N° attribué par la bibliothèque

année : 2000

THÈSE

présentée à

l'ÉCOLE POLYTECHNIQUE

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité

Informatique

soutenue par

Bruno Blanchet

le 7 décembre 2000

Titre

Analyse d'échappement. Applications à ML et Java™
Escape Analysis. Applications to ML and Java™

Directeur de thèse : Patrick Cousot

Jury

Président	David Schmidt
Rapporteurs	Chris Hankin Nevin Heintze
Examinateurs	Patrick Cousot Alain Deutsch Vania Joloboff Xavier Leroy Mary Lou Soffa

Remerciements

M. David Schmidt me fait l'honneur d'être président de mon jury de thèse. Je l'en remercie sincèrement.

MM. Chris Hankin et Nevin Heintze ont été les rapporteurs de cette thèse. Je tiens à les remercier pour le travail considérable de lecture qu'ils ont effectué. Je voudrais également remercier M. Nevin Heintze pour ses commentaires pertinents qui ont contribué à améliorer ma thèse.

Je voudrais dire un très grand merci à MM. Patrick Cousot et Alain Deutsch, sans qui cette thèse n'aurait jamais existé.

M. Patrick Cousot, en tant que professeur à l'École Normale Supérieure, m'a fait découvrir l'interprétation abstraite dont il est l'inventeur. Cette théorie de l'analyse de programme est à la base de cette thèse. C'est grâce à lui que j'ai découvert ce domaine si passionnant. Il a bien voulu ensuite être mon directeur de thèse. Je tiens à l'en remercier chaleureusement.

M. Alain Deutsch a poursuivi ma formation à l'interprétation abstraite au DEA de Sémantique, Preuves et Programmation. C'est lui qui m'a encadré durant mon stage de DEA et ma thèse. Il est à l'origine de mon travail sur l'analyse d'échappement. Je lui adresse toute ma gratitude.

Mme Mary Lou Soffa et MM. Vania Joloboff et Xavier Leroy ont bien voulu faire partie de mon jury de thèse. Qu'ils en soient ici remerciés. M. Vania Joloboff m'a donné accès au compilateur turboJ que j'ai utilisé pour expérimenter mon analyse d'échappement pour Java. M. Xavier Leroy est le principal concepteur du langage Objective Caml, auquel j'ai appliqué mes analyses de programmes. Il m'a également permis d'approfondir mes connaissances en compilation grâce à son cours de DEA. Je tiens à remercier Mme Mary Lou Soffa et MM. Nevin Heintze et David Schmidt d'avoir fait le voyage depuis les Etats-Unis pour ma soutenance.

Mes remerciements vont également à tous les membres du projet Para (devenu maintenant Moscova) de l'INRIA, dans lequel j'ai été accueilli pour ma thèse, et plus généralement à tous ceux que j'ai côtoyés dans les bâtiments 8 et 11. Ils m'ont permis d'effectuer mon travail dans les meilleures conditions, et dans une ambiance très agréable.

Je tiens également à remercier MM. Sylvain Conchon, Ian Mackie et François Pottier pour leurs commentaires sur des premières versions de cette thèse. Ils ont largement contribué à améliorer la rédaction et la clarté des explications.

Contents

1 Introduction (Français) **9**

2 Introduction (English) **17**

I Escape Analysis for ML **23**

3 Design and correctness proof of escape analyses **25**
3.1 Preliminaries 25
3.1.1 Complete partial orders (CPO) 25
3.1.2 Abstract interpretation 25
3.2 Analyzed language 28
3.2.1 Syntax 28
3.2.2 Typing 31
3.2.3 Hierarchy of analyses 35
3.3 Analysis E 36
3.3.1 Introduction 36
3.3.2 Formal definition and properties 40
3.4 Towards efficient escape analyses 46
3.4.1 Adapting analysis E to types 46
3.4.2 A general framework for escape analysis 48
3.4.3 Properties of analysis E 52
3.5 Analysis based on annotated types 53
3.5.1 Definition of the graph of a type 53
3.5.2 Abstraction 56
3.5.3 Correctness 66
3.5.4 Properties of special kinds of types 67
3.6 Analyses based on integer contexts 68
3.6.1 Analysis L (closures represented by ∞) 68
3.6.2 Analysis F (without enumeration of free variables) 77
3.6.3 Other integer escape analyses 78

4 Implementation of analyses L and F **79**
4.1 Computing type heights 79
4.2 Building the equations of the analysis 80
4.3 Solving of equations 86
4.4 Program transformation 88

4.5 Complexity of the analyses . . . 94
4.6 A few points to note . . . 94
4.7 Conclusion . . . 96

5 Implementation of analysis T 97
5.1 Building of the graph . . . 97
5.1.1 Definition of the exploded super-graph . . . 97
5.1.2 Algorithm to build the exploded super-graph . . . 105
5.2 Computing summary edges . . . 111
5.3 Computing escaping parts . . . 112
5.3.1 Algorithm . . . 112
5.3.2 Correctness . . . 113
5.3.3 Optimized algorithm . . . 113
5.4 Complexity . . . 114
5.5 Optimizations . . . 115
5.5.1 Reducing the graph . . . 115
5.5.2 Compacting the graph . . . 115
5.5.3 Recursive types . . . 116
5.5.4 A difference with the real implementation . . . 117
5.5.5 Program transformation . . . 119

6 Benchmarks and comparisons 121
6.1 Our benchmark programs . . . 121
6.2 Comparison between different escape analyses . . . 121
6.3 The effects of stack allocation . . . 127
6.4 Comparison between different machines . . . 137
6.5 Conclusion . . . 140

II Escape Analysis for Java 141

7 Design of escape analysis for Java 143
7.1 Notations . . . 143
7.2 Analyzed language . . . 143
7.2.1 Syntax and informal semantics . . . 143
7.2.2 Analysis hierarchy . . . 146
7.3 Alias Analysis . . . 147
7.4 Analysis $\mathbb{E}$: Escape Analysis using Access Paths . . . 151
7.4.1 Abstract lattice for values . . . 151
7.4.2 The analysis . . . 153
7.4.3 Correctness proof . . . 157
7.5 Analysis $\mathbb{L}$: Escape Analysis using Integer Contexts . . . 160
7.6 Representation of context transformers . . . 167
7.7 Extensions . . . 168

8 Implementation **171**
8.1 Computing type levels . . . 171
8.2 The analysis algorithm . . . 171
8.3 Program transformation . . . 172
8.3.1 Stack allocation . . . 172
8.3.2 Reuse allocated space in loops . . . 173
8.3.3 Inlining . . . 174
8.3.4 Synchronization elimination . . . 178
8.4 Complexity . . . 181

9 Benchmarks **183**
9.1 Benchmark programs . . . 183
9.2 Results . . . 183
9.3 Analysis speed . . . 192
9.4 Conclusion . . . 193

10 Related and future work **195**
10.1 Comparison with related work . . . 195
10.1.1 Various escape analyses and similar analyses . . . 195
10.1.2 Program transformations . . . 200
10.1.3 Study of the impact of stack allocation . . . 202
10.2 Future work . . . 202

11 Conclusion **205**

A Proofs for analysis E for ML **207**
A.1 Correctness proof . . . 207
A.1.1 Semantics . . . 208
A.1.2 Correctness . . . 211
A.2 Adapting analysis E to types . . . 233
A.3 From monomorphic to polymorphic properties . . . 236

B Proofs for analysis T **239**
B.1 Definition of the graph of a type . . . 239
B.2 Generalization . . . 241
B.3 Instantiation . . . 241

C Other integer escape analyses **251**
C.1 Analysis L' (finite contexts only) . . . 251
C.2 Analysis L_1 (analysis L' limited to one step) . . . 254
C.3 Comparison between different integer escape analyses . . . 256

D Correctness of the implementation of analysis T **263**
D.1 Graph building . . . 263
D.2 Computing summary edges . . . 279
D.3 Correctness of the whole algorithm . . . 283

E Proofs of escape analysis for Java **285**
E.1 Correctness of alias analysis 285
E.2 Correctness proof for analysis $\mathbb{E}$ 289
E.3 Inlining 297
E.4 Global synchronization analysis 299

Chapitre 1

Introduction (Français)

The English speaking reader will find the translation of this chapter in Chapter 2.

Dans beaucoup de langages de programmation modernes, la gestion de la mémoire est automatique. C'est le cas notamment dans des langages fonctionnels comme ML, et dans des langages orientés-objet comme Java™ ou certaines implantations de C++. Cette gestion automatique se fait à l'aide d'un glaneur de cellules (GC, *garbage collector*). Le programmeur indique explicitement chaque donnée qu'il souhaite allouer, mais il n'indique plus quand la donnée doit être désallouée. Le GC parcourt les données présentes en mémoire pour détecter les données qui ne sont plus accessibles, et libère celles-ci.

Ces techniques de GC ont de très importants avantages. En particulier, elles évitent de laisser au programmeur l'initiative de désallouer lui-même les données ce qui était la source de nombreuses erreurs difficiles à corriger. Le programmeur peut en effet oublier de libérer des données, ce qui est la source de fuites de mémoire : la taille mémoire disponible diminue de façon excessive. Il peut aussi, chose plus grave, libérer trop tôt une donnée. Ceci conduit en général à l'arrêt du programme sur une erreur mémoire. La sécurité qu'apporte le GC, qui est souhaitable pour tous les langages, est même indispensable pour le langage Java car il a été conçu pour pouvoir exécuter des applications potentiellement hostiles sans risque pour la machine de l'utilisateur. L'exécution de ces applications ne doit donc pas causer d'erreurs mémoire.

Cependant, l'utilisation d'un GC a aussi un coût, qui peut être important dans certains cas. La première motivation de ce travail de thèse est d'étudier une autre méthode de gestion de la mémoire : l'allocation en pile. Les données sont allouées dans une pile et libérées dans l'ordre inverse de leur allocation. Comme on utilise la même pile pour allouer les données et empiler les adresses de retour des fonctions, on ne peut allouer en pile qu'une donnée qui devient inutile avant le retour de la fonction dans laquelle elle est allouée. Pour conserver les avantages du GC, on ne va pas laisser au programmeur l'initiative de décider quels objets il va allouer en pile. On va utiliser une analyse statique du programme pour déterminer des données que l'on peut allouer en pile. Cette analyse s'appelle l'*analyse d'échappement*. Son but est de déterminer les données dont la durée de vie (le temps pendant lequel elles doivent être conservées) ne dépasse pas la portée statique (la portée de leur déclaration). En Java, la portée statique est la méthode dans laquelle l'objet est alloué. En ML, la portée statique d'une variable `x` définie par `let x = M in N` est le corps `N` du `let`. On peut modifier cette portée en déplaçant le `let` (tout en restant bien sûr à l'intérieur de la même fonction).

Considérons par exemple le programme suivant :

```
let rec filter f l =
    match l with
        [] -> []
      | (a::r) -> if f a
                  then a :: filter f r
                  else filter f r


filter (fun x -> x mod 2 = 0) [1;2;3;4]
```

La fonction `filter f l` filtre les éléments `a` de la liste `l` tels que `f a` est vrai. La fermeture `fun x -> x mod 2 = 0` et la liste `[1 ;2 ;3 ;4]` ne sont plus accessibles après l'appel à `filter`. Elles peuvent donc être allouées en pile. Au contraire, la liste créée par `filter` ne peut pas être allouée en pile car elle est encore accessible à la fin de la fonction `filter`.

On introduit une nouvelle expression `letstack x = M in N`, qui signifie que la valeur construite par `M` est allouée en pile (*stack*) et désallouée à la fin de `N`. Le programme transformé serait alors (ceci est légèrement simplifié par rapport à la transformation réelle) :

```
letstack f = fun x -> x mod 2 = 0 in
letstack l = [1;2;3;4] in
   filter f l
```

En Java, on peut considérer par exemple la classe suivante, qui définit des vecteurs de taille bornée `n`.

```
class LimVect {
  int count = 0;
  Object[] el;

  LimVect(int n) { el = new Object[n]; }

  void put(Object o) { el[count++] = o; }

  Object get(int n) { return el[n]; }

  static Object run() {
    LimVect local = new LimVect(4);
    local.put(new Integer(1));
    return local.get(0);
  }
}
```

L'attribut `count` correspond au nombre d'objets insérés dans le vecteur, et l'attribut `el` au tableau des objets insérés. La méthode `put` insère un objet supplémentaire, alors que `get(n)` retourne le `n`^{e} objet inséré.

Dans la méthode `run`, le vecteur `local` devient inaccessible à la fin de `run`, et peut donc être alloué en pile. Le tableau d'objets à l'intérieur de ce vecteur est évidemment encore utile à la fin du constructeur `LimVect(int n)`, mais devient aussi inaccessible à la fin de `run`. On peut donc l'*inliner* pour effectuer l'allocation en pile : l'appel au constructeur est remplacé par une copie de son code dans la méthode `run`. Le tableau d'objets est alors alloué en pile dans la méthode `run` transformée. Ceci donne le code transformé suivant, dans lequel les allocations en pile sont représentées par `alloca_new` :

```
static Object run() {
  LimVect local = alloca_new LimVect();
  local.el = alloca_new Object[4];
  local.put(new Integer(1));
  return local.get(0);
}
```

Nous avons appliqué l'analyse aux deux langages ML (plus précisément son dialecte Objective Caml) et Java. Java est d'abord compilé en bytecode et les impératifs de sécurité exigent que l'exécution de ce bytecode soit sans danger pour le système. La correction des allocations en pile est, on l'a vu, indispensable pour assurer cette sécurité. Il faut donc soit vérifier cette correction au moment de l'exécution du bytecode, soit effectuer l'analyse sur le bytecode lui-même, pour qu'un pirate ne puisse pas introduire de fausses allocations en pile dans du bytecode de sa fabrication. Nous avons ici choisi d'effectuer l'analyse sur le bytecode lui-même.

Plusieurs méthodes sont possibles pour exécuter le bytecode Java. La première chronologiquement, celle utilisée par le *Java Development Kit (JDK)* de *Sun* version 1.0.2, est l'interprétation. Ici, on ne considérera pas cette méthode car le surcoût dû à l'interprétation est très important et risquerait de rendre négligeable toute amélioration dans l'allocation des données. Une autre méthode est d'utiliser un compilateur *Just in Time* : le code est compilé avant chaque exécution. Cette méthode donne des performances nettement meilleures (l'exécution est une dizaine de fois plus rapide) et est utilisée dans des *JDK* plus récents (1.1.7 sous Windows par exemple). Enfin, on peut aussi traduire l'application en un code exécutable directement, comme cela est fait pour les programmes C. Notre travail pourrait s'appliquer à ces deux dernières techniques. Notre implantation est effectuée dans un compilateur qui produit du code exécutable à partir du bytecode, en passant par l'intermédiaire du langage C.

Même si l'analyse d'échappement poursuit un but analogue pour les deux langages ML et Java, il y a de très importantes différences entre ces langages. Les difficultés que l'on rencontre dans la conception de l'analyse d'échappement pour chacun de ces langages ne sont donc pas les mêmes.

- ML est un langage fonctionnel, qui utilise l'ordre supérieur, ce qui n'est pas le cas de Java. Il est important de traiter l'ordre supérieur de façon assez précise car les fermetures sont des données souvent allouables en pile. La présence de l'ordre supérieur complique beaucoup la définition de la correction de l'analyse d'échappement, et l'implantation de l'analyse.

  La construction qui correspond en Java est l'utilisation de méthodes virtuelles (c'est-à-dire des méthodes qui sont attachées aux objets et non aux classes). Mais elles sont plus faciles à traiter que l'ordre supérieur de ML car la classe, le nom et les types des paramètres de la méthode donnent un ensemble de méthodes susceptibles d'être appelées.
- Le polymorphisme prend une forme radicalement différente dans les deux langages. En ML, il apparaît sous forme de variables de types. Ainsi la fonction identité est de type $\beta \to \beta$ où β est une variable de type. Ceci signifie qu'elle peut prendre un argument de n'importe quel type et redonner un résultat du même type. En Java, le polymorphisme passe par l'intermédiaire du sous-typage[1]. Une méthode qui prend des arguments d'un

[1] Si une classe C hérite d'une classe C' par C `extends` C', C est aussi dite sous-classe (ou sous-type) de

certain type τ peut en fait être utilisée avec tous les sous-types de τ. Toutes les méthodes sont donc potentiellement polymorphes : elles peuvent être appelées avec des arguments de plusieurs types. La fonction identité en Java aurait le type `Object` $\rightarrow$ `Object` (cette notation est inhabituelle en Java, mais je l'utilise ici par analogie avec ML). Elle peut prendre n'importe quel objet en argument, car tout type objet τ est un sous-type de `Object`, et elle rend un `Object`. On a donc moins d'information que pour ML, où on savait que le résultat était du type τ de l'argument.

L'analyse du polymorphisme de Java est assez simple, mais conduit à une inévitable perte de précision. Une telle analyse serait utilisable aussi pour ML, mais on a pu mettre au point une analyse plus précise du polymorphisme à la ML.

- Les opérations impératives (affectations) sont présentes dans les deux langages, mais n'ont pas la même utilité. En ML, elles sont très peu utilisées, et l'approximation simple suivante convient tout à fait : on considère que toute donnée apparaissant dans un membre droit d'affectation est susceptible d'être utile jusqu'à la fin du programme. Au contraire, l'affectation est très utilisée en Java, et une telle approximation conduirait à une perte de précision très importante. C'est pourquoi on a conçu une analyse plus précise, mais aussi plus compliquée des affectations pour Java.

A ces différences intrinsèques entre les deux langages s'ajoutent aussi des différences au niveau du GC qu'elles utilisent dans les implantations considérées. Le GC d'Objective Caml est beaucoup plus efficace pour les données à courte durée de vie que le GC du JDK 1.1.5 que nous avons utilisé. On peut donc s'attendre à ce que ces différences se traduisent dans les résultats expérimentaux par des effets différents de l'allocation en pile.

L'allocation en pile peut avoir plusieurs effets positifs sur l'exécution d'un programme. Elle diminue tout d'abord la charge du GC qui n'a pas à gérer les données allouées en pile. Cela peut se traduire par une diminution du temps de GC. Elle améliore également la localité des données. En effet, les données utiles se situent au sommet de pile, et celui-ci a tendance à rester en permanence dans le cache car il est très souvent utilisé. On peut donc s'attendre à une diminution du nombre d'échecs lors des accès au cache de données. Enfin, l'allocation en pile est très rapide à effectuer (simplement un registre à changer). On pourra donc observer une diminution du temps d'allocation. On s'attachera dans la suite à étudier ces différents effets de l'allocation en pile, et à comparer ces effets sur nos deux plateformes d'expérimentation.

Ces différences importantes justifient la séparation en deux parties de cette thèse. La première partie concerne ML alors que la seconde concerne Java. On n'omettra cependant pas tout au long de la thèse d'effectuer des comparaisons entre les deux langages.

Nous avons aussi remarqué que l'analyse d'échappement peut avoir plusieurs applications :

- L'allocation en pile, qui était notre motivation initiale.
- L'élimination des synchronisations sur des objets locaux à un *thread* (processus léger), en Java. En effet, Java peut exécuter plusieurs *threads* en parallèle, et il faut alors effectuer des synchronisations entre les accès aux objets partagés entre plusieurs *threads* pour garantir la cohérence des données. Chaque objet a un verrou associé. Quand on appelle une méthode (virtuelle) synchronisée, le verrou de l'objet récepteur `this` est acquis. Ce verrou est ensuite libéré à la fin de l'appel de la méthode. Ceci fournit un mécanisme d'exclusion mutuelle entre les méthodes synchronisées qui travaillent sur le même objet.

C'. On étend cette relation de sous-typage au types tableaux : si τ est un sous-type de τ', τ`[]` est un sous-type de τ'`[]`, et par transitivité : si τ est sous-type de τ' et τ' sous-type de τ'', τ est un sous-type de τ''.

Ces synchronisations sont évidemment inutiles pour les objets qui ne sont accédés que par un unique *thread*. Or, si notre analyse d'échappement découvre qu'un objet a une durée de vie limitée à une certaine méthode, alors cet objet est aussi local au *thread* dans lequel il a été alloué. On peut ainsi éliminer des synchronisations inutiles.

Un exemple de ceci peut être trouvé dans le test Symantec (un peu simplifié) :

```
public void setData(int p1) {
    a = new int[p1];
    Random r = new Random(seed);
    for(int i = 0; i < a.length; i++) {
        a[i] = r.nextInt();
    }
}
```

Les appels à `r.nextInt()` appellent `Random.next`, qui est synchronisée.

```
public int nextInt() {  return next(32); }

synchronized int next(int bits) {
    long nextseed = (seed * multiplier + addend) & mask;
    seed = nextseed;
    return (int)(nextseed >>> (48 - bits));
}
```

`r` ne s'échappe pas de la méthode `setData`, donc est local au thread courant, donc on peut appeler une version non synchronisée de `Random.next`.

Ceci est une optimisation importante en Java car les méthodes des bibliothèques sont conçues pour être correctes même en cas d'exécutions sur des objets partagés entre plusieurs threads, et donc sont très souvent synchronisées. Ces synchronisations sont la plupart du temps inutiles. De plus, l'opération de synchronisation est coûteuse.

Nous avons utilisé ML comme plateforme d'expérimentation dans laquelle nous avons implémenté essentiellement deux classes différentes d'algorithmes d'analyse d'échappement, qui se distinguent par des niveaux de précision différents (avec des variantes mineures à l'intérieur de ces deux classes). Nous avons ensuite choisi l'algorithme qui offrait le meilleur compromis entre coût et précision pour l'adapter à Java.

Pour ML, notre première contribution est une preuve de correction de l'analyse d'échappement à partir d'une sémantique dénotationnelle. Nous prouvons la correction de plusieurs analyses d'échappement. La première et la plus précise est une analyse dans laquelle les parties des données sont représentées par des chemins d'accès. Cette analyse est trop complexe pour être implémentée mais elle sert de base pour les analyses qui suivent. Nous prouvons ensuite la correction d'une analyse d'échappement qui représente les parties des données par des types annotés, et de plusieurs analyses utilisant des entiers. Ces analyses traitent l'ordre supérieur, les affectations, le polymorphisme, les types récursifs. L'analyse à types annotés étend et améliore les travaux de Markus Mohnen [Moh95a, Moh95b] en prouvant la correction des extensions au polymorphisme, en proposant une extension plus efficace aux fonctions d'ordre supérieur, en étendant l'analyse aux opérations impératives. Nous proposons aussi une implantation nettement plus efficace de cette analyse que celle de Mohnen, qui lui permet de passer à l'échelle pour analyser de gros programmes. Les analyses à entiers sont des extensions du travail d'Alain Deutsch [Deu97] qui y ajoutent les types récursifs généraux, et une preuve de correction à partir de la sémantique. Nous traitons les extensions suggérées par Alain Deutsch (polymorphisme, ordre supérieur, affectations) plus en détail.

Notre deuxième contribution est une preuve de correction de l'implantation de l'analyse qui utilise des types annotés. Etant donné la complexité de cet algorithme, certaines optimisations (cruciales pour permettre l'exécution de cet algorithme à un coût raisonnable) ne sont pas prouvées.

Enfin, notre troisième contribution est une implantation complète dans le langage Objective Caml 1.05, qui nous a permis d'effectuer des tests expérimentaux détaillés sur de grosses applications. Cette implantation comporte plusieurs extensions : l'expansion en ligne des petites fonctions est utilisé pour augmenter les opportunités d'allocation en pile ; la portée des données allouées en pile est automatiquement choisie de façon à minimiser leur durée de vie tout en permettant l'allocation en pile ; si possible, les données allouées en pile sont libérées juste avant un éventuel appel terminal de la fonction, de façon à pouvoir coder cet appel terminal par un simple saut au lieu d'un véritable appel de fonction. Trois analyses ont été implémentées : celle sur les types annotés et deux analyses utilisant des entiers. La comparaison entre ces différentes analyses montre que l'analyse à entiers la plus rapide offre déjà une précision tout à fait satisfaisante en pratique. Cette analyse est implémentée avec une complexité $\mathcal{O}(n \log^2 n)$ où n est la taille du programme, donnée par Alain Deutsch [Deu97]. Nous avons également analysé les causes de l'accélération de l'exécution obtenue par nos optimisations, et effectué des comparaisons entre plusieurs architectures de processeurs pour voir l'influence de facteurs tels que la vitesse du cache de niveau 2 et la politique en cas d'échec en écriture.

En ce qui concerne l'analyse d'échappement pour Java, notre première contribution est la conception d'une analyse qui traite précisément les affectations tout en utilisant des entiers pour représenter les parties de données. Cette représentation à l'aide d'entiers offre un très bon compromis entre vitesse d'analyse et précision comme l'étude sur le langage ML l'a montré.

Nous avons également effectué la preuve de correction de cette analyse à partir d'une sémantique opérationnelle à petits pas. Comme pour ML, cette preuve utilise comme intermédiaire une analyse d'échappement à chemins d'accès. De plus, elle utilise aussi une analyse d'alias.

Enfin, notre dernière contribution est une implantation complète pour le langage Java. Cette implantation a été effectuée dans le compilateur turboJ [W+98] de Silicomp Research Institute (`http://www.ri.silicomp.com`). Ce compilateur compile du bytecode Java en C. Notre implantation utilise l'expansion en ligne des petites méthodes pour augmenter les possibilités d'allocation en pile. Quand une allocation a lieu dans une boucle, le même espace mémoire est réutilisé à l'itération suivante si possible. On effectue également de l'élimination de synchronisations, grâce à une analyse globale qui détecte les objets locaux à un *thread*. Nous avons effectué des tests sur de grosses applications, et analysé les causes de l'accélération obtenue.

Plan

Le premier chapitre d'introduction est inclus à la fois en français et en anglais.

La suite de la thèse est divisée en deux parties. La première partie concerne le langage ML. Le chapitre 3 décrit les différentes analyses d'échappement pour ML, et donne une preuve de leur correction. Le chapitre 4 donne tous les détails d'implantation concernant les analyses d'échappement à entiers pour ML. Le chapitre 5 traite de l'implémentation de l'analyse à types annotés. Enfin, le chapitre 6 détaille nos résultats expérimentaux. Les preuves détaillées de la plupart des résultats, ainsi que la description de certaines analyses à entiers, ont été reportées en appendice pour faciliter la lecture.

La deuxième partie concerne Java. Elle a sensiblement la même structure que la partie traitant de ML. Elle commence par le chapitre 7 qui décrit notre analyse et prouve sa correction. Ensuite, le chapitre 8 donne son implantation, et le chapitre 9 décrit les résultats expérimentaux.

Le corps de la thèse se termine par un chapitre de comparaison avec les autres travaux concernant l'analyse d'échappement et des travaux similaires, aussi bien pour des langages fonctionnels que pour Java.

Lectures

Le lecteur intéressé par l'analyse de ML lira les chapitres 3, 4 et 6. La lecture du chapitre 5 peut être omise sans nuire à la compréhension de l'ensemble.

Les chapitres 7, 8 et 9 concernent l'analyse de Java. La lecture des chapitres concernant ML n'est pas indispensable pour comprendre cette partie, si l'on excepte les paragraphes « comparaison avec ML ». Cependant, le lecteur qui désire des rappels d'interprétation abstraite lira la section 3.1 et pourra consulter les références qu'elle cite.

La lecture des appendices n'est destinée qu'au spécialiste qui desire avoir une preuve détaillée de la correction des analyses utilisées.

L'index en fin de thèse permettra au lecteur de retrouver facilement les principales notations et les définitions.

Chapter 2

Introduction (English)

In many modern programming languages, memory management is automatic. This is the case for instance in functional languages such as ML, and in object-oriented languages such as Java^{TM} or some implementations of C++. This automatic management is achieved by using a garbage collector (GC). The programmer mentions explicitly each value he wants to allocate, but does not indicate when the value should be deallocated. The GC scans the data in memory to detect unreachable data and deallocates them.

These GC techniques have very important advantages. In particular, they avoid leaving to the programmer the task of deallocating himself data, which was the source of many difficult-to-correct errors. The programmer can indeed forget to deallocate data, which is the source of memory leaks: the available memory size decreases excessively. He can also, and this is more serious, deallocate data too early. This often leads to the termination of the program with an error (memory fault). The safety brought by the GC, which is desirable for all programming languages, is even necessary for the Java language, since it has been designed to be able to execute potentially hostile programs without risk for the user. The execution of these programs must therefore not cause memory faults.

However, using a GC also has a cost, which may be large in some cases. The first goal of this work is to study another memory management technique: stack allocation. Data are allocated in a stack, and deallocated in the reverse order of their allocation. Since we use the same stack for allocating data and for storing the return addresses of the functions, a value can only be stack allocated if it becomes useless before the return of the function in which it is allocated. To preserve the advantages of the GC, we do not leave to the programmer the decision of which objects he will stack allocate. We are going to use a static analysis of the program to determine data that can be stack allocated. This analysis is called *escape analysis*. Its goal is to determine data whose lifetime (the time during which they have to be kept in memory) does not exceed their static scope (the scope of their declaration). In Java, the static scope is the method in which the object is allocated. In ML, the static scope of a variable `x` defined by `let x = M in N` is the body of the `let`. This scope can be modified by moving the `let` (of course, we have to stay in the same function).

Let us consider for example the following Caml program:

```
let rec filter f l =
    match l with
        [] -> []
      | (a::r) -> if f a then a :: filter f r else filter f r
```

```
filter (fun x -> x mod 2 = 0) [1;2;3;4]
```

The function `filter f l` filters the elements `a` of list `l` such that `f a` is true. The closure `fun x -> x mod 2 = 0` and the list `[1;2;3;4]` are not reachable any more after the call to `filter`. They can therefore be stack allocated. On the contrary, the list created by `filter` cannot be stack allocated since it is still reachable at the end of the `filter` function.

We introduce a new expression `letstack x = M in N`, which means that the value constructed by `M` is stack allocated, and deallocated at the end of `N`. The transformed program would then be (this is slightly simplified with respect to the real transformation):

```
letstack f = fun x -> x mod 2 = 0 in
letstack l = [1;2;3;4] in
   filter f l
```

In Java, consider for example the following class, which defines vectors of maximum size `n`.

```
class LimVect {
  int count = 0;
  Object[] el;

  LimVect(int n) { el = new Object[n]; }

  void put(Object o) { el[count++] = o; }

  Object get(int n) { return el[n]; }

  static Object run() {
    LimVect local = new LimVect(4);
    local.put(new Integer(1));
    return local.get(0);
  }
}
```

The field `count` corresponds to the number of objects inserted in the vector, and the field `el` to the array of inserted objects. The method `put` inserts an object in the vector, whereas `get(n)` returns the `n`th inserted object.

In the method `run`, the vector `local` becomes unreachable at the end of `run`, and can therefore be stack allocated. The array of objects inside this vector is of course still useful at the end of the constructor `LimVect(int n)`, but becomes unreachable at the end of `run`. Therefore, the constructor can be inlined to perform the stack allocation: the call to the constructor is replaced with a copy of its code in the method `run`. The array of objects is then stack allocated in the transformed method `run`. This yields the following transformed program, in which stack allocations are represented by `alloca_new`:

```
static Object run() {
  LimVect local = alloca_new LimVect();
  local.el = alloca_new Object[4];
  local.put(new Integer(1));
  return local.get(0);
}
```

We have applied the analysis to both languages ML (more precisely its dialect Objective Caml) and Java. Java is first compiled into bytecode, and safety constraints demand that the execution of the bytecode does not jeopardize the system. The correctness of stack allocations is, as we have seen, necessary to achieve this safety. Therefore, either this correctness must be checked when running the bytecode, or the analysis must be performed on the bytecode itself, so that a hacker cannot introduce wrong stack allocations in its own bytecode. We have chosen here to perform the analysis on the bytecode itself.

Several methods are possible to run Java bytecode. Chronologically the first one is interpretation, used in the *Java Development Kit (JDK)* from *Sun*, version 1.0.2. Here, we will not consider this technique, since the cost of interpreting the bytecodes is very large, and would make negligible any improvement in data management. Another technique is to use a *Just in Time* compiler: the code is compiled before each execution. This method yields much better results (the execution is about ten times faster), and is used in more recent *JDK*s (1.1.7 under Windows for instance). The program can also be translated into native code directly executable by the processor, as it is done for C programs. Our work could be applied to the latter two techniques. Our analysis is implemented in a compiler which outputs native code from the bytecode, using C as an intermediate language.

Even if escape analysis seeks a similar goal for both languages ML and Java, there are very important differences between these languages. The difficulties we meet for designing escape analysis for each of these languages are therefore not the same.

- ML is a functional language, with higher-order functions, which is not the case for Java. It is important to handle higher-order functions precisely enough since closures can often be stack allocated. Higher-order functions complicate the definition of the correctness of escape analysis, and the implementation of the analysis.

  The corresponding concept in Java uses virtual methods (i.e. methods which are attached to objects and not to classes). However, they are easier to handle than higher-order functions in ML, since the class, name and types of parameters of the method yield a set of methods that may be called.

- The polymorphism takes a radically different form in the two languages. In ML, it appears under the form of type variables. The identity function is then of type $\beta \to \beta$ where β is a type variable. This means that it can take an argument of any type, and returns a result of the same type. In Java, the polymorphism comes from subtyping[1]. A method which takes arguments of a given type τ can in fact be used with all subtypes of τ. All methods are therefore potentially polymorphic: they can be used with arguments of several types. The identity function in Java would have type `Object` $\to$ `Object` (this notation is unusual in Java, but I use it here by analogy with ML). It can take any object as argument, since every object type τ is a subtype of `Object`, and it returns an `Object`. We have therefore less information than in ML, where we knew that the result is of the type τ of the argument.

  The analysis of Java polymorphism is simple enough, but leads to an unavoidable loss of precision. Such an analysis would also be usable for ML, but we have been able to design a more precise analysis of the ML polymorphism.

[1] If a class C inherits from a class C' by C `extends` C', C is also said to be a subclass (or subtype) of C'. This subtyping relation is extended to arrays: if τ is a subtype of τ', τ`[]` is a subtype of τ'`[]` and by transitivity: if τ is a subtype of τ' and τ' a subtype of τ'', τ is a subtype of τ''.

- Imperative operations (assignments) are present in both languages, but they are not equally useful in ML and Java. In ML, they are not very often used, and the following simple approximation is perfectly suitable: all data appearing in the right-hand side of an assignment are assumed to be reachable until the end of the program. On the contrary, assignments are very often used in Java, and such an approximation would lead to a very important loss of precision. That is why we have designed a more precise, but also more complicated, analysis of the assignments for Java.

On top of these intrinsic differences between both languages, there are also differences between the GC they use in the considered implementations. The GC of Objective Caml is much more efficient for short lived data than the GC of the JDK 1.1.5, the JDK we have used. We can expect that these differences lead to different effects of stack allocation in the experiments.

Stack allocation can have several positive effects on the execution of a program. First, it reduces the GC workload, since it does not have to manage stack allocated data. This can lead to a decrease of the GC time. Stack allocation also improves data locality. Useful data are at the top of the stack, which tends to stay in the cache since it is very often used. We can therefore expect a decrease of the number of data cache misses. At last, allocation on the stack can be performed very quickly (only shifting a register). Allocation time should therefore decrease. In the following, we will study the different effects of stack allocation, and compare these effects on our two experimental platforms.

These important differences justify the splitting of this thesis into two parts. The first part deals with ML, whereas the second one deals with Java. However, we shall not forget to compare between both languages during the whole thesis.

We have also noticed that escape analysis can have several applications:

- Stack allocation, which was our first motivation.

- Synchronization elimination on thread-local objects, in Java. Indeed, Java can run several threads (light-weight processes) in parallel, and synchronizations have to be performed when objects shared between several threads are accessed to guarantee data coherence. Each object has an associated lock. When a synchronized (virtual) method is called, the lock of the receiver `this` is acquired. This lock is then released at the end of the method call. This provides a mutual exclusion mechanism between synchronized methods which work on the same object. These synchronizations are obviously useless for objects which are accessed by a single thread. If our escape analysis discovers that an object has a lifetime limited to a certain method, then this object is also local to the thread in which it has been allocated. Useless synchronizations can therefore be eliminated.

  An example of this can be found in the Symantec benchmark (slightly simplified):

  ```
  public void setData(int p1) {
      a = new int[p1];
      Random r = new Random(seed);
      for(int i = 0; i < a.length; i++) {
          a[i] = r.nextInt();
      }
  }
  ```

  The calls to `r.nextInt()` call `Random.next`, which is synchronized.

```
public int nextInt() {  return next(32); }

synchronized int next(int bits) {
    long nextseed = (seed * multiplier + addend) & mask;
    seed = nextseed;
    return (int)(nextseed >>> (48 - bits));
}
```

`r` does not escape from method `setData`, hence is local to the current thread. A version of `Random.next` without synchronization can therefore be called.

This is an important optimization in Java since methods of libraries are designed to be correct even if they are called on objects shared between several threads, and are therefore very often synchronized. These synchronizations are most of the time useless. Moreover, the synchronization operation is costly.

We have used ML as an experimentation platform in which we have implemented essentially two classes of escape analysis algorithms, which differ by their level of precision (with minor variants inside these classes). We have chosen the algorithm which gave the best cost/precision compromise to adapt it to Java.

For ML, our first contribution is a correctness proof of escape analysis from a denotational semantics. We prove the correctness of several escape analyses. The first and most precise is an analysis in which parts of data are represented by access paths. This analysis is too complex to be implemented, but it is the basis for the following analyses. Then, we prove the correctness of an escape analysis which represents parts of data by annotated types, and several analyses using integers. These analyses handle higher-order functions, assignments, polymorphism, recursive types. The annotated types analysis extends and improves upon work by Markus Mohnen [Moh95a, Moh95b] by proving the correctness of the polymorphic extensions, by proposing a more efficient extension to higher-order functions, and by extending the analysis to imperative operations. We also give a much more efficient implementation of this analysis than Mohnen's, which enables it to scale to the analysis of large programs. The integer escape analyses are extensions of Alain Deutsch's work [Deu97] which add general recursive types, and a proof of correctness from the semantics. We handle the extensions suggested by Alain Deutsch (polymorphism, higher-order, assignments) in more detail.

Our second contribution is a correctness proof of the implementation of the analysis using annotated types. Because of the complexity of this algorithm, some optimizations (crucial to be able to execute this algorithm at a reasonable cost) are not proved.

At last, our third contribution is a full implementation in the language Objective Caml 1.05, which has enabled us to perform detailed experimental tests on large programs. This implementation contains several extensions: inlining of small functions to increase stack allocation opportunities; the scope of stack allocated data is automatically chosen to minimize their lifetime while allowing stack allocation; when possible, stack allocated data are deallocated just before a tail call of the function, to be able to code this tail call by a jump instead of a real function call. Three analyses have been implemented: the one based on annotated types, and two analyses using integers. The comparison between these different analyses shows that the fastest integer analysis already offers a very satisfactory precision in practice. This analysis is implemented in a complexity $\mathcal{O}(n \log^2 n)$ where n is the size of the program, given by Alain Deutsch [Deu97]. We have also analyzed the causes of the speedups obtained by our optimizations, and compared between several processor architectures to analyze the influence of factors such as the speed of the L2 cache and the write miss policy.

For the escape analysis for Java, our first contribution is the design of an analysis which handles assignments precisely while using integers to represent parts of data. This representation using integers yields a very good compromise between analysis speed and precision. This has been demonstrated by the study on the ML language.

We have also performed the correctness proof of this analysis from a small step operational semantics. As for ML, this proof uses an escape analysis based on access paths as intermediate step. Moreover, it also uses an alias analysis.

Our last contribution is a full implementation for the Java language. This implementation has been integrated in the Java-to-C compiler turboJ [W+98] from Silicomp Research Institute (`http://www.ri.silicomp.com`). Our implementation uses inlining of small methods to increase stack allocation opportunities. When an allocation takes place in a loop, the same memory space is reused at the next iteration when possible. Synchronization elimination is also performed, thanks to a global analysis which detects thread-local objects. We have performed tests on large programs, and analyzed the causes of the speedups.

Overview

The first chapter (introduction) is included both in French and English.

The following of the thesis is split into two parts. The first part deals with the ML language. Chapter 3 describes the different escape analyses for ML and proves their correctness. Chapter 4 details the implementation of the escape analyses for ML based on integers. Chapter 5 deals with the implementation of the analysis based on annotated types. At last, Chapter 6 details our experimental results. The detailed proofs of most results as well as the description of some integer escape analyses have been postponed to appendices to make reading easier.

The second part deals with Java. It has about the same structure as the ML part. It begins with Chapter 7 which describes our analysis and proves its correctness. Chapter 8 describes its implementation and Chapter 9 our experimental results.

The body of the thesis ends with a chapter of comparison with related work about escape analysis and similar analyses, for functional languages as well as for Java.

Readings

The reader interested in the analysis of ML will read Chapters 3, 4 and 6. The ML part can be understood without reading Chapter 5.

Chapters 7, 8 and 9 deal with the analysis of Java. The chapters about ML are not necessary to understand this part, except the paragraphs "Comparison with ML". However, the reader who wants some information about abstract interpretation will read Section 3.1 and may consult the references it quotes.

The reading of appendices is only intended for specialists who wish to have a detailed proof of correctness of the analyses.

The index at the end of the thesis will enable the reader to find easily the main notations and definitions.

Part I

Escape Analysis for ML

Chapter 3

Design and correctness proof of escape analyses

3.1 Preliminaries

General mathematical notations are recalled in Figure 3.1. Other notations will be introduced as they are needed.

3.1.1 Complete partial orders (CPO)

The semantics of recursive programs and of loops is in general given by a fixed point computation. The framework of complete partial orders and continuous functions is well-suited for such fixed point computations, as Kleene's theorem shows.

Definition 3.1 (CPO) Let $S(\leq)$ be a partially ordered set (poset). A ***chain*** $C = (x_n)_{n\in\mathbb{N}}$ is a monotone sequence of elements of S: $x_0 \leq x_1 \leq ... \leq x_n \leq x_{n+1} \leq ....$

A ***complete partial order*** (CPO) is a poset $S(\leq)$ such that S has a least element $\perp$ and every chain C has a least upper bound $\sqcup C$.

Let $S(\leq)$ and $S'(\leq')$ be two CPOs. An operator $f : S \to S'$ is ***continuous*** if and only if for all chains $C \subseteq S$, $f(\sqcup C) = \sqcup' f(C)$.

Theorem 3.2 (Kleene) *Let $S(\leq)$ be a CPO. Let $f : S \to S$ be a continuous operator. Then f has a least fixed point:*

$$\mathrm{lfp}(f) = \sqcup\{f^i(\perp) | i \in \mathbb{N}\}.$$

Proposition 3.3 *Let S be a set. We define $S_\perp = S \cup \{\perp\}$ ordered by $x \leq y$ if and only if $x = \perp$ or $x = y$. Then $S_\perp$ is a CPO.*

3.1.2 Abstract interpretation

For the design of our analyses, we use the general framework of abstract interpretation, by Patrick and Radhia Cousot [CC77, Cou78, CC79, CC92]. In general, properties of programs cannot be determined exactly, since most of them are undecidable. Approximations have

$[\![a,b]\!] = \{a,...,b\}$	Interval in $\mathbb{N}$
Functions	
$\{x_1 \mapsto y_1,...,x_n \mapsto y_n\}$	Function which maps x_i to y_i for $i \in [\![1,n]\!]$, undefined otherwise.
$f[x_1 \mapsto y_1,...,x_n \mapsto y_n]$	Extension of f which maps x_i to y_i for $i \in [\![1,n]\!]$. If f was already defined at some x_i, the new value replaces the old one.
$f_{\vert E}$	Restriction of f to the set E
$\mathrm{Dom}(f)$	Definition domain of f
Lists	
S^*	Set of lists (or words, or tuples) whose elements are in S
$[]$	Empty list
$[p_1,\ldots,p_n]$	List of elements $p_1,\ldots,p_n$.
$p_1 : l$	List l at the head of which p_1 has been concatenated.
$l(i)$	ith element of the list l.
$\mathrm{Dom}([p_1,...,p_n]) = [\![1,n]\!]$	Set of i such that $l(i)$ is defined
Expressions	
$FV(M)$	Free variables of M.
$M[N/x]$	Expression M after the substitution of variable x by N.

Figure 3.1: Notations

therefore to be performed. The idea is then to start from an exact (or concrete) semantics of the studied language, and to define an approximate (or abstract) semantics which only retains the properties in which we are interested. The abstract semantics can be systematically computed from the concrete semantics. The approximations are performed in such a way that we may answer "don't know", but if a definite answer is given, it will always be correct.

Concrete properties are represented by the elements of a set L_1, ordered by $\leq$ where $c \leq c'$ means that property c is more precise than c'. If c is correct, then c' greater than c is also correct, but contains less information. In the following, the sets used to represent properties will be complete lattices:

Definition 3.4 (Complete lattice) A ***complete lattice*** is a partially ordered set $L(\leq)$ such that every subset X of L has a least upper bound $\sqcup X$ and a greatest lower bound $\sqcap X$. In particular, L has a least element $\perp = \sqcup\emptyset$ and a greatest element $\top = \sqcup L$. This lattice will be denoted by $L(\leq,\perp,\top,\sqcup,\sqcap)$.

Proposition 3.5 *1. If S is a set, $\mathcal{P}(S)(\subseteq,\emptyset,S,\cup,\cap)$ is a complete lattice.*

2. Let $L_1(\leq_1,\perp_1,\top_1,\sqcup_1,\sqcap_1)$ and $L_2(\leq_2,\perp_2,\top_2,\sqcup_2,\sqcap_2)$ be complete lattices. The set of monotone functions $(L_1 \stackrel{\mathrm{m}}{\to} L_2)(\leq_{\mathrm{f}},\perp_{\mathrm{f}},\top_{\mathrm{f}},\sqcup_{\mathrm{f}},\sqcap_{\mathrm{f}})$ ordered pointwise is a complete lattice. $\phi_1 \leq_{\mathrm{f}} \phi_2 \Leftrightarrow \forall x \in X, \phi_1(x) \leq \phi_2(x)$, $\perp_{\mathrm{f}} = \lambda x.\perp$, $\top_{\mathrm{f}} = \lambda x.\top$, $\sqcup_{\mathrm{f}} F = \lambda x. \sqcup \{f(x) \mid f \in F\}$, $\sqcap_{\mathrm{f}} F = \lambda x. \sqcap \{f(x) \mid f \in F\}$.

3. $([\![0,n]\!] \cup \{\infty\})(\leq,0,\infty,\sqcup,\sqcap)$ with the natural ordering is a complete lattice.

4. $([\![0,n]\!])(\leq,0,n,\sqcup,\sqcap)$ with the natural ordering is a complete lattice.

Theorem 3.6 (Tarski) *The set of fixed points of a monotone operator f on a complete lattice $L(\leq,\perp,\top,\sqcup,\sqcap)$ is a non-empty complete lattice, ordered by $\leq$. The least fixed point of*

f is

$$\mathrm{lfp}(f) = \sqcap\{x \in L | f(x) \leq x\}$$

and its greatest fixed point is

$$\mathrm{gfp}(f) = \sqcup\{x \in L | x \leq f(x)\}.$$

The least fixed point could in fact be obtained by a transfinite iteration of f from $\perp$. If $f^\lambda(\perp)$, where λ is an ordinal, is defined by

$$\begin{aligned} &f^0(\perp) = \perp \\ &f^{\lambda+1}(\perp) = f(f^\lambda(\perp)) \\ &f^\lambda(\perp) = \sqcup\{f^\beta(\perp) | \beta < \lambda\} \text{ if } \lambda \text{ is a limit ordinal,} \end{aligned}$$

this transfinite sequence eventually stabilizes, and its limit is $\mathrm{lfp}(f)$ (this is a particular case of [Cou78, Th. 2.5.2.0.2]).

A complete lattice is also a CPO, then Kleene's theorem can be applied: it shows that the above sequence stops at ordinal ω if f is continuous.

Like concrete properties, abstract properties are also represented by a complete lattice L_2. The correspondence between the concrete and abstract properties is formalized by semi-dual Galois connections.

Definition 3.7 (Galois connection [CC79, Definition 5.3.0.1][CC92, Example 10]) Let $L_1(\leq_1)$ and $L_2(\leq_2)$ be posets. (α, γ) is a semi-dual ***Galois connection*** (or pair of adjoined functions) between L_1 and L_2 if and only if $\alpha \in L_1 \to L_2$, $\gamma \in L_2 \to L_1$ and

$$\forall x \in L_1, \forall y \in L_2, \alpha(x) \leq_2 y \Leftrightarrow x \leq_1 \gamma(y).$$

This is denoted by:

$$(L_1, \leq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L_2, \leq_2).$$

In this case, α and γ are monotone, $(\alpha \circ \gamma)(y) \leq_2 y$, $(\gamma \circ \alpha)(x) \geq_1 x$ [CC79, Theorem 5.3.0.4].

The two properties $\alpha(x) \leq_2 y$ and $x \leq_1 \gamma(y)$ both mean that y is a correct approximation of the concrete property x. $\alpha(x)$ is the most precise approximation of $x \in L_1$ in L_2. $\gamma(y)$ is the least precise element of L_1 which can be correctly approximated by $y \in L_2$. α is said to be the ***abstraction*** and γ the ***concretization***.

Proposition 3.8 *Let $L_1(\leq_1, \perp_1, \top_1, \sqcup_1, \sqcap_1)$ and $L_2(\leq_2, \perp_2, \top_2, \sqcup_2, \sqcap_2)$ be complete lattices. Let (α, γ) be a semi-dual Galois connection between L_1 and L_2.*

1. *[CC79, Corollary 5.3.0.5, (3)]: Each function in the pair (α, γ) uniquely determines the other:*

$$\begin{aligned} \alpha &= \lambda x. \sqcap_2 \{y \in L_2 | x \leq_1 \gamma(y)\}, \\ \gamma &= \lambda y. \sqcup_1 \{x \in L_1 | \alpha(x) \leq_2 y\}. \end{aligned}$$

2. *[CC79, Corollary 5.3.0.5, (4)] α is a complete join-morphism (i.e. is additive), $\alpha(\perp_1) = \perp_2$, γ is a complete meet-morphism, $\gamma(\top_2) = \top_1$.*

3. *[CC79, Theorem 5.3.0.6, (1)]* α *is onto if and only if* γ *is one-to-one if and only if* $\alpha \circ \gamma = \lambda y.y$.

The last property $\alpha \circ \gamma = \lambda y.y$ means intuitively that the concrete properties of L_1 are more precise than the abstract properties of L_2 (every abstract property has a corresponding concrete property which has exactly the same meaning, whereas in general one can only find an approximate abstract property $\alpha(x)$ corresponding to a concrete property x).

The design of an analysis is then defining α or γ and computing for each construct of the analyzed language the image of its semantics by the abstraction α.

There exist systematic techniques to build semi-dual Galois connections, for example:

Proposition 3.9 *[CC92, (17)] Let* L_1, L_1', L_2, L_2' *be complete lattices. If* (α, γ) *is a semi-dual Galois connection,*

$$(L_1, \leq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L_2, \leq_2)$$

and (α', γ') *is a semi-dual Galois connection,*

$$(L_1', \leq_1') \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (L_2', \leq_2')$$

then we have a semi-dual Galois connection:

$$(L_1 \overset{\mathrm{m}}{\to} L_1', \leq_1'') \underset{\alpha''}{\overset{\gamma''}{\leftrightarrows}} (L_2 \overset{\mathrm{m}}{\to} L_2', \leq_2'')$$

where the function lattices are ordered pointwise, $\alpha''(f) = \alpha' \circ f \circ \gamma$ *and* $\gamma''(g) = \gamma' \circ g \circ \alpha$.

If $\alpha \circ \gamma = \mathrm{Id}$ *and* $\alpha' \circ \gamma' = \mathrm{Id}$ *then* $\alpha'' \circ \gamma'' = \mathrm{Id}$.

3.2 Analyzed language

Our implementation analyzes Objective Caml programs. This language is however too complex to be able to perform formal proofs of a reasonable length. That is why, for the theoretical description, we restrict ourselves to a small but representative subset of Objective Caml, which contains higher-order functions, recursive types, assignments, polymorphism. The goal of this section is to explain the syntax, informal semantics and typing of this subset. There is nothing new in this section; for definitions of similar languages see for example [Ler92, MTHM97]. The reader who is not familiar with the ML programming language can find more information in [LW93, CM95, CM98, WL99, Pau96].

3.2.1 Syntax

Our small language handles the following ***types***: integers (type `int`), booleans (type `bool`), type `unit` (a type which contains one single value NIL; this value is returned by expressions which return no result), functions, and constructed types. Functions that take an argument of type τ_1 and return a result of type τ_2 are of type $\tau_1 \to \tau_2$. Constructed types are given a name $\tau_{\mathrm{N}} \in \mathit{TyConstr}$. Moreover, the ML language is polymorphic. This means that the same expression may belong to several different types. This is represented by type variables which may represent any type. The syntax of type expressions is therefore:

Type	β `list`		β `ref`	$\tau_1 * \tau_2$
Constructor C	`::`	`[]`	`ref`	$(\_, \_)$
Destructor $\overline{C_i}$	$\overline{C_1}$ = `hd`, $\overline{C_2}$ = `tl`		`!`	$\overline{C_1}$ = `fst`, $\overline{C_2}$ = `snd`
Test $C?$		`null`		
Assignment $\overline{C_i}(v) \leftarrow w$			v`:=`w	

Figure 3.2: Constructed types

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} & \text{Base types} \\
& \mid & \beta & \text{Type variable} \\
& \mid & \tau_1 \to \tau_2 & \text{Function type} \\
& \mid & (\tau_1, ..., \tau_p)\tau_{\mathrm{N}} & \text{Constructed type}
\end{array}
$$

Constructed types are declared by ***type declarations*** of the form:

$$
\begin{array}{rcl}
\texttt{type } (\beta_1, ..., \beta_p)\tau_{\mathrm{N}} & = & C \texttt{ of } \tau_1 * \tau_2 * ... * \tau_k \\
& \mid & C' \texttt{ of } \tau'_1 * ... * \tau'_{k'} \\
& \mid & ...
\end{array}
$$

where τ_i, τ'_i, ... are types, and $\beta_1, ..., \beta_p$ are the only type variables that may occur in τ_i, τ'_i, ... The identifier τ_{N} is said to be a ***type constructor***. The ***constructors*** of type constructor τ_{N} are C, C', ... Values of type $(\beta_1, ..., \beta_k)\tau_{\mathrm{N}}$ are built by expressions of the form $C(M_1, ..., M_k)$ where M_i is an expression of type τ_i. We do not consider local type declarations [MTHM97] in our language; all type declarations are at top-level.

Example 3.10 If we have the following type declaration:

$$
\begin{array}{rcl}
\texttt{type } \beta\,\texttt{t} & = & \texttt{Val of int} \\
& \mid & \texttt{P of } \beta \\
& \mid & \texttt{Add of } \beta\,\texttt{t} * \beta\,\texttt{t}
\end{array}
$$

then `P(3)` is a value of type `int t` (parentheses may be omitted when there is only one parameter), `Add(P(true), Val(2))` is a value of type `bool t`.

For each constructor C with k parameters, we define k ***destructors*** $\overline{C_i}$, by: if $x = C(x_1, ..., x_k)$, $\overline{C_i}(x) = x_i$ $(i \in [\![1, k]\!])$. A ***test function*** $C?$ is also associated with each constructor, such that $C?(x)$ is true if and only if x is a value constructed by constructor C. For types which have a single constructor, it is useless to define a test function. An ***assignment instruction*** $\overline{C_i}(v) \leftarrow w$ is associated with each destructor, which enables the user to change a constructed value after its creation. In ML, few values are mutable. In Objective Caml, these are records declared as such, and references are defined from them. In SML, these are only references. In ML, other types have no assignment instruction and are immutable. Tuples can be regarded as constructed types with a single constructor $(\_, ..., \_)$, and n destructors `fun` $(x_1, ..., x_n) \to x_i$. Figure 3.2 gives some examples of constructed types. We define:

$\tau \in Type$	Set of types
$\tau_{\mathrm{N}} \in TyConstr$	Set of type constructors
$C \in Constr(\tau_{\mathrm{N}})$	Set of constructors of type constructor τ_{N}
$C \in Constr$	Set of all constructors
$\overline{C_i} \in Destr(\tau_{\mathrm{N}})$	Set of destructors of type constructor τ_{N}
$\overline{C_i} \in Destr$	Set of all destructors

Our language contains the ***expressions*** generated by the following grammar, where $x, f \in Var$ are ***variables***:

$$
\begin{array}{rcll}
M, N, P & ::= & x & \text{variable} \\
& \mid & M\ N & \text{application} \\
& \mid & \texttt{fun}\ x \rightarrow M & \text{function definition} \\
& \mid & \texttt{let}\ x = M\ \texttt{in}\ N & \text{let binding} \\
& \mid & \texttt{let rec}\ f = \texttt{fun}\ x \rightarrow M\ \texttt{in}\ N & \text{recursive function definition} \\
& \mid & \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P & \text{condition} \\
& \mid & M;\ N & \text{sequence} \\
& \mid & \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \ldots & \text{constants} \\
& \mid & C(M_1, ..., M_n) & \text{construction of a value} \\
& \mid & C? & \text{test} \\
& \mid & \overline{C_i} & \text{destructor} \\
& \mid & \overline{C_i}(M) \leftarrow N & \text{assignment}
\end{array}
$$

In program examples, we use a syntax closer to Objective Caml: we write `let` $f\ x$ `=` M `in` N instead of `let` f $=$ `fun` $x \rightarrow M$ `in` N, and similarly for `let rec`. We also write `fun` $x\ y \rightarrow M$ instead of `fun` $x \rightarrow$ `fun` $y \rightarrow M$.

The application $M\ N$ first evaluates M, then N, and applies the result of M (which must be a function) to the result of N. Our language uses call-by-value evaluation: the parameter N is evaluated before passing it to the function M. The expression `fun` $x \rightarrow M$ creates a function such that applying this function to a value v means evaluating M with x bound to v. Functions are represented in the implementation by ***closures*** which contain the values of the free variables of the function and a pointer to the code of the function. The expression `let` $x = M$ `in` N first evaluates M, and evaluates N with x bound to the result of M. The recursive definition `let rec` $f =$ `fun` $x \rightarrow M$ `in` N creates a function f such that applying f to a value v means evaluating M with x bound to v. The expression M itself can refer to f. The expression N is evaluated with f bound to this function. The `let rec` expression can easily be generalized to mutually recursive functions. We do not consider here the recursive definition of values, since special restrictions need to be put on such definitions, and this complex case has no particular importance for escape analysis. The conditional `if` M `then` N `else` P first evaluates the boolean expression M. If the result of M is `true`, N is then evaluated. If the result of M is `false`, P is evaluated. The sequence $M; N$ first evaluates M, discards its results and then evaluates N. The result of the sequence is the result of N. The expression $C(M_1, ..., M_n)$ evaluates $M_1, ..., M_n$ and builds a constructed value $C(r_1, ..., r_n)$ where r_i is the result of M_i.

A very useful construct which exists in ML is pattern matching. A simplified version of its syntax is:

$$\begin{array}{l}\texttt{match}\ M\ \texttt{with}\\ \quad C(x_1,...,x_k) \to N\\ \quad \mid C'(x'_1,...,x'_{k'}) \to N'\\ \quad \mid ...\end{array}$$

where $M, N, N', ...$ are expressions and the type τ of the expression M is of the form:

$$\begin{array}{rcl}\texttt{type}\ (\beta_1,...,\beta_p)\tau_{\mathrm{N}} & = & C\ \texttt{of}\ \tau_1 * \tau_2 * ... * \tau_k\\ & \mid & C'\ \texttt{of}\ \tau'_1 * ... * \tau'_{k'}\\ & \mid & ...\end{array}$$

When such a pattern matching is evaluated, M is first evaluated. If the constructor of its result is C, the first branch N is executed, with the variables $x_1, ..., x_k$ bound to the components of the result of M: $x_i = \overline{C_i}(M)$. Otherwise, if the constructor of its result is C', the second branch is executed, and so on. In ML, a more general pattern matching can be used, in which the patterns $C(x_1, ..., x_k)$ can be replaced with variables or constructor patterns $C(p_1, ..., p_k)$ where C is a contructor and $p_1, ..., p_k$ are patterns. In our language, pattern matching can be implemented thanks to destructors $\overline{C_i}$ and test functions $C?$.

A program is a pair $(tdecls, P)$ where $tdecls$ is a list of type declarations and P is an expression. The expression P is the body of the program.

The reader who is interested in a formal semantics of this language can find one in Appendix A.1.1 (but it is better understood by first reading the design of the analysis in the next section). There are two kinds of data structures in this language: constructed values are built by $C(M_1, ..., M_n)$, and functions are represented by closures. All these data structures are normally allocated on the heap and deallocated by a garbage collector. Our goal will be to allocate them on the stack.

We consider that all functions only take one parameter. A function taking several parameters will be coded as follows:

```
f = fun x -> fun y -> M
```

that is when applying `f` to two parameters `N` and `P`, we first apply `f` to the first parameter `N`. This creates a new closure which is then applied to `P`. This is of course not what happens in Objective Caml (it would be very inefficient). In the real implementation, a function has an arity, which corresponds to its expected number of arguments (the number of imbricated `fun` expressions in the definition of the function). If a function f of arity n is applied to exactly n arguments, then no closure is created. The application of f to m arguments $M_1, ..., M_m$ with $m < n$ is considered as equivalent to $(\texttt{fun}\ x_1...x_m \to (\texttt{fun}\ x_{m+1}...x_n \to f x_1...x_n))M_1...M_m$. For simplicity, we will not consider this mechanism in our theoretical study, but of course, we take it into account in our implementation. Closures created by partial application are indeed candidates for stack allocation. On the other hand, functions of arity n must not be stack allocated if they are only applied to less than n arguments (f escapes from $f M_1...M_m = (\texttt{fun}\ x_1...x_m \to (\texttt{fun}\ x_{m+1}...x_n \to f x_1...x_n))M_1...M_m$ if $m < n$).

3.2.2 Typing

We have already seen in the syntax of the language the definitions of types or type expressions. We will now give typing rules which associate a type with each expression.

Preliminary definitions

Type variables can represent any type. The type represented by a type variable is given by a ***substitution*** which is a finite mapping from type variables to type expressions:

$$\sigma = \{\beta_1 \mapsto \tau_1, ..., \beta_n \mapsto \tau_n\}.$$

A substitution σ can naturally be extended to type expressions by

$$\begin{aligned}
&\sigma\tau = \tau \text{ if } \tau \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\}\\
&\sigma\beta_i = \tau_i\\
&\sigma\beta = \beta \text{ if } \beta \notin \mathrm{Dom}(\sigma)\\
&\sigma(\tau_1 \to \tau_2) = \sigma\tau_1 \to \sigma\tau_2\\
&\sigma((\tau_1, ..., \tau_p)\tau_{\mathrm{N}}) = (\sigma\tau_1, ..., \sigma\tau_p)\tau_{\mathrm{N}}
\end{aligned}$$

We define ***free variables*** of τ, $FV(\tau)$, to be the set of type variables which appear in τ.

A type scheme is defined as follows:

$$\tau_{\mathrm{S}} = \forall\beta_1...\beta_k.\tau$$

Type schemes which differ only by renaming of type variables $\beta_1...\beta_k$ are regarded as equivalent. A type scheme $\forall\beta_1...\beta_k.\tau$ represents intuitively any type obtained from τ by substituting variables β_1, ..., β_k by any type. We can define a substitution of type schemes, but we must take care of name captures by first renaming bound variables of the type scheme:

$$\begin{aligned}
\sigma(\forall\beta_1...\beta_k.\tau) = \forall\beta_1...\beta_k.\sigma\tau \text{ if } &\mathrm{Dom}(\sigma) \cap \{\beta_1, ..., \beta_k\} = \emptyset \text{ and}\\
&\{\beta_1, ..., \beta_k\} \cap \cup_{\beta\in\mathrm{Dom}(\sigma)} FV(\sigma\beta) = \emptyset.
\end{aligned}$$

The ***free variables*** of a type scheme are: $FV(\forall\beta_1...\beta_k.\tau) = FV(\tau) - \{\beta_1, ..., \beta_k\}$.

A ***typing environment*** Γ gives for each variable its type scheme:

$$\Gamma = \{x_1 \mapsto \tau_{\mathrm{S}1}, ...\}.$$

We extend the definition of free variables to environments: $FV(\Gamma) = \cup_{x\in\mathrm{Dom}(\Gamma)} FV(\Gamma(x))$. Substitutions are defined on environments by: $\sigma\Gamma = \{x \mapsto \sigma(\Gamma(x)) \text{ for all } x \in \mathrm{Dom}(\Gamma)\}$.

The generic ***instantiation*** of a type scheme is defined by: τ_2 is a generic instance of the type scheme $\forall\beta_1...\beta_k.\tau$ if and only if:

$$\tau_2 \leq \forall\beta_1...\beta_k.\tau \Leftrightarrow \exists\sigma \text{ substitution such that } \mathrm{Dom}(\sigma) \subseteq \{\beta_1, ..., \beta_k\} \text{ and } \sigma\tau = \tau_2.$$

The ***generalization*** is defined by:

$$\mathrm{Gen}(\tau, \Gamma) = \forall\beta_1...\beta_k.\tau \text{ where } \{\beta_1, ..., \beta_k\} = FV(\tau) - FV(\Gamma).$$

A typing judgment is: $\Gamma \vdash M : \tau$ which means "expression M has type τ in environment Γ".

Typing rules

The typing rules are given under the form of inference rules:

$$\frac{J_1 \qquad \dots \qquad J_n}{J}$$

which means that J is true if the hypotheses J_1, ..., J_n are satisfied. When there is no hypothesis, we simply have an axiom J which means that J is true. Variables that appear in a rule are implicitly universally quantified at the head of the rule.

The inference rules are as follows:

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}$$

The typing rule for variables means that x is of type τ for all types τ which are instances of the type scheme $\Gamma(x)$. The variable x can then be used with several different types in the same expression.

$$\frac{\Gamma[x \mapsto \tau_1] \vdash M : \tau_2}{\Gamma \vdash (\texttt{fun}\ x \to M) : \tau_1 \to \tau_2}$$

A function has type $\tau_1 \to \tau_2$ if its body has type τ_2 when its parameter has type τ_1.

$$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\,N : \tau_2}$$

A function application is well typed if the argument has the same type as the formal parameter of the function, and the type of the application is then the type of the result of the function.

$$\frac{\Gamma \vdash M : \tau_1 \qquad \Gamma[x \mapsto \mathrm{Gen}(\tau_1, \Gamma)] \vdash N : \tau_2}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau_2} \text{ where } M \text{ is } \texttt{fun}\ y \to P\text{, a constant, or a variable}$$

$$\frac{\Gamma \vdash M : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash N : \tau_2}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau_2} \text{ where } M \text{ is another expression}$$

Only expressions M which are syntactically known not to create mutable data structures (such expressions are said to be ***non-expansive***) have their type generalized. This restriction, introduced by Wright [Wri95], avoids problems of unsoundness of polymorphic typing in the presence of mutable data structures. It works well because nearly all interesting values of polymorphic types are functions. In ML, this restriction is slightly relaxed by allowing generalization when M is $C(M_1, ..., M_k)$ where C is a constructor which does not create mutable values and $M_1, ..., M_k$ satisfy the same restriction as M.

When generalization is allowed, the `let` expression introduces new type schemes. The variable x can be used with different types in expression N.

$$\frac{\Gamma[f \mapsto (\tau_1 \to \tau_2), y \mapsto \tau_1] \vdash M : \tau_2 \qquad \Gamma[f \mapsto \mathrm{Gen}(\tau_1 \to \tau_2, \Gamma)] \vdash N : \tau_3}{\Gamma \vdash \texttt{let rec}\ f = \texttt{fun}\ y \to M\ \texttt{in}\ N : \tau_3}$$

The type of f is not generalized during the typing of M, therefore there is no polymorphic recursion in our language: f can only be used with one single type in M.

$$\frac{\Gamma \vdash M : \texttt{bool} \qquad \Gamma \vdash N : \tau \qquad \Gamma \vdash P : \tau}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P : \tau}$$

$$\frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash N : \tau_2}{\Gamma \vdash M; N : \tau_2}$$

$$\Gamma \vdash n : \texttt{int} \text{ if } n \in \mathbb{Z} \qquad \Gamma \vdash \texttt{true} : \texttt{bool} \qquad \Gamma \vdash \texttt{false} : \texttt{bool}$$

If there exists a type declaration

$$\begin{array}{lcl} \texttt{type } (\beta_1, ..., \beta_p)\tau_{\mathrm{N}} & = & C \texttt{ of } \tau_1 * \tau_2 * ... * \tau_k \\ & | & ... \end{array}$$

then for each declared constructor C, we have the rule:

$$\frac{\forall i \in [\![1, k]\!], \Gamma \vdash M_i : \sigma\tau_i}{\Gamma \vdash C(M_1, ..., M_k) : (\sigma\beta_1, ..., \sigma\beta_p)\tau_{\mathrm{N}}}$$

For each constructor C, for each i,

$$\frac{\Gamma \vdash M : (\sigma\beta_1, ..., \sigma\beta_p)\tau_{\mathrm{N}} \qquad \Gamma \vdash N : \sigma\tau_i}{\Gamma \vdash \overline{C_i}(M) \leftarrow N : \texttt{unit}}$$

and the following values are in the initial typing environment Γ_0:

$$\Gamma_0(C?) = \forall \beta_1 ... \beta_p.(\beta_1, ..., \beta_p)\tau_{\mathrm{N}} \to \texttt{bool} \qquad \Gamma_0(\overline{C_i}) = \forall \beta_1 ... \beta_p.(\beta_1, ..., \beta_p)\tau_{\mathrm{N}} \to \tau_i$$

(the expressions $C?$ and $\overline{C_i}$ are then considered as particular cases of the variable expression x).

If we want to use a monomorphic typing, we can keep the same inference rules, but just change the definition of the generalization: $\mathrm{Gen}(\tau, \Gamma) = \tau$. Then no type scheme with universally quantified type variables is introduced, and the typing rule for variables is equivalent to: $\Gamma \vdash x : \Gamma(x)$. A variable x can then only be used with one type, the type recorded in the environment.

The most important property of typing is that, informally, when an expression M of type τ is evaluated, and M terminates, its result is a value of type τ. This is called safety of typing. This property protects the programmer against a large class of errors. If an expression M is of type $\tau_1 \to \tau_2$, its result will be a closure at runtime, and the application $M\,N$ will not cause an error. Applying a destructor may cause an error (for types which have several constructors, even if the value is of the right type, the destructor may not correspond to the constructor of the value). However, if destructors $\overline{C_i}$ are always applied after first testing that the constructor of their argument is C, no error can happen. This is the case in particular in pattern matching[1]. In the following, we assume that this is always the case. Then, the safety of typing enables us to leave the semantics undefined in case of error, since such errors do not happen.

A typing judgment $\Gamma \vdash M : \tau$ is said to be more general than the typing judgment $\Gamma' \vdash M : \tau'$ if there exists a substitution σ such that $\Gamma' = \sigma\Gamma$ and $\tau' = \sigma\tau$. We can prove the existence of a most general typing judgment, and it is the judgment computed by type inference.

When typing a program $(tdecls, P)$, the body P of the program is typechecked in the environment Γ_0 containing the types of destructors and test functions generated by the type

[1]We assume that all possible constructors are mentioned in the pattern matching, otherwise another error may happen if the constructor of the matched value is not mentioned.

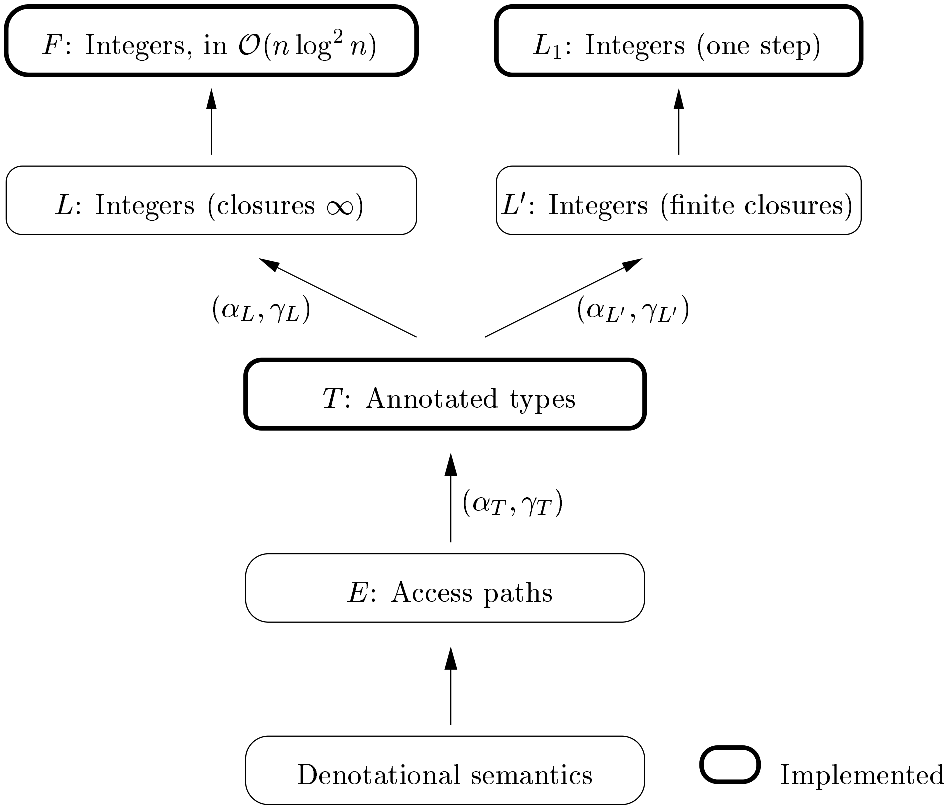


Figure 3.3: Hierarchy of semantics and analyses.

declarations *tdecls*: $\Gamma_0 \vdash P : \tau$. During the computation of this type, we have one type judgment for each subexpression M of P : $\Gamma \vdash M : \tau'$. Then τ' is said to be the type of expression M, and is denoted τ_M. (More precisely, τ' is not only related to the expression M, but to the occurrence of M in the program. Two different occurrences may have different types.) If y is a variable, it is added in the typing environment Γ during the analysis of its binder, and $\Gamma(y) = \forall \beta_1 ... \beta_p . \tau''$. The type τ'' is said to be the general type of y, τ_y^{g}. The types of the occurrences of y in the program are of the form $\sigma \tau_y^{\mathrm{g}}$ where $\mathrm{Dom}(\sigma) \subseteq \{\beta_1, ..., \beta_p\}$. When y is bound by `fun`, σ must be the identity, and τ_y^{g} is also the type of occurrences of y in the program. In this case, we can omit the exponent g and denote by τ_y the type of y. The type of the ith parameter of M is denoted by $\tau_{M,i}$, i.e. $\tau_M = \tau_{M,1} \rightarrow \tau_{M,2} \rightarrow ... \tau_{M,k} \rightarrow \tau_r$ where τ_r is not a function type. The type $\tau_{y,i}^{\mathrm{g}}$ is defined in the same way from τ_y^{g}.

3.2.3 Hierarchy of analyses

Our strategy to prove the correctness of the analyses is to start from the exact denotational semantics of our language, and to approximate it by several analyses from the more precise to the less precise, as shown in Figure 3.3. Analyses represented inside of a bold frame have been implemented. The name of the analysis is mentioned in the frame, with the kind of abstract values it uses. Arrows labelled by a semi-dual Galois connection (α, γ) mean that such a Galois connection has been used to define the approximate analysis. The rest of this chapter will be dedicated to the design of these analyses and their correctness proofs. The comparison between the different escape analyses will enable us to find which one gives the best cost/precision tradeoff.

3.3 Analysis E

3.3.1 Introduction

Analysis E can be applied to any functional language, even untyped. It is also very precise, so it can be used as a basis for several less precise analyses that can be defined by abstract interpretation. The cost of this is that it cannot be implemented directly because the fixed point iteration may not terminate. Therefore we will have to define more approximate analyses (in the next sections).

Goal of escape analysis

The goal of escape analysis is to determine which pieces of data can be stack allocated. When a value is stack allocated, it is deallocated at a fixed program point by popping it from the stack. Of course, values must not be accessed after they are deallocated. Therefore, we must determine whether a given value will be reachable after a given program point. If it is not reachable, it may be deallocated at that point. We consider in particular expressions of the form:

$$\texttt{let}\ x = C(M_1, ..., M_n)\ \texttt{in}\ M$$

and similar expressions in which the constructor C is replaced with the creation of a closure by $\texttt{fun}\ y \to N$. Our goal is to determine whether the constructor C can be stack allocated and deallocated at the end of M. Does the constructor C have a lifetime longer than M ? If we are sure that the lifetime of the constructor in shorter than M, then we can perform the stack allocation. The expression is then transformed into

$$\texttt{letstack}\ x = C(M_1, ..., M_n)\ \texttt{in}\ M$$

Otherwise, we say that this constructor may escape from M, and the stack allocation is unsafe. Using `let` expressions is useful to fix the deallocation point: stack allocated values will be deallocated at the end of the body of the `let`. This will be generalized to other expressions in our implementations by first transforming them to introduce `let`s.

Consider for example,

```
let x = [3] in print_int(hd x)
```

In this case, the list constructor is not reachable at the end of `print_int(hd x)`, and we can perform the stack allocation, and deallocate the list constructor at the end of `print_int(hd x)`. On the contrary, consider

```
let x = [a] in b :: x
```

In this case, the list constructor is reachable from the result of `b :: x`, and it cannot be deallocated at the end of `b :: x`.

Representing the escape information

Intuitively, escape analysis is done by marking data that may still be live after the program point at which we want to deallocate values. If we want to deallocate data at the end of M in the expression `let` $x = C(M_1, ..., M_n)$ `in` M considered above, we mark the result of M. We also mark conservatively data stored by assignments during the execution of M, because their lifetime is more difficult to determine and they may be live after the end of M. The marks are then propagated by following the data flow. Unmarked data do not escape. In the example,

```
let x = [3] in print_int(hd x)
```

nothing is marked, and the list does not escape. On the contrary, in the example

```
let x = [a] in b :: x
```

the result of `b :: x` is marked, and therefore also `x`. The list constructor cannot be stack allocated.

Moreover, simply marking values as "escaping" or "not escaping" would lead to a very imprecise analysis. Indeed, consider the following expression:

```
let x = ([a],b) in snd x
```

A part of `x` escapes from this expression (the second element of the pair) but the first element does not escape. The pair and the list constructor could therefore be stack allocated. If we could not distinguish parts of `x`, we could not determine that this stack allocation is correct.

Therefore, we have to be able to say that only a part of a value escapes, by marking this part. To represent the marking of parts of values, we will for the moment use sets of access paths leading to marked parts of data. The advantage of this representation is that it does not depend on types. Therefore, we can design an escape analysis without dealing with polymorphism and recursive types which are complex points. They will be introduced in the next section. This way, problems are split in several steps and each step is simpler.

We first have to define access paths. Values of constructed types are accessed by destructors $\overline{C_i} \in \mathit{Destr}$. This yields a first kind of ***accessors***: the destructors. However, our language also allocates closures in the heap. Closures are not accessed by destructors, but by function application. Therefore, we define another accessor app to represent function application. The accessors are then $\delta \in \mathit{Access} = \mathit{Destr} \cup \{\mathrm{app}\}$.

Access paths are defined as follows:

$$\mathit{Path} = \overline{C_i}.\mathit{Path} | \mathrm{app}.\mathit{Path} | \epsilon \quad \text{that is} \quad \mathit{Path} = \mathit{Access}^*$$

where $\overline{C_i} \in \mathit{Destr}$. The empty path ϵ means that the top location of the value is marked; $\overline{C_i}$ indicates that the marked part is in the ith element of a value constructed by constructor C. An example is represented in Figure 3.4. The accessor app is useful to mark functions. If f is a function, f is marked with the path app.p when f is applied to one (unspecified) argument, and the result of this application is marked with path p. The accessor app is useful for stack allocating closures.

We give types to accessors: δ has type $\tau_1 \to \tau_2$ if the access of a value of type τ_1 by accessor δ yields a value of type τ_2. The destructor $\overline{C_i}$ has type $(\beta_1, ..., \beta_p)\tau_{\mathrm{N}} \to \tau_i$ where $C : \tau_1 \times ... \times \tau_k \to (\beta_1, ..., \beta_p)\tau_{\mathrm{N}}$, since it takes a parameter of type $(\beta_1, ..., \beta_p)\tau_{\mathrm{N}}$ and yields a

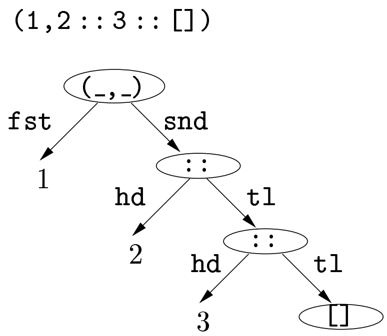


Figure 3.4: Paths. For example, 2 is represented by path `snd.hd`. The `::` location at the top of `2 :: 3 :: []` is represented by `snd`.

result of type τ_i. Similarly, we give to app the type $(\tau_1 \to \tau_2) \to \tau_2$ (for any τ_1, τ_2) since the application of a function of type $\tau_1 \to \tau_2$ to one argument yields a result of type τ_2.

Consider for example the expression

```
let x = ([a],b) in snd x
```

All paths in the result of `snd x` are marked, but only paths beginning with `snd` are marked in `x`. Therefore, the pair constructor and the list constructor do not escape. A similar example with the accessor app is:

```
let f = fun x -> y -> ...
in f [3]
```

The marked paths of `f` are the paths beginning with app, since `f` is always applied. For the expression `f [3] [4]`, the marked paths of `f` would begin with app.app (`f` is applied twice).

Remark. Access paths are usually used in the field of alias analysis, see [Deu94] for example. Two values are said to be aliased if they correspond to the same location. An alias between a part of x and another part of y may be represented by:

$$x.p \sim y.p'$$

with $p, p' \in Path$. It is natural to find access paths in escape analysis, since this analysis in fact discovers a particular kind of alias properties: may a value be aliased to the result of an expression or to data stored in assignments ? The fact that we look for particular aliases (those with the result of an expression) can however be exploited to design much more efficient analyses. This is our goal in this study.

A ***context*** is the representation of the escaping (marked) part of a value. As we have seen, the marking of data is represented by sets of paths. Therefore, contexts associated with data are here sets of access paths: $Ctx_E = \mathcal{P}(Path)$. Contexts are ordered by inclusion. By Proposition 3.5, Ctx_E is then a complete lattice, which has $\emptyset$ as least element, $Path$ as greatest element. The upper bound is $\cup$ and the lower bound is $\cap$.

We define a context ***construction*** by $\delta.c = \{\delta.p | p \in c\}$ where c is a context and δ an accessor. If the marked part of x_i is represented by context c, then the part represented by

$\overline{C_i}.c$ must be marked in $y = C(x_1, ..., x_k)$. Similarly, if the marked part of the result of the application $f\ v$ is context c, then the marked part of function f is app.c. Conversely, if we know the marked part c of $y = C(x_1, ..., x_k)$ and we wish to compute the marked part of x_i, we have to use a context ***restriction*** $\overline{C_i}^{-1}.c$ defined as follows:

$$\delta^{-1}.c = \{p | \delta.p \in c\}.$$

Such a restriction is also used with the app accessor: if a function $f = \texttt{fun}\ x \to M$ is marked with context c, the context of M is $\text{app}^{-1}.c$.

Example 3.11 Let us consider again the data structure of Figure 3.4. If `x = [2;3]` and the marked part of `x` is represented by $c = \texttt{tl}.Path$, then the marked part of `(1,x)` is represented by $\texttt{snd}.c = \texttt{snd}.\texttt{tl}.Path$ (the list `[3]` is marked).

Conversely, if the marked part of `x` is $c' = \{\texttt{hd}, \texttt{tl}.\texttt{hd}\}$, then the marked part of `[3] = tl x` is $\texttt{tl}^{-1}.c' = \{\texttt{hd}\}$ (3 is marked in `tl x = [3]`, 2 and 3 are marked in `x = [2;3]`).

Assume that `f` is a function: `f = fun x -> fun y -> [x;y]...`. If the marked part of the result of `f [3] [4]` is $c = \texttt{hd}.Path$, then the marked part of `f` is $c' = \text{app}.\text{app}.c = \text{app}.\text{app}.\texttt{hd}.Path$: `f` is applied twice, and the head of the result escapes. The marked part of the body of `f [x;y]` is then $\text{app}^{-1}.\text{app}^{-1}.c' = \texttt{hd}.Path$: the head of the body escapes.

Avoiding multiple analyses of code

If we want to know whether we can allocate a value `v` on the stack in an expression `let x = v in` M, the reference scope is expression M. Then the result of M escapes and is marked. It has context $Path$. But assume that we want to analyze an expression containing M, such as

```
let y = v' in
fst (let x = v in M)
```

to know whether `v'` can be stack allocated. In this case the result of M does not escape entirely. Only the first element of M escapes from the new reference scope `fst (let x = v in` M`)`: the context of M is $\texttt{fst}.Path$. That is, we have to analyze M in several contexts. However, for efficiency reasons, we do not want to analyze M several times. So, what can we do ? The idea is that the analysis of M is a function of the context representing the escaping part of its result. Therefore, the escaping part of a value inside M will not be represented by a single context, but by a ***context transformer***. A context transformer is a function from the escape context of the result of M yielding the escape context of the considered value.

Example 3.12 The escaping part of `v` in the expression `let x = hd v in (x,x)` is represented by the context transformer $\phi = \{c \mapsto \texttt{hd}.\texttt{fst}^{-1}.c \cup \texttt{hd}.\texttt{snd}^{-1}.c\}$. If the result escapes entirely, $c = Path$, and the escaping part of `v` is $\phi(c) = \texttt{hd}.Path$. Only the head of `v` escapes.

If we consider the expression (assuming that `hd v` is a pair)

```
fst (fst (let x = hd v in (x,x)))
```

the escape context of the result of `let x = hd v in (x,x)` will be $c = \texttt{fst}.\texttt{fst}.Path$, and the escape context of `v` is $\phi(c) = \texttt{hd}.\texttt{fst}.Path$.

The escape analysis of an expression M yields for each free variable of M its corresponding context transformer. The analysis of `let x = hd v in (x,x)` would therefore be $\{\mathtt{v} \mapsto \phi\}$ with the notations of the above example.

Moreover, we also have to represent the escape effect of functions. Indeed, in the expression

```
let x = [a] in f(x)
```

we have to know whether the parameter of `f` may be reachable from its result (or stored in a mutable data structure during the runtime of `f`). If the parameter of `f` is reachable from the result, `x` cannot be stack allocated. The effect of functions is also represented by context transformers, one for each parameter of the function. Parameters are numbered by a parameter index $i \in \mathit{Ind} = \mathbb{N} - \{0\}$ (the first parameter has index 1). For example, the analysis of the function `fun l -> hd l` is then $\{1 \mapsto \phi'\}$ where $\phi'(c) = \mathtt{hd}.c$. The parameter index counts the number of imbricated function definitions `fun`. The context transformers giving the escape information of a function can be computed once, and used every time the function is called. Therefore, using context transformers enables us to avoid reanalyzing functions several times.

Escape analysis is therefore a backward analysis [Hug88]: escape information propagates from the result to the free variables or the parameters of functions. At first sight, it could seem more natural to compute the escape information forward. One would consider an allocation or a parameter or a free variable, and determine which part of this element escapes by following the data flow forward. However, this would require reanalyzing the expression for each considered element. By using a backward analysis, we can determine in a single analysis which part of each element escapes, as noticed by Deutsch [Deu97]. We consider the result of the expression and follow the data flow backward to find out which element may flow to the result.

3.3.2 Formal definition and properties

The design described in the introduction can be summarized by the following definitions:

$\delta \in \mathit{Access} = \mathit{Destr} \cup \{\mathrm{app}\}$	Accessors
$p \in \mathit{Path} = \mathit{Access}^*$	Paths
$c \in \mathit{Ctx}_E = \mathcal{P}(\mathit{Path})$	Contexts
$\mathit{Tr}_E = \mathit{Ctx}_E \to \mathit{Ctx}_E$	Context transformers
$i \in \mathit{Ind} = \mathbb{N} - \{0\}$	Parameter indices
$\mathit{Exp}_E = (\mathit{Var} \cup \mathit{Ind}) \to \mathit{Tr}_E$	Abstract domain of an expression
$\mathit{Val}_E = \mathit{Ind} \to \mathit{Tr}_E$	Abstract domain of a value
$\rho \in \mathit{Env}_E = \mathit{Var} \to \mathit{Val}_E$	Abstract environment
$E[\![M]\!] : \mathit{Env}_E \to \mathit{Exp}_E$	Abstract semantic function

The abstract value associated with an expression yields for each free variable of the expression its corresponding context transformer. The abstract value associated with a value or an expression which represents a function yields a context transformer for each parameter of the function. The parameters are represented by their parameter index. The abstract environment gives for each variable the corresponding abstract value.

We write $f = \{x \mapsto f(x), ...\}$ for a function $f : (\mathit{Var} \cup \mathit{Ind}) \to \mathit{Tr}_E$ defined for each point. Abstract values f are total functions which have value $\lambda c.\emptyset$ when they are not explicitly defined. $f = \{\}$ is therefore the constant function $f(x) = \lambda c.\emptyset$ for all $x \in \mathit{Var} \cup \mathit{Ind}$. The

$E[\![M]\!] : Env_E \to Exp_E$

$E[\![y]\!]\rho = \{y \mapsto \lambda c.c, i \mapsto \rho[\![y]\!]i \text{ if } i \in Ind\}$

$E[\![M\ N]\!]\rho = \{i \mapsto \phi(i+1) \text{ if } i \in Ind, x \mapsto \phi(x) \sqcup \psi(x) \text{ if } x \in Var\}$
where $\phi xc = E[\![M]\!]\rho x(\text{app}.c)$ and $\psi x = (E[\![N]\!]\rho x) \circ (\phi(1))$

$E[\![\texttt{fun}\ y \to M]\!]\rho = \{1 \mapsto \phi(y), i \mapsto \phi(i-1) \text{ if } i > 1, x \mapsto \phi(x) \text{ if } x \in Var - \{y\}\}$
$\sqcup\{x \mapsto \lambda c.\text{if } \epsilon \in c \text{ and } x \in FV(\texttt{fun}\ y \to M) \text{ then } Path \text{ else } \emptyset\}$
where $\phi x = E[\![M]\!]\,\rho[y \mapsto \rho_\top]\,x \circ (\lambda c.\text{app}^{-1}.c)$ and $\rho_\top = \{i \mapsto \lambda c.Path \text{ if } i \in Ind\}$.

$E[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]\rho = \{x \mapsto E[\![N]\!]\rho' x \sqcup (E[\![M]\!]\rho x \circ E[\![N]\!]\rho' y) \text{ if } x \in Var - \{y\},$
$i \mapsto E[\![N]\!]\rho' i \text{ if } i \in Ind\}$ where $\rho' = \rho[y \mapsto (E[\![M]\!]\rho)_{|Ind}]$

$E[\![\texttt{let rec}\ f = \texttt{fun}\ y \to M\ \texttt{in}\ N]\!]\rho = \{x \mapsto R(x) \text{ if } x \in Var - \{f\}, i \mapsto E[\![N]\!]\rho' i \text{ if } i \in Ind\}$
where $\rho' = \text{lfp}(\lambda\rho_1.\rho[f \mapsto (E[\![\texttt{fun}\ y \to M]\!]\rho_1)_{|Ind}])$
and $R = \text{lfp}(\lambda R.\{x \mapsto E[\![N]\!]\rho' x \sqcup (E[\![\texttt{fun}\ y \to M]\!]\rho' x \circ R(f))\})$.

$E[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]\rho x = [(E[\![M]\!]\rho)_{|Var} x \circ (\lambda c.\emptyset)] \sqcup E[\![N]\!]\rho x \sqcup E[\![P]\!]\rho x$

$E[\![M; N]\!]\rho x = [(E[\![M]\!]\rho)_{|Var} x \circ (\lambda c.\emptyset)] \sqcup E[\![N]\!]\rho x$

$E[\![C(M_1, ..., M_n)]\!]\rho x = \sqcup_{i \in [\![1,n]\!]}[(E[\![M_i]\!]\rho)_{|Var} x \circ (\lambda c.\overline{C_i}^{-1}.c)]$

$E[\![\overline{C_i}(M) \leftarrow N]\!]\rho x = [(E[\![M]\!]\rho)_{|Var} x \circ (\lambda c.\emptyset)] \sqcup [(E[\![N]\!]\rho)_{|Var} x \circ (\lambda c.Path)]$

Initial environment: $\rho_0[\![\overline{C_i}]\!] = \{1 \mapsto \lambda c.\overline{C_i}.\text{app}^{-1}.c, k \mapsto \lambda c.Path \text{ if } k > 1\}$
$\rho_0[\![C?]\!] = \{\}$
$\rho_0[\![a]\!] = \{\}$ where $a \in \{\texttt{true}, \texttt{false}\} \cup \mathbb{Z}$

Figure 3.5: Analysis E

environment ρ is a partial function (defined for variables that have been bound). In the following definitions $x \in Var \cup Ind$, $y \in Var$.

Definition 3.13 Analysis E is defined in Figure 3.5. Let M be a given subexpression of the body P of the program. Analyzing M means computing $E[\![M]\!]\rho$. Notice that the value of ρ cannot be chosen arbitrarily: ρ gives the escape information for the variables bound in the environment of M. When computing $E[\![P]\!]\rho_0$ where ρ_0 is the initial environment, each subexpression of P is analyzed. In particular, $E[\![M]\!]\rho$ is evaluated for a certain value of ρ. This value of ρ must be used for all analyses of M. That is, the analysis of M is the value of $E[\![M]\!]\rho$ computed during the analysis of the whole body P of the program (the computation of $E[\![P]\!]\rho_0$).

More precisely, the value of ρ can be computed as follows. Let ρ_1 be the environment just before binding y. The environment ρ_2 with y bound is given by: $\rho_2 = \rho_1[y \mapsto \rho_\top]$ if y is bound by $\texttt{fun}\ y \to N$, $\rho_2 = \rho_1[y \mapsto (E[\![N]\!]\rho_1)_{|Ind}]$ if y is bound by $\texttt{let}\ y = N$, and $\rho_2 = \text{lfp}(\lambda\rho_2.\rho_1[y \mapsto (E[\![\texttt{fun}\ z \to N]\!]\rho_2)_{|Ind}])$ if y is bound by $\texttt{let rec}\ y = \texttt{fun}\ z \to N\ \texttt{in}\ ....$

Let us explain the intuition behind the formulas in Figure 3.5.

If the expression M is a variable y, the escaping part of y is the context c corresponding to the result of the expression. The escaping part of other variables is empty. If y is a function, the escaping parts of its parameters are given by $\rho[\![y]\!]$.

For an application $M\ N$, a variable x may escape because it appears in M or because it appears in N. The first case is taken into account by $E[\![M]\!]\rho x(\text{app}.c)$: if c is the context of the result of $M\ N$, then app.c is the context of the result of M, and $E[\![M]\!]\rho x(\text{app}.c)$ yields the corresponding escaping part of x. In the second case (x escapes because it appears in N), the escaping part of N is the escaping part of the first parameter of M ($\phi(1) = \lambda c.E[\![M]\!]\rho 1(\text{app}.c)$), and the escaping part of x is the composition of the escaping part of x in N ($E[\![N]\!]\rho x$) and the escaping part of N in $M\ N$ ($E[\![M]\!]\rho 1(\text{app}.c)$). The parameters of $M\ N$ are the parameters of M, except that their indices are shifted (the first parameter of $M\ N$ is the second parameter of M).

In the closure case `fun` $y \to M$, if the context contains the ϵ path, i.e. if the closure escapes, then all free variables of the closure also escape (they are stored in the environment inside the closure). This is handled by the second line of the formula. If y is a function (i.e. `fun` $y \to M$ is a higher-order function), its escape information is given by $\rho_\top = \{i \mapsto \lambda c.\mathit{Path} \text{ if } i \in \mathit{Ind}\}$. The value of $\rho_\top$ comes from the fact that if y is a function, we do not know exactly which function it will be at runtime, and we have to assume conservatively that all its parameters escape. Such an approximation is necessary to have a reasonable complexity (Alain Deutsch [Deu97] has proved that a more precise escape analysis for higher-order functions is DEXPTIME-complete, even when using much simpler lattices such as integers to represent the escaping part of values).[2] The environment ρ contains only the information $\rho[\![y]\!] = \rho_\top$ when y is bound by `fun`; it will only contain a precise escape information for variables bound by `let` or `let rec`. The escaping part of the first parameter of the function is the escaping part of y in M ($\phi(y)$). The escaping part of the other parameters (if M itself is a function) are the escaping parts of the corresponding parameters of M (indices are shifted by -1 because the second parameter of `fun` $y \to M$ corresponds to the first parameter of M).

In the `let` $y = M$ `in` N case, a variable x may escape because it appears in N ($E[\![N]\!]\rho' x$) or because it appears in M. In this case, its escaping part is the composition of the escaping part of x in M ($E[\![M]\!]\rho x$) and of y in N ($E[\![N]\!]\rho' y$). Indeed, even if N escapes entirely, only a part of y may escape, as in

```
let y = [x] in hd (fst y)
```

$E[\![N]\!]\rho'\mathtt{y}\mathit{Path} = \mathtt{hd}.\mathtt{fst}.\mathit{Path}$, and $E[\![M]\!]\rho\mathtt{x}(\mathtt{hd}.\mathtt{fst}.\mathit{Path}) = \mathtt{fst}.\mathit{Path}$. The first element of the pair `x` escapes.

If M is a function, it may be applied in N, and therefore we need the escape information for its parameters. This information is added in ρ' by $\rho' = \rho[y \mapsto E[\![M]\!]\rho_{|\mathit{Ind}}]$. For example, in

```
let f = fun x -> [x] in f(3)
```

$\rho'[\![\mathtt{f}]\!]1 = E[\![\mathtt{fun\ x\ \text{->}\ [x]}]\!]\rho 1 = E[\![\mathtt{[x]}]\!]\rho\mathtt{x} = \lambda c.\mathtt{hd}^{-1}.c$.

The `let rec` case is somewhat similar to the `let` case, but we have to perform fixed point iterations to find the escape information. We have two fixed points. The fixed point in the definition of ρ' computes the escape information for the parameters of the function f, whereas the fixed point in R deals with the free variables. The definition of R can be rephrased as

$$\begin{aligned} R(x) &= E[\![N]\!]\rho' x \sqcup (E[\![\mathtt{fun}\ y \to M]\!]\rho' x) \circ \mathrm{lfp}(\lambda R_2.E[\![N]\!]\rho' f \sqcup (E[\![\mathtt{fun}\ y \to M]\!]\rho' f) \circ R_2) \\ &= E[\![N]\!]\rho' x \sqcup (E[\![\mathtt{fun}\ y \to M]\!]\rho' x) \circ \sqcup_{n\in\mathbb{N}}(E[\![\mathtt{fun}\ y \to M]\!]\rho' f)^n \circ E[\![N]\!]\rho' f \end{aligned}$$

[2]Of course, we could also have used control flow analysis to determine a set of functions that may be called, and thus get a more precise escape information for calls to unknown functions. However, this would require to analyze the whole program at once, whereas our escape analysis can analyze modules separately, in dependency order.

which gives a better intuition: x may escape either because it appears in N (first term) or because f appears in N and x escapes from f (second term).

Remark. These two fixed points come from the fact that the escape information is split into two parts. The escape information for the parameters of functions in ρ propagates forward (the escape information for the parameters of functions is computed at their definition site, and must propagate from the definition towards the uses). On the contrary, the escape information for free variables propagates backward (when a variable is used, we update its escape information, and propagate it towards the definition of the variable). We have one fixed point for each part.

However, this is not the only possible presentation. In [Bla98], escape analysis is presented with only one fixed point in the analysis of `let rec`, by grouping together the parameters and the free variables in the environment ρ. Having two fixed points instead of one is in fact a technical detail, even if changing the presentation requires a few mathematical tricks.

In the `if` case, we do not know which branch of the `if` will be executed. Therefore, we take the upper bound of the escape behaviours of both branches, to be correct in all cases.

Notice that when functions are used inside data structures by expressions $C(M_1, ..., M_n)$, we forget their escape behaviour, and assume the worst when the function is extracted from the data structure by $\overline{C_i}$: the parameters escape, with context $Path$. In most cases, the function that will be extracted from the data structure cannot be known statically (with an analysis of a reasonable complexity). Forgetting the behaviour of functions stored in data structures is then a reasonable approximation.

In the definition of the analysis of the assignment $\overline{C_i}(M) \leftarrow N$, we can notice that the analysis of N is composed with $\lambda c.Path$. This means that the result of N always escapes, and corresponds to the approximation: the right-hand side of an assignment always escapes. Assignments could be handled in a much more precise way, by still using techniques of escape analysis. Such a technique will used in the second part of this thesis, to deal with the Java language. However, since ML is a functional language, assignments are not used too often. Therefore, we have decided to avoid the additional complexity required to take precisely into account assignments, and to use this simple analysis.

Example 3.14 Let us consider the following example using the `map` function:

```
let l =
   let a = 1 :: 2 :: 3 :: [] in
   let g = fun x -> x+1 in
   let rec map f l =
      if null l
      then []
      else (f (hd l)) :: (map f (tl l))
   in map g a
in ...
```

Let M be the expression `let rec map = ... in map g a`. Let ρ_0 be the environment containing only predefined variables.

$$\rho = \rho_0[\mathtt{a} \mapsto \{\}, \mathtt{g} \mapsto (E[\![\mathtt{fun\ x\ \text{->}\ x+1}]\!]\rho_{0|Ind})]$$

$$E[\![M]\!]\rho x = \text{lfp}(\lambda R.\{x \mapsto E[\![\texttt{map g a}]\!]\rho' x \sqcup (E[\![\texttt{fun f l -> if } \ldots]\!]\rho' x) \circ (R(f))\})(x)$$

with $\rho' = (\text{lfp}(\lambda\rho_1.\rho[\texttt{map} \mapsto E[\![\texttt{fun f l -> if } \ldots]\!]\rho_{1|Ind}]))$. A simple computation shows that:

$$E[\![\texttt{fun f l -> if } \ldots]\!]\rho_1 1c = (\text{app}.\texttt{hd}^{-1}.\text{app}^{-2}.c) \sqcup \rho_2[\![\texttt{map}]\!]1c'$$

where $\rho_2 = \rho_1[\texttt{f} \mapsto \rho_\top, \texttt{l} \mapsto \rho_\top]$ and $c' = \text{app}^2.\texttt{tl}^{-1}.\text{app}^{-2}.c$. Similarly,

$$E[\![\texttt{fun f l -> if } \ldots]\!]\rho_1 2c = \texttt{hd}.Path \sqcup \texttt{tl}.\rho_2[\![\texttt{map}]\!]2c'$$

A fixed point computation then shows:

$$\rho'[\![\texttt{map}]\!] = \{1 \mapsto \lambda c.\text{app}.\texttt{hd}^{-1}.(\texttt{tl}^{-1})^*.\text{app}^{-2}.c, 2 \mapsto \lambda c.\texttt{tl}^*.\texttt{hd}.Path\}$$

($\texttt{tl}^*$ is the set of words containing any number of $\texttt{tl}$). $\texttt{tl}^*.\texttt{hd}.Path$ means that we take any number of times the tail of the list, then we take an element of the list, therefore this context designates an element of the list parameter of `map`. The context $\text{app}.\texttt{hd}^{-1}.(\texttt{tl}^{-1})^*.\text{app}^{-2}.c$ begins with app therefore it contains at most the function `g` applied once, therefore the closure of `g` does not escape. Moreover, if the part of result kept after the call to `map` is represented by c', $c = \text{app}.\text{app}.c'$, so the escaping part of the function `g` is $\text{app}.\texttt{hd}^{-1}.(\texttt{tl}^{-1})^*.c'$. If c' contains paths $\texttt{tl}^*.Path$, then the escaping part of `g` is $\text{app}.Path$, so the result of `g` may escape. Otherwise, c' is of the form $c' = \texttt{tl}^*.\texttt{hd}.c''$, and the escaping part of `g` becomes $\text{app}.c''$: the part of the result of `g` that escapes is the part c'' of the elements of the result of `map` that escapes.

We also have: $E[\![\texttt{fun f l -> if } \ldots]\!]\rho' xc = \emptyset$ if $x \in Var - \{\texttt{map}\}$. We get:

$$E[\![M]\!]\rho\texttt{a}Path = \rho_1[\![\texttt{map}]\!]2(\text{app}.\text{app}.Path) = \texttt{tl}^*.\texttt{hd}.Path$$

and

$$E[\![M]\!]\rho\texttt{g}Path = \rho_1[\![\texttt{map}]\!]1(\text{app}.\text{app}.Path) = \text{app}.Path$$

therefore the spine of `a` and the `g` closure can be stack allocated (here, the function `map` can also be stack allocated but in a real program, `map` is a library function and therefore a global declaration which can never be stack allocated).

Let us notice interesting properties of our analysis.

Definition 3.15 A context $c \in Ctx_E$ is ***closed*** if and only if

$$\forall p \in c, \forall p' \in Path, p.p' \in c.$$

Intuitively, contexts should be closed, since if a value v is reachable at a given program point, all values reachable from v are also reachable at this point. Indeed, we can prove the following lemma.

Proposition 3.16 (Closure) *If $c \in Ctx_E$ is closed, then $\forall x \in Var \cup Ind, E[\![M]\!]\rho xc$ is also closed.*

Proof: By cases on the operations used to compute analysis E.

- The constants $\emptyset$ and $Path$ are obviously closed.

- Construction: if c is closed, $\delta.c$ is closed since $\forall p \in \delta.c, \forall p' \in Path, \exists p'' \in c$ such that $p = \delta.p''$. $p''.p' \in c$ since c is closed. Therefore, $p.p' = \delta.p''.p' \in \delta.c$.

- Restriction: if c is closed, $\delta^{-1}.c$ is closed since $\forall p \in \delta^{-1}.c, \forall p' \in Path, \delta.p \in c$ hence $\delta.p.p' \in c$ since c is closed. Therefore $p.p' \in \delta^{-1}.c$.

- This property is obviously preserved by upper bound, composition and fixed point. It is obviously true for identity.

This gives the required result. $\square$

Proposition 3.17 *Analysis E is additive, i.e. predicate transformers $\phi_E \in Tr_E$ of analysis E are additive:*

$$\phi_E(\cup_{i \in I} c_i) = \cup_{i \in I} \phi_E(c_i).$$

I is a finite or infinite set.

Proof: It is easy to prove the additivity of elementary operations of analysis E. Additivity is preserved by composition, upper bound and fixed point, therefore all context transformers of analysis E are additive. $\square$

We shall see in the following that all our escape analyses are additive. This is a very important property since it enables us to find the image of each element of a context independently. Therefore, we will be able to give a much faster implementation. Even if we do not implement analysis E, this result shows that additivity does not come from further approximations, but from the definition of the most precise analysis itself.

We can also notice that analysis E is polymorphically invariant [Abr86], since types do not intervene in its definition. ***Polymorphic invariance*** means that the result given by the analysis of a general version (with type variables) is the same as the one given by the analysis of an instantiated version (where type variables have been substituted by types).

The following theorem gives a formal definition of the correctness of our escape analysis, which will be proved in Appendix A.1.

Theorem 3.18 (Correctness of analysis E) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let us assume that $\epsilon \notin E[\![N]\!]\rho\, x\, Path$.*

- `let rec` $x =$ `fun` $y \to M$ `in` N. *In this case, let ℓ be the location of the closure x. We assume $\epsilon \notin R(x) Path$ where R is as in the definition of analysis E.*

In these two cases, ℓ can be stack allocated, and deallocated at the end of N.

Intuitively, this theorem means that, if we assume that the result of N escapes (i.e. the reference scope is N) and we prove that the top of x does not escape, then the location ℓ at the top of x can be stack allocated, and deallocated at the end of N. This theorem yields the key criterion to decide stack allocation.

3.4 Towards efficient escape analyses

We have defined in the previous section an analysis for an untyped language. It cannot be implemented directly because the fixed point iteration may not terminate. Therefore we will define more approximate analyses. However, these analyses will be based on types. Our first task is therefore to adapt analysis E to a typed language. We shall then define a general framework, that can represent all the different escape analyses we are going to consider in the following. The formulas of escape analysis will remain the same, only the lattice of context transformers will change.

3.4.1 Adapting analysis E to types

We have to perform two changes to adapt analysis E to types. First, if M is of type $\tau_1 \to \ldots \to \tau_n$ where τ_n is not a function type, M can be given at most $n-1$ parameters. Then $E[\![M]\!]\rho$ will be defined on $\mathrm{Dom}(\rho) \cup [\![1, n-1]\!]$, and not on $\mathit{Var} \cup \mathit{Ind}$. Similarly, $\rho[\![y]\!]$ will be defined on $[\![1, n-1]\!]$ if y is of type $\tau_1 \to \ldots \to \tau_n$ where τ_n is not a function type. We can easily check that $E[\![M]\!]\rho$ and $\rho[\![y]\!]$ are never used outside this definition set, except in one case: if the language is polymorphic, and y is of type $\tau_1 \to \ldots \to \tau_{n-1} \to \beta$ where β is a type variable, y may be instantiated to a type $\sigma\tau_1 \to \ldots \to \sigma\tau_{n-1} \to \sigma\beta$ where $\sigma\beta$ is a function type. In this case, y has more parameters than expected according to its general type. We assume that the new parameters always escape, which is certainly correct:

$$E[\![y : \sigma\tau]\!]\rho = \{y \mapsto \lambda c.c, i \mapsto \lambda c.\rho[\![y : \tau]\!]ic \text{ if } i \in [\![1, n-1]\!], i \mapsto \lambda c.\mathit{Path} \text{ if } i \in [\![n, p-1]\!]\}$$

with $\tau = \tau_1 \to \ldots \to \tau_n$, $\sigma\tau_n = \tau'_n \to \ldots \to \tau'_p$, where τ_n and τ'_p are not function types.

Assuming that the new parameters always escape does not lead to a loss of precision. Indeed, assume that we analyze a function M of type $\tau_1 \to \ldots \to \tau_n$ where $\tau_n = \beta$ is a type variable. Assume that M is instantiated by substitution σ such that $\sigma\beta$ is a function type. This means that M will return a function f, but since the type of the result of M is β, f cannot come from inside the expression M. It always comes from a parameter of M. Therefore, we do not have escape information for the parameters of f (the escape information for f is $\rho_\top$). This is formalized by the following result:

Proposition 3.19 *If M is of type $\tau_1 \to \ldots \to \tau_n$ where τ_n is a type variable, then for all $i \geq n$, $E[\![M]\!]\rho ic = \mathit{Path}$ (if ρ has the corresponding property: if $\tau_y^{\mathrm{g}} = \tau_1 \to \ldots \to \tau_n$ where τ_n is a type variable, then for all $i \geq n$, $\rho[\![y]\!]ic = \mathit{Path}$).*

Proof: Case y is by hypothesis on ρ; cases $M\ N$ and `fun` $y \to M$ are by induction on M (and we check that $\rho_\top$ satisfies the hypothesis); cases `let` and `let rec` are satisfied by induction on N, and since $\rho[\![y]\!]$ satisfies the hypothesis by induction on M; case `if` is by induction on N (or on P); case $M; N$ is by induction on N; cases $C(M_1, \ldots, M_n)$ and $\overline{C_i}(M) \leftarrow N$: the expression is not of type $\tau_1 \to \ldots \to \tau_n$ where τ_n is a type variable.

$\rho[\![\overline{C_i}]\!]$ satisfies the hypothesis. $C?$, `true`, `false` and n, $n \in \mathbb{Z}$ are not of type $\tau_1 \to \ldots \to \tau_n$ where τ_n is a type variable. □

The second change takes into account the fact that not all paths really correspond to elements of a value of type τ. For example, for a closure, the first element of the path should be app, for a list it should be either `hd` or `tl`. In the general case, we define the paths corresponding to a value of type τ by: $Path(\tau)$ is the smallest set of paths such that

$$\begin{aligned}Path(\beta) &= Path\\ Path(\tau) &= \{\epsilon\} \cup \{\delta.p | \delta : \tau \to \tau', p \in Path(\tau')\}\end{aligned}$$

where τ is not a type variable, and δ is an accessor that can be applied to values of type τ. In particular,

$$\begin{aligned}&Path(\tau_1 \to \tau_2) = \{\epsilon\} \cup \{\text{app}.p | p \in Path(\tau_2)\},\\ &Path(\tau\ \texttt{list}) = \{\epsilon\} \cup \{\texttt{hd}.p | p \in Path(\tau)\} \cup \{\texttt{tl}.p | p \in Path(\tau\ \texttt{list})\},\\ &Path(\tau) = \{\epsilon\} \text{ if } \tau \in \{\texttt{int}, \texttt{bool}, \texttt{unit}\}.\end{aligned}$$

This is a definition by fixed point equations, since in the case of recursive types, $Path(\tau)$ is defined using itself.

If v is a value of type τ, only paths in $Path(\tau)$ are interesting. Then we can add intersections with $Path(\tau)$ everywhere in the definition of analysis E, to define an analysis $\tilde{E}$. We denote by $\tilde{\rho}$ the environment for analysis $\tilde{E}$. We define $Ctx^{\tau}_{E} = \mathcal{P}(Path(\tau))$, ordered by inclusion. This is a complete lattice by Proposition 3.5. We can easily check that, in the monomorphic case, analysis E never creates paths outside Ctx^{τ}_{E} for a value of type τ, except when using the context $Path$. In analysis $\tilde{E}$, this context should be replaced with $Path(\tau)$ where τ is the type of the corresponding value. In the polymorphic case, a variable y of type τ may be used with type $\sigma\tau$. Since $Path(\sigma\tau) \subseteq Path(\tau)$, we have $\tilde{E}[\![y : \sigma\tau]\!]\tilde{\rho} y c = c \subseteq Path(\tau)$ if $c \subseteq Path(\sigma\tau)$. However, $E[\![y : \sigma\tau]\!]\rho i c = \rho[\![y]\!] i c \not\subseteq Path(\sigma\tau_i)$ where $\tau = \tau_1 \to \ldots \to \tau_n$. Indeed, $\rho[\![y]\!] i c$ may contain every path in $Path(\tau_i)$. Therefore, we have to modify the analysis in the case of a variable:

$$\begin{aligned}\tilde{E}[\![y : \sigma\tau]\!]\tilde{\rho} = \{&y \mapsto \lambda c.c, i \mapsto \lambda c.\tilde{\rho}[\![y : \tau]\!] i c \cap Path(\sigma\tau_i) \text{ if } i \in [\![1, n-1]\!],\\ &i \mapsto \lambda c.Path(\tau'_i) \text{ if } i \in [\![n, p-1]\!]\}\end{aligned}$$

with $\tau = \tau_1 \to \ldots \to \tau_n$, $\sigma\tau_n = \tau'_n \to \ldots \to \tau'_p$, where τ_n and τ'_p are not function types.

Analysis $\tilde{E}$ is exactly the same as analysis E, except these two differences: $Path$ replaced with $Path(\tau)$, and the analysis of a variable defined above.

This new version of analysis E will be very useful in the following, because it will allow us to define analyses based on types, in which paths outside of $Path(\tau)$ have no representation.

Theorem 3.20 (Correctness of analysis $\tilde{E}$) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *Let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let us assume that $\epsilon \notin \tilde{E}[\![N]\!]\tilde{\rho}\, x\, (Path(\tau_N))$.*
- `let rec` x = `fun` $y \to M$ `in` N. *Let ℓ be the location of the closure x. We assume $\epsilon \notin R(x)(Path(\tau_N))$ where R is defined in the* `let rec` *case of the definition of analysis $\tilde{E}$.*

In these two cases, ℓ can be stack allocated, and deallocated at the end of N.

This theorem is proved in Appendix A.2. From now on, we will only consider analysis $\tilde{E}$, and we will denote it simply by E.

3.4.2 A general framework for escape analysis

We now define a general framework for escape analysis that we will use to represent all our escape analyses, of different precision levels. We first need the following definition.

Definition 3.21 If a context transformer ϕ analyses an expression of type τ and gives the escape information for a variable of type τ', we write $\phi : \tau[\tau']$. For example, $E[\![M]\!]\rho x : \tau_M[\tau_x^{\mathrm{g}}]$. This can be seen as an abbreviation of $E[\![M]\!]\rho x : \mathit{Ctx}_E^{\tau_M} \to \mathit{Ctx}_E^{\tau_x^{\mathrm{g}}}$. But for some analyses, the set of contexts does not depend on the type, and the notation $\phi : \tau[\tau']$ has therefore a stronger meaning. For operations which apply to context transformers $\phi : \tau[\tau']$, we write $\tau[\tau']$ as an exponent. The set of context transformers $\phi : \tau[\tau']$ for analysis E is $\mathit{Tr}_E^{\tau[\tau']} = \mathit{Ctx}_E^{\tau} \xrightarrow{\mathrm{m}} \mathit{Ctx}_E^{\tau'}$.

Our general framework for escape analysis is defined in Figure 3.6. We first define which operations on the context transformers Tr_E we will need. These operations are not yet defined in our general framework; they will be defined differently for each instance of this framework. Once these operations are known, the escape analysis can be computed by the given formulas. We recover the particular case of analysis E previously defined when we use the definition of Tr_E of Figure 3.7. Other escape analyses will be defined by only changing the definition of Tr_E, all formulas remaining the same. We are going to use somewhat simplified notations when the context is clear. When $\circ_E$ is the ordinary composition, the subscript E will be omitted. Similarly when $\sqcup_E$ is the least upper bound. The substitution σ will be omitted from $G_E^{\sigma,\tau}$ and $I_E^{\sigma,\tau[\tau']}$ when it is clear from the context.

The upper bound of the family $(\phi_i)_i$, $\sqcup_E(\phi_i)_i$ is used in two cases: first with a finite number of elements in expressions such as $\phi_1 \sqcup_E \phi_2$; second, to compute the least fixed point: $\mathrm{lfp}(f) = \sqcup_E(f^n(\bot^{\tau}_{\mathit{Tr}_E}))_{n\in\mathbb{N}}$ by Kleene's theorem (Theorem 3.2) since f is continuous[3]. Therefore, all properties of the fixed point can be inferred from properties of the infinite upper bound.

We can easily check, thanks to the typing rules given in Section 3.2.2, that the types associated with the contexts are coherent (they are the same as the types of the corresponding value): $E[\![M]\!]\rho x \in \mathit{Tr}_E^{\tau_M[\tau_x^{\mathrm{g}}]}$ if $x \in \mathit{Var}$, $E[\![M]\!]\rho i \in \mathit{Tr}_E^{\tau_M[\tau_{M,i}]}$ if $i \in \mathit{Ind}$.

Let σ be a given substitution on type variables. If a variable y of type τ is used in expression M with type $\sigma\tau$, then the escaping part of y in M is computed with type $\sigma\tau$ and we need to convert it to type τ. This conversion operation is what we call ***generalization***. The generalization operation is $G_E^{\tau} : \mathit{Ctx}_E^{\sigma\tau} \to \mathit{Ctx}_E^{\tau}$. If y is a function, we also have to convert the escape information for the parameters of y computed from the definition of y of type τ to the uses of y of type $\sigma\tau$. If the ith parameter of y is of type τ_i, the escape information is of type $\mathit{Ctx}_E^{\tau} \to \mathit{Ctx}_E^{\tau_i}$, and we convert it to type $\mathit{Ctx}_E^{\sigma\tau} \to \mathit{Ctx}_E^{\sigma\tau_i}$ by the ***instantiation*** $I_E^{\tau[\tau_i]}$. Such a conversion is very easy in the case of analysis E, because types intervene in a very simple manner in the definition of contexts. It will become much more complicated in the following escape analyses, since these analyses are really defined from types.

The general strategy to define other escape analyses follows the techniques of abstract interpretation. Assume that we want to define an approximate analysis T from analysis E.

[3]Indeed, $\rho \mapsto \rho[\![y]\!]i$ is a continuous function of ρ; the context transformers that do not depend on ρ are continuous functions of ρ; if $\forall k, \rho \mapsto \phi_k(\rho)$ is a continuous function of ρ, the composition $\rho \mapsto \phi_1(\rho) \circ \phi_2(\rho)$, the upper bound $\rho \mapsto \sqcup_k \phi_k(\rho)$ and the instantiation $\rho \mapsto I_T^{\sigma,\tau[\tau']}(\phi_1(\rho))$ are also continuous functions of ρ. Therefore when we compute $\mathrm{lfp}(f)$ in the `let rec` case of our analysis, f is continuous.

The family of complete lattices of context transformers $Tr_E^{\tau[\tau']}$ comes with 10 operations:

Construction: $\mathrm{Constr}_E(\delta,\tau) \in Tr_E^{\tau'[\tau]}$	Restriction: $\mathrm{Restr}_E(\delta,\tau) \in Tr_E^{\tau[\tau']}$ where $\delta : \tau \to \tau'$
Identity: $\mathrm{Id}_E^{\tau} \in Tr_E^{\tau[\tau]}$	Step: $\mathrm{test}_E^{\tau_y\to\tau,\tau'} \in Tr_E^{\tau_y\to\tau[\tau']}$
Greatest element: $\top_{Tr_E}^{\tau'} \in Tr_E^{\tau[\tau']}$	Upper bound: $\sqcup_E(\phi_i)_i \in Tr_E^{\tau[\tau']}$ if $\phi_i \in Tr_E^{\tau[\tau']}$
Least element: $\perp_{Tr_E}^{\tau'} \in Tr_E^{\tau[\tau']}$	Composition: $\circ_E \in Tr_E^{\tau'[\tau'']} \times Tr_E^{\tau[\tau']} \to Tr_E^{\tau[\tau'']}$
Generalization: $G_E^{\sigma,\tau} \in Tr_E^{\sigma\tau[\tau]}$	Instantiation: $I_E^{\sigma,\tau[\tau']} \in Tr_E^{\tau[\tau']} \to Tr_E^{\sigma\tau[\sigma\tau']}$

The analysis is defined as follows:

$i \in Ind = \mathbb{N} - \{0\}$	Parameter indices
$Exp_E = (Var \cup Ind) \to Tr_E$	Abstract domain of an expression
$Val_E = Ind \to Tr_E$	Abstract domain of a value
$\rho \in Env_E = Var \to Val_E$	Abstract environment
$E[\![M]\!] : Env_E \to Exp_E$	Abstract semantic function

In the monomorphic case:
$E[\![y]\!]\rho = \{y \mapsto \mathrm{Id}_E^{\tau}, i \mapsto \rho[\![y]\!]i \text{ if } i \in Ind\}$
In the polymorphic case:
$E[\![y : \sigma\tau]\!]\rho = \{y \mapsto G_E^{\sigma,\tau}, i \mapsto I_E^{\sigma,\tau[\tau_i]}(\rho[\![y : \tau]\!]i) \text{ if } i \in [\![1, n-1]\!],$
$\qquad i \mapsto \top_{Tr_E}^{\tau_i'} \text{ if } i \in [\![n, p-1]\!]\}$
with $\tau = \tau_1 \to \ldots \to \tau_n$, $\sigma\tau_n = \tau_n' \to \ldots \to \tau_p'$, where τ_n and τ_p' are not function types.

$E[\![M\ N]\!]\rho = \{i \mapsto \phi(i+1) \text{ if } i \in Ind, x \mapsto \phi(x) \sqcup \psi(x) \text{ if } x \in Var\}$
where $\phi(x) = (E[\![M]\!]\rho x) \circ_E \mathrm{Constr}_E(\mathrm{app}, \tau_M)$ and $\psi(x) = (E[\![N]\!]\rho x) \circ_E (\phi(1))$

$E[\![\texttt{fun}\ y \to M]\!]\rho = \{1 \mapsto \phi(y), i \mapsto \phi(i-1) \text{ if } i > 1, x \mapsto \phi(x) \text{ if } x \in Var - \{y\}\ \}$
$\qquad \sqcup \{x \mapsto \mathrm{test}_E^{\tau_y\to\tau_M,\tau_x^{\mathrm{g}}} \text{ if } x \in FV(\texttt{fun}\ y \to M)\ \}$
where $\phi(x) = (E[\![M]\!]\,\rho[y \mapsto \rho_\top]\,x) \circ_E \mathrm{Restr}_E(\mathrm{app}, \tau_y \to \tau_M)$ and $\rho_\top = \{i \mapsto \top_{Tr_E}^{\tau_{y,i}} \text{ if } i \in Ind\}$.

$E[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]\rho = \{x \mapsto E[\![N]\!]\rho' x \sqcup_E (E[\![M]\!]\rho x \circ_E E[\![N]\!]\rho' y) \text{ if } x \in Var - \{y\},$
$\qquad i \mapsto E[\![N]\!]\rho' i \text{ if } i \in Ind\}$ where $\rho' = \rho[y \mapsto (E[\![M]\!]\rho)_{|Ind}]$

$E[\![\texttt{let rec}\ f = \texttt{fun}\ y \to M\ \texttt{in}\ N]\!]\rho = \{x \mapsto R(x) \text{ if } x \in Var - \{f\}, i \mapsto E[\![N]\!]\rho' i \text{ if } i \in Ind\}$
where $\rho' = \mathrm{lfp}(\lambda\rho_1.\rho[f \mapsto (E[\![\texttt{fun}\ y \to M]\!]\rho_1)_{|Ind}])$
and $R = \mathrm{lfp}(\lambda R.\{x \mapsto E[\![N]\!]\rho' x \sqcup_E (E[\![\texttt{fun}\ y \to M]\!]\rho' x \circ_E R(f))\})$.

$E[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]\rho x = [(E[\![M]\!]\rho)_{|Var} x \circ_E \perp_{Tr_E}^{\tau_M}] \sqcup_E E[\![N]\!]\rho x \sqcup_E E[\![P]\!]\rho x$

$E[\![M; N]\!]\rho x = [(E[\![M]\!]\rho)_{|Var} x \circ_E \perp_{Tr_E}^{\tau_M}] \sqcup_E E[\![N]\!]\rho x$

$E[\![C(M_1, \ldots, M_n)]\!]\rho x = \sqcup_E((E[\![M_i]\!]\rho)_{|Var} x \circ_E \mathrm{Restr}_E(\overline{C_i}, \tau'))_{i\in[\![1,n]\!]}$
where $C : \tau_1' \times \ldots \times \tau_n' \to \tau'$.

$E[\![\overline{C_i}(M) \leftarrow N]\!]\rho x = [(E[\![M]\!]\rho)_{|Var} x \circ_E \perp_{Tr_E}^{\tau_M}] \sqcup_E [(E[\![N]\!]\rho)_{|Var} x \circ_E \top_{Tr_E}^{\tau_N}]$

$\rho_0[\![\overline{C_i}]\!] = \{1 \mapsto \mathrm{Constr}_E(\overline{C_i}, \tau') \circ_E \mathrm{Restr}_E(\mathrm{app}, \tau' \to \tau_i'), k \mapsto \top_{Tr_E}^{\tau_k} \text{ if } k \in [\![2, r-1]\!]\}$
where $C : \tau_1' \times \ldots \times \tau_n' \to \tau'$, $\tau_i' = \tau_2 \to \ldots \to \tau_r$, τ_r is not a function type.
$\rho_0[\![C?]\!] = \{\}$
$\rho_0[\![a]\!] = \{\}$ where $a \in \{\texttt{true}, \texttt{false}\} \cup \mathbb{Z}$

Figure 3.6: General framework of escape analysis for a typed language.

Contexts:	$Ctx_E^\tau = \mathcal{P}(Path(\tau))$
Context transformers:	$Tr_E^{\tau[\tau']} = Ctx_E^\tau \xrightarrow{\mathrm{m}} Ctx_E^{\tau'}$
Identity:	$\mathrm{Id}_E^\tau = \lambda c.c$
Greatest element:	$\top_{Tr_E}^\tau = \lambda c.Path(\tau)$
Least element:	$\bot_{Tr_E}^\tau = \lambda c.\emptyset$
Construction:	$\mathrm{Constr}_E(\delta,\tau) = \lambda c.\{\delta.p \mid p \in c\}$
Restriction:	$\mathrm{Restr}_E(\delta,\tau) = \lambda c.\{p \mid \delta.p \in c\}$
Step:	$\mathrm{test}_E^{\tau_y \to \tau,\tau'} = \lambda c.\mathrm{if}\ \epsilon \in c\ \mathrm{then}\ Path(\tau')\ \mathrm{else}\ \emptyset$
Upper bound:	$\sqcup_E(\phi_i)_i = \lambda c. \cup_i \phi_i(c)$
Composition:	$\phi \circ_E \psi = \lambda c.\phi(\psi(c))$
Generalization:	$G_E^{\sigma,\tau} = \lambda c.c$
Instantiation:	$I_E^{\sigma,\tau[\tau']}(\phi) = \lambda c.\phi(c) \cap Path(\sigma\tau')$

Figure 3.7: Definition of the lattice of context transformers Tr_E for analysis E.

We define a semi-dual Galois connection between the complete lattices $Ctx_E^\tau(\subseteq)$ and $Ctx_T^\tau(\leq)$:

$$(Ctx_E^\tau, \subseteq) \underset{\alpha_T^\tau}{\overset{\gamma_T^\tau}{\leftrightarrows}} (Ctx_T^\tau, \leq) \quad \text{that is} \quad \alpha_T^\tau(x) \leq y \Leftrightarrow x \subseteq \gamma_T^\tau(y).$$

Furthermore, $(\alpha_T^\tau, \gamma_T^\tau)$ will be such that $\alpha_T^\tau \circ \gamma_T^\tau = \mathrm{Id}$. By Proposition 3.8, this is equivalent to α_T^τ surjective, and to γ_T^τ injective. By Proposition 3.9, we have a semi-dual Galois connection between the complete lattices $Tr_E^{\tau[\tau']} = Ctx_E^\tau \xrightarrow{\mathrm{m}} Ctx_E^{\tau'}$ and $Tr_T^{\tau[\tau']} = Ctx_T^\tau \xrightarrow{\mathrm{m}} Ctx_T^{\tau'}$ ordered pointwise:

$$(Tr_E^{\tau[\tau']}, \leq_1'') \underset{\alpha_T^{\tau[\tau']}}{\overset{\gamma_T^{\tau[\tau']}}{\leftrightarrows}} (Tr_T^{\tau[\tau']}, \leq_2'')$$

where $\alpha_T^{\tau[\tau']}(\phi_E) = \alpha_T^{\tau'} \circ \phi_E \circ \gamma_T^\tau$ and $\gamma_T^{\tau[\tau']}(\phi_T) = \gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^\tau$.

Proposition 3.22 *We assume that all elementary operations of analysis T are the image of the corresponding operation of analysis E by the abstraction. That is* $\mathrm{Constr}_T(\delta,\tau) = \alpha_T^{\tau'[\tau]}(\mathrm{Constr}_E(\delta,\tau))$ *where* $\delta : \tau \to \tau'$ *and similarly for* $\mathrm{Restr}_T(\delta,\tau)$ *and* $G_T^{\sigma,\tau}$. *Assume that*

$$\mathrm{Id}_E^\tau = \mathrm{Id}$$

$$\mathrm{test}_E^{\tau_y \to \tau,\tau'} = \lambda c.\mathrm{if}\ c \supseteq c_0\ \mathrm{then}\ \top_{Ctx_E^{\tau'}}\ \mathrm{else}\ \bot_{Ctx_E^{\tau'}}\ (c_0 \textit{ is a constant})$$

$\top_{Tr_E}^\tau = \lambda c.\top_{Ctx_E^\tau}$ *is the greatest element of* $Tr_E^{\tau'[\tau]}$

$\bot_{Tr_E}^\tau = \lambda c.\bot_{Ctx_E^\tau}$ *is the least element of* $Tr_E^{\tau'[\tau]}$

$\circ_E = \circ$ *is the ordinary composition of functions*

$\sqcup_E = \sqcup$ *is the upper bound of* $Tr_E^{\tau[\tau']}$

and define

$$\mathrm{Id}_T^\tau = \alpha_T^{\tau[\tau]}(\mathrm{Id}_E^\tau) = \mathrm{Id}$$

$\mathrm{test}_T^{\tau_y \to \tau, \tau'} = \alpha_T^{\tau_y \to \tau[\tau']}(\mathrm{test}_E^{\tau_y \to \tau, \tau'}) = \lambda c.\mathrm{if}\ c \geq \alpha_T^{\tau_y \to \tau}(c_0)\ \mathrm{then}\ \top_{Ctx_T^{\tau'}}\ \mathrm{else}\ \bot_{Ctx_T^{\tau'}}$

$\top_{Tr_T}^{\tau} = \alpha_T^{\tau'[\tau]}(\top_{Tr_E}^{\tau}) = \lambda c.\top_{Ctx_T^{\tau}}$ *is the greatest element of* $Tr_T^{\tau'[\tau]}$

$\bot_{Tr_T}^{\tau} = \alpha_T^{\tau'[\tau]}(\bot_{Tr_E}^{\tau}) = \lambda c.\bot_{Ctx_T^{\tau}}$ *is the least element of* $Tr_T^{\tau'[\tau]}$

$\circ_T = \circ$ *is the ordinary composition of functions*

$\sqcup_T = \sqcup$ *is the upper bound of* $Tr_T^{\tau[\tau']}$

We also assume

$$I_T^{\sigma,\tau[\tau']} \geq \alpha_T^{\sigma\tau[\sigma\tau']} \circ I_E^{\sigma,\tau[\tau']} \circ \gamma_T^{\tau[\tau']}$$

or the slightly weaker property[4]*: for all ϕ_T and ϕ_E that may be parameters of the instantiation,*

$$\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E) \Rightarrow I_T^{\sigma,\tau[\tau']}(\phi_T) \geq \alpha_T^{\sigma\tau[\sigma\tau']} \circ I_E^{\sigma,\tau[\tau']}(\phi_E).$$

Then for all context transformers ϕ_T of analysis T corresponding to ϕ_E of analysis E (i.e. $\phi_T = T[\![M]\!]\rho_T x$ and $\phi_E = E[\![M]\!]\rho_E x$, or $\phi_T = \rho_T[\![x]\!]i$ and $\phi_E = \rho_E[\![x]\!]i$), we have $\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E)$.

Let us consider two kinds of expressions:

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let $\phi_E = E[\![N]\!]\rho_E x$ and $\phi_T = T[\![N]\!]\rho_T x$.*
- `let rec` $x = $ `fun` $y \to M$ `in` N. *In this case, let ℓ be the location of the closure x. Let $\phi_E = R_E(x)$ and $\phi_T = R_T(x)$ where R is defined in the* `let rec` *case of the definition of analysis E, R_E is the value of R for analysis E, R_T for analysis T.*

We assume that, if $\phi_E(\top_{Ctx_E^{\tau_N}}) \not\supseteq c_0$, then ℓ can be stack allocated, and deallocated at the end of N (this will be given by the correctness theorem of analysis E).

In this case, if $\phi_T(\top_{Ctx_T^{\tau_N}}) \not\geq \alpha_T^{\tau_x^{\mathrm{g}}}(c_0)$, then ℓ can be stack allocated, and deallocated at the end of N.

Proof: The computation of the image by $\alpha_T^{\tau[\tau']}$ of identity, step, and constants proves that for these operations $\phi_T = \alpha_T^{\tau[\tau']}(\phi_E)$. The fact that $\alpha_T^{\tau'[\tau]}(\bot_{Tr_E}^{\tau})$ is the least element of $Tr_T^{\tau'[\tau]}$ comes from Proposition 3.8. $\alpha_T^{\tau'[\tau]}(\top_{Tr_E}^{\tau})$ is the greatest element of $Tr_T^{\tau'[\tau]}$ because $\alpha_T^{\tau'[\tau]}$ is onto. By hypothesis, we also have $\phi_T = \alpha_T^{\tau[\tau']}(\phi_E)$ for the construction, restriction and generalization.

For the composition, if $\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E)$ and $\psi_T \geq \alpha_T^{\tau'[\tau'']}(\psi_E)$, then $\psi_T \circ \phi_T \geq \alpha_T^{\tau''} \circ \psi_E \circ \gamma_T^{\tau'} \circ \alpha_T^{\tau'} \circ \phi_E \circ \gamma_T^{\tau} \geq \alpha_T^{\tau''} \circ (\psi_E \circ \phi_E) \circ \gamma_T^{\tau}$ since $\gamma_T^{\tau'} \circ \alpha_T^{\tau'} \geq \mathrm{Id}$ by Definition 3.7. Therefore, $\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E)$ and $\psi_T \geq \alpha_T^{\tau'[\tau'']}(\psi_E)$ implies $\psi_T \circ \phi_T \geq \alpha_T^{\tau[\tau'']}(\psi_E \circ \phi_E)$.

For the upper bound, if $\phi_{Tk} \geq \alpha_T^{\tau[\tau']}(\phi_{Ek})$, $\sqcup_k \phi_{Tk} \geq \sqcup_k \alpha_T^{\tau[\tau']}(\phi_{Ek}) = \alpha_T^{\tau[\tau']}(\sqcup_k \phi_{Ek})$ since $\alpha_T^{\tau[\tau']}$ is additive (Proposition 3.8).

[4] The first property is equivalent to the second one if all functions may be parameters of the instantiation, but we shall see that this is not the case in general.

For the instantiation, $\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E)$ implies $I_T^{\sigma,\tau[\tau']}(\phi_T) \geq \alpha_T^{\sigma\tau[\sigma\tau']} \circ I_E^{\sigma,\tau[\tau']}(\phi_E)$ by definition of Galois connections.

By induction on the building of expressions of our analysis, we have therefore $\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E)$ for all context transformers.

If $\phi_E(\top_{Ctx_E^{\tau_N}}) \supseteq c_0$, then $\phi_T(\top_{Ctx_T^{\tau_N}}) \geq (\alpha_T^{\tau_x^{\mathrm{g}}} \circ \phi_E \circ \gamma_T^{\tau_N})(\top_{Ctx_T^{\tau_N}}) \geq \alpha_T^{\tau_x^{\mathrm{g}}}(c_0)$. Therefore, $\phi_T(\top_{Ctx_T^{\tau_N}}) \not\geq \alpha_T^{\tau_x^{\mathrm{g}}}(c_0)$ implies $\phi_E(\top_{Ctx_E^{\tau_N}}) \not\supseteq c_0$. $\square$

This proposition shows that once the Galois connection is defined, the design of the analysis is complete: it only remains to compute context transformers. This computation is easy in most cases, the only really difficult case will be instantiation. This proposition will be used three times, to prove the correctness of T with respect to E and of L and L' with respect to T. It is the key of our design of escape analyses.

3.4.3 Properties of analysis E

Analysis E adapted to types is obviously still additive. We have to adapt the definition of a closed context to these new contexts.

Definition 3.23 A context $c \in Ctx_E^\tau$ is ***closed*** if and only if, for all $p \in c$, for all $p' \in Path$ such that $p.p' \in Path(\tau)$, $p.p' \in c$.

Proposition 3.16 does not extend to this new definition, in the case of a polymorphic language. Indeed, in the variable case, $E[\![y : \sigma\tau]\!]\rho(y : \tau)c = c$, $c \in Ctx_E^{\sigma\tau}$ is closed does not imply that $c \in Ctx_E^\tau$ is closed (for example, if $\tau = \beta$ and $\epsilon \in c$, c should contain all paths, and since $c \subseteq Path(\sigma\beta)$, it cannot contain all paths). However, we have the following property:

Proposition 3.24 *Let us consider a property $P^{\tau[\tau']}$ of context transformers $\phi : \tau[\tau'] \in Tr_E$. We assume that $P^{\tau[\tau']}$ is true for constants ($P^{\tau[\tau']}(\lambda c.\emptyset)$ and $P^{\tau[\tau']}(\lambda c.Path(\tau'))$), identity ($P^{\tau[\tau]}(\lambda c.c)$), restriction ($P^{\tau[\tau']}(\lambda c.\delta^{-1}.c)$ where $\delta : \tau \to \tau'$), construction ($P^{\tau[\tau']}(\lambda c.\delta.c)$ where $\delta : \tau' \to \tau$), step ($P^{\tau[\tau']}(\lambda c.\text{if } \epsilon \in c \text{ then } Path(\tau') \text{ else } \emptyset)$). We also assume that $P^{\tau[\tau']}$ is preserved by composition ($P^{\tau[\tau']}(\phi)$ and $P^{\tau'[\tau'']}(\phi')$ implies $P^{\tau[\tau'']}(\phi' \circ \phi)$) and finite and infinite upper bound ($\forall j, P^{\tau[\tau']}(\phi_j)$ implies $P^{\tau[\tau']}(\sqcup_j \phi_j)$). These are all properties we prove when we want to prove property $P^{\tau[\tau']}$ in the monomorphic case.*

Then, in the case of a polymorphic language, $P^{\tau_M[\tau_x]}(E[\![M]\!]\rho x)$ where $x \in Var$ bound by a `fun` *expression (it is wrong in general if x is bound by a* `let` *or* `let rec` *expression), $P^{\tau_M[\tau_{M,i}]}(E[\![M]\!]\rho i)$ where $i \in Ind$, $P^{\tau_y^{\mathrm{g}}[\tau_{y,i}^{\mathrm{g}}]}(\rho[\![y]\!]i)$ for all $i \in Ind$.*

The proof is in Appendix A.3.

Proposition 3.25 *In the case of a monomorphic language, if $c \in Ctx_E^\tau$ is closed then $E[\![M]\!]\rho x c \in Ctx_E^{\tau'}$ is closed for all $x \in Var \cup Ind$ and $\rho[\![y]\!]ic$ is closed for all $i \in Ind$.*

In the case of a polymorphic language, if $c \in Ctx_E^\tau$ is closed then $E[\![M]\!]\rho x c \in Ctx_E^{\tau'}$ is closed for all $x \in Ind$ or $x \in Var$ bound by a `fun` *expression (it is wrong in general if x is bound by a* `let` *or* `let rec` *expression) and $\rho[\![y]\!]ic$ is closed for all $i \in Ind$.*

Proof: Taking into account the properties proved in Proposition 3.16, we can immediately apply Proposition 3.24 which gives the result. $\square$

3.5 Analysis based on annotated types

In the preceding section, parts of values were represented by sets of access paths, which were intuitively equivalent to marking a part of the value. Abstract values were then functions from $\mathcal{P}(Path)$ to $\mathcal{P}(Path)$, therefore infinite objects. The goal of this section and of the following ones is to give good finite representations of these functions (even by losing some precision) to make sure that the fixed point iteration terminates.

In this section, we have chosen to represent parts of values by marking their type. This is equivalent to the marking by access paths in the case of non recursive types. But for recursive types, there are cycles in the type and we cannot distinguish any more certain parts of data. The advantage is that types (at least those considered here) are finite objects. Therefore the analysis terminates, at the cost of a loss in precision. Types can be easily represented.

3.5.1 Definition of the graph of a type

Informally, we want to mark types by underlining part of them. The underlined part of the type will correspond to the escaping part of the value. For example, if v is of type `int list` $*$ `int`, and we underline the `int list` : $\underline{\texttt{int list}} * \texttt{int}$, then the first component of the pair v escapes. However underlining the type term itself is not very adequate since the type may be a simple name even if it represents a complex data structure. Therefore, we build a representation of types which better reflects the structure of data. This representation is a graph defined as follows.

Definition 3.26 (Graphs of τ) For each type τ, a graph G of τ is defined as follows. G contains nodes labelled by types (the node n labelled τ is written $n[\tau]$) and edges labelled by accessors (an edge labelled δ is denoted by $n_1 \xrightarrow{\delta} n_2$). G contains a special node $n_0[\tau]$ which is the top of G. For each node $n'[\tau']$ in G, for each accessor $\delta : \tau' \to \tau''$, G contains a node $n''[\tau'']$ and an edge $n' \xrightarrow{\delta} n''$, and these are the only outgoing edges of n'. All nodes of G are added by this process, starting from the only node n_0. Therefore, the nodes of G are all reachable from $n_0[\tau]$.

Such a graph is of course not unique since in the above definition $n''[\tau'']$ is not necessarily a new node. Figure 3.8 illustrates this for two examples. However, we want to select a unique graph for each type (this will make the presentation much simpler). Moreover, the graph defined above is not necessarily finite. For example, if n'' is always a new node, then the graph is infinite for every recursive type. So we really have to merge nodes in this case. We first need to define preliminary mathematical notions, to have tools to manipulate these graphs.

Definition 3.27 (Graph morphism) A graph morphism from G to G' is a function ϕ such that for each node $n[\tau]$ of G, $\phi(n)$ is a node $n'[\tau]$ of G', and for all edges $n_1 \xrightarrow{\delta} n_2$ of G, there is an edge $\phi(n_1) \xrightarrow{\delta} \phi(n_2)$ in G'. If n_0 is the top node of G, $\phi(n_0)$ is the top node of G'.

A graph isomorphism is a bijective (one-to-one and onto) graph morphism.

Lemma 3.28 *If there is a morphism ϕ from G to G', and another graph morphism ϕ' from G' to G, then $\phi = \phi'^{-1}$ and ϕ is an isomorphism from G to G'.*

This result is proved in Appendix B.1.

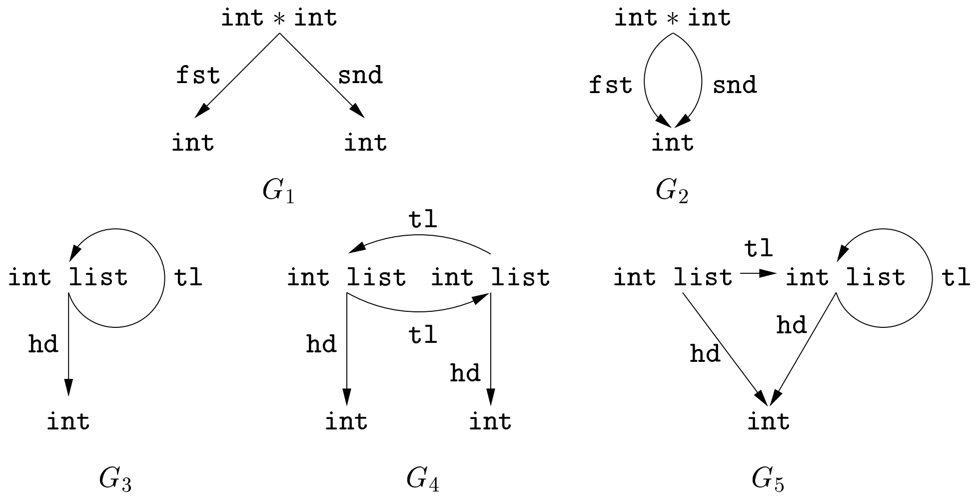


Figure 3.8: Several graphs for types `int * int` and `int list`.

Definition 3.29 The relation $G \geq G'$ defined by "there is a graph morphism from G to G'" is a preorder relation.

Proof: Identity is a morphism from G to G, hence $\geq$ is reflexive. If ϕ is a morphism from G to G' and ϕ' from G' to G'', then $\phi' \circ \phi$ is a morphism from G to G''. Then $\geq$ is transitive. $\square$

Definition 3.30 (The chosen graph of τ) Among the graphs of τ, we choose the graph G which has the following properties:

1. There is no acyclic path containing two nodes labelled by the same type. Intuitively, this property can be enforced by merging nodes if at the beginning there exists such an acyclic path.

2. G is the greatest such graph, for the $\geq$ relation. This prevents ourselves from merging too many nodes, which would lead to a less precise analysis.

For each graph, there is a unique corresponding type, the type which labels the top node of the graph.

Theorem 3.31 (Existence and uniqueness) *For each type τ, there exists a unique (up to isomorphism) graph satisfying Definition 3.30.*

The proof can be found in Appendix B.1. Since we have the uniqueness of the graph of τ satisfying Definition 3.30, $\mathrm{Gr}(\tau)$ can be defined to be this graph.

The chosen graph associated with $\tau = \tau_1 \to \tau_2$ is $n_0[\tau] \xrightarrow{\text{app}} n_1[\tau_2]$, to which we add the graph of τ_2 (this last graph may reuse n_0 in case of recursion). The chosen graph associated with a constructed type

$$\begin{array}{lcl} \texttt{type } (\beta_1, ..., \beta_p)\tau_{\mathrm{N}} & = & C \texttt{ of } \tau_1 * \tau_2 * ... * \tau_k \\ & | & C' \texttt{ of } \tau'_1 * ... * \tau'_{k'} \\ & | & ... \end{array}$$

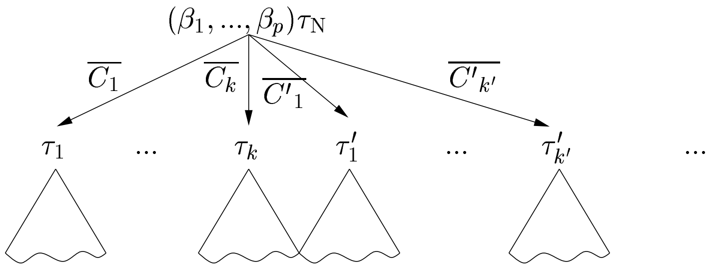


Figure 3.9: Graph of a constructed type

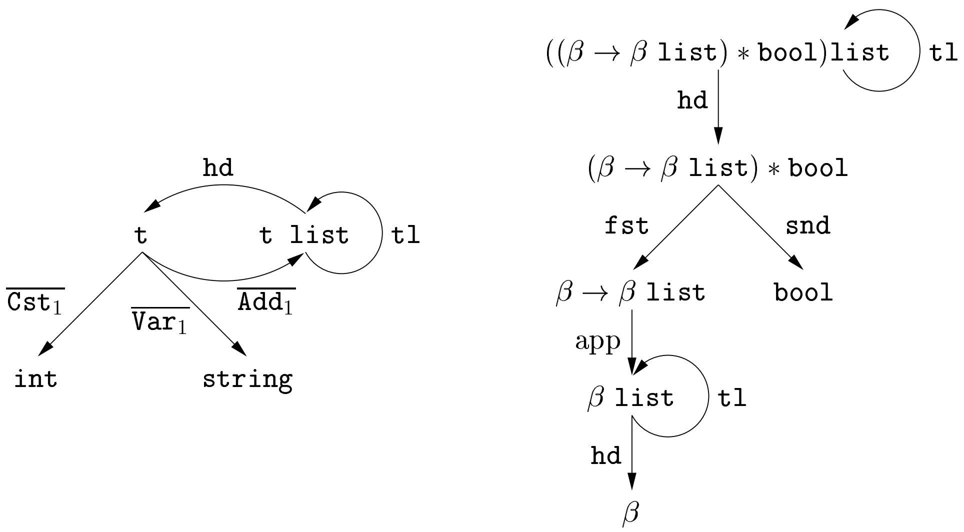


Figure 3.10: Examples of graphs

is represented in Figure 3.9, assuming there is no recursion. Otherwise, nodes associated by the recursion have to be merged (i.e. if there is an occurrence of $(\beta_1, ..., \beta_p)\tau_N$ in τ_i, τ'_i, ... it is represented by the top node of the graph).

Example 3.32 In Figure 3.8, there is a morphism from graph G_1 to graph G_2 (just map the top node of G_1 to the top node of G_2 and the bottom nodes of G_1 to the bottom node of G_2). Therefore, G_2 cannot be chosen as graph for `int` $*$ `int`. The only possible graphs for type `int` $*$ `int` are G_1 and G_2, and by Definition 3.30, we choose G_1 as unique graph for this type.

The chosen graph for `int list` is G_3. The first property of the definition of the graph of τ (Definition 3.30) fails for G_4 and G_5.

Figure 3.10 shows the chosen graph for two more examples of types:

```
type t  =  Add of t list
        |  Cst of int
        |  Var of string
```

and $((\beta \to \beta\ \texttt{list}) * \texttt{bool})\texttt{list}$.

3.5.2 Abstraction

Each path $p \in \mathit{Path}$ can be associated to a node in the graph G, or the empty set if the path p does not correspond to an element of a value of the type τ of G (i.e. $p \notin \mathit{Path}(\tau)$). In the latter case, the path p is said to be forbidden. The NofP (node of path) function computes the node corresponding to a path.

$$\mathrm{NofP}_G(n, \epsilon) = \{n\}$$
$$\mathrm{NofP}_G(n, p) = \{n\} \text{ if } n \text{ is labelled by a type variable}$$
$$\mathrm{NofP}_G(n, \delta.c) = \mathrm{NofP}_G(n', c) \text{ if there exists an edge } n \xrightarrow{\delta} n' \text{ in } G.$$
$$\mathrm{NofP}_G(n, \delta.c) = \emptyset \text{ otherwise (forbidden path).}$$
$$\mathrm{NofP}_G(p) = \mathrm{NofP}_G(n_0, p) \text{ where } n_0 \text{ is the top node of } G.$$

A context for analysis T, for a value of type τ is then a set of nodes of the graph $\mathrm{Gr}(\tau)$. Such a context is said to be an ***annotated type***. Let $\mathrm{Nodes}(G)$ be the set of nodes of graph G. We define Ctx_T^τ to be the set of annotations of $\mathrm{Gr}(\tau)$, that is:

$$\mathit{Ctx}_T^\tau = \mathcal{P}(\mathrm{Nodes}(\mathrm{Gr}(\tau))).$$

Let $\mathit{Ctx}_T = \cup_{\tau \in \mathit{Type}} \mathit{Ctx}_T^\tau$.

Annotated types are represented by underlining marked parts of the type (i.e. nodes that are in the context), when this is possible. For example, the graph of $\mathtt{int} * \mathtt{int}$ contains three nodes $n_1[\mathtt{int} * \mathtt{int}]$, $n_2[\mathtt{int}]$ at the end of the edge labelled $\mathtt{fst}$ and $n_3[\mathtt{int}]$ at the end of the edge labelled $\mathtt{snd}$, $\underline{\mathtt{int}} * \mathtt{int} = \{n_2\}$, $\mathtt{int}\underline{*}\mathtt{int} = \{n_1\}$ and $\mathtt{int} * \underline{\mathtt{int}} = \{n_3\}$. The graph of $\beta \to \beta$ contains the nodes $n_1[\beta \to \beta]$ and $n_2[\beta]$, $\beta\underline{\to}\beta = \{n_1\}$, $\beta \to \underline{\beta} = \{n_2\}$. If the marking is not visible on the type itself, we have to represent the graph. For example, this is the case if we want to mark a part of type $\mathtt{t}$ of Figure 3.10. The only contexts that can be represented by underlining $\mathtt{t}$ are the empty context $\mathtt{t} = \emptyset$ and the full context $\underline{\mathtt{t}} = \mathrm{Nodes}(\mathrm{Gr}(\mathtt{t}))$.

The set of contexts Ctx_T^τ ordered by inclusion is a complete lattice (Proposition 3.5). The upper bound of annotated types is the union, the lower bound is the intersection.

We define the abstraction $\alpha_T^\tau : \mathit{Ctx}_E^\tau \to \mathit{Ctx}_T^\tau$ by

$$\alpha_T^\tau(c) = \cup_{p \in c} \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p).$$

The abstraction of the context $c \in \mathit{Ctx}_E^\tau$, $\alpha_T^\tau(c)$, is the set of nodes of $\mathrm{Gr}(\tau)$ obtained by following from the top of the graph, all paths labelled by $p \in c$.

Ctx_E^τ and Ctx_T^τ are complete lattices, then by Proposition 3.8, the concretization function is:

$$\gamma_T^\tau(c') = \sqcup\{c \in \mathit{Ctx}_E^\tau | \alpha_T^\tau(c) \subseteq c'\} = \{p \in \mathit{Path}(\tau) | \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) \subseteq c'\}$$

$\gamma_T^\tau(c')$ is the set of paths p going from the top of $\mathrm{Gr}(\tau)$ to a node marked in c'. $(\alpha_T^\tau, \gamma_T^\tau)$ is a semi-dual Galois connection (Definition 3.7)

$$(\mathit{Ctx}_E^\tau, \subseteq) \underset{\alpha_T^\tau}{\overset{\gamma_T^\tau}{\leftrightarrows}} (\mathit{Ctx}_T^\tau, \subseteq), \qquad \alpha_T^\tau(x) \subseteq y \Leftrightarrow x \subseteq \gamma_T^\tau(y).$$

We also have $\alpha_T^\tau \circ \gamma_T^\tau = \mathrm{Id}$, which is equivalent to α_T^τ onto, equivalent to γ_T^τ one-to-one (Proposition 3.8).

For the function lattice $\mathit{Tr}_T^{\tau[\tau']} = \mathit{Ctx}_T^\tau \xrightarrow{\mathrm{m}} \mathit{Ctx}_T^{\tau'}$, the abstraction is then (Proposition 3.9):

$$\alpha_T^{\tau[\tau']}(\phi_E) = \alpha_T^{\tau'} \circ \phi_E \circ \gamma_T^\tau.$$

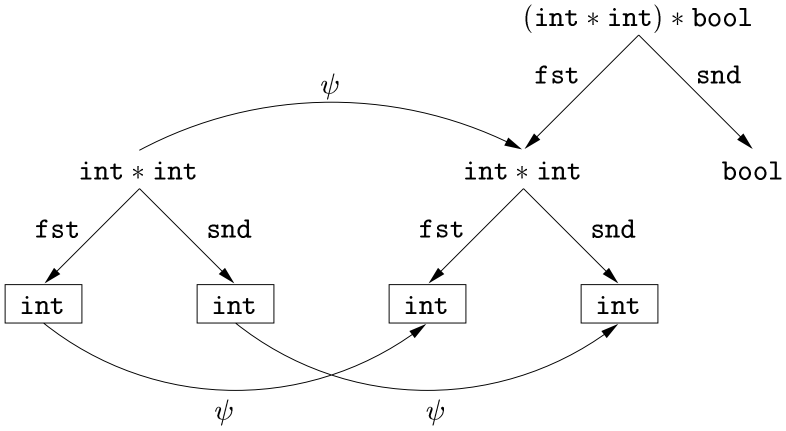


Figure 3.11: Example of construction. Nodes contained in the context are boxed in this figure.

Constructions Let $\delta : \tau \to \tau'$ be an accessor of type τ.

$$\mathrm{Constr}_T(\delta,\tau)(c) = \alpha_T^{\tau'[\tau]}(\lambda c.\delta.c)(c) = \alpha_T^{\tau}(\delta.\gamma_T^{\tau'}(c))$$

Let $n_0[\tau]$ be the top of $\mathrm{Gr}(\tau)$. The graph $\mathrm{Gr}(\tau)$ contains an edge $n_0 \xrightarrow{\delta} n_1$. The subgraph G'' of $\mathrm{Gr}(\tau)$ reachable from n_1 is isomorphic to $\mathrm{Gr}(\tau')$ (by uniqueness of $\mathrm{Gr}(\tau')$ up to isomorphism, Theorem 3.31). Let $\psi : \mathrm{Gr}(\tau') \to G''$ be this isomorphism. We extend it such that $\psi(\{n\}) = \{\psi(n)\}$ and $\psi(\emptyset) = \emptyset$. Then:

Lemma 3.33 $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(\delta.p) = \psi(\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p))$.

Proof: Because of the isomorphism ψ, for every n node of $\mathrm{Gr}(\tau')$, for every $p \in \mathit{Path}$, $\psi(\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p,n)) = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p,\psi(n))$. Therefore $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(\delta.p) = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(\delta.p,n_0) = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p,n_1)$ and $\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p) = \mathrm{NofP}_{\mathrm{Gr}(\tau')}(p,\psi(n_1))$. This gives the result. ☐

$$\begin{aligned}\mathrm{Constr}_T(\delta,\tau)(c) &= \cup\{\mathrm{NofP}_{\mathrm{Gr}(\tau)}(\delta.p) \mid \mathrm{NofP}_{\mathrm{Gr}(\tau')}(p) \subseteq c\} \\ &= \cup\{\psi(\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p)) \mid \mathrm{NofP}_{\mathrm{Gr}(\tau')}(p) \subseteq c\}\end{aligned}$$

Hence,

$$\mathrm{Constr}_T(\delta,\tau)(c) = \{\psi(n) \mid n \in c\}.$$

Example 3.34 Let us consider the example of Figure 3.11:

$$\mathrm{Constr}_T(\mathtt{fst}, (\mathtt{int} * \mathtt{int}) * \mathtt{bool})(\underline{\mathtt{int}} * \underline{\mathtt{int}}) = (\underline{\mathtt{int}} * \underline{\mathtt{int}}) * \mathtt{bool}.$$

In the case of the recursive type `t` of Figure 3.10, `t` and `t list` have the same graph, except that the top node is different: the top node of $\mathrm{Gr}(\mathtt{t})$ is the node labelled `t` on the left of the figure, whereas the top node of $\mathrm{Gr}(\mathtt{t\ list})$ is the node labelled `t list` (on the right). Hence, the same nodes are marked: $\mathrm{Constr}_T(\mathtt{hd}, \mathtt{t\ list})(c) = c$.

Restrictions Let $\delta : \tau \to \tau'$ be an accessor of type τ.

$$\mathrm{Restr}_T(\delta, \tau)(c) = \alpha_T^{\tau[\tau']}(\lambda c.\delta^{-1}.c)(c) = \alpha_T^{\tau'}(\delta^{-1}.\gamma_T^{\tau}(c))$$

Let n_0 be the top of $\mathrm{Gr}(\tau)$. $\mathrm{Gr}(\tau)$ contains an edge $n_0[\tau] \xrightarrow{\delta} n_1[\tau']$. As we have seen in the case of constructions, there exists an isomorphism ψ from $\mathrm{Gr}(\tau')$ to the subgraph G'' of $\mathrm{Gr}(\tau)$ reachable from n_1.

$$\begin{aligned}\mathrm{Restr}_T(\delta, \tau)(c) &= \cup\{\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p) | \mathrm{NofP}_{\mathrm{Gr}(\tau)}(\delta.p) \subseteq c\} \\ &= \cup\{\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p) | \psi(\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p)) \subseteq c\}\end{aligned}$$

by Lemma 3.33. Hence,

$$\mathrm{Restr}_T(\delta, \tau)(c) = \{\psi^{-1}(n) | n \in c \cap \mathrm{Nodes}(G'')\}.$$

Intuitively, $\mathrm{Restr}_T(\delta, \tau)(c)$ is the context corresponding to nodes marked by c which are in $\mathrm{Gr}(\tau')$.

Other monomorphic cases Other context transformers except generalization and instantiation are easy to compute. Their values are summarized in Figure 3.15, page 66.

Generalization Let σ be a given substitution on type variables. The generalization is defined by:

$$G_T^{\tau} = \alpha_T^{\sigma\tau[\tau]}(\mathrm{Id}) = \alpha_T^{\tau} \circ \gamma_T^{\sigma\tau} = \lambda c. \cup \{\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) | p \in \mathit{Path}(\sigma\tau), \mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) \subseteq c\}.$$

It is not easy to give a more explicit formula, but the G_T^{τ} function can be computed by the following algorithm:

Assume $c \in \mathit{Ctx}_T^{\sigma\tau}$, n node of $\mathrm{Gr}(\sigma\tau)$, n' node of $\mathrm{Gr}(\tau)$.
search(n, n') =
  if the current call comes from a call to search(n, n') **then return**
  if $n \in c$ **then** $n' \in G_T^{\tau}(c)$
  if n' is labelled by a type variable **then**
    foreach δ, for $n \xrightarrow{\delta} n_1$ in $\mathrm{Gr}(\sigma\tau)$, search(n_1, n')
  else
    foreach δ, for $n \xrightarrow{\delta} n_1$ in $\mathrm{Gr}(\sigma\tau)$, for $n' \xrightarrow{\delta} n'_1$ in $\mathrm{Gr}(\tau)$, search(n_1, n'_1)
search(n_0, n'_0) where n_0 is the top node of $\mathrm{Gr}(\sigma\tau)$, n'_0 the top node of $\mathrm{Gr}(\tau)$.

If we call search(n, n') then there exists a path p such that $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) = \{n\}$ and $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) = \{n'\}$. Conversely, for all paths p such that $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) = \{n\}$ and $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) = \{n'\}$, we call search$(n, n')$ (by induction on the length of the path). Then it is easy to see that our algorithm computes $G_T^{\tau}(c)$.

Lemma 3.35 *If two paths p and p' lead to the same node in $\mathrm{Gr}(\tau)$ (i.e. $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p')$), and they do not go inside a type variable substituted by σ (i.e. $\exists n, \tau_1$, such that $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) = \{n[\sigma\tau_1]\}$, and idem for p') then they also lead to the same node in $\mathrm{Gr}(\sigma\tau)$.*

This lemma is proved in Appendix B.2.

Then, in the above algorithm, if the call to search(n, n') comes from a call to search(x, n') for any x, and n is not inside a type variable, then $x = n$ and the call comes in fact from a call to search(n, n'). The algorithm can then be rewritten into:

search(n, n') =
  if the current call comes from a call to search$(_, n')$ **then return**
  if $n \in c$ **then** $n' \in G_T^\tau(c)$
  if n' is labelled by a type variable **then**
    if there exists a node n_1 reachable from n in $\mathrm{Gr}(\sigma\tau)$ such that $n_1 \in c$ **then** $n' \in G_T^\tau(c)$
  else
    foreach δ, for $n \xrightarrow{\delta} n_1$ in $\mathrm{Gr}(\sigma\tau)$, for $n' \xrightarrow{\delta} n'_1$ in $\mathrm{Gr}(\tau)$, search(n_1, n'_1)
search(n_0, n'_0) where n_0 is the top node of $\mathrm{Gr}(\sigma\tau)$, n'_0 the top node of $\mathrm{Gr}(\tau)$.

The loop test "**if** the current call comes from a call to search$(_, n')$" can be implemented efficiently by marking nodes n' which appear in the second parameter of search.

Instantiation Let σ be a given substitution on type variables. The expected formula of instantiation is $I_T^{\tau[\tau']} = \alpha_T^{\sigma\tau[\sigma\tau']} \circ I_E^{\tau[\tau']} \circ \gamma_T^{\tau[\tau']}$ and the direct concretization we find for functions is (Proposition 3.9)

$$\gamma_T^{\tau[\tau']}(\phi_T) = \gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^\tau.$$

We obtain

$$I_T^{\tau[\tau']}(\phi_T)(c) = \alpha_T^{\sigma\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^\tau \circ \gamma_T^{\sigma\tau})(c) \cap \mathit{Path}(\sigma\tau')).$$

However, this instantiation function is not very precise. Indeed, in the resulting context $I_T^{\tau[\tau']}(\phi_T)(c)$, the graph of $\sigma\beta$ (for each type variable β) is either entirely marked, or not marked at all. For instance, consider the identity function $f = \texttt{fun}\ y \to y$ of type $\beta \to \beta$. The context transformer is $\phi_T = \rho_T[\![f]\!]1 = \lambda c.c$. Then

$$I_T^{\beta[\beta]}(\phi_T)(c) = \alpha_T^{\sigma\beta}((\gamma_T^\beta \circ \alpha_T^\beta \circ \gamma_T^{\sigma\beta})(c) \cap \mathit{Path}(\sigma\beta)).$$

As soon as $c \neq \emptyset$, $\gamma_T^{\sigma\beta}$ contains a path, and $(\alpha_T^\beta \circ \gamma_T^{\sigma\beta})(c) = \{n[\beta]\}$. Therefore, $(\gamma_T^\beta \circ \alpha_T^\beta \circ \gamma_T^{\sigma\beta})(c) = \mathit{Path}$, and $I_T^{\beta[\beta]}(\phi_T)(c)$ contains all nodes of $\mathrm{Gr}(\sigma\beta)$. However, we can see that $\lambda c.c$ is also a correct context transformer for the identity function of type $\sigma\beta \to \sigma\beta$, and it yields much more precise results.

Therefore, we are going to define a more precise instantiation function. Notice that, once the abstraction is defined, there is a single corresponding concretization which is (Proposition 3.8):

$$\gamma_T^{\tau[\tau']}(\phi_T) = \sqcup\{\phi_E \in S | \alpha_T^{\tau[\tau']}(\phi_E) \leq \phi_T\}$$

where S is the set of concrete functions ϕ_E. Therefore, if we manage to reduce the set of possible ϕ_E, we can have a more precise concretization, and thus a more precise instantiation. Our goal in the following lemmas is to reduce the set S, infer an explicit formula for the concretization, and finally find a precise instantiation. This restricted set S does not contain all context transformers of analysis E, but it contains all context transformers $\rho[\![y]\!]i$ that may be parameters of instantiation. This is enough for our goal. The proofs of the following results can be found in Appendix B.3.

Definition 3.36 Let p be a path. If $\alpha_T^\tau(p) = \{n\}$ is labelled by a type variable, then p is of the form $p = p_0.p'$ where p_0 is the shortest prefix of p such that $\alpha_T^\tau(p_0) = \{n\}$. We define $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = p'$ (remaining path of p). If $\alpha_T^\tau(p)$ is not labelled by a type variable, we define $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = NV$ (NV is a special value which means "No Variable" to distinguish this situation from the previous one).

Let G be the graph of τ, n_0 its top node. Another definition of $\mathrm{RemPath}_G(p)$ is

$$\mathrm{RemPath}_G(n, \epsilon) = NV \text{ if } n \text{ is not labelled by a type variable;}$$
$$\mathrm{RemPath}_G(n, p) = p \text{ if } n \text{ is labelled by a type variable;}$$
$$\mathrm{RemPath}_G(n, \delta.c) = \mathrm{RemPath}_G(n', c) \text{ if there is an edge } n \xrightarrow{\delta} n' \text{ in } G;$$
$$\mathrm{RemPath}_G(n, \delta.c) = NV \text{ otherwise.}$$
$$\mathrm{RemPath}_G(p) = \mathrm{RemPath}_G(n_0, p).$$

If $c \in Ctx_E$, we define $\mathrm{Ins}_{p'}^\tau(c) = \{p.p' | p \in c, \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon\}$ (insertion of path p' in context c) if $p' \in Path$. $\mathrm{Ins}_{NV}^\tau(c) = c$.

Lemma 3.37 *Assume that $\phi_E : \tau[\tau']$. We say that $\phi_E \in Tr'_E$ if ϕ_E satisfies the property:*

If $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon$, let $c = \phi_E(p) - \phi_E(\emptyset)$. Then $\forall p'' \in c, \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p'') = \epsilon$ and $\forall p' \in Path, \phi_E(p.p') = \mathrm{Ins}_{p'}^{\tau'}(c) \cup \phi_E(\emptyset)$.

In the monomorphic case, all context transformers ϕ_E are in Tr'_E. In the polymorphic case, we only have $E[\![M]\!]\rho x \in Tr'_E$ if $x \in Ind$ or $x \in Var$ bound by a `fun` expression (it is wrong in general if x is bound by a `let` or `let rec` expression) and $\rho[\![y]\!]i \in Tr'_E$ if $i \in Ind$.

The fundamental idea behind this lemma is that expressions cannot manipulate data inside values whose types are type variables. Assume that p represents a value whose type is a type variable (i.e. $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon$), and that the corresponding image is $\phi_E(p)$. If we consider the simplified situation without constant context transformers $\lambda c.\underline{\tau}$ (i.e. $\phi_E(\emptyset) = \emptyset$), we would have $\phi_E(p.p') = \phi_E(p).p'$: ϕ_E can only transmit the path p' without changing it. The above lemma is a rigorous formulation of this result, taking also into account constant context transformers.

Definition 3.38

$$\gamma_T^{\tau[\tau']}(\phi_T) = \lambda c. \cup \{\mathrm{Ins}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^\tau)(p) - \gamma_T^{\tau'}(\phi_T(\tau))) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau))$$

The instantiation function for predicate transformers is then:

$$I_T^{\tau[\tau']} = \alpha_T^{\sigma\tau[\sigma\tau']} \circ I_E^{\tau[\tau']} \circ \gamma_T^{\tau[\tau']}.$$

We extend the definition of α_T^τ to all contexts $c \in Ctx_E$ by the same formula $\alpha_T^\tau(c) = \cup_{p \in c} \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p)$. This avoids using always the intersection $c \cap Path(\sigma\tau')$ before applying $\alpha_T^{\sigma\tau'}$. Paths that are not in $Path(\sigma\tau')$ always give an empty result with this definition of $\alpha_T^{\sigma\tau'}$. Then

$$I_T^{\tau[\tau']}(\phi_T)(c) = \alpha_T^{\sigma\tau'}(\cup \{\mathrm{Ins}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}^{\tau'}(\gamma_T^{\tau'}(\phi_T \circ \alpha_T^\tau(p)) - \gamma_T^{\tau'}(\phi_T(\tau))) | p \in \gamma_T^{\sigma\tau}(c)\}$$
$$\cup\, \gamma_T^{\tau'}(\phi_T(\tau))).$$

We also define the instantiation function for contexts by: $I_T^\tau = \alpha_T^{\sigma\tau} \circ \gamma_T^\tau$.

The instantiation function for predicate transformers is a complicated formula. We will give an intuition for it when we have a little more information about its behavior. We need to prove a few lemmas before coming to the correctness of instantiation itself. These lemmas also give properties interesting for themselves.

Proposition 3.39 *The context transformers $\phi : Ctx_T^{\tau} \to Ctx_T^{\tau'}$ for analysis T are additive, that is: $\phi(\cup_i c_i) = \cup_i \phi(c_i)$. They can be put into the form:*

$$\lambda c. \cup \{i_s | s \in c\} \cup i_0 = \lambda c. \cup \{\phi(s) | s \in c\} \cup \phi(\emptyset)$$

where $i_s, i_0 \in Ctx_T^{\tau'}$.

Additivity is a very important property of our escape analysis, since it enables us to find the image of each node of a context independently, thus leading to a much faster implementation.

Definition 3.40 The set of types labelling nodes of a context c is

$$\mathrm{Types}(c) = \{\tau | n[\tau] \in c\}.$$

These types are said to be the types marked in context c.

The following lemma shows that generalization and instantiation behave as expected with respect to types.

Lemma 3.41 $\mathrm{Types}(G_T^{\tau}(c)) = \{\tau' | \sigma\tau' \in \mathrm{Types}(c)\} \cup \{\beta$ *type variable* $| \exists \tau' \in \mathrm{Types}(c), \tau'$ *is the label of a node of* $\mathrm{Gr}(\sigma\beta)\}$.

Similarly for instantiation, $\mathrm{Types}(I_T^{\tau}(c)) = \{\sigma\tau' | \tau' \in \mathrm{Types}(c)\} \cup \{$ *types labelling* $\mathrm{Gr}(\sigma\beta) |$ $\beta \in \mathrm{Types}(c), \beta$ *type variable* $\}$.

Proposition 3.42 *If the language is monomorphic,*

$$\text{if } c \text{ does not contain a node labelled by a function type, } \mathrm{Types}(\phi(c) - \phi(\tau)) \subseteq \mathrm{Types}(c). \tag{3.1}$$

In the case of a polymorphic language, (3.1) *is only true if $\phi = T[\![M]\!]\rho x$ with $x \in Ind$ or x has been bound in by a* `fun` *expression, or $\phi = \rho[\![y]\!]i$ with $i \in Ind$. In particular, this is true for all functions ϕ that may be the parameters of $I_T^{\tau[\tau']}$. It is not always true for $\phi = T[\![M]\!]\rho x$ if x has been bound by a* `let` *or* `let rec` *expression.*

This property means that if a value of type τ escapes, i.e. it is in the context $\phi(c)$, then this value also appears with the same type in the context c, except in two cases:

- constants which are subtracted by $-\phi(\tau)$ (they essentially correspond to values stored in assignments);
- tests "if $c \geq \tau_y \underline{\to} \tau$ then $\underline{\tau'}$ else $\emptyset$" which have no contribution when c does not contain a node labelled by a function type (they correspond to values which appear in environments of closures, and whose type is hidden in the function type).

In the case of a polymorphic language, the difficulty comes from the fact that (3.1) is not true for G_T^τ (the generalization may replace a type with a type variable).

This property is really expected for functions which are instantiated by $I_T^{\tau[\tau']}$. Indeed, with $p' = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$, we have $p = p''.p'$ and p'' corresponds to a node n labelled by a type variable β in τ. During the computation of $\gamma^{\tau[\tau']}(\phi_T)$, $\mathrm{Ins}_{p'}^{\tau'}$ concatenates p' to paths which correspond to nodes n' in the image of ϕ_T. The same path p' is used to represent both a part of a value corresponding to $n[\beta]$ in the input of ϕ_T and a part of a value corresponding to n' in the output of ϕ_T. For the definition of instantiation to be really meaningful, the nodes n and n' must therefore be labelled by the same type β, and be instantiated to the same type $\sigma\beta$.

Let us assume that a value v_2 of type τ_2 is inside a value v_1 of type τ_1 (i.e. the graph of the type contains an edge $n_1[\tau_1] \to n_2[\tau_2]$). Then, if v_1 is reachable, so is v_2. This implies that if a node n of the graph is marked, all nodes reachable from n are also marked. This means that contexts are downwards closed.

Definition 3.43 A context $c \in Ctx_T^\tau$ is ***closed*** if and only if

$$\forall n \in c, \forall n' \text{ reachable from } n \text{ in the graph of } \tau, n' \in c.$$

Proposition 3.44 (Relation between closure for T and closure for E) *If $c \in Ctx_T^\tau$ is closed, $\gamma_T^\tau(c) \in Ctx_E^\tau$ is closed. Conversely, if $c \in Ctx_E^\tau$ is closed, $\alpha_T^\tau(c) \in Ctx_T^\tau$ is also closed.*

Proposition 3.45 (Closure) *If $c \in Ctx_T^{\tau_M}$ is closed, then $\forall x \in Var \cup Ind, T[\![M]\!]\rho x c$ is also closed.*

This implies that if a node of a strongly connected component of the graph of the type is marked, then the entire strongly connected component is also marked. Then a single node can be used to represent it. This will be done in the implementation, but we do not consider this optimization in this theoretical study. The fact that all useful contexts are closed is also a key point for other optimizations, and it is the reason why the approximations using integer contexts described in the following sections yield precise results (these approximations can only represent closed contexts).

We can now prove the main theorem, which gives the correctness of our instantiation:

Theorem 3.46 *For context transformers corresponding to $E[\![M]\!]\rho x$, $x \in Var$ bound by* `fun` *or $x \in Ind$, and for context transformers $\rho[\![y]\!]i$, $i \in Ind$ (and in particular for context transformers that may be parameters of $I_T^{\tau[\tau']}$), $\gamma_T^{\tau[\tau']}$ is a correct concretization:*

$$\gamma_T^{\tau[\tau']}(\phi_T) = \sqcup\{\phi_E \in Tr'_E | \alpha_T^{\tau[\tau']}(\phi_E) \leq \phi_T\}.$$

Proposition 3.47

$$I_T^{\tau[\tau']}(\phi_T)(\sigma\tau) = I_T^{\tau'}(\phi_T(\tau)), \quad I_T^{\tau[\tau']}(\phi_T)(\underline{\sigma\tau}) = I_T^{\tau'}(\phi_T(\underline{\tau})), \quad I_T^{\tau[\tau']}(\phi_T) \leq I_T^{\tau'} \circ \phi_T \circ G_T^\tau.$$

If we want to compute $I_T^{\tau[\tau']}(\phi_T)(c)$, we use the following procedure. Let $n \in c$. Let $p \in \gamma_T^{\sigma\tau}(\{n\})$, and $\{n_1\} = \alpha_T^\tau(p)$. Therefore $n_1 \in G_T^\tau(\{n\})$. The contribution of path p in $I_T^{\tau[\tau']}(\phi_T)(c)$ is

$$i(p) = \alpha_T^{\sigma\tau'}(\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau'}(\phi_T(n_1) - \phi_T(\tau))))$$

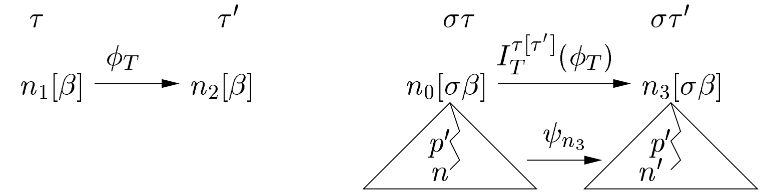


Figure 3.12: Algorithm for computing the instantiation

and $I_T^{\tau[\tau']}(\phi_T)(c) = \cup\{i(p) | p \in \gamma_T^{\sigma\tau}(c)\} \cup I_T^{\tau'}(\phi_T(\tau)))$. We explain below how to compute $i(p)$ from n and n_1.

If n_1 is not labelled by a type variable, then $\mathrm{RemPathGr}(\tau)(p) = NV$, and $i(p) = I_T^{\tau'}(\phi_T(\{n_1\}) - \phi_T(\tau))$. We have seen the algorithm to compute G_T^{τ}. A similar algorithm computes $I_T^{\tau'}$. We can therefore compute $i(p) = I_T^{\tau'}(\phi_T(\{n_1\}) - \phi_T(\tau))$.

To have a more intuitive situation, we can assume that no node $n_1 \in G_T^{\tau}(\{n\})$ is labelled by a type variable. In this case, $\mathrm{RemPathGr}(\tau)(p) = NV$ for all paths p, and $I_T^{\tau[\tau']}(\phi_T)(\{n\}) = (I_T^{\tau'} \circ \phi_T \circ G_T^{\tau})(\{n\})$. We first convert the context $\{n\}$ to type τ, take the image by ϕ_T and convert back the result to type $\sigma\tau'$. In this case, the precision of the instantiation is the same as with the naive concretization.

Assume now that n_1 is labelled by a type variable β. This case is represented in Figure 3.12. The nodes $n_2 \in N_2 = \phi_T(n_1) - \phi_T(\emptyset)$ are labelled β (because ϕ_T satisfies (3.1)). Let N_3 be the set of nodes of $I_T^{\tau'}(N_2)$ that are labelled by $\sigma\beta$. Then $I_T^{\tau'}(N_2)$ is the set of nodes reachable from the nodes of N_3. Let $p' = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$ and p'' such that $p = p''.p'$. Let $\{n_0[\sigma\beta]\} = \mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p'')$. For each $n_3 \in N_3$, the graphs reachable from n_3 and n_0 are isomorphic (uniqueness of the graph of $\sigma\beta$), let ψ_{n_3} be the isomorphism. Let us show that $i(p) = \{\psi_{n_3}(n) | n_3 \in N_3\}$. Indeed, the path p' goes from n_0 to n in $\mathrm{Gr}(\sigma\tau)$ (since $\{n_0\} = \mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p'')$ and $\{n\} = \mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p''.p')$). Therefore p' also goes from $n_3 = \psi_{n_3}(n_0)$ to $n' = \psi_{n_3}(n)$ in $\mathrm{Gr}(\sigma\tau')$ by the isomorphism ψ_{n_3}. We have $\alpha_T^{\sigma\tau'}(\{p_2 \in \gamma_T^{\tau'}(N_2) | \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p_2) = \epsilon\}) = N_3 = \{\psi_{n_3}(n_0) | n_3 \in N_3\}$ therefore $i(p) = \alpha_T^{\sigma\tau'}(\mathrm{Ins}_{p'}^{\tau'}(\gamma_T^{\tau'}(N_2))) = \alpha_T^{\sigma\tau'}(\{p_2.p' | p_2 \in \gamma_T^{\tau'}(N_2), \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p_2) = \epsilon\}) = \{\psi_{n_3}(n) | n_3 \in N_3\}$ since p' goes from n_0 to n in $\mathrm{Gr}(\sigma\tau)$. Hence, we have the result.

Intuitively, assume that $\phi_T(\emptyset) = \emptyset$, $\phi_T(\{n_1[\beta]\}) = \{n_2[\beta]\}$. If we map type τ to $\sigma\tau$, n_1 corresponds to $n_0[\sigma\beta]$, and if we map type τ' to $\sigma\tau'$, n_2 corresponds to $n_3[\sigma\beta]$. If only a part of the graph of $\sigma\beta$ reachable from n_0 is marked in context c, then the *same part* of the graph of $\sigma\beta$ reachable from n_3 is marked. The "same part" is formalized by the isomorphism ψ_{n_3}. In this case, we have much improved precision with respect to the naive concretization (it would have marked all the nodes of the graph reachable from n_3).

Applying this algorithm for each node $n \in c$, and for each $n_1 \in G_T^{\tau}(\{n\})$, we compute $i(p)$ for each $p \in \gamma_T^{\sigma\tau}(c)$. We take the union of all these $i(p)$ and add $I_T^{\tau[\tau']}(\phi_T)(\sigma\tau) = I_T^{\tau'}(\phi_T(\tau))$ to get the instantiation $I_T^{\tau[\tau']}(\phi_T)(c)$.

Example 3.48 Let us consider again the `map` function of Example 3.14. Let us assume that `map` has type $(\tau_1 \to \tau_2) \to \tau_1$ `list` $\to \tau_2$ `list`. We obtain the following result thanks to a

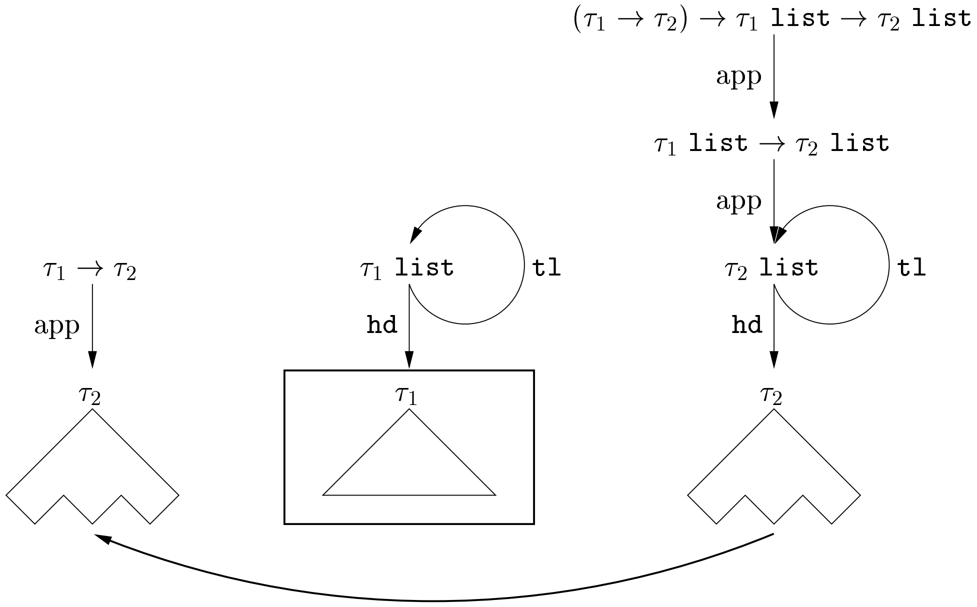


Figure 3.13: Representation of the context transformers for the `map` function. The box means that the elements of the list, of type τ_1, always escape. The arrow from τ_2 on the right to τ_2 on the left means that if a node of the graph of τ_2 is marked on the right, the corresponding node is marked on the left.

computation similar to analysis E:

$$\begin{aligned}\rho_T[\![\texttt{map}]\!] = \{&\texttt{map} \mapsto \lambda c.c, 1 \mapsto \mathrm{Constr}_T(\mathrm{app}, \tau_1 \to \tau_2) \circ \mathrm{Restr}_T(\texttt{hd}, \tau_2\ \texttt{list}) \\ &\circ \mathrm{Restr}_T(\mathrm{app}, \tau_1\ \texttt{list} \to \tau_2\ \texttt{list}) \circ \mathrm{Restr}_T(\mathrm{app}, (\tau_1 \to \tau_2) \to \tau_1\ \texttt{list} \to \tau_2\ \texttt{list}), \\ &2 \mapsto \lambda c.\underline{\tau_1}\ \texttt{list}\}.\end{aligned}$$

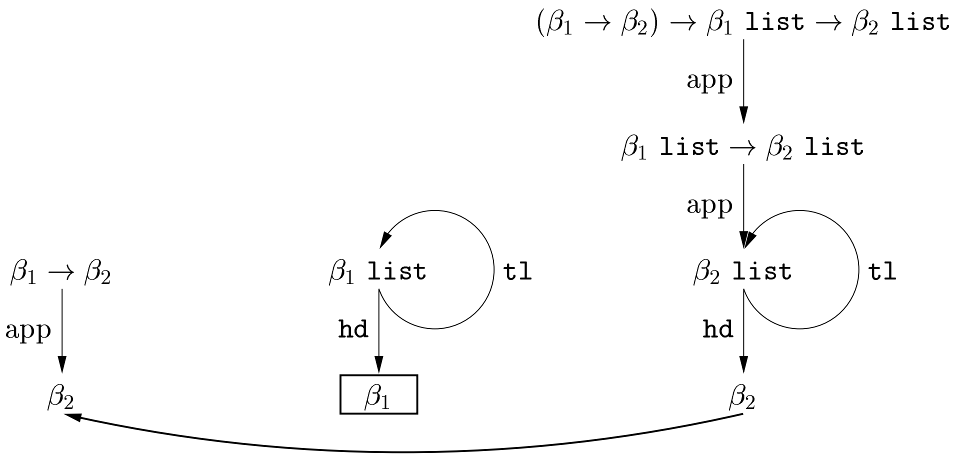
We can notice that $\mathrm{Restr}_T(\texttt{tl}, \tau\ \texttt{list}) = \mathrm{Id}$ and $\mathrm{Constr}_T(\texttt{tl}, \tau\ \texttt{list}) = \mathrm{Id}$. Therefore, the $(\texttt{tl}^{-1})^*$ and $\texttt{tl}^*$ of analysis E disappear in analysis T. This way, the contexts become finite. We can give a graphical representation of this escape information (Figure 3.13). The closure of parameter 1 never escapes. The result of parameter 1 escapes if and only if the elements of the result of `map` escape. The elements of the second parameter always escape because they are given as parameters to an unknown function.

The escape information can also be summarized as follows

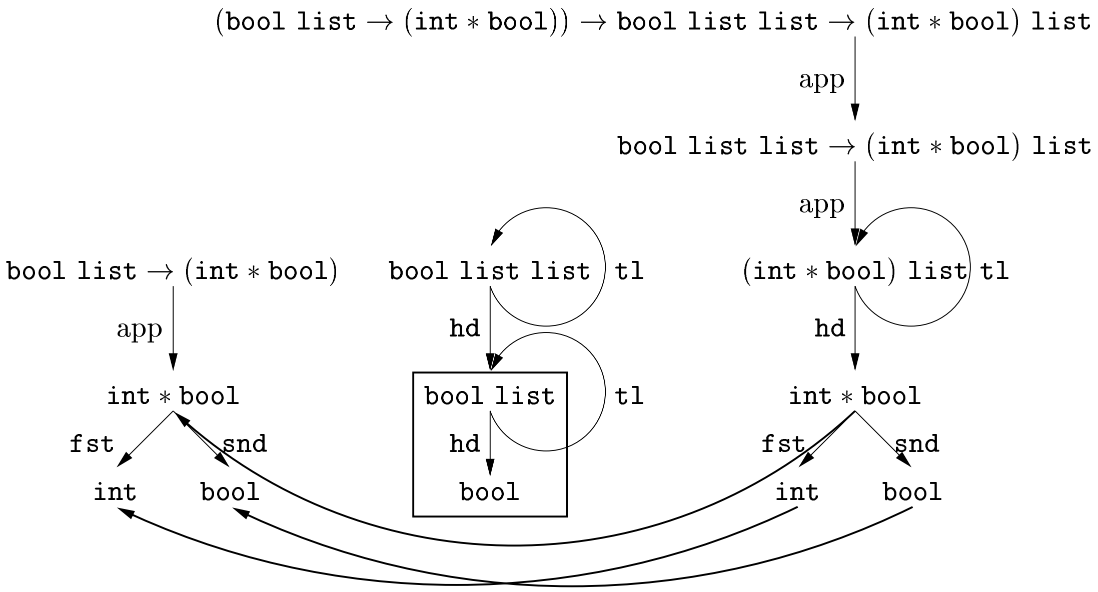
$$\texttt{map} : (\tau_1 \to \tau_2) \to \underline{\tau_1}\ \texttt{list} \to \tau_2\ \texttt{list}$$
$$\texttt{map} : (\tau_1 \to \underline{\tau_2}) \to \underline{\tau_1}\ \texttt{list} \to \underline{\tau_2\ \texttt{list}}$$

In the first line, the underlined parts correspond to the case when the result of `map` does not escape at all. In the second line, the result of `map` escapes entirely. This symbolic representation does not represent what happens when only part of the result escapes.

The context transformers for the polymorphic version of $\texttt{map} : \tau = (\beta_1 \to \beta_2) \to \beta_1\ \texttt{list} \to \beta_2\ \texttt{list}$ are displayed in Figure 3.14, together with the result of instantiating these context transformers to type $\sigma\tau = (\texttt{bool list} \to (\texttt{int} * \texttt{bool})) \to \texttt{bool list list} \to (\texttt{int} * \texttt{bool})\texttt{list}$.



Graph for the `map` function yielded by the analysis of the polymorphic code.



Graph yielded by instantiation of the above graph

Figure 3.14: Instantiation of escape information for the `map` function. The conventions of representation are the same as in Figure 3.13.

Contexts:	$Ctx_T^\tau = \mathcal{P}(\text{Nodes}(\text{Gr}(\tau)))$
Context transformers:	$Tr_T^{\tau[\tau']} = Ctx_T^\tau \xrightarrow{\text{m}} Ctx_T^{\tau'}$
Identity:	$\text{Id}_T^\tau = \lambda c.c$
Greatest element:	$\top_{Tr_T}^\tau = \lambda c.\underline{\tau}$
Least element:	$\perp_{Tr_T}^\tau = \lambda c.\emptyset$
Construction:	$\text{Constr}_T(\delta, \tau) = \lambda c.\{\psi(n) \mid n \in c\}$
Restriction:	$\text{Restr}_T(\delta, \tau) = \lambda c.\{\psi^{-1}(n) \mid n \in c \cap G''\}$

where $\text{Gr}(\tau)$ contains an edge $n_0[\tau] \xrightarrow{\delta} n_1[\tau']$, G'' is the subgraph of $\text{Gr}(\tau)$ reachable from n_1, ψ is the isomorphism from $\text{Gr}(\tau')$ to G''.

Step:	$\text{test}_T^{\tau_y \to \tau, \tau'} = \lambda c.\text{if } c \geq \underline{\tau_y \to \tau} \text{ then } \underline{\tau'} \text{ else } \emptyset$
Upper bound:	$\sqcup_T(\phi_i)_i = \lambda c. \cup_i \phi_i(c)$
Composition:	$\phi \circ_T \psi = \lambda c.\phi(\psi(c))$
Generalization:	$G_T^{\sigma,\tau} = \alpha_T^\tau \circ \gamma_T^{\sigma\tau}$
Instantiation:	

$$I_T^{\sigma,\tau[\tau']}(\phi) = \lambda c.\alpha_T^{\sigma\tau'}(\cup\{\text{Ins}_{\text{RemPath}_{\text{Gr}(\tau)}(p)}^{\tau'}(\gamma_T^{\tau'}(\phi \circ \alpha_T^\tau(p)) - \gamma_T^{\tau'}(\phi(\tau))) \mid p \in \gamma_T^{\sigma\tau}(c)\} \cup \gamma_T^{\tau'}(\phi(\tau)))$$

Figure 3.15: Lattice of context transformers for analysis T

Let us consider for example the context c containing the `bool` node of the graph of the type $\sigma\tau$ (instantiated type of `map`), and compute $I_T^{\tau[\tau']}(\rho[\![\texttt{map}]\!]1)(c)$ where $\tau' = \beta_1 \to \beta_2$. We have $\gamma_T^{\sigma\tau}(c) = \{\text{app.app.}\texttt{hd}.\texttt{snd}\}$. Let $p = \text{app.app.}\texttt{hd}.\texttt{snd}$. $p' = \text{RemPath}_{\text{Gr}(\tau)}(p) = \texttt{snd}$. $\gamma_T^{\tau'} \circ (\rho[\![\texttt{map}]\!]1) \circ \alpha_T^\tau(p) = \text{app.}Path$ (the analysis of the general version of `map` indicates that the type variable β_2 of the first parameter escapes if the elements of the resulting list escape). $\text{Ins}_{p'}^{\tau'}(\text{app.}Path) = \{\text{app.}p'\}$ concatenates the path p' to the path needed to reach the type variable, hence $I_T^{\tau[\tau']}(\rho[\![\texttt{map}]\!]1)(c) = \alpha_T^{\sigma\tau'}(\{\text{app.}\texttt{snd}\})$ which corresponds to the `bool` node of the first parameter of `map`.

The computation of the images of other nodes is similar.

3.5.3 Correctness

The context transformers of analysis T are summarized in Figure 3.15.

Theorem 3.49 (Correctness of analysis T) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let n_0 be the node at the top of the graph $\text{Gr}(\tau_x^{\text{g}})$. Let us assume that $n_0 \notin T[\![N]\!]\rho\, x\, \underline{\tau_N}$.*
- `let rec` $x =$ `fun` $y \to M$ `in` N. *In this case, let ℓ be the location of the closure x. Let n_0 be the node at the top of the graph $\text{Gr}(\tau_x^{\text{g}})$. We assume $n_0 \notin R(x)\underline{\tau_N}$ where R is defined in the* `let rec` *case of Figure 3.6.*

In these two cases, ℓ can be stack allocated, and deallocated at the end of N.

Proof: By the correctness of analysis E (Theorem 3.18) and the properties of the abstraction, we can apply the general correctness proposition (Proposition 3.22). For the instantiation case, notice that, for all ϕ_E and ϕ_T that may be parameters of instantiation, if $\phi_T \geq \alpha_T^{\tau[\tau']}(\phi_E)$ then $I_T^{\tau[\tau']}(\phi_T) \geq \alpha_T^{\sigma\tau[\sigma\tau']}(\lambda c.\phi_E(c) \cap Path(\sigma\tau'))$. Indeed,

$$\alpha_T^{\sigma\tau[\sigma\tau']} \circ \gamma_T^{\tau[\tau']}(\phi_T) \geq \alpha_T^{\sigma\tau[\sigma\tau']}(\phi_E)$$

by the semi-dual Galois connection property for $(\alpha_T^{\tau[\tau']}, \gamma_T^{\tau[\tau']})$.

This proposition yields the expected result, noticing that $\alpha_T^{\tau_{\tau x}^{g}}(\{\epsilon\}) = \{n_0\}$. □

3.5.4 Properties of special kinds of types

Non recursive monomorphic types The graph of a type is in fact a tree, and there is a bijective mapping between allowed (i.e. not forbidden) paths corresponding to a value and the nodes of the graph of its type. Analyses E and T therefore give the same result:

$$\gamma_T^\tau \circ \alpha_T^\tau = \mathrm{Id}_{Ctx_E^\tau}.$$

Monomorphic recursive types For recursive types, the graph of a type is in general cyclic. Therefore, analysis T is less precise than analysis E. For example, if we consider a list, all paths `tl`* refer to the same node, the top node of the graph of the list type. Then, we cannot distinguish the different cons locations of the list. Similarly, we do not distinguish the different elements of the list. This loss of precision is necessary to have a finite analysis. For monomorphic recursive types, the graph is always finite, since there is a finite number of types in the program, and any path in the graph cannot contain several times the same type (the length of a path is bounded, the out degree of the nodes is bounded and all nodes are reachable from a single node, then the graph is finite).

Polymorphic recursive types The graph of a type is always finite, except for ***non-monotone*** recursive types: in the definition of a polymorphic type constructor, we can call this constructor with different parameters, for example:

```
type β t = None | Some of β * β list t
```

The cycle check is therefore not enough, as we use infinitely many different types β `list` ... `list t`. Our analysis cannot handle this case precisely if we want it to remain finite. For a value of a non-monotone recursive type, we only consider two contexts: either the value escapes (entirely or part of it), or it does not escape at all. However, non-monotone recursive types have no practical use, since no iterator can be written for them without polymorphic recursion (and there is no polymorphic recursion in Objective Caml). In our implementation, the case when the definition of `t` uses several but a finite number of instances of `t` is handled in the same way as the infinite case for simplicity. We could do better here, but these types are very rarely used. For the theoretical study of analysis T, we can handle non-monotone recursive types by considering an infinite graph. In the following, we will exclude non monotone recursive types, and assume **that all graphs of types are finite**.

Analysis T is not polymorphically invariant with recursive types. If we consider for example type `t` of Figure 3.10:

$$\begin{array}{rcl} \texttt{type t} & = & \texttt{Add of t list} \\ & | & \texttt{Cst of int} \\ & | & \texttt{Var of string} \end{array}$$

Consider the function `fun x -> let y = [x] in hd y` of type $\beta \to \beta$. The list constructor does not escape from it: $T[\![\texttt{hd y}]\!]\rho y \underline{\beta} = \text{Constr}_T(\texttt{hd}, \beta\ \texttt{list})(\underline{\beta}) = \underline{\beta}\ \texttt{list}$. But if we analyze the same function instantiated to type $\texttt{t} \to \texttt{t}$, then $T[\![\texttt{hd y}]\!]\rho y \underline{\texttt{t}} = \text{Constr}_T(\texttt{hd}, \texttt{t list})(\underline{\texttt{t}}) = \underline{\texttt{t list}}$ because `t` and `t list` are in the same strongly connected component, and therefore `t list` is underlined in $\underline{\texttt{t}}$. Then we can no longer detect that the list constructor does not escape.

Another example is the following:

$$\begin{array}{rcl} \texttt{type}\ \beta\ \texttt{t}_1 & = & \beta * \beta \\ \texttt{type}\ \texttt{t}_2 & = & C\ \texttt{of}\ \texttt{t}_2\ \texttt{t}_1 | C'\ \texttt{of int} \end{array}$$

If we consider `f = fun x -> let y = (x,C' 1) in snd y`, of type $\beta \to \texttt{t}_2$, $\rho[\![f]\!]1c = \emptyset$: the first parameter does not escape. This remains true when instantiating this analysis with $I_T^{\texttt{t}_2[\beta]}(\rho[\![f]\!]1)$ for any substitution σ. However, if we analyze `f` instantiated to type $\texttt{t}_2 \to \texttt{t}_2$, then $\rho[\![f]\!]1c = (\text{Restr}_T(\texttt{fst}, \texttt{t}_2\ \texttt{t}_1) \circ \text{Constr}_T(\texttt{snd}, \texttt{t}_2\ \texttt{t}_1))(c)$. If $\texttt{t}_2$ is underlined in c so is $\texttt{t}_2\ \texttt{t}_1$, and $\texttt{t}_2$ is also underlined in $\rho[\![f]\!]1c$. The first parameter escapes if the result escapes.

In these two examples, directly analyzing the instantiated expression is less precise than instantiating the analysis of a more general expression. The loss of precision comes from the fact that types which are in different strongly connected components in the general type may be in the same component in the instantiated type, and therefore, if one these types is marked in the instantiated type, all of them are marked (since contexts are closed).

3.6 Analyses based on integer contexts

3.6.1 Analysis L (closures represented by ∞)

The preceding analysis is precise, but its complexity is not very satisfying. It is therefore interesting to design a more approximate analysis, having a more efficient implementation. We consider as contexts finite or infinite integers: $Ctx_L = \mathbb{N} \cup \{\infty\}$ ordered with the natural order.

Historically, the use of integers in similar analyses has been introduced for lists [Hug92, Deu97]. It is relatively natural to merge all elements of a list, to have an analysis of reasonable complexity. Then we obtain a linear lattice which can easily be mapped to integers: 0 means nothing escapes, 1 means elements *el* of lists may escape, 2 means *el* `list` may escape, ..., n means *el* `list` ... `list` ($n-1$ occurrences of `list`) may escape. When the only constructed types are lists, integers are equivalent to the annotated types of the previous section. Deutsch [Deu97] has suggested a generalization to cartesian products. Our contribution is to give a general definition for all types (except non-monotone recursive types, see page 67).

We assume that all graphs of types are finite. The ***height*** of the type τ is the least integer

$\top_L[\tau]$ such that

$$\top_L[\tau] \geq 1 \tag{3.2}$$
$$\top_L[\tau_1 \to \tau_2] = \infty \tag{3.3}$$
$$\top_L[\tau] \geq \top_L[\tau'] \text{ if there exists an accessor } \delta : \tau \to \tau' \tag{3.4}$$
$$\top_L[\tau] \geq \top_L[\tau'] + 1 \text{ if there exists an accessor } \delta : \tau \to \tau', \tag{3.5}$$
$$\text{and this rule does not contradict the preceding ones.}$$

The last rule is only applicable outside of a cycle of the graph of τ, since in a cycle, (3.4) implies the equality of all type heights. Rule (3.4) is necessary for the correctness of the analysis, whereas (3.5) is useful to make the analysis as precise as possible, by distinguishing as many levels as possible.

The main idea behind this definition of heights of types is to make sure that when a value v of type τ is contained in another value v' of type τ', then τ' is higher than τ. This gives immediately (3.4). This also implies (3.3) because a closure contains its free variables and the types of the free variables cannot be determined from the type of the closure: contrary to other data structures, the type of a closure does not determine the types of its components. Then a closure has to be higher than all types. The property "if v of type τ is contained in v' of type τ', then τ' is higher than τ" will enable us to use a fast solving algorithm by Knuth (see Section 4.3), which can only handle functions whose result is smaller than the parameters, and upper bounds of these functions with constants.

We obtain for example the following results (assuming that the only recursion is in the definition of lists):

$$\top_L[\texttt{int}] = \top_L[\texttt{bool}] = \top_L[\beta] = 1$$
$$\top_L[\tau_1 * \tau_2] = 1 + \max(\top_L[\tau_1], \top_L[\tau_2])$$
$$\top_L[\tau\ \texttt{list}] = 1 + \top_L[\tau]$$
$$\top_L[\tau\ \texttt{ref}] = 1 + \top_L[\tau]$$

The height of an annotated type (context for analysis T) is then the greatest height of a type marked in the considered context:

$$\alpha_L(c) = \sqcup\{\top_L[\tau] | n[\tau] \in c\}.$$

Example 3.50 If we consider for example the type `t` of Example 3.32 (see Figure 3.16), by (3.2), $\top_L[\texttt{int}] = \top_L[\texttt{string}] = 1$ (`string` is a simple type like `bool` or `int` in ML). A value of type `t` may contain a `t list` and a `t list` contains elements of type `t` (more formally, there are accessors $\overline{\texttt{Add}_1} : \texttt{t} \to \texttt{t list}$ and $\texttt{hd} : \texttt{t list} \to \texttt{t}$). Then $\top_L[\texttt{t}] = \top_L[\texttt{t list}]$ by (3.4). Therefore, we cannot apply (3.5) between `t` and `t list`. By (3.5), $\top_L[\texttt{t}] \geq 2$. Hence $\top_L[\texttt{t}] = \top_L[\texttt{t list}] = 2$.

A similar reasoning yields the heights of types for the function type represented on the left of Figure 3.16.

If the `int` or `string` types are marked in a value of type `t`, then the corresponding abstract value for analysis L will be 1. If the `t` type and the `t list` are marked (they cannot be marked independently because contexts are closed, Proposition 3.45) then the abstract value is 2.

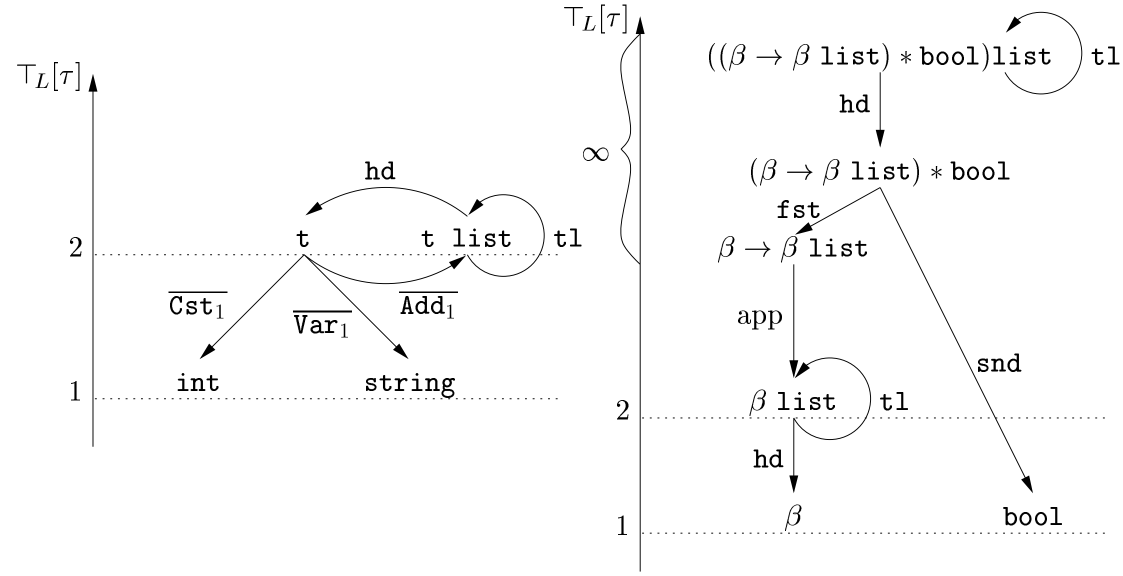


Figure 3.16: Types levels

This defines an abstraction $\alpha_L : Ctx_T \to Ctx_L$. Only closed contexts can be represented by this abstraction. This is in fact a very good property of this analysis, because all interesting contexts are closed, and therefore this analysis does not carry useless information. We lose precision "horizontally" (that is, we cannot distinguish types of same height) but not "vertically". The information for the top of data structures in still represented as precisely as in analysis T, which is very important since this is the part of data structures that can be most often stack allocated. The more we go inside complex data structures, the more we lose precision, but also the less stack allocation is probable anyway, even with a precise analysis. This gives an analysis of reasonable precision, as our benchmarks will demonstrate.

We define:

$$\top_f[\tau] = \sqcup\{\top_L[\tau'] | \tau' \text{ label of a node of } \mathrm{Gr}(\tau), \top_L[\tau'] < \infty\}.$$

The contexts useful for a type τ are, if $\top_L[\tau] < \infty$, $Ctx^\tau_L = [\![0, \top_f[\tau]]\!]$, and if $\top_L[\tau] = \infty$, $Ctx^\tau_L = [\![0, \top_f[\tau]]\!] \cup \{\infty\}$. Then we have more precisely an abstraction $\alpha_L : Ctx^\tau_T \to Ctx^\tau_L$. Ctx^τ_T and Ctx^τ_L are complete lattices, then by Proposition 3.8, the corresponding concretization is defined by:

$$\gamma^\tau_L(c') = \sqcup\{c \in Ctx^\tau_T | \alpha_L(c) \leq c'\} = \{n[\tau] \text{ node of } \mathrm{Gr}(\tau) | \top_L[\tau] \leq c'\}.$$

This gives a semi-dual Galois connection (Definition 3.7):

$$(Ctx^\tau_T, \subseteq) \underset{\alpha_L}{\overset{\gamma^\tau_L}{\leftrightarrows}} (Ctx^\tau_L, \leq).$$

Furthermore, α_L maps Ctx^τ_T *onto* Ctx^τ_L. Indeed, for every height h such that $1 \leq h \leq \top_f[\tau]$, there exists a corresponding node $n[t]$ in the graph of τ' (if there was an unused height h, all heights greater than h could be diminished by 1, and therefore heights would not be as small as possible, contrary to their definition). Then $h = \alpha_L(\{n\})$. We also have $0 = \alpha_L(\emptyset)$ and $\top_L[\tau] = \alpha_L(\{n_0\})$ where n_0 is the top node of $\mathrm{Gr}(\tau)$.

Construction Let $\delta : \tau \to \tau'$.

$$\mathrm{Constr}_L(\delta,\tau)(c) = \alpha_L(\mathrm{Constr}_T(\delta,\tau)(\gamma_L^{\tau'}(c))) = \alpha_L(\gamma_L^{\tau'}(c)) = c$$

since nodes marked in c' and in $\mathrm{Constr}_T(\delta,\tau)(c')$ have the same type labels. Therefore $\mathrm{Constr}_L(\delta,\tau) = \mathrm{Id}$.

Restriction Let $\delta : \tau \to \tau'$. If $c = \top_L[\tau]$,

$$\mathrm{Restr}_L(\delta,\tau)(c) = \alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c)) = \alpha_L(\mathrm{Restr}_T(\delta,\tau)(\underline{\tau})) = \alpha_L(\underline{\tau'}) = \top_L[\tau'].$$

Otherwise, $c \leq \top_L[\tau']$ and $c < \infty$,

$$\mathrm{Restr}_L(\delta,\tau)(c) = \alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c)) \leq c$$

since all types labelling nodes of $\mathrm{Restr}_T(\delta,\tau)(c')$ also label nodes of $c' = \gamma_L^{\tau}(c)$.

$$\mathrm{Restr}_L(\delta,\tau)(c) = \alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c)) \leq \top_f[\tau']$$

since all types labelling nodes of $\mathrm{Restr}_T(\delta,\tau)(c')$ are types of the graph of τ', and of finite height since they label nodes in $\gamma_L^{\tau}(c)$ with $c < \infty$.

On the other hand, since $c \sqcap \top_f[\tau'] \leq \top_f[\tau']$, and $\alpha_L : Ctx_T^{\tau'} \to Ctx_L^{\tau'}$ is onto, there exists $c'' \in Ctx_T^{\tau'}$ such that $c \sqcap \top_f[\tau'] = \alpha_L(c'')$. In the proof of analysis T, we have seen that $\mathrm{Gr}(\tau)$ contains an edge $n_0[\tau] \xrightarrow{\delta} n_1[\tau']$ and that there exists an isomorphism ψ from $\mathrm{Gr}(\tau')$ to the part G'' of $\mathrm{Gr}(\tau)$ reachable from n_1. Since all heights of types in c'' are at most c, $\psi(c'') \subseteq c' = \gamma_L^{\tau}(c)$, therefore $c'' \subseteq \mathrm{Restr}_T(\delta,\tau)(c')$ hence $\alpha_L(\mathrm{Restr}_T(\delta,\tau)(c')) \geq \alpha_L(c'') = c \sqcap \top_f[\tau']$.

Finally, $\alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c)) = c \sqcap \top_f[\tau']$.

Three cases have to be distinguished: if $\top_L[\tau'] = \infty$ and $c = \infty$, $\alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c))) = \top_L[\tau'] = \infty$. If $\top_L[\tau'] = \infty$ and $c < \infty$, $\alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c))) = c \sqcap \top_f[\tau']$. Otherwise, $\alpha_L(\mathrm{Restr}_T(\delta,\tau)(\gamma_L^{\tau}(c))) = c \sqcap \top_f[\tau']$.

Therefore, if $\top_L[\tau'] = \infty$,

$$\mathrm{Restr}_L(\delta,\tau)(c) = (\text{if } c = \infty \text{ then } \infty \text{ else } 0) \sqcup (c \sqcap \top_f[\tau'])$$

and if $\top_L[\tau'] < \infty$,

$$\mathrm{Restr}_L(\delta,\tau)(c) = c \sqcap \top_f[\tau'].$$

Other monomorphic cases Other context transformers except generalization and instantiation are easy to compute. Their values are summarized in Figure 3.17.

Generalization Let σ be a given substitution on type variables. The generalization is defined by: $G_L^{\tau} = \alpha_L \circ G_T^{\tau} \circ \gamma_L^{\sigma\tau}$.

Lemma 3.51

$$G_L^{\tau}(c) = \begin{cases} \sqcup\{\top_L[\tau'] \mid \tau' \textit{ label of a node of } \mathrm{Gr}(\tau), \top_L[\sigma\tau'] \leq c\} \sqcup 1 & \textit{if } c > 0 \\ 0 & \textit{if } c = 0 \end{cases}$$

The $\sqcup 1$ in this formula comes from the following fact. It may happen that for all types τ', $\top_L[\sigma\tau'] > c$, but if $c > 0$, there is always a type τ'_1 in the graph of $\sigma\tau$ such that $\top_L[\tau'_1] \leq c$. Then $\tau'_1 \in \mathrm{Types}(\gamma_L^{\sigma\tau}(c))$. The corresponding type in the graph of τ is then a type variable, of height 1. This type variable will be marked in $G_T^\tau(\gamma_L^{\sigma\tau}(c))$, therefore $G_L^\tau(c) \geq 1$.

Proof: If $c = 0$, $\gamma_L^{\sigma\tau}(c) = \emptyset$, $G_T^\tau \circ \gamma_L^{\sigma\tau} = \emptyset$ and $G_L^\tau(c) = 0$.

Let us assume that $c > 0$. By definition of α_L, $G_L^\tau(c) = \sqcup\{\top_L[\tau'_0] | n'[\tau'_0] \in G_T^\tau(\gamma_L^{\sigma\tau}(c))\}$. If $\top_L[\sigma\tau'] \leq c$ (where τ' is the label of a node of $\mathrm{Gr}(\tau)$) then there exists $n[\sigma\tau'] \in \gamma_L^{\sigma\tau}(c)$, then by Lemma 3.41, $\tau' \in \mathrm{Types}(G_T^\tau(\gamma_L^{\sigma\tau}(c))$ and $G_L^\tau(c) \geq \top_L[\tau']$. Then

$$G_L^\tau(c) \geq \sqcup\{\top_L[\tau'] \mid \tau' \text{ label of a node of } \mathrm{Gr}(\tau), \top_L[\sigma\tau'] \leq c\}.$$

There exists a label τ' of a node of $\mathrm{Gr}(\sigma\tau)$ such that $\top_L[\tau'] = 1 \leq c$. Then $\exists n[\tau'] \in \gamma_L^{\sigma\tau}(c)$. Either τ' is of the form $\sigma\tau_1$ and $\mathrm{Types}(G_T^\tau(\{n\}))$ contains τ_1, or τ' is the label of a node of $\mathrm{Gr}(\sigma\beta)$ where β is a type variable, and $\mathrm{Types}(G_T^\tau(\{n\}))$ contains β. Anyway, $\alpha_L(G_T^\tau(\{n\})) \geq 1$. Hence $G_L^\tau(c) \geq 1$.

Conversely, if $n'[\tau_0] \in G_T^\tau(\gamma_L^{\sigma\tau}(c))$, $\exists n[\tau'_0] \in \gamma_L^{\sigma\tau}(c)$ such that $n' \in G_T^\tau(\{n\})$. By Lemma 3.41, $\tau'_0 = \sigma\tau_0$ or τ_0 is a type variable and τ'_0 is a type of the graph of $\sigma\tau_0$. Then $\top_L[\sigma\tau_0] \leq \top_L[\tau'_0] \leq c$, or $\top_L[\tau_0] \leq 1$. Therefore we have $\top_L[\tau_0] \leq \sqcup\{\top_L[\tau'] \mid \tau'$ label of a node of $\mathrm{Gr}(\tau)$, $\top_L[\sigma\tau'] \leq c\} \sqcup 1$ and

$$G_L^\tau(c) \leq \sqcup\{\top_L[\tau'] \mid \tau' \text{ label of a node of } \mathrm{Gr}(\tau), \top_L[\sigma\tau'] \leq c\} \sqcup 1.$$

$\square$

Instantiation

Lemma 3.52 *The general form of the context transformers $Ctx_L^\tau \to Ctx_L^{\tau'}$ of analysis L for a monomorphic language is*

$$\lambda c.(\text{if } c = \infty \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$$

where $i, f, i^+ \in Ctx_L$. The effect of imperative operations (assignments) and of higher-order functions is denoted by i, the effect of first-order purely functional operations by f. The step part corresponds to the free variables of closures. The context transformers can be put into the canonical form:

$$\phi(c) = (\text{if } c = \infty \text{ then } \phi(\infty) \text{ else } 0) \sqcup (c \sqcap m) \sqcup \phi(0)$$

where m is the greatest (finite) integer for which the value of ϕ changes: $\phi(m-1) \neq \phi(m)$ ($m = 0$ if ϕ is constant on $[\![0, \top_f[\tau]]\!]$).

Proof: This is true for elementary operations (step, constructions which are in fact identity, restriction, constants) and preserved by composition, upper bound and therefore fixed point since iterations are finite (lattice of finite height for each type). Identity is represented by

$$\mathrm{Id}_L^\tau(c) = (\text{if } c = \infty \text{ then } \infty \text{ else } 0) \sqcup (c \sqcap \top_f[\tau]) \sqcup 0.$$

For the composition, if

$$\phi_1(c) = (\text{if } c = \infty \text{ then } i_1^+ \text{ else } 0) \sqcup (c \sqcap f_1) \sqcup i_1$$

and

$$\phi_2(c) = (\text{if } c = \infty \text{ then } i_2^+ \text{ else } 0) \sqcup (c \sqcap f_2) \sqcup i_2$$

then

$$\phi_1(\phi_2(c)) = (\text{if } c = \infty \text{ then } \phi_1(i_2^+ \sqcup f_2) \text{ else } 0) \sqcup (c \sqcap (f_1 \sqcap f_2)) \sqcup \phi_1(i_2).$$

The other points are obvious. □

The result will enable us to design an efficient implementation of this analysis. This will be detailed in Chapter 4. We can compare this general form with other properties that enable fast implementations of analyses:

1. The context transformers are additive. This is equivalent to being monotone since Ctx_L^τ is a linear lattice.

2. The context transformers are not reductive (ϕ is reductive if $\forall c \in Ctx_L^\tau, \phi(c) \leq c$). Notice that in the case of a purely functional language, they were reductive [Deu97].

3. They satisfy Kam-Ullman rapidity: $\forall \phi \in Tr_L^{\tau[\tau']}, \forall c \in Ctx_L^\tau, \phi(c) \leq c \sqcup \phi(0)$ [KU76, page 166]. This property is equivalent to the property required to apply Knuth's algorithm. However, Knuth's algorithm will be not applied to the context transformers themselves. It will be applied to functions computing the integers i^+, f, i of the representation of context transformers.

Remark. The generalization is not of the form of the above lemma. However, we have $G_L^\tau(c) \leq c$ since $\top_L[\tau] \leq \top_L[\sigma\tau]$. Then,

$$\begin{aligned} & G_L^\tau(\text{if } c = \infty \text{ then } i^+ \text{ else } 0 \sqcup (c \sqcap f) \sqcup i) \\ & \quad = \text{if } c = \infty \text{ then } G_L^\tau(i^+) \text{ else } 0 \sqcup (G_L^\tau(c) \sqcap G_L^\tau(f)) \sqcup G_L^\tau(i) \\ & \quad \leq \text{if } c = \infty \text{ then } G_L^\tau(i^+) \text{ else } 0 \sqcup (c \sqcap G_L^\tau(f)) \sqcup G_L^\tau(i). \end{aligned}$$

Then we can always put $G_L^\tau \circ \phi$ into the form of Lemma 3.52 by losing a bit of precision. We use this property here to define our instantiation only on context transformers of the form of Lemma 3.52. In our implementation (see Chapter 4), we will put the analysis of an expression $\texttt{fun}\ x \to M$ or $\texttt{let}\ x = N\ \texttt{in}\ M$ into the form:

$$L[\![M]\!]\rho x = \sqcup_{\text{occurrences } x_i \text{ of } x \text{ in } M} G_L^{\sigma_i,\tau} \circ \phi_i$$

where $\tau_{x_i} = \sigma_i \tau_x^{\mathrm{g}}$ is the type of the ith occurrence x_i of x, and $G_L^{\sigma_i,\tau}$ designates the generalization corresponding to substitution σ_i. We compute ϕ_i in the form of Lemma 3.52, and use the above property to get the result of the whole analysis in the same form.

Then we can define the instantiation only on context transformers of the form of Lemma 3.52.

Definition 3.53 The instantiation function for contexts is defined by: $I_L^\tau = \alpha_L \circ I_T^\tau \circ \gamma_L^\tau$.

Let $\alpha_f(c) = \sqcup\{\top_f[\tau] | n[\tau] \in c\}$ similar to $\alpha_L(c)$ except that $\top_L[\tau]$ is replaced by $\top_f[\tau]$. Then we define $I_f^\tau = \alpha_f \circ I_T^\tau \circ \gamma_L^\tau$.

The instantiation of predicate transformers $\phi : \tau[\tau']$ is

$$I_L^{\tau[\tau']}(\phi) = \lambda c.(\text{if } c = \infty \text{ then } I_L^{\tau'}(\phi(\infty)) \text{ else } 0) \sqcup (c \sqcap I_f^{\tau'}(m)) \sqcup I_L^{\tau'}(\phi(0))$$

where m is the greatest integer for which the value of ϕ changes: $\phi(m-1) \neq \phi(m)$ ($m = 0$ if ϕ is constant on $[\![0, \top_f[\tau']]\!]$).

Lemma 3.54

$$I_L^\tau(c') = \sqcup\{\top_L[\sigma\tau'] \mid \tau' \textit{ label of a node of } \mathrm{Gr}(\tau), \top_L[\tau'] \le c'\}$$
In particular, $I_L^\tau(0) = 0$
$$I_f^\tau(c') = \sqcup\{\top_f[\sigma\tau'] \mid \tau' \textit{ label of a node of } \mathrm{Gr}(\tau), \top_L[\tau'] \le c'\}$$

Proof: Similar to Lemma 3.51. □

Analysis L is not polymorphically invariant [Abr86]. If we reuse the same example as for analysis T page 67, we have the same result: in this example, the analysis of the instantiated expression is less precise than the analysis of the polymorphic expression. It may also be more precise. For example: in `fun x -> let y = ([x],[3]) in snd y` of type $\beta \to \beta$, the list `[x]` escapes (escape context 2, since this is the height of `snd y = [3]`), whereas if `x` is of type `int list` for example, the height of `[x]` is 3, and the list constructor of `[x]` does not escape with the same escape context 2.

The instantiation by $I_L^{\tau[\tau']}$ may lead to a loss of precision with respect to the direct analysis of the considered instance. For example, if $\tau = \beta * \beta'$ and it is instantiated by $\sigma = \{\beta \mapsto \texttt{bool list list}, \beta' \mapsto \texttt{bool}\}$, if we know that the level 1 of τ escapes, which corresponds to β or β', the analysis will consider that level 3 of $\sigma\tau$ escapes, whereas if β' only escapes, the direct analysis would have indicated that only level 1 of $\sigma\tau$ escapes which is more precise.

But the analysis obtained by instantiation may also be more precise than the direct analysis. For example,

```
phi = fun x -> fun y ->
        let z = (x,y) in
          a := snd z;
          fst z
```

Here, the analysis yields $L[\![\texttt{phi}]\!]\rho 1 c = c \sqcup \top_L[\tau]$ if `y` is of type τ. Assume that the analysis is performed on the general version $\tau = \beta$. Then $L[\![\texttt{phi}]\!]\rho 1 c = c \sqcup 1$ and the instantiation leaves this value unchanged (since we instantiate relatively to the type of `x` independently of the one of `y`). On the other hand, the direct analysis with $\tau = \texttt{int list}$ for example yields $L[\![\texttt{phi}]\!]\rho 1 c = c \sqcup 2$ which is less precise.

Finally, the analysis yielded by the instantiation and the direct analysis are not comparable. However, the correctness can be proved with respect to analysis T.

Theorem 3.55 (Instantiation) *Let* $\phi_L = \rho_L[\![y]\!]i$ *and* $\phi_T = \rho_T[\![y]\!]i : \tau[\tau']$.

$$\phi_L \ge \alpha_L \circ \phi_T \circ \gamma_L^\tau \Rightarrow I_L^{\tau[\tau']}(\phi_L) \ge \alpha_L \circ I_T^{\tau[\tau']}(\phi_T) \circ \gamma_L^{\sigma\tau}$$

Proof:

- If $c = \infty$,

$$I_L^{\tau[\tau']}(\phi_L)(\infty) \ge I_L^{\tau'}(\phi_L(\infty)) \ge (I_L^{\tau'} \circ \alpha_L \circ \phi_T \circ \gamma_L^\tau)(\infty) \ge (\alpha_L \circ I_T^{\tau'} \circ \phi_T)(\underline{\tau})$$

  and on the other hand, by Proposition 3.47,

$$I_T^{\tau[\tau']}(\phi_T)(\underline{\sigma\tau}) = (I_T^{\tau'} \circ \phi_T)(\underline{\tau})$$

therefore

$$(\alpha_L \circ I_T^{\tau[\tau']}(\phi_T) \circ \gamma_L^{\sigma\tau})(\infty) = (\alpha_L \circ I_T^{\tau'} \circ \phi_T)(\underline{\tau}) \leq I_L^{\tau[\tau']}(\phi_L)(\infty).$$

This gives the correctness in the case $c = \infty$.

- If $c < \infty$, $\phi_L(c) = (c \sqcap m) \sqcup \phi_L(0)$, $I_L^{\tau[\tau']}(\phi_L)(c) = (c \sqcap I_f^{\tau'}(m)) \sqcup I_L^{\tau'}(\phi_L(0))$. In the case $c = 0$,

$$\begin{aligned} I_L^{\tau'}(\phi_L(0)) &= (\alpha_L \circ I_T^{\tau'} \circ \gamma_L^{\tau'} \circ \phi_L)(0) \\ &\geq (\alpha_L \circ I_T^{\tau'} \circ \phi_T)(\tau) && \text{since } \phi_L(0) \geq (\alpha_L \circ \phi_T)(\tau) \\ &\geq [\alpha_L \circ I_T^{\tau[\tau']}(\phi_T)](\sigma\tau) && \text{by Proposition 3.47} \end{aligned}$$

In the general case,

$$\begin{aligned} I_L^{\tau[\tau']}(\phi_L)(c) &= (c \sqcap I_f^{\tau'}(m)) \sqcup I_L^{\tau'}(\phi_L(0)) \\ &\geq (I_f^{\tau'}[(G_L^{\tau}(c) \sqcap m) \sqcup \phi_L(0)] \sqcap c) \sqcup I_L^{\tau'}(\phi_L(0)). \end{aligned}$$

Indeed, $c \geq I_f^{\tau'}[G_L^{\tau}(c) \sqcap m] \sqcap c$, $I_f^{\tau'}(m) \geq I_f^{\tau'}[G_L^{\tau}(c) \sqcap m] \geq I_f^{\tau'}[G_L^{\tau}(c) \sqcap m] \sqcap c$ and $I_L^{\tau'}(\phi_L(0)) \geq I_f^{\tau'}[\phi_L(0)] \sqcap c$.

$$\begin{aligned} I_L^{\tau[\tau']}(\phi_L)(c) &\geq [(I_f^{\tau'} \circ \phi_L \circ G_L^{\tau})(c) \sqcap c] \sqcup I_L^{\tau'}(\phi_L(0)) \\ & \qquad \text{since } \phi_L(c') = (c' \sqcap m) \sqcup \phi_L(0) \text{ with } c' = G_L^{\tau}(c) \\ &\geq [(\alpha_f \circ I_T^{\tau'} \circ \gamma_L^{\tau'} \circ \alpha_L \circ \phi_T \circ \gamma_L^{\tau} \circ \alpha_L \circ G_T^{\tau} \circ \gamma_L^{\sigma\tau})(c) \sqcap c] \sqcup I_L^{\tau'}(\phi_L(0)) \\ & \qquad \text{by definition of } I_f^{\tau'}, G_L^{\tau} \text{ and correctness of } \phi_L \\ &\geq [(\alpha_f \circ I_T^{\tau'} \circ \phi_T \circ G_T^{\tau} \circ \gamma_L^{\sigma\tau})(c) \sqcap c] \sqcup [\alpha_L \circ I_T^{\tau[\tau']}(\phi_T)](\sigma\tau) \\ & \qquad \text{using the case } c = 0 \text{ above} \\ &\geq [(\alpha_f \circ I_T^{\tau[\tau']}(\phi_T) \circ \gamma_L^{\sigma\tau})(c) \sqcap c] \sqcup [\alpha_L \circ I_T^{\tau[\tau']}(\phi_T)](\sigma\tau) \\ & \qquad \text{since } I_T^{\tau[\tau']}(\phi_T) \leq I_T^{\tau'} \circ \phi_T \circ G_T^{\tau} \\ &\geq \alpha_f[I_T^{\tau[\tau']}(\phi_T)(\gamma_L^{\sigma\tau}(c)) - I_T^{\tau[\tau']}(\phi_T)(\sigma\tau)] \sqcup [\alpha_L \circ I_T^{\tau[\tau']}(\phi_T)](\sigma\tau) \end{aligned}$$

since $I_T^{\tau[\tau']}(\phi_T)(c_T) - I_T^{\tau[\tau']}(\phi_T)(\sigma\tau) \leq c_T$ where $c_T = \gamma_L^{\sigma\tau}(c)$ does not contain any function. Moreover, the parameter of α_f does not underline any function, therefore α_f can be replaced by α_L, then both terms can be grouped and we get the required result.

□

Correctness We obtain analysis L defined in Figure 3.17.

Theorem 3.56 (Correctness of L) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let us assume that $L[\![N]\!]\rho\, x\, \top_L[\tau_N] < \top_L[\tau_x^{\mathrm{g}}]$.*

Contexts:	$Ctx_L^{\tau} = [\![0, \top_f[\tau]]\!] \cup \{\top_L[\tau]\}$
Context transformers:	$Tr_L^{\tau[\tau']} = Ctx_L^{\tau} \xrightarrow{\mathrm{m}} Ctx_L^{\tau'}$
Identity:	$\mathrm{Id}_L^{\tau} = \lambda c.c$
Greatest element:	$\top_{Tr_L}^{\tau} = \lambda c.\top_L[\tau]$
Least element:	$\bot_{Tr_L}^{\tau} = \lambda c.0$
Construction:	$\mathrm{Constr}_L(\delta, \tau) = \lambda c.c$
Restriction:	Let $\delta : \tau \to \tau'$. If $\top_L[\tau'] < \infty$, $\mathrm{Restr}_L(\delta, \tau) = \lambda c.c \sqcap \top_f[\tau']$

If $\top_L[\tau'] = \infty$, $\mathrm{Restr}_L(\delta, \tau) = \lambda c.(\text{if } c = \infty \text{ then } \infty \text{ else } 0) \sqcup (c \sqcap \top_f[\tau'])$

Step:	$\mathrm{test}_L^{\tau_y \to \tau, \tau'} = \lambda c.\text{if } c = \infty \text{ then } \top_L[\tau'] \text{ else } 0$
Upper bound:	$\sqcup_L(\phi_i)_i = \lambda c. \sqcup_i \phi_i(c)$
Composition:	$\phi \circ_L \psi = \lambda c.\phi(\psi(c))$

Generalization:

$$G_L^{\sigma,\tau} = \lambda c. \begin{cases} \sqcup\{\top_L[\tau'] \mid \tau' \text{ label of a node of } \mathrm{Gr}(\tau), \top_L[\sigma\tau'] \le c\} \sqcup 1 & \text{if } c > 0 \\ 0 & \text{if } c = 0 \end{cases}$$

Instantiation:

$I_L^{\sigma,\tau[\tau']}(\phi) = \lambda c.(\text{if } c = \infty \text{ then } I_L^{\sigma,\tau'}(\phi(\infty)) \text{ else } 0) \sqcup (c \sqcap I_f^{\sigma,\tau'}(m)) \sqcup I_L^{\sigma,\tau'}(\phi(0))$
where m is the greatest integer for which the value of ϕ changes: $\phi(m-1) \neq \phi(m)$.

Figure 3.17: Lattice of context transformers for analysis L

- `let rec` $x =$ `fun` $y \to M$ `in` N. *In this case, let ℓ be the location of the closure x. We assume $R(x)\top_L[\tau_N] < \top_L[\tau_x^{\mathrm{g}}]$ where R is defined in the* `let rec` *case of Figure 3.6.*

In these two cases, ℓ can be stack allocated, and deallocated at the end of N.

Proof: By the correctness of analysis T (Theorem 3.49) and the properties of the abstraction, we can apply the general correctness proposition (Proposition 3.22). This yields the expected result, noticing that $\alpha_L(\{n_0\}) = \top_L[\tau_x^{\mathrm{g}}]$ if n_0 is the top node of $\mathrm{Gr}(\tau_x^{\mathrm{g}})$. □

Example 3.57 If we consider again the `map` function from Example 3.14 of type $(\tau_1 \to \tau_2) \to \tau_1$ `list` $\to \tau_2$ `list`, analysis L yields:

$$\begin{aligned} &L[\![\texttt{fun f l -> if ...}]\!]\rho = \\ &\quad \{1 \mapsto (\lambda c.\text{if } c = \infty \text{ then } \infty \text{ else } 0 \sqcup (c \sqcap \top_L[\tau_2])) \sqcup \rho[\![\texttt{map}]\!]\, 1, \\ &\quad\ 2 \mapsto (\lambda c.\top_L[\tau_1]) \sqcup \rho[\![\texttt{map}]\!]\, 2, \texttt{map} \mapsto (\lambda c.c)\}. \end{aligned}$$

We can then compute the fixed points:

$$\rho_L[\![\texttt{map}]\!] = \{1 \mapsto \lambda c.\text{if } c = \infty \text{ then } \infty \text{ else } 0 \sqcup (c \sqcap \top_L[\tau_2]), 2 \mapsto \lambda c.\top_L[\tau_1]\}.$$

If a closure may escape in the result (i.e. `map` is partially applied) $c = \infty$ and $\rho_L[\![\texttt{map}]\!]1c = \infty$: the closure of the first parameter escapes. Otherwise, the result of the first parameter escapes if and only if the elements of result escape ($c \sqcap \top_L[\tau_2]$). The elements of the second parameter of `map` may always escape, because they are given to an unknown function (the first parameter of `map`).

In we consider the polymorphic version of `map`$:(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list` the analysis yields:

$$\rho_L[\![\texttt{map}]\!] = \{1 \mapsto \lambda c.\text{if } c = \infty \text{ then } \infty \text{ else } 0 \sqcup (c \sqcap 1), 2 \mapsto \lambda c.1\}.$$

When we instantiate $\alpha = \tau_1, \beta = \tau_2$, where τ_1 and τ_2 do not contain functions, we get the same result as the direct analysis of the instantiated function:

$$\rho_L[\![\mathtt{map}]\!] = \{1 \mapsto \lambda c.\text{if } c = \infty \text{ then } \infty \text{ else } 0 \sqcup (c \sqcap \top_L[\tau_2]), 2 \mapsto \lambda c.\top_L[\tau_1]\}.$$

3.6.2 Analysis F (without enumeration of free variables)

Enumeration of free variables of all closures leads to a quadratic complexity. (This comes from overlapping closures. For example, in an expression of size $\mathcal{O}(n)$ such as $\mathtt{fun}\ x_1 \to ...\mathtt{fun}\ x_n \to x_1...x_n$, there are n closures, therefore, with the obvious algorithm, enumerating all their free variables means scanning the expression $x_1...x_n$ n times which takes n^2 operations.) We can eliminate this enumeration as follows.

We define a new analysis F with the same definition as L except that

$$\begin{aligned} Ctx_F^\tau &= [\![0, \top_f[\tau]]\!] \cup \{\infty\} \\ \phi \circ_F \psi &= \phi \circ (\psi \sqcup s_\infty) \end{aligned}$$

where $s_\infty = \lambda c.\text{if } c = \infty \text{ then } \infty \text{ else } 0$. We extend context transformers ϕ which where defined on Ctx_L^τ (with $\top_L[\tau] < \infty$) to Ctx_F^τ by taking $\phi(\infty) = \sqcup_{n\in\mathbb{N}}\phi(n)$. We have $\phi \circ_F \psi \geq \phi \circ_L \psi$, hence the analysis is correct.

Lemma 3.58 *If M is not a variable, the analysis of M can be put into the form:*

$$F[\![M]\!]\rho x = \sqcup_F(F[\![M_i]\!]\rho_i x \circ_F \phi_i')_i$$

where M_i are the subexpressions of M.

Proof: In most cases, the analysis is already in this form. Just notice that $\phi \circ_F \mathrm{Id}_F^\tau = \phi$ to put the `if`, $M; N$ and `let` cases into this form. In the `let rec` case,

$$F[\![\mathtt{let\ rec}\ f = \mathtt{fun}\ y \to M\ \mathtt{in}\ N]\!]\rho x = R(x) = F[\![N]\!]\rho' x \sqcup_F (F[\![\mathtt{fun}\ y \to M]\!]\rho' x) \circ_F (R(f))$$

hence the expected form. In the `fun` case, we have to consider the free variables of M as subexpressions of M. □

The following lemma is the key point to eliminate the enumeration of free variables in the function case. It means that if $x \in FV(M)$, then $F[\![M]\!]\rho x\infty = \top_L[\tau_x^{\mathrm{g}}]$, hence if a closure is in the input context of M, all free variables x of M escape.

Lemma 3.59 *If $x \in FV(M)$ is a variable, then $F[\![M]\!]\rho x \geq \lambda c.\text{if } c = \infty \text{ then } \top_L[\tau_x^{\mathrm{g}}] \text{ else } 0$.*

Proof: By induction on M. If M is a variable, since $x \in FV(M)$, $M = x$, and $F[\![M]\!]\rho x = G_F^{\sigma,\tau_x^{\mathrm{g}}} \geq \lambda c.\text{if } c = \infty \text{ then } \top_L[\tau_x^{\mathrm{g}}] \text{ else } 0$, hence the result.

In the other cases, we use the above lemma:

$$F[\![M]\!]\rho x = \sqcup_F(F[\![M_i]\!]\rho_i x \circ_F \phi_i')_i$$

where M_i are the subexpressions of M. Since $x \in FV(M)$, $\exists i, x \in FV(M_i)$. Then

$$\begin{aligned} F[\![M]\!]\rho x &\geq F[\![M_i]\!]\rho_i x \circ_F \phi_i' \geq (\lambda c.\text{if } c = \infty \text{ then } \top_L[\tau_x^{\mathrm{g}}] \text{ else } 0) \circ s_\infty \\ &\geq \lambda c.\text{if } c = \infty \text{ then } \top_L[\tau_x^{\mathrm{g}}] \text{ else } 0 \end{aligned}$$

using the induction hypothesis on M_i. □

The formula of the `fun` case is:

$$\begin{aligned}F[\![\texttt{fun}\ y \to M]\!]\rho = &\{1 \mapsto \phi(y), i \mapsto \phi(i-1)c \text{ if } i > 1, x \mapsto \phi(x) \text{ if } x \in \mathit{Var} - \{y\}\ \} \\ &\sqcup \{x \mapsto \lambda c.\text{if } c = \infty \text{ then } \top_L[\tau_x^{\mathrm{g}}] \text{ else } 0 \text{ if } x \in FV(\texttt{fun}\ y \to M)\ \}\end{aligned}$$

where $\phi(x) = (F[\![M]\!]\rho[y \mapsto \rho_\top]x) \circ_F \mathrm{Restr}_F(\mathrm{app}, \tau_y \to \tau_M)$ and $\rho_\top = \{i \mapsto \top_{Tr_F}^{\tau_y, i} \text{ if } i \in \mathit{Ind}\}$. The above lemma proves that in this formula the second term is smaller than the first one, then this formula can be changed into:

$$F[\![\texttt{fun}\ y \to M]\!]\rho = \{1 \mapsto \phi(y), i \mapsto \phi(i-1)c \text{ if } i > 1, x \mapsto \phi(x) \text{ if } x \in \mathit{Var} - \{y\}\ \}.$$

Now, the enumeration of free variables is eliminated.

3.6.3 Other integer escape analyses

Using height ∞ for closures leads to a relatively imprecise analysis for stack allocation of closures, and for all data structures which contain functions. We have designed a slightly more precise analysis still using integers. This analysis is detailed in Appendix C. In practice, the results are very similar to the ones of analysis F.

The next chapter will show how the analyses L and F are implemented, and the different costs of these implementations. Chapter 5 will handle analysis T. Our goal is to determine which analysis gives the best cost/precision tradeoff.

Chapter 4

Implementation of analyses L and F

The same technique is used to implement both integer escape analyses L (Section 3.6.1) and F (Section 3.6.2). Analysis F avoids an explicit enumeration of the free variables of the closures. This enumeration is quadratic in the worst case, and increases the complexity of analysis L. The small differences between the algorithms of L and F will be detailed in the following of this chapter.

The analysis program works in two passes. The first pass, the analysis itself, takes as argument the abstract syntax tree of the program to be analyzed. It yields a new syntax tree labelled by escape information for each expression. The second pass takes this labelled tree, and decides how to transform the program to optimize it, by introducing stack allocations.

4.1 Computing type heights

As we have seen in Section 3.6, we associate with each type τ, its height $\top[\tau]$. This height has to satisfy the properties:

$$\top[\tau] \geq 1 \tag{3.2}$$

$$\top[\tau_1 \rightarrow \tau_2] = \infty \tag{3.3}$$

$$\top[\tau] \geq \top[\tau'] \text{ if there exists an accessor } \delta : \tau \rightarrow \tau' \tag{3.4}$$

$$\top[\tau] \geq \top[\tau'] + 1 \text{ if there exists an accessor } \delta : \tau \rightarrow \tau', \tag{3.5}$$

$$\text{and this rule does not contradict the preceding ones.}$$

To compute this height, we build a graph whose nodes are types and which contains edges $\tau_1 \xrightarrow{\delta} \tau_2$ for each accessor $\delta : \tau_1 \rightarrow \tau_2$. This graph is similar to the graph of a type for analysis T, but it merges all nodes labelled by the same type. By (3.4), the height of all types in a strongly connected component of this graph is the same. Therefore, we compute the strongly connected components of this graph by Tarjan's algorithm [Tar72]. If we merge all nodes of the same strongly connected component, we get a DAG (Directed Acyclic Graph). We give heights to the nodes in reverse topological order (if there is an edge $n_1 \xrightarrow{\delta} n_2$, the height of n_2 is determined before the height of n_1). The leaves (nodes without outgoing edges) have height 1 (to satisfy (3.2)), the following nodes have height 1 plus the maximum height of their sons, except for functions, which have height ∞ to satisfy (3.3).

Example 4.1 Let us consider the following type declaration:

```
type term =
    Var of int
  | Prop of head * term list
and head =
    { name: string;
      props: (term * term) list }
```

The types `term`, `term list`, `term` $*$ `term`, (`term` $*$ `term`)`list` and `head` are in the same strongly connected component, therefore $\top[\texttt{term}] = \top[\texttt{term list}] = \top[\texttt{term} * \texttt{term}] = \top[(\texttt{term} * \texttt{term})\texttt{list}] = \top[\texttt{head}] = 2$ and $\top[\texttt{head list}] = 1 + \top[\texttt{head}] = 3$ since this type is not in the same strongly connected component.

The building of the graph can be done incrementally. A type is added when it is met during the analysis. Adding a type does not modify already existing strongly connected components, because when adding a type τ, all types of the same strongly connected component are added at the same time (they are reachable from τ). New types are not reachable from types already in the graph.

Two special cases have been excluded from the theoretical study. If τ is an abstract type, $\top[\tau] = 1$. Nothing can be extracted from τ by standard expressions of the language. If a function (implemented out of Objective Caml, in C for example) extracts a part of a value of type τ, its parameter of type τ is considered as entirely escaping.

The other special case is the one of non-monotone recursive types (page 67). Such types are detected by our analyzer, and are given height 1, in a way similar to abstract types. If a standard destructor of the language is used to access such types, the parameter of the destructor escapes entirely as soon as a part of the result escapes.

This computation of type heights is relatively complex and time consuming. It is optimized by translating type expressions to types labelled by their height. This avoids recomputing the height several times for the same type.

4.2 Building the equations of the analysis

In this section, A designates an analysis, which may be L or F. The algorithm is based on work by Tarjan [Tar79].

Semantic equations are represented by an ***anti-tree*** (tree with pointers from sons to fathers), whose nodes are occurrences of $A[\![M]\!]\rho$ and whose edges are such that for all $x \in \mathit{Var}$, $A[\![M]\!]\rho x = \sqcup_i A[\![M_i]\!]\rho_i x \circ_A op_i$ if $A[\![M]\!]\rho$ is linked by edges labelled op_i to $A[\![M_i]\!]\rho_i$ (the M_i are subexpressions of M). We can easily see that the equations of the analysis can be put into this form (the proof is the same as in Lemma 3.58).

$$\begin{tikzcd}
 & A[\![M]\!]\rho & \\
A[\![M_1]\!]\rho_1 \arrow[ur, "op_1"] & \cdots & A[\![M_n]\!]\rho_n \arrow[ul, "op_n"']
\end{tikzcd}$$

With this definition of the tree, we can easily see that

$$A[\![M]\!]\rho x = \sqcup_{\text{occurrences } x_i \text{ of } x \text{ in } M}((G_A^{\sigma_i,\tau} \circ_A \phi'_{i1}) \circ_A \ldots) \circ_A \phi'_{ij_i} \tag{4.1}$$

where $\tau_{x_i} = \sigma_i \tau_x^{\mathrm{g}}$ is the type of the ith occurrence x_i of x, and $G_L^{\sigma_i,\tau}$ designates the generalization corresponding to substitution σ_i, the $\phi'_{i1}, ..., \phi'_{ij_i}$ are the labels of a path from the node representing the occurrence x_i of x to the node representing M.

Two operations can be used to manipulate the tree: eval and link, defined as in Tarjan's paper [Tar79]. eval(n) computes the composition of context transformers labelling edges on a path from node n to the root of the tree, doing path compression: all nodes on the path from n to the root are linked directly to the root after a call to eval(n) and labels of edges are correctly updated. link(n_1, n_2, op) links n_1 to node n_2 with the context transformer op as label (n_1 must be a root of a tree, since each node has at most one father).

Let $\varrho(x)$ be a set containing a pair (n, τ_x) for each occurrence of x in M, where n is the node of the anti-tree that corresponds to this occurrence, and $\tau_x = \sigma\tau_x^{\mathrm{g}}$ is the type in which x is used at this occurrence. The type τ_x^{g} is the type of the definition of x. For analysis L, the above relation (4.1) proves that

$$A[\![M]\!]\rho x = \sqcup_{(n,\sigma\tau_x^{\mathrm{g}})\in\varrho(x)} G_A^{\sigma,\tau_x^{\mathrm{g}}} \circ_A \mathrm{eval}(n) \tag{4.2}$$

when analyzing M. This formula enables us to compute A. The escaping part of a variable x is $\sqcup_{(n,\sigma\tau_x^{\mathrm{g}})\in\varrho(x)} G_A^{\sigma,\tau_x^{\mathrm{g}}} \circ_A \mathrm{eval}(n)$ where the eval have to go up to the binder of x in the tree. This can easily be inferred from the definition of labels of edges. To perform this computation, we memorize the value of $\varrho(x)$ for each variable, and the computation itself is postponed until we come back to the binder of x.

For analysis F, we have to take into account the fact that $\circ_F$ is not associative.

$$F[\![M]\!]\rho x = \sqcup_{\text{occurrences } x_i \text{ of } x \text{ in } M}(G_L^{\sigma_i,\tau} \circ (\phi'_{i1} \sqcup s_\infty) \circ ... \circ (\phi'_{ij_i} \sqcup s_\infty))$$

Therefore, we put as labels on the tree $\phi'_{ik} \sqcup s_\infty$ instead of ϕ'_{ik} itself, and use the ordinary composition $\circ$. With this composition, we can use formula (4.2).

By Lemma 3.52, the general form of context transformers is $\lambda c.(\text{if } c = \infty \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$. They can therefore be represented by a triple of integers (f, i, i^+).[1] The operations on context transformers are defined on the corresponding integers. For instance, the composition of context transformers is

$$\begin{aligned}(f_1, i_1, i_1^+) \circ (f_2, i_2, i_2^+) = (&f_1 \sqcap f_2, (i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 = \infty \text{ then } i_1^+ \text{ else } 0),\\ &(\text{if } f_2 \sqcup i_2^+ = \infty \text{ then } i_1^+ \text{ else } 0) \sqcup (i_2^+ \sqcap f_1)).\end{aligned}$$

If some of these values are unknowns, new unknowns f_3, i_3, i_3^+ have to be created to represent the composed operation, and equations are emitted to code the above formula. That is: $(f_1, i_1, i_1^+) \circ (f_2, i_2, i_2^+) = (f_3, i_3, i_3^+)$ with the inequations $f_3 \geq f_1 \sqcap f_2$, $i_3 \geq (i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 = \infty \text{ then } i_1^+ \text{ else } 0)$, $i_3^+ \geq (\text{if } f_2 \sqcup i_2^+ = \infty \text{ then } i_1^+ \text{ else } 0) \sqcup (i_2^+ \sqcap f_1)$ (when solving the inequations, we obtain the least solution, in which we have in fact equality). The upper bound is handled in the same way.

The algorithm is detailed in Figure 4.3. The analysis function anal takes as parameters the expression M to analyze, the environment ρ giving for each variable the escape information

[1] Our implementation of analysis F in fact uses a slightly different representation, based on splitting the analysis into two analyses F_1 and F_2 [Bla98]. We have found that this splitting is in fact not necessary. It gives exactly the same precision as the implementation described here, and has the same complexity, but is more complicated to explain. We have chosen the simplest presentation. It would be possible to reimplement analysis F in this simpler framework.

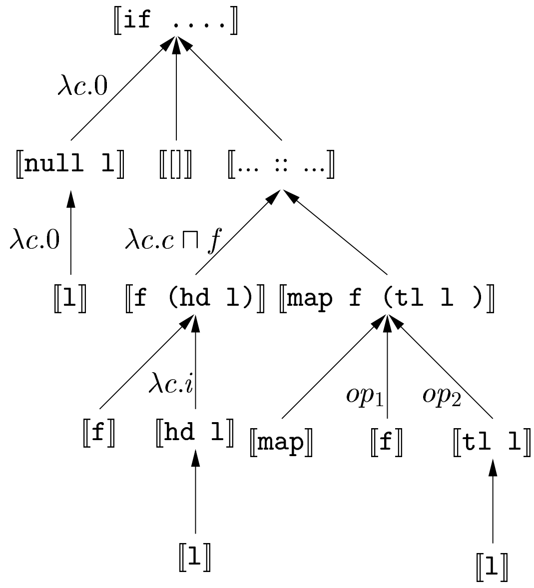


Figure 4.1: Tree built for Example 3.14, for analysis L (the tree for analysis F is quite similar, but of course context transformers have to be adapted). The identity operation has been omitted. Operations op_1 and op_2 are unknowns representing the escaping part of parameters of the `map` function. The `map` function is assumed to have type $(\tau_1 \rightarrow \tau_2) \rightarrow \tau_1$ `list` $\rightarrow \tau_2$ `list` where τ_1 and τ_2 do not contain functions. Then $i = \top_L[\tau_1]$, $f = \top_L[\tau_2]$.

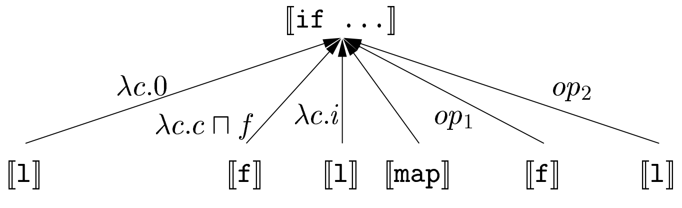


Figure 4.2: Figure 4.1 after path compression

Node is the set of nodes of Tarjan's tree.
$\rho : Var \to {Tr_A}^*$
$\varrho : Var \to (Node \times Type)^*$
$\mathrm{anal}(M, \rho, \varrho) =$
  match M **with**
  $y \to$
    let n be a new node, σ such that $\tau_y = \sigma\tau_y^{\mathrm{g}}$
    $\varrho(y) \leftarrow (n, \tau_y) : \varrho(y)$
    return $(n, [i \mapsto I_A^{\sigma,\tau_y^{\mathrm{g}}[\tau_{y,i}^{\mathrm{g}}]}(\rho(y)i)$ if $i < n$,
      $i \mapsto \top_{Tr_E}^{\tau_i'}$ if $n \le i < p])$[a]
    with $\tau_y^{\mathrm{g}} = \tau_1 \to ... \to \tau_n$, $\sigma\tau_n = \tau_n' \to ... \to \tau_p'$,
      where τ_n and τ_p' are not function types.
  $|N\ P \to$
    $(t_N, \rho_N) = \mathrm{anal}(N, \rho, \varrho)$
    $(t_P, \rho_P) = \mathrm{anal}(P, \rho, \varrho)$
    let $\rho_1 : \rho_M = \rho_N$
    return $\left(\begin{tikzcd} & n_M & \\ t_N \arrow[ur, "\mathrm{Id}_A"] & & t_P \arrow[ul, "\rho_1"'] \end{tikzcd}, \rho_M \right)$
  $|(\texttt{fun}\ y \to N) \to$
    $\varrho_1 = \varrho[y \mapsto [\,]]$
    $(t_N, \rho_N) = \mathrm{anal}(N, \rho[y \mapsto \rho_\top], \varrho_1)$
    $p_1 = \sqcup_{(n,\tau) \in \varrho_1(y)} \mathrm{eval}(n)$
    foreach $z \in FV(M)$,
      let n_z be a new node, $\varrho(z) \leftarrow (n_z, \tau_z^{\mathrm{g}}) : \varrho(z)$
    return $\left(\begin{tikzcd} & n_M & \\ n_z \arrow[ur, "\mathrm{test}_A^{\tau_M,\tau_z^{\mathrm{g}}}"] & & t_N \arrow[ul, "\mathrm{Restr}_A(\mathrm{app},\tau_M)"'] \end{tikzcd}, p_1 : \rho_N \right)$
  $|\texttt{let}\ y = N\ \texttt{in}\ P \to$
    $(t_N, \rho_N) = \mathrm{anal}(N, \rho, \varrho)$
    $\varrho_1 = \varrho[y \mapsto [\,]]$
    $(t_P, \rho_P) = \mathrm{anal}(P, \rho[y \mapsto \rho_N], \varrho_1)$
    $e_y = \sqcup_{(n,\sigma\tau_y^{\mathrm{g}}) \in \varrho_1(y)} G_A^{\sigma,\tau_y^{\mathrm{g}}} \circ \mathrm{eval}(n)$
    return $\left(\begin{tikzcd} & n_M & \\ t_P \arrow[ur, "\mathrm{Id}_A"] & & t_N \arrow[ul, "e_y"'] \end{tikzcd}, \rho_P \right)$
  $|(\texttt{let rec}\ f = \texttt{fun}\ y \to N\ \texttt{in}\ P) \to$
    $\varrho_1 = \varrho[f \mapsto [\,]]$
    let u and u' be new unknowns
    $(t_Q, \rho_Q) = \mathrm{anal}(\texttt{fun}\ y \to N, \rho[f \mapsto u], \varrho_1)$
    emit equations $\rho_Q = u$
    $(t_P, \rho_P) = \mathrm{anal}(P, \rho[f \mapsto u], \varrho_1)$
    $t_M = \begin{tikzcd} & n_M & \\ t_P \arrow[ur, "\mathrm{Id}_A"] & & t_Q \arrow[ul, "u'"'] \end{tikzcd}$
    emit equation
      $u' = \sqcup_{(n,\sigma\tau_f^{\mathrm{g}}) \in \varrho_1(f)} G_A^{\sigma,\tau_f^{\mathrm{g}}} \circ \mathrm{eval}(n)$
    return (t_M, ρ_P)
  $|\texttt{if}\ N\ \texttt{then}\ P\ \texttt{else}\ Q \to$
    $(t_N, []) = \mathrm{anal}(N, \rho, \varrho)$
    $(t_P, \rho_P) = \mathrm{anal}(P, \rho, \varrho)$
    $(t_Q, \rho_Q) = \mathrm{anal}(Q, \rho, \varrho)$
    return
      $\left(\begin{tikzcd} & n_M & \\ t_N \arrow[ur, "\bot_{Tr_A}^{\tau_N}"] & t_P \arrow[u, "\mathrm{Id}_A"'] & t_Q \arrow[ul, "\mathrm{Id}_A"'] \end{tikzcd}, \rho_P \sqcup_A \rho_Q \right)$
  $|N; P \to$
    $(t_N, \rho_N) = \mathrm{anal}(N, \rho, \varrho)$
    $(t_P, \rho_P) = \mathrm{anal}(P, \rho, \varrho)$
    return $\left(\begin{tikzcd} & n_M & \\ t_N \arrow[ur, "\bot_{Tr_A}^{\tau_N}"] & & t_P \arrow[ul, "\mathrm{Id}_A"'] \end{tikzcd}, \rho_P \right)$
  $|C(M_1, ..., M_n) \to$
    foreach $i \in \{1, ..., n\}$,
      $(t_{M_i}, \rho_{M_i}) = \mathrm{anal}(M_i, \rho, \varrho)$
    return
      $\left(\begin{tikzcd} & n_M & \\ t_{M_1} \arrow[ur, "\mathrm{Restr}_A(\overline{C_1},\tau_M)"] & ... & t_{M_n} \arrow[ul, "\mathrm{Restr}_A(\overline{C_n},\tau_M)"'] \end{tikzcd}, [] \right)$
  $|(\overline{C_i}(N) \leftarrow P) \to$
    $(t_N, []) = \mathrm{anal}(N, \rho, \varrho)$
    $(t_P, \rho_P) = \mathrm{anal}(P, \rho, \varrho)$
    return $\left(\begin{tikzcd} & n_M & \\ t_P \arrow[ur, "\top_{Tr_A}^{\tau_P}"] & & t_N \arrow[ul, "\bot_{Tr_A}^{\tau_N}"'] \end{tikzcd}, [] \right)$

[a] $[i \mapsto x_i]$ is the list whose ith element is x_i

Figure 4.3: Algorithm to compute the escape analysis

of its parameters ($\rho(y)$ is empty when the variable y is not a function), and the environment ϱ giving for each variable y the set of nodes of the tree corresponding to y, together with the type of the corresponding occurrence of y. It yields the tree t_M and the escape information for the parameters of M, ρ_M. The environment ρ is exactly equivalent to the environment considered in the definition of the analyses, but in these definitions, $\rho(y)$ is a function $\rho(y) \in Ind \to Tr_A$, whereas in the implementation $\rho(y)$ is represented by a list $\rho(y) \in {Tr_A}^*$.

In the variable case, the analysis creates a new node, and registers this node in the environment ϱ, together with the type in which the variable is used. The returned escape information for the parameters of y is the instantiation of the information found in the environment ρ. In the application case, $\rho_1 = A[\![N]\!]\rho 1$, then the returned tree is the expected translation of the formula: $A[\![N\ P]\!]\rho x = (A[\![N]\!]\rho x) \circ_A \mathrm{Constr}_A(\mathrm{app}, \tau_N) \sqcup_A (A[\![P]\!]\rho x) \circ_A (A[\![N]\!]\rho 1) \circ_A \mathrm{Constr}_A(\mathrm{app}, \tau_N)$ since the constructions are the identity. In the function case, by formula (4.2), $p_1 = A[\![M]\!]\rho y$ (the generalisation is useless here, since y cannot be used with different instances of its type). The algorithm creates a new node for each free variable, to take into account the $\{x \mapsto \mathrm{test}_A^{\tau_y \to \tau_N, \tau_x^{\mathrm{g}}}$ if $x \in FV(\texttt{fun}\ y \to N)\ \}$ part of the formula. This part can be omitted in the case of analysis F. In the `let` case, $e_y = A[\![P]\!]\rho y$ by formula (4.2). Then the returned tree corresponds to the expected translation of the analysis formula $A[\![N]\!]\rho' x \sqcup_A (A[\![M]\!]\rho x) \circ_A (A[\![N]\!]\rho' y)$. In the `let rec` case, we have to introduce unknowns to translate the fixed point computation. These unknowns will be determined by solving equations, thanks to the method described in the next section. We will have $u' = R(f)$. Then the tree is the translation of the formula: $A[\![N]\!]\rho' x \sqcup_A (A[\![M]\!]\rho x) \circ_A R(f)$ which is equivalent to $R(x)$.

In the case of analysis F, our goal is to have a very fast analysis, of worst case complexity $\mathcal{O}(n \log^2 n)$. However, the upper bound $\rho_P \sqcup_A \rho_Q$ in the `if` case, and the instantiation $I_A^{\sigma, \tau_y^{\mathrm{g}}[\tau_{y,i}^{\mathrm{g}}]}(\rho(y)i)$ in the variable case do not fit in this complexity (consider for example a variable y with p parameters, and build a tree of `if` with b branches, each branch returning y. Then the complexity of the algorithm is at least bp for a program of size $b + p$, then it is quadratic). To solve this problem, we only return the full escape information for the parameters of M if M is a `fun` expression. If M is a variable and the containing expression of M is an application $MN_1...N_p$, we return p escape informations for the parameters. Otherwise, we return an empty escape information. If an application does not find the escape information it expected (i.e. ρ_N is empty in the application case), the parameter is assumed to escape. This leads to a reasonable loss of precision, and the complexity bound is then satisfied.

This algorithm is very similar to that of Alain Deutsch [Deu97], except that in his case context transformers could be represented by one single integer. Alain Deutsch has proved that his algorithm is sound and complete.

In our program transformation, we may move `let`s to increase the scope of a variable and allow the stack allocation of its value. That is:

$$C[\texttt{let}\ x = M\ \texttt{in}\ N]$$

may be transformed into

$$\texttt{let}\ x = M\ \texttt{in}\ C[N]$$

if x escapes from N but not from $C[N]$. Therefore, we have to compute the escaping part of x in N, $A[\![N]\!]\rho x$, but also in $C[N]$. However, this transformation must not change the order of evaluation of expressions or the number of times they are evaluated. Therefore, some kinds of expressions cannot be contained in the context $C[]$. Moreover, it is useless to move a `let` when we know that the escape context will not change any more (this is the case for the right hand side of `<-`, since the whole right hand side escapes). To limit the context $C[]$, we have chosen the following heuristic definition. The $[]$ of $C[]$ must not be in the following positions:

$$\begin{array}{l}
\texttt{fun}\ y \to [] \\
\texttt{let rec}\ x = []\ \texttt{in}\ M \\
\texttt{if}\ []\ \texttt{then}\ []\ \texttt{else}\ [] \\
\overline{C_i}(M) \leftarrow [] \\
[]; []
\end{array}$$

This means that if a `let` is in one of these $[]$, it cannot be moved out of it. The outermost expression in which we can move a `let` is said to be the ***control expression*** of this `let` binding.

The escaping part of an expression M in one of its containing expressions N is the composition of the context transformers labelling the path from node n which represents M to the node which represents N. This can be computed by evaluating $\text{eval}(n)$ during the analysis of N. For each expression, we wish to compute its escaping part in the immediately containing expression, which is simply the label of the edge linking n to its father, but also in the control expression. This escape information enables us to decide whether a value can be stack allocated: it must not escape from the control expression. The escaping part with respect to the immediately containing expression will be useful to determine the more precise scope to give to the value. (It will enable us to compute the escaping part of the value with respect to intermediate containing expressions between the original body of the `let` and the control expression. The `let` binding will be moved to the smallest expression which makes sure that the value does not escape.)

In ML, ***tail calls*** (calls which take place just before the return from the function) are optimized: they are replaced by a simple jump to the called function. If we perform a stack allocation in tail position, we have to deallocate the stack allocated value after the return from the tail call. This prevents the normal tail call optimization. To avoid losing this optimization, we compute another context: the escaping part in tail calls. The computation is the same as the preceding one, except that when analyzing a tail call $MN_1...N_n$, we consider that all parameters $N_1, ..., N_n$ and the function M escape. If a value does not escape with respect to this tail call analysis, it can be deallocated before the tail call, and the tail call optimization can be performed as it were without stack allocation. To perform this computation, we add to each context transformer op, represented by a tuple of integers, one integer element i_t, since only the imperative escaping part i differs between the normal escape analysis and the escape analysis for tail call preservation. For example the composition formula becomes:

$$\begin{aligned}
(f_1, i_1, i_1^+, i_{t1}) \circ (f_2, i_2, i_2^+, i_{t2}) = (& f_1 \sqcap f_2, (i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 = \infty \text{ then } i_1^+ \text{ else } 0), \\
& (\text{if } f_2 \sqcup i_2^+ = \infty \text{ then } i_1^+ \text{ else } 0) \sqcup (i_2^+ \sqcap f_1), \\
& (i_{t2} \sqcap f_1) \sqcup i_{t1} \sqcup (\text{if } i_{t2} = \infty \text{ then } i_1^+ \text{ else } 0)).
\end{aligned}$$

Example 4.2 Let us consider again the example of the `map` function and assume that expression `map g a` is in tail position. Let $\texttt{g} : \tau_1 \to \tau_2$. Then with this representation of analysis

F by quadruples of integers, the analysis is coded as follows:

$$F[\![\texttt{map g a}]\!]\rho = \{\texttt{g} \mapsto (\top_f[\tau_2], 0, \infty, \infty), \texttt{a} \mapsto (0, \top_L[\tau_1], \top_L[\tau_{\texttt{a}}], \top_L[\tau_{\texttt{a}}]),$$
$$\texttt{map} \mapsto (\top_f[\tau_{\texttt{map}}], 0, \infty, \infty)\}.$$

Analysis F expresses that the closures of `map` and `g` and the list `a` escape in the tail call since the last element of the quadruple, i_t, is ∞ for `map` and `g`, and $\top_L[\tau_{\texttt{a}}]$ for `a`. On the other hand, the result of `g` and the result of `map` escape when the result of `map g a` escapes (value of f) whereas only the elements of `a` escape even if the result does not escape (value of i) because they are given as parameters to an unknown function.

4.3 Solving of equations

The particular properties of the equations systems enable us to use a very efficient solving algorithm, *a generalization of Dijkstra's shortest paths algorithm* given by Knuth in [Knu77].

We compute the least fixed point of an equations system $Y = \sqcup_i g_i(X_1, \ldots, X_k)$ where functions g_i are inferior: $g_i(x_1, \ldots, x_k) \leq \min(x_1, \ldots, x_k)$. (Notice that $\sqcap$, $\lambda c.\text{if } c = \infty \text{ then } x \text{ else } 0$ and the generalization G_L^τ are inferior, therefore Knuth's algorithm can solve all equations containing these functions and the upper bound. The instantiation does not fit in this framework, we will see at the end of this section how it is handled.) Let G be the grammar whose non-terminals are unknowns X_i and production rules are $Y \longrightarrow g_i(X_1, \ldots, X_k)$. Let T be the terminal alphabet, containing function symbols g, parentheses and comma. Let $L(Y) = \{\alpha \mid \alpha \in T^* \text{ and } Y \longrightarrow^* \alpha\}$ be the language generated by Y. Let $val(\alpha)$ be the value of the string α. The least fixed point is then: $m(Y) = \max\{val(\alpha) \mid \alpha \in L(Y)\}$.

This algorithm uses a priority queue which can be efficiently implemented by a binary heap, or better a Fibonacci heap, as explained in [CLR94]. The algorithm is the following:

Let P be the priority queue.
At the beginning, P contains for each non-terminal the key 0.
foreach equation $Y = g$ (constant), update $\text{key}(Y) = \max(\text{key}(Y), g)$.
while P is not empty,

- Extract the non-terminal Y of maximum key from P. We have then $m(Y) = \text{key}(Y)$
- **foreach** non-terminal Y' which is still in P, update

$$\text{key}(Y') = \max(\text{key}(Y'), g(\text{key}(X_1), \ldots, \text{key}(X_k)))$$

  for each equation $Y' = g(X_1, \ldots, X_k)$ such that $X_1, \ldots, X_k$ are not in P and Y is one of the $X_1, \ldots, X_k$.

The correctness of this algorithm is proved in [Knu77]. Each equation $Y \geq g(X_1, \ldots, X_k)$ is used exactly once, when the last non-terminal parameter of g is extracted from the priority queue P. The idea is therefore to maintain for each non-terminal the list of equations containing it, with a counter indicating the number of non-terminals of this equation which are in P. The equation is applied when the counter becomes zero.

Let us assume that the complexity of the priority queue operations are given by T_{max} for the time of extraction of the maximum and T_{maj} for the time of updating a key. The complexity is then:

- Time for extracting maximums = number of unknowns (denoted by u) $\times T_{max}$
- Time for computing values for updates = total compute time of equations (denoted by t)
- Time for updating keys = number of equations (denoted by e) $\times T_{maj}$
- Time for updating counters = total length of equations = $\mathcal{O}(t)$

hence the total complexity $C = \mathcal{O}(u \times T_{max} + e \times T_{maj} + t)$. For a priority queue implemented by a binary heap: $T_{max} = \mathcal{O}(\log u)$, $T_{maj} = \mathcal{O}(\log u)$ therefore $C = \mathcal{O}(e \times \log u + t)$ and for a Fibonacci heap (amortized time): $T_{max} = \mathcal{O}(\log u)$, $T_{maj} = \mathcal{O}(1)$ therefore $C = \mathcal{O}(u \times \log u + t)$. Knowing the size of equations systems (rarely more than 10 equations) and the fact that in general, the number of equations is of the same order as the number of unknowns, we have thought that the improvement of the complexity of the Fibonacci heap implementation was not worth the effort, all the more so as the constant hidden in the $\mathcal{O}$ is greater for a Fibonacci heap than for a binary heap. We have therefore used a binary heap.

The instantiation operation of analyses L and F does not satisfy Knuth's condition: it is not inferior. To be able to use Knuth's algorithm, we first split the equations system into strongly connected components. Each component is solved separately by Knuth's algorithm. Inside a component, the instantiation is approximated by the constant function equal to the height of the type (when it is not identity). This leads to a loss of precision, but is certainly correct. Between components, we can use the precise instantiation. The loss of precision due to this technique is in fact very small. The Objective Caml language does not contain polymorphic recursion, therefore a type variable introduced in a `let rec f` definition cannot be instantiated inside the definition. Then only a type variable β introduced in another binding (inside the `let rec` definition) can be instantiated inside the `let rec` definition itself. Let us assume that β has been introduced to type a variable `y`. Normally, the escaping part of `y` does not depend on the function `f`, since `y` cannot be given to `f` as a parameter (otherwise, its type would not be a type variable β introduced inside the `let rec` binding), and therefore the instantiation would not be in a strongly connected component of the equations system. However, our analysis can introduce spurious dependencies if the analyzed program builds a data structure using `y`, and then extracts another part of this data structure to give it to `f` as a parameter. In this case, the instantiation can be in a strongly connected component. For example,

```
let rec f x n =
    if n = 0 then x else
        let g(y) = f (fst(x,y)) (n-1) in
            g([x])
in
....
```

In this expression, `f` : $\tau \to$ `int` $\to \tau$, `g` : $\beta \to \tau$. The `g` function is instantiated to type `g` : τ `list` $\to \tau$. A very precise escape analysis would discover that `y` does not escape from `g`, and therefore the analysis of `g` would not depend on unknowns introduced in the analysis of `f` and the instantiation would not be in a strongly connected component. However, if op_1 is the unknown escaping part of `x` in `f`, $op_1 = \rho[\![\texttt{f}]\!]1$, our analysis yields:

$$\rho[\![\texttt{g}]\!]1c = op_1(c \sqcap \top[\tau])$$

because the expression builds a pair `(x,y)` instead of simply passing `x` to `f`. We have to instantiate $\rho[\![\mathtt{g}]\!]1$ to have the escaping part of `[x]`, $I_L^{\sigma,\tau[\beta]}(\rho[\![\mathtt{g}]\!]1)$ where $\sigma = \{\beta \mapsto \tau\ \mathtt{list}\}$, and op_1 of course also depends on this escape information. Then the instantiation is in a strongly connected component. This case is very rare in practice.

4.4 Program transformation

We use a slightly modified version of Objective Caml, which implements an expression $\mathtt{letstack}\ x = M\ \mathtt{in}\ N$ equivalent to $\mathtt{let}\ x = M\ \mathtt{in}\ N$ except that the top-most constructor of x will be stack allocated, and deallocated at the end of the execution of N. We have added a second expression to perform stack allocation: $\mathtt{letstack}'\ x = M\ \mathtt{in}\ N$. This expression is useful when N contains function calls which are in tail position in the function containing N. These function calls are replaced by jumps by tail call optimization. In this case, $\mathtt{letstack}'\ x = M\ \mathtt{in}\ N$ stack allocates the last constructor of x, but deallocates it before the tail calls of N, which enables us to preserve tail call optimizations. When N contains no tail call, $\mathtt{letstack}'$ is equivalent to `letstack`.

Example 4.3 Taken from a Knuth-Bendix completion program:

```
let kb_completion =
  let rec kbrec j rules =
  let rec process failures (k,l) eqs =
   ...
   let enter_rule(left,right) =
     letstack' left_reducible rule =... in
     letstack' right_reduce rule =... in
       ...
       kbrec (j+1) (new_rule::irreds) []
       (k,l) (eqs @ eqs' @ failures)
   in
  ...
```

Here, `left_reducible` and `right_reduce` are deallocated before the recursive tail call to `kbrec`, which avoids a useless increase of the stack size.

An expression of the form $\mathtt{let}\ x = M\ \mathtt{in}\ N$ is transformed into $\mathtt{letstack}\ x = M\ \mathtt{in}\ N$ if the top of x can be stack allocated, i.e. using Theorem 3.56, if we have a certain condition on $A[\![N]\!]\rho x$ which is determined by escape analysis. To be able to perform this transformation, the expression M also has to be of a stack allocatable type (not integer, boolean, character or `unit`), and to be an allocator: a constructor or a standard primitive performing an allocation.

According to command line options, a `letstack` may be forbidden if it prevents a tail call optimization or a recursive tail call optimization. In this case, only `letstack'` is possible, if the escape information allows it. Indeed, if a `letstack` is put in tail position in a recursive loop, the stack size increases at each recursive call whereas with tail call optimization, it would not have increased. This can lead to the program failure by stack overflow, and should therefore be avoided (see Chase's paper [Cha88]). Detecting tail calls is very easy, it is enough to simply scan the expression to see whether the last evaluated part is a function call. Detecting recursive tail calls is slightly more difficult. We use the following safe criterion:

- An expression e is said to perform an unknown tail call if it contains in tail position a call to a function which is not statically known (i.e. it is not a `fun` expression or a variable which refers to a known `fun` expression. Essentially, it has been given to e as a parameter or it is extracted from a data structure) or a call to a function which itself performs an unknown tail call.

- A tail call may be recursive if it is an unknown tail call according to the above definition, or if there is a `let rec` binding containing both the current program point and the declaration of the called function. More precisely, the second case is the following:

  `let rec` $x_1 = M_1$ `and` ... `and` $x_k = M_k$ `in`
  M

  If the current program point is in one of the M_i and the called function is an x_j, or a function defined in the M_j, then the called function may call back x_i and execute M_i again.

If our criterion claims that the tail call cannot be recursive, then it is a call to a function of a preceding declaration block, and all functions called by tail calls are known functions, therefore functions of the same or of a preceding declaration block. Therefore the current function cannot be reached by such a tail call, which is what we wanted.

A `let` structure which defines several variables is split into several `lets` to be able to allocate some variables on the stack if other variables escape (such a splitting is of course not possible for a `let rec`).

The program transformation may also add `lets` to put an expression into the form `let` $x = M$ `in` N and be able to allocate the last constructor of M on the stack. For example, the expression $M_0 M_1 \dots M_n$ can be transformed into

$$\begin{aligned}
\texttt{let } x_0 &= M_0 \texttt{ in}\\
\texttt{let } x_n &= M_n \texttt{ in}\\
&\dots\\
\texttt{let } x_{i+1} &= M_{i+1} \texttt{ in}\\
\texttt{letstack } x_i &= M_i \texttt{ in}\\
x_0 M_1 \dots M_{i-1} x_i \dots x_n
\end{aligned}$$

according to information given by escape analysis: this transformation can be performed for x_i ($i \in [\![1, n]\!]$) if the stack allocation condition of Theorem 3.56 is satisfied after adding the `lets` (which can easily be checked without performing this transformation explicitly: we check that the ith parameter of M_0 does not escape thanks to the value of $A[\![M_0]\!]\rho i$). Moreover, `lets` have to be added for x_0 and for x_j with $j > i$ to avoid changing the order of evaluation: x_j is evaluated before x_i if $j > i$ or $j = 0$. (This is the order of evaluation of the Objective Caml native code compiler. In Objective Caml, the order of evaluation is not specified, but we have preferred not to change it to avoid breaking programs which wrongly rely on it.)

Inlining is also performed to increase stack allocation opportunities. Small functions which allocate their result are automatically inlined which moves the allocator into the `let` binding and allows stack allocation. The inlining is only done when it really leads to stack allocation. Otherwise, the non-inlined version is kept, to avoid transforming the analyzed program too much, and to avoid increasing the code size. An example of inlining is the following:

```
let f x = [x];; hd (f 3)
```

becomes

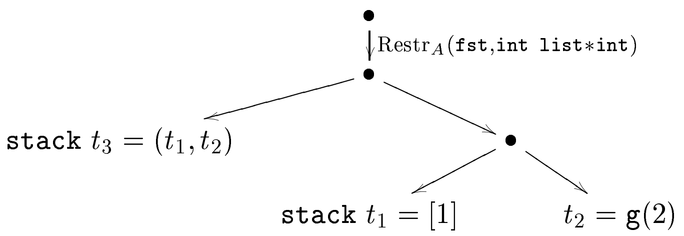
```
hd (let x = 3 in [x])
```

then

```
let x = 3 in letstack %t1 = [x] in hd %t1.
```

Our program transformation algorithm is summarized in Figures 4.4 and 4.5. The program transformation function Transf (Figure 4.5) takes as parameter an expression to transform M and yields the transformed expression and the set of `let` bindings to be put outside the current expression. These `let` bindings are collected in a tree whose edges are labelled by context transformers defining the escaping part of the `let` bindings. These context transformers have been computed for each expression during the analysis phase. Only the most interesting cases have been mentioned in the algorithm.

Consider for example the expression $M = \mathtt{fst}\ ([1], \mathtt{g}(2))$ where `g` is any function from `int` to `int` (in real programs, the function `fst` would be replaced with a more complex function with the same escape behaviour). Assume that the list [1] and the pair constructor do not escape from the control expression (i.e. they can be stack allocated). Transf(M) yields $\mathtt{fst}(t_3)$ with the bindings $\mathtt{stack}\ t_1 = [1]$, $t_2 = \mathtt{g}(2)$ (useful to preserve evaluation order) and $\mathtt{stack}\ t_3 = (t_1, t_2)$ represented by the tree



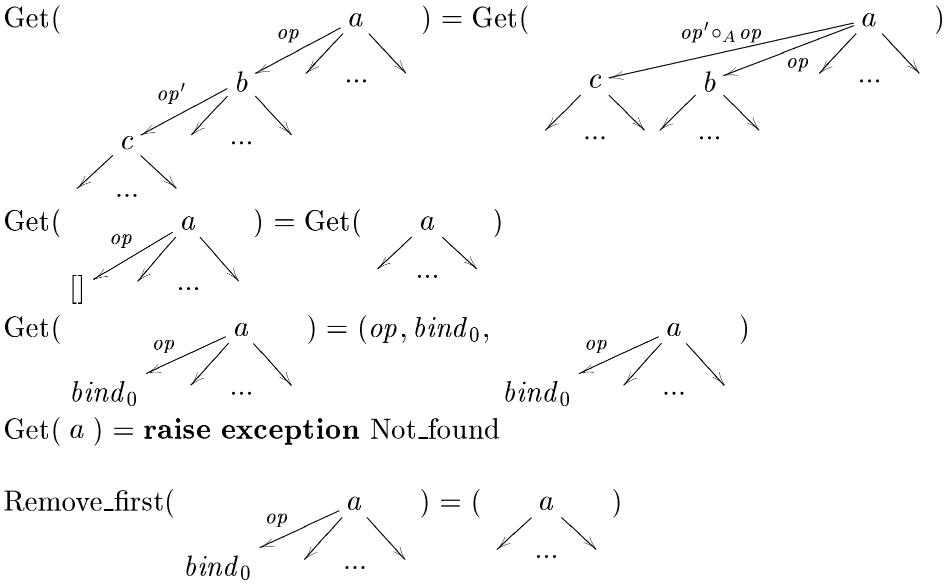
Identity context transformers have been omitted for clarity.

The Get function performs path compression in the tree using Tarjan's algorithm again, to update context transformers as we go up in the abstract syntax tree. It returns the `let` binding that should be inserted in the innermost expression in the program (the last in evaluation order) and the compressed tree. The binding is not removed from the tree. The Get function raises the Not_found exception when the tree contains no binding.

For example, on the above tree, Get will return the binding $\mathtt{stack}\ t_3 = (t_1, t_2)$, with the escape context transformer $\mathrm{Restr}_A(\mathtt{fst}, \mathtt{int\ list} * \mathtt{int})$.

As soon as the last `let` (in evaluation order) does not escape from the current expression, it is extracted from the tree, and inserted into the program by the Put function. This enables us to insert the `letstack`s such that stack allocated data have the shortest possible lifetime, since we do not wait to be in the control expression before writing the `letstack`. The Put function returns the expression with the new bindings that have been inserted into the program and the remaining bindings in the tree. These bindings will be written in an expression containing M.

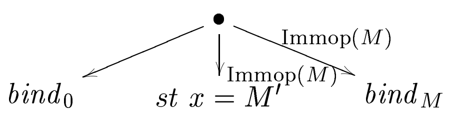
In the above example, Put inserts the first binding into the expression, since the context transformer $\mathrm{Restr}_A(\mathtt{fst}, \mathtt{int\ list} * \mathtt{int})$ proves that it does not escape. The obtained expression is then $\mathtt{letstack}\ t_3 = (t_1, t_2)\ \mathtt{in\ fst}\ t_3$. The program tries to insert more bindings by recursively calling Put, but the next binding $\mathtt{stack}\ t_1 = [1]$ escapes: it will be stack allocated in a greater expression. Therefore, Put returns the expression $\mathtt{letstack}\ t_3 = (t_1, t_2)\ \mathtt{in\ fst}\ t_3$.



Put$(M, bind_M)$ =
  try
    $(op, bind_1, bind_M) = \mathrm{Get}(bind_M)$
    if op allows stack allocation **or**
      $bind_1$ is not stack allocated **then**
      return Put(`let` $(bind_1)$ `in` M,
                Remove_first$(bind_M)$)
    else
      return $(M, bind_M)$
  with exception Not_found$\rightarrow$
    return $(M, bind_M)$

Put_all$(M, bind_M)$ =
  try
    $(op, bind_1, bind_M) = \mathrm{Get}(bind_M)$
    return Put_all(`let` $(bind_1)$ `in` M,
                Remove_first$(bind_M)$)
  with exception Not_found$\rightarrow$
    return M

Transf_exp(M) = **return** Put_all(Transf(M))

Transf_stack$(bind_0, M, decompose)$ =
  st = **if** M can be stack allocated (i.e. M is a constructor or a `fun` expression,
      and it does not escape from its control expression) **then** "`stack`" **else** ""
  $(M', bind_M) = \mathrm{Transf}(M)$
  if st ="" **then** $(M', bind_M) = \mathrm{Put}(M', bind_M)$
  if $decompose$ **or** st ="`stack`" **then**
    return ( , x, true)

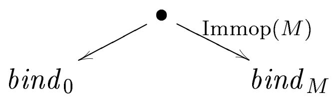
  else
    return ( , M', $bind_M$ not empty)

Figure 4.4: Program transformation algorithm (1).

Transf_rands$(M_1, ..., M_n) =$
  $bind_0 = [], decompose =$ false
  for $i = 1$ **to** n **do**
    $(bind_0, M'_i, decompose) =$ Transf_stack$(bind_0, M_i, decompose)$
  return $((M'_1, ..., M'_n), bind_0, decompose)$

Transf$(M) =$
  match M **with**
    $y \to$ **return** $(y, [])$
    $|N\ P \to (bind_P, P', decompose) =$ Transf_stack$([], P, \text{false})$
        $(bind_N, N', decompose) =$ Transf_stack$(bind_P, N, decompose)$
        return $(N'\ P', bind_N)$
    $|$(`fun` $y \to N$) $\to$ **return** (`fun` $y \to$ Transf_exp$(N), [])$
    $|$`let` $y = N$ `in` $P \to (bind_P, P', decompose) =$ Transf_stack$([], P, \text{false})$
        $(bind_N, N', decompose) =$ Transf_stack$([], N, decompose)$
        return $(P',$

$$\begin{tikzcd} & \bullet \arrow[dl] \arrow[d, "\mathrm{Immop}(N)"] \arrow[dr, "\mathrm{Immop}(N)"] & \\ bind_P & y = N' & bind_N \end{tikzcd}$$

        $)$
    $|$(`let rec` $y =$ `fun` $x \to N$ `in` P) $\to (bind_P, P', decompose) =$ Transf_stack$([], P, \text{false})$
        $N' =$ Transf_exp(N)
        $st =$ **if** `fun` $x \to N$ can be stack allocated **then** "`stack`" **else** ""
        return $(P',$

$$\begin{tikzcd} & \bullet \arrow[dl] \arrow[dr, "\mathrm{Immop}(N)"] & \\ bind_P & & st\ \texttt{rec}\ y = \texttt{fun}\ x \to N' \end{tikzcd}$$

        $)$
    $|C(M_1, ..., M_n) \to ((M'_1, ..., M'_n), bind_0, decompose) =$ Transf_rands$(M_1, ..., M_n)$
        return $(C(M'_1, ..., M'_n), bind_0)$

Figure 4.5: Program transformation algorithm (2).

The Put_all function handles the case when M is a control expression. No `let` binding can then moved out of M, and we are sure that if a binding has been inserted in the tree and marked for stack allocation, it can be stack allocated in the control expression (definition of st in Transf_stack). Therefore, we can write all `let` bindings in the program, and return the transformed expression.

The Transf_exp function performs the program transformation when M is a control expression, by calling Put_all.

The Transf_stack function handles the other cases. It takes as parameters the bindings $bind_0$ that should be evaluated after M, an expression M to transform, and a flag *decompose* which is true when the whole M expression should be transformed into a binding. The goal of this flag can be better understood on an example. Assume that we transform $M\ P$, and that a value created in a subexpression N of P escapes from P but not from $M\ P$. We want to write something like `letstack` $y = N$ `in` $M\ P'$. But this would not preserve the order of evaluation (N is evaluated before M in the transformed expression, and N was evaluated after M in the original expression). Therefore, we have to modify the expression again, and generate: `let` $x = M$ `in` `letstack` $y = N$ `in` $x\ P'$. Essentially, *decompose* is true when there exists a binding b (such as `stack` $y = N$ in the example above) that should be evaluated after M and that should have a scope greater than the expression $C[M]$ immediately containing M for stack allocation purposes. These two contradictory requirements are reconciled by decomposing M into the variable x and the binding $x = M$.

The Transf_stack function first determines whether M will be stack allocated. Then it transforms M, perhaps creating new bindings $bind_M$. If M can be stack allocated, we do not put the bindings of M immediately in the program, to avoid creating expressions such as `letstack` $x =$ `let` $(bind_M)$ `in` M' `in` ... in which the stack allocated constructor is not the outermost expression of the `letstack` binding. We prefer writing: `let` $(bind_M)$ `in letstack` $x = M'$ `in` ... The last step is to create the new tree of bindings and the transformed expression.

- If M can be stack allocated, or M should be decomposed into a binding (the flag *decompose* is true), then the transformed expression is simply a new variable x, and everything is put in the bindings. In these bindings, Immop(M) denotes the context transformer representing the escaping part of M in its immediately containing expression. When drawing trees, the identity context transformer has been omitted.

- Otherwise, the transformed expression is M'.

The last component of the result of Transf_stack is the new value of the *decompose* flag, which is true either if it was already true, or if a new binding has been created.

For example, consider Transf_stack($\bullet, ([1], \mathtt{g}(2)), \mathrm{false}$) when the pair constructor can be stack allocated. Then $st =$ `stack`. We have

$$\mathrm{Transf}(M) = (\,(t_1, t_2),\quad \begin{tikzcd}[ampersand replacement=\&] \& \bullet \arrow[dl] \arrow[dr] \& \\ \mathtt{stack}\ t_1 = [1] \& \& t_2 = \mathtt{g}(2) \end{tikzcd}\quad).$$

The resulting output is therefore (b, t_3, true) where b is the tree above, with one more binding `stack` $t_3 = (t_1, t_2)$ on the left.

Transf_rands($M_1, ..., M_n$) applies the Transf_stack function to several expressions M_i (such that M_n is evaluated first, M_1 last). It collects the bindings in $bind_0$. The *decompose* flag is false at the beginning and becomes true as soon as a binding has been put in $bind_0$. Indeed, to avoid changing the evaluation order, if a binding is inserted when transforming M_i, the expressions M_j, $j > i$ evaluated before M_i should be transformed into a binding `let` $x_j = M_j$ `in` $...x_j...$ such that M_j is still evaluated before the bindings generated by M_i. Transf_rands($M_1, ..., M_n$) returns the transformed expressions, the bindings to insert and the *decompose* flag which is true if and only if a binding has been inserted.

Example 4.4 Let us consider the following function:

```
let rec tak p =
  if p.x>p.y then
    tak
     (letstack %t1={x = p.x-1, y = p.y, z = p.z} in tak %t1,
      letstack %t2={x = p.y-1, y = p.z, z = p.x} in tak %t2,
      letstack %t3={x = p.z-1, y = p.x, z = p.y} in tak %t3)
  else
      p.z
```

The intermediate data structures `%t1`, `%t2` and `%t3` are each deallocated before allocating the next one, which reduces the stack size. (Experiments confirm that the maximum stack allocated size is 3 times smaller with the above program than when all `letstack`s are at the beginning.)

4.5 Complexity of the analyses

Let n be the size of the program. For analysis F, the number of calls to eval and to link is bounded by $\mathcal{O}(n)$, therefore by Theorem 1 of [Tar79], the length of path compressions is in the worst case $\mathcal{O}(n \log n)$. The number of generated equations and unknowns is therefore bounded by $\mathcal{O}(n \log n)$. The solving time is then at most $\mathcal{O}(n \log^2 n)$ (computing strongly connected components of a graph is linear, therefore dominated by Knuth's algorithm).

For analysis L, the number of calls to link and to eval is bounded by $\mathcal{O}(n^2)$ because the number of free variables of `fun` $y \rightarrow M$ is at most $\mathcal{O}(n)$ and there may be $\mathcal{O}(n)$ closures in the program. The length of paths compressions is then at most $\mathcal{O}(n^2 \log n^2) = \mathcal{O}(n^2 \log n)$. The number of generated equations and unknowns is within the same bound, and the solving time by Knuth's algorithm is bounded by $\mathcal{O}(n^2 \log^2 n)$.

The computation of type heights is dominated by the size of types (more precisely, by the sum of sizes of instances of types declarations. I.e. each type declaration is reanalyzed when it is instantiated into a new type, and the analysis is linear).

The program transformation costs at most $\mathcal{O}(n \log n)$ following the same reasoning on Tarjan's algorithm.

Finally, the complexity of the analysis is $\mathcal{O}(n \log^2 n + t)$ for F, $\mathcal{O}(n^2 \log^2 n + t)$ for L, where t is the compute time of type heights. The quadratic behavior due to the enumeration of free variables of closures, or to the upper bound in the `if` case or to the instantiation rarely appears. In practice, the complexity is more likely to be $\mathcal{O}(n \log^2 n + t)$ for L.

When inlining is activated, the complexity obviously increases. Let I be the maximal size of an inlined function (can be chosen by the user). The size of the resulting program is at most $n' = \mathcal{O}(nI)$, and the analysis time is $\mathcal{O}(nI + n' \log n' + n \log^2 n)$ for F, $\mathcal{O}(nI + n' \log n' + n^2 \log^2 n)$ for L. (The inlined program is only post-transformed. All operations done for inlining are linear if the search time of an identifier in a table is considered as constant.)

4.6 A few points to note

Escape analysis is modular In the Objective Caml language, declarations are grouped by blocks `let` $x = M$ or `let rec` $x_1 = M_1$ `and` ... `and` $x_i = M_i$. Once the analysis of such a block is complete, we can solve the equations, and compute the escape information for each expression of the block. Being able to solve equations separately for each block is also a reason why the analysis scales well. To analyze subsequent blocks, we do not need to keep the whole equations system. It is enough to add the escape information for the declared functions to the environment ρ and to discard all other information.

This remark is a fortiori true between different modules. Each module is analyzed separately, knowing only the escape transformers for the functions declared in imported modules. The modules have to be analyzed in dependency order (the same ordering as for the compilation). To be able to take into account escape information computed in imported modules, the analyzer writes the escape information to the disk at the end of the analysis of a module, and reads it when the module is used.

When a module is changed, all dependent modules must be reanalyzed, since the escape information may have changed. Even without escape analysis, the Objective Caml native code compiler needs to recompile dependent modules when a module is changed, so this requirement is not an issue.

Adapting the garbage collector (GC) We first describe the GC of Objective Caml.

The memory space in which values are allocated (the heap) is split into two parts. A small 128 kb part named ***minor heap***, and a larger part named ***major heap*** which can grow as needed. We also say that the GC has two ***generations***. The minor heap is the young generation, the major heap the old generation.

All values (except very large arrays) are allocated in the minor heap, one after the other, until the minor heap is full. Then, some garbage collection has to be performed. The GC of the minor heap is a ***copy collector***. Reachable data are copied from the minor heap into the the major heap. More precisely, data considered as reachable are data reachable from ***roots***: registers of the CPU, values in call frames (in the call stack), pointers from the major heap to the minor heap. We can prove that such pointers can only be created by an assignment operation. They can be recorded by instrumenting each assignment to store the assigned address in a list. Once values have been copied into the major heap, the minor heap can be emptied, and a new allocation cycle begins.

The major heap uses a ***mark and sweep collector***. Reachable values are first marked. The GC follows pointers from the roots (registers of the CPU and values in call frames), and changes a bit in the header of the values it can reach. This is the marking phase. Then the whole major heap is scanned. Unmarked values are added to a list of free blocks, and marked values are left unchanged. This is the sweeping phase. In Objective Caml, the mark and sweep collector is ***incremental***: a small part of the major garbage collection is performed after each minor collection. Such a part is called a slice.

This GC is exact: it can distinguish integers from pointers. The lower bit of a pointer is 0. The lower bit of an integer is 1, and only the 31 (or 63 on 64 bits CPUs) higher bits are significant. The GC can therefore follow only pointers when searching reachable data.

This GC is very efficient for fast allocating languages such as ML, because most values have a short lifetime, and are therefore unreachable at the time of the minor collection. Such data are not scanned by the minor GC, and have therefore no cost for this GC. They would have a cost with a mark and sweep collector, because they would be scanned by the sweeping phase. However, the assignment operation becomes more costly with such a GC, because it has to keep track of pointers from the major to the minor heap. This is not a problem for ML, since assignments are very rare.

This GC is described in full detail in [Dol95].

We had to modify the garbage collector to adapt it to stack allocated data, and make sure that managing stack allocated data is as efficient as possible. The naive way of adapting the GC would be to consider all stack allocated data as roots. However, this would lead to copying dead data which are reachable from stack allocated data. Therefore, we have modified the minor GC so that it follows pointers through stack allocated data in the same way as through heap allocated data. Only reachable data are then copied to the major heap.

In contrast, for the marking phase of the major GC, stack allocated data are still considered as roots. The major GC cannot manage stack allocated data in the same way as heap allocated data. Indeed, assume that a slice of the major GC marks data D on the stack as reachable but does not yet scan data reachable from D. Then, D can become unreachable, and be deallocated before the next marking slice. The value D can then be replaced by any binary value. This binary value can look like a pointer towards any location in the heap, even a location in the middle of an object. The next slice of marking will find this random binary value and will start marking random parts of the heap and finding incorrect value headers. This is why we had to consider stack allocated data as roots for the major GC.

4.7 Conclusion

We have given very efficient implementations of our escape analyses based on integers. However, we had to perform some more approximations with respect to the theoretical part (Section 3.6). Here is a summary of these approximations:

- The generalization is approximate to have a simple general form for context transformers (see page 73) for all integer escape analyses;
- The instantiation inside a strongly connected component is replaced with a constant to be able to apply Knuth's algorithm, for analyses L and F;
- The escape information for the parameters of functions is removed except for variables or expressions which are syntactically a function, in analysis F.

We have already noticed that these approximations have been chosen so that they do not lead to losing two much precision. We shall see in the benchmarks that even the most approximate analysis is still precise.

Chapter 5

Implementation of analysis T

Our implementation of analysis T uses an algorithm which decides graph accessibility, due to Reps, Sagiv and Horwitz [RSH94]. The first step is therefore to build a graph, such that a piece of data escapes according to analysis T if and only if the corresponding node in the graph is reachable by certain paths, called realizable paths. In Appendix D, we prove that the graph built by our algorithm satisfies this property. The second step is to compute realizable paths. We also prove that the algorithm correctly computes realizable paths in Appendix D. Finally, our algorithm decides to allocate a value on the stack if and only if analysis T also takes this decision.

Moreover, several optimizations are very important to be able to analyze large programs.

- Once the analysis of a declaration of recursive functions is complete, the graph of these functions can be reduced. This yields a simpler graph containing the escape information for the parameters of these functions, without the now useless information about the escaping part of each subexpression of these functions.
- The graph is compacted by representing several nodes and edges by one single node and edge, thus reducing the memory footprint and the compute time.
- The handling of recursive types is also optimized by using a single node for a strongly connected component.

These optimizations are explained in more detail at the end of this chapter.

5.1 Building of the graph

Our algorithm, which is detailed below, is based on the demand-driven [DGS97] version of Reps, Sagiv and Horwitz's algorithm [RSH94, HRS95]. However there are important differences between this algorithm and our framework. These differences will be detailed in paragraphs "Comparison with [RSH94]".

5.1.1 Definition of the exploded super-graph

We recall the general framework of [RSH94] with the necessary changes to adapt it to our situation. The goal of [RSH94] is to solve the so called IFDS problems, which can be defined as follows:

Definition 5.1 (Interprocedural, finite, distributive subset problems (IFDS)) An IFDS problem is

$$IP = (Node_0, Edge_0, D, Fun)$$

where $(Node_0, Edge_0)$ is a directed graph, which is called the ***super-graph***: $Node_0$ is the set of nodes, $Edge_0$ is the set of edges. $Edge_0 \subseteq Node_0 \times Node_0 \times \{\iota, (_i,)_i | i \in \mathbb{N}\}$. The edge (n, m, l) from n to m labelled l will be denoted by $n \xrightarrow{l} m$. n is the source of the edge, m is its target. In imperative programming, a node represents a program point, an edge represents the fact that one can go from the program point represented by its source to the program point represented by its target by executing one instruction. In our case, a node will correspond to an expression M or a variable x, and an edge will link a variable or a subexpression to the expression(s) containing it.

Ordinary edges have the empty label ι. They represent the control-flow inside a function. The edges labelled $(_i$ are said to be ***call edges***, the edges labelled $)_i$ are ***return edges***. When a function is called, an edge labelled $(_i$ is created from the call point to the entry point of the called function, and an edge labelled $)_i$ is created from the return point of the function to the successor of the call point. In a valid path, if on the call, we have traversed an edge labelled $(_i$, on the return, we must traverse an edge labelled $)_i$ for the same i. This makes sure that a function always returns to the point from which it has been called.

By convention, if there is a call edge from n_1 to n_2, $n_1 \xrightarrow{(_i} n_2$, n_1 is a call node and will be denoted by c, and n_2 is the entry node of function, and will be denoted by s (for start). Similarly, a return edge will be denoted $e \xrightarrow{)_i} r$, e being the exit node of the function and r being the return node. However, the same node may for example be an exit node and a return node (seen from different edges). This notation is therefore only an indication to make the understanding of the role of the different nodes easier.

$D : Node_0 \to \textit{Finite sets}$ yields for each node $n \in Node_0$ of the graph a finite set D_n. These sets D_n are pairwise disjoint.

$Fun : Edge_0 \to \textit{Distributive functions}$ (distributive is a synonym for additive) maps each edge $e = n \xrightarrow{l} m \in Edge_0$ to a distributive function $Fun(e) \in \mathcal{P}(D_m) \to \mathcal{P}(D_n)$. (The distributive function goes from D_m to D_n whereas the edge goes from n to m because escape analysis is a backward analysis.)

The link with the definition of analysis T is as follows: each node $n \in Node_0$ represents a value $v(n) = T[\![M]\!]\rho$ or $\rho[\![x]\!]$. If $v(n) = T[\![M]\!]\rho$, D_n is a copy of $\mathrm{Nodes}(\mathrm{Gr}(\tau_M))$, and if $v(n) = \rho[\![x]\!]$, D_n is a copy of $\mathrm{Nodes}(\mathrm{Gr}(\tau_x^{\mathrm{g}}))$. Two nodes n and m are linked by an edge $e : n \xrightarrow{l} m$ if the value associated with m depends on the value associated with n, and $Fun(e)$ is the function which expresses this dependency. Essentially,

$$\forall x \in Var, v(m)(x) \geq \sqcup_{l,n} v(n)(x) \circ Fun(n \xrightarrow{l} m)$$

except that we take into account the fact that not all paths correspond to real execution paths (a function always returns to its call site).

Example 5.2 Let us consider the following list reversal function:

```
let rec rev_append accu l =
  match l with
    [] -> accu
  | a::l' -> rev_append (a::accu) l'

let rev l = rev_append [] l
```

In our small language, `rev_append` is translated into

```
let rec rev_append = fun accu -> fun l ->
  if null l then
    accu
  else
    let a = hd l in
    let l' = tl l in
    rev_append (a :: accu) l'

let rev = fun l -> rev_append [] l
```

The corresponding IFDS problem is represented in Figure 5.1. The edge label ι has been omitted. Moreover, an edge e has been labelled by the function $Fun(e)$ which is associated with it. The identity operation has been omitted to simplify the figure. Edges go from the parameters to the result of the analyzed functions (`rev` and `rev_append`) but from the result to the parameter of context transformers, because escape analysis is a backward analysis.

Definition 5.3 (Valid path) A path q from node m to node n is a sequence of edges, denoted by $e_1 : e_2 : \ldots : e_j$ (or $m : e_1 : e_2 : \ldots : e_j : n$ to emphasize the source m and target n of the path), such that e_i is an edge of the graph and, if $e_i = s_i \xrightarrow{l_i} t_i$, for all $i, 1 \leq i \leq j-1$, $t_i = s_{i+1}$. The concatenation of paths q_1 and q_2 will be denoted by $q_1 : q_2$. The empty path from m to m is $m : \epsilon : m$.

A path q is a same level interprocedurally valid path if it is generated by one of the following rules, where $SLIVP(m,n)$ denotes the set of same level interprocedurally valid paths from m to n and $SLIVPEL(m,n)$ is the set of elementary paths of $SLIVP(m,n)$ (an $SLIVP$ path is the concatenation of a certain number of $SLIVPEL$ paths).

$$m \xrightarrow{\iota} n \in SLIVPEL(m,n)$$
$$q \in SLIVP(n,p) \Rightarrow m \xrightarrow{(_i} n : q : p \xrightarrow{)_i} r \in SLIVPEL(m,r)$$
$$m : \epsilon : m \in SLIVP(m,m)$$
$$q_1 \in SLIVP(m,n), q_2 \in SLIVPEL(n,p), \text{there exists no path } m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n'$$
$$\Rightarrow q_1 : q_2 \in SLIVP(m,p)$$

A path q is interprocedurally valid if it is generated by the following rules, where $IVP(m,n)$

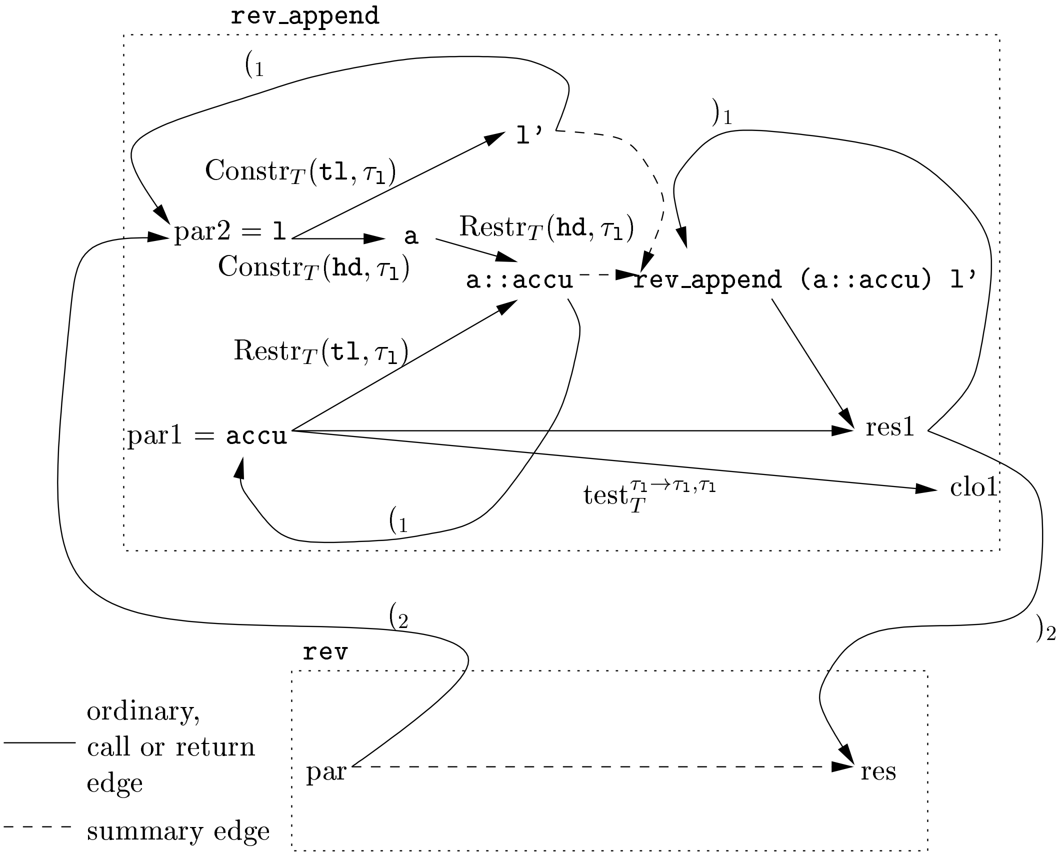


Figure 5.1: IFDS problem associated with the list reversal function. The dashed edges are summary edges which will be added when solving the IFDS problem. They represent the effect of function calls. They will be studied in Section 5.2. The "par" node represents the parameter of `rev`, "par1" the first parameter of `rev_append`, "par2" the second one, "res" the result of `rev` and "res1" the result of `rev_append`. The "clo1" node represents the closure created when `rev_append` is applied to one parameter only.

denotes the set of interprocedurally valid paths from m to n.

$$q \in \mathit{SLIVP}(m,n), \text{there exists no path } m' \stackrel{(_i}{\to} m : q : n \stackrel{)_i}{\to} n' \Rightarrow q \in \mathit{IVP}(m,n)$$

$$q_1 \in \mathit{SLIVP}(m,n), q_2 \in \mathit{IVP}(p,r), \text{there exists no path } m' \stackrel{(_i}{\to} m : q_1 : n \stackrel{)_i}{\to} n'$$
$$\Rightarrow q_1 : n \stackrel{(_i}{\to} p : q_2 \in \mathit{IVP}(m,r)$$

Paths in *SLIVP* correspond to paths which start in a function and return to the same function: the calls $(_i$ and returns $)_i$ must match. Paths in *IVP* correspond to paths which start in a function f and end in a function g called from f. For such paths, a return must always correspond to a previous call, but there may exist calls from which we do not return (last rule).

Comparison with [RSH94] This definition differs from [RSH94] by the condition "there exists no path $m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n'$". In [RSH94], it is assumed that the end nodes of procedures are not linked by other edges than return edges. This hypothesis is not satisfied in our framework: we have in parallel edges labelled ι and edges labelled $)_i$ in the graph. Indeed, a function $y = \texttt{fun}\ z \to M$ may escape either because the closure escapes (ι edges) or because it is applied (edges labelled $)_i$). This makes the condition "there exists no path $m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n'$" necessary. This condition means that if a node m is the entry node of a function (i.e. m represents a parameter of the function, that is there exists an edge $m' \xrightarrow{(_i} m$) then when exiting the function by node n we take a return edge, and not an ordinary edge. When determining the escaping part of the parameters of a function, the function is applied, hence its closure does not escape: we should therefore exit the function by returning from it (by an edge labelled $)_i$, not by an ordinary edge).

This is the most important difference between our framework and [RSH94].

Definition 5.4 (Path function) Let $IP = (Node_0, Edge_0, D, Fun)$ be an IFDS problem.

Let $q = e_1 : e_2 : \ldots : e_j$ be a non-empty path. The path function associated with q is $pf_q = Fun(e_1) \circ \cdots \circ Fun(e_j)$. If q is an empty path, $pf_q = \mathrm{Id}$.

Definition 5.5 (Solution of an IFDS problem) Let $IP = (Node_0, Edge_0, D, Fun)$ be an IFDS problem. The solution of IP is the upper bound of the path functions over all valid paths:

$$MVP_{m,n} = \sqcup_{q \in IVP(m,n)} pf_q$$

Our goal in the following will be to compute these solutions. To achieve this goal, we first define another graph, called the exploded super-graph, such that the solution can be computed only by testing the reachability of nodes of the exploded super-graph. The first step is to translate a distributed function into a graph. Then we will build the exploded super-graph by combining the graphs representing the transfer functions $Fun(e)$ corresponding to each edge e.

Definition 5.6 (Representation of a distributive function) Let f be a distributive function of $\mathcal{P}(D_n) \to \mathcal{P}(D_m)$. The representation of f, denoted by R_f, is a pair containing a part of D_m and a binary relation between elements of D_m and of D_n:

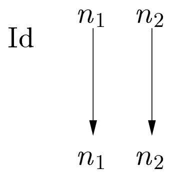
$$R_f \in \mathcal{P}(D_m) \times \mathcal{P}(D_m \times D_n)$$
$$R_f = (f(\emptyset), \{(y,x) | y \in f(x) \text{ and } y \notin f(\emptyset)\})$$

In the graph, $R_f = (E_f, F_f)$ will be represented by marking (by a big black dot) nodes of E_f and drawing edges $y \to x$ for each pair $(y,x) \in F_f$.

Comparison with [RSH94] In [RSH94], the image of $\emptyset$ is represented by linking the nodes of the image to a particular node $\Lambda_n \in D_n$.

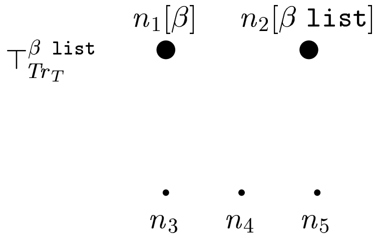
We use a slightly different representation of distributive functions, which enables us to take into account more efficiently the image of $\emptyset$. (Our algorithm can determine that it is included in the result without traversing the whole graph.)

Example 5.7 The identity function $\mathrm{Id} : \mathcal{P}(\{n_1, n_2\}) \to \mathcal{P}(\{n_1, n_2\})$ defined by $\mathrm{Id}(S) = S$ is represented by $(\emptyset, \{(n_1,n_1), (n_2,n_2)\})$.

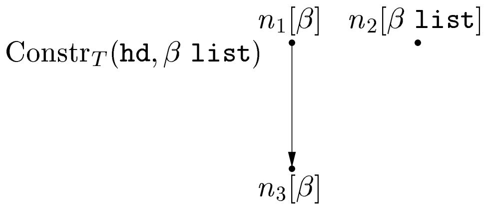


Notice that on the graphical representation the image domain is above the source domain. The image domain is the starting point of arrows contrary to what we could think. This comes from the fact that escape analysis is a backward analysis, therefore predicate transformers go in the opposite direction with respect to a forward analysis.

The constant function $\top_{Tr_T}^{\beta\ \mathtt{list}} : \mathcal{P}(\mathrm{Nodes}(\mathrm{Gr}(\tau))) \to \mathcal{P}(\mathrm{Nodes}(\mathrm{Gr}(\beta\ \mathtt{list})))$ equal to $\mathrm{Nodes}(\mathrm{Gr}(\beta\ \mathtt{list}))$ is represented by $(\mathrm{Nodes}(\mathrm{Gr}(\beta\ \mathtt{list})), \emptyset)$.



On the graph below, the construction $\mathrm{Constr}_T(\mathtt{hd}, \beta\ \mathtt{list}) : \mathcal{P}(\mathrm{Nodes}(\mathrm{Gr}(\beta))) \to \mathcal{P}(\mathrm{Nodes}(\mathrm{Gr}(\beta\ \mathtt{list})))$ is represented by $(\emptyset, \{(n_1[\beta], n_3[\beta])\})$.



Definition 5.8 (Interpretation of a representation) The interpretation of the representation $R = (E, F) \in \mathcal{P}(D_m) \times \mathcal{P}(D_m \times D_n)$ is

$$\begin{aligned} &[\![R]\!] : \mathcal{P}(D_n) \to \mathcal{P}(D_m) \\ &[\![R]\!] : \lambda X.E \cup \{y \mid \exists x \in X \text{ such that } (y, x) \in F\}. \end{aligned}$$

Theorem 5.9 $[\![R_f]\!] = f$

Proof: Similar to [RSH94, Theorem 3.3]. □

Definition 5.10 (Composition of representations) The composition of two representations $R_f = (E_f, F_f) \in \mathcal{P}(D_1) \times \mathcal{P}(D_1 \times D_2)$ and $R_g = (E_g, F_g) \in \mathcal{P}(D_2) \times \mathcal{P}(D_2 \times D_3)$ is

$$\begin{aligned} &R_f; R_g \in \mathcal{P}(D_1) \times \mathcal{P}(D_1 \times D_3) \\ &R_f; R_g = (E_f \cup \{x \mid \exists y \in E_g, (x, y) \in F_f\}, \\ &\qquad \{(x, y) \in D_1 \times D_3 \mid \exists z \in D_2, (x, z) \in F_f, (z, y) \in F_g\}). \end{aligned}$$

When representations are composed, the edges are therefore concatenated: the edges $x \to y$ and $y \to z$ yield $x \to z$. Composing several transfer functions yields to considering paths.

Theorem 5.11 $[\![R_f; R_g]\!] = f \circ g$

Proof: Similar to [RSH94, Theorem 3.5]. □

The exploded super-graph is built by replacing each node n of the super-graph by its domain D_n and each edge e of the super-graph by the representation of the corresponding function $Fun(e)$. This is formalized in the following definition:

Definition 5.12 (Exploded super-graph) Let $IP = (Node_0, Edge_0, D, Fun)$ be an IDFS problem. The exploded super-graph of IP, denoted by $G^{\#}_{IP}$, is:

$$G^{\#}_{IP} = (Node, Edge, Mark)$$

with

$$\begin{aligned}
Node &= \cup_{n \in Node_0} D_n \\
Edge &= \{d_1 \xrightarrow{l} d_2 | \exists e = n \xrightarrow{l} m \in Edge_0, (d_1, d_2) \in F_{Fun(e)}\} \\
Mark &= \cup_{e \in Edge_0} E_{Fun(e)}
\end{aligned}$$

where $R_{Fun(e)} = (E_{Fun(e)}, F_{Fun(e)})$.

The successors of a node n are defined as nodes reachable from n by following one edge of $Edge$ labelled ι, i.e.: $\text{Succ}(n) = \{m \in Node | n \xrightarrow{\iota} m \in Edge\}$.

The set of called nodes associated with a node c is the set Callee(c) of target nodes of call edges starting from c: $\text{Callee}(c) = \{s | \exists i, c \xrightarrow{(_i} s \in Edge\}$.

The set of return edges associated with node e is Ret(e), the set of return edges of $Edge$ starting from node e: $\text{Ret}(e) = \{e \xrightarrow{)_i} r \in Edge\}$.

The set of call nodes corresponding to a return edge $e \xrightarrow{)_i} r$ and a node s is the set Callers(i, s) of nodes c such that there exists a call edge $c \xrightarrow{(_i} s$ in $Edge$: $\text{Callers}(i, s) = \{c | c \xrightarrow{)_i} s \in Edge\}$.

The set $Mark$ is said to be the set of marked nodes. We define Mark(n) by Mark(n) if and only if $n \in Mark$.

Definition 5.13 (Realizable path) A path q in the exploded super-graph $G^{\#}_{IP} = (Node, Edge, Mark)$ is said to be a same-level realizable path if it is generated by the following rules, where $SLRP(m, n)$ is the set of same-level realizable paths from m to n:

$$m \xrightarrow{\iota} n \in SLRPEL(m, n) \tag{5.1}$$

$$q \in SLRP(n, p) \Rightarrow m \xrightarrow{(_i} n : q : p \xrightarrow{)_i} r \in SLRPEL(m, r) \tag{5.2}$$

$$m : \epsilon : m \in SLRP(m, m) \tag{5.3}$$

$$\begin{aligned}
&q_1 \in SLRP(m, n), q_2 \in SLRPEL(n, p), \text{there exists no path } m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n' \\
&\qquad \Rightarrow q_1 : q_2 \in SLRP(m, p)
\end{aligned} \tag{5.4}$$

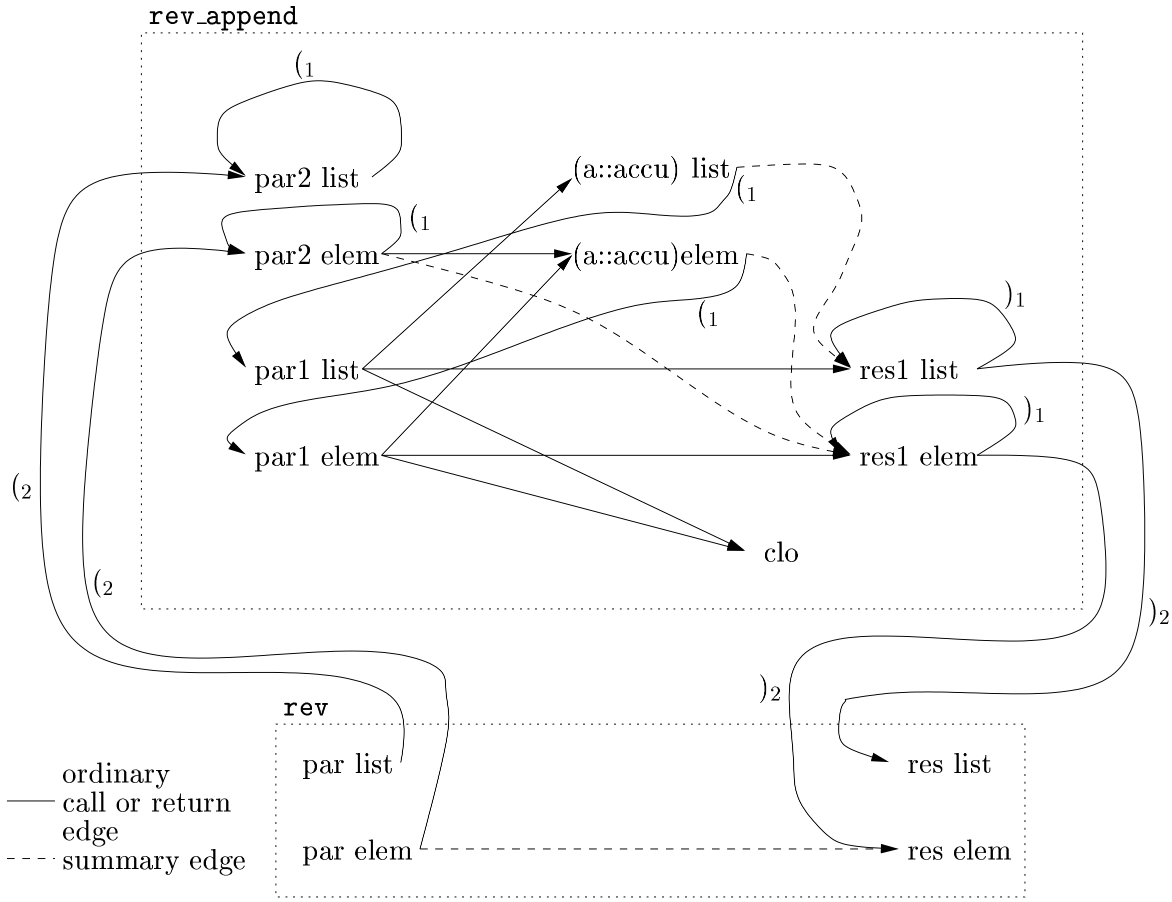


Figure 5.2: Let us consider again Example 5.2, Figure 5.1. The exploded super-graph is represented above. The domain D_n corresponding to data of type β `list` contains two nodes, denoted by "n list" (corresponds to the node labelled by type β `list` in Gr(β `list`); it represents the spine of the list) and "n elem" (corresponds to the node labelled β in Gr(β `list`); it represents the elements of the list).

A path q is said to be realizable if it is generated by the following rules, where $RP(m, n)$ is the set of realizable paths from m to n.

$$q \in SLRP(m,n), \text{there exists no path } m' \overset{(_i}{\rightarrow} m : q : n \overset{)_i}{\rightarrow} n' \Rightarrow q \in RP(m,n) \tag{5.5}$$

$$\begin{aligned} &q_1 \in SLRP(m,n), q_2 \in RP(p,r), \text{there exists no path } m' \overset{(_i}{\rightarrow} m : q_1 : n \overset{)_i}{\rightarrow} n' \\ &\quad \Rightarrow q_1 : n \overset{(_i}{\rightarrow} p : q_2 \in RP(m,r) \end{aligned} \tag{5.6}$$

Realizable paths and valid paths formalize the same idea: the fact that not every path really corresponds to an execution path. The name "realizable" will be used in connection with the exploded super-graph, whereas valid paths will be paths in the super-graph.

It is proved in the framework of [RSH94] that reachability by realizable paths yields the solution of IFDS problems [RSH94, Theorem 3.10]. Adapted to our framework, this result would be: let m, n be two nodes of the super-graph, $d \in D_m$, $d' \in D_n$ two nodes of the exploded super-graph. Then $d' \in MVP_{m,n}(\{d\})$ if and only if there exists a realizable path from d' to d, or from d' to a marked node $d'' \in Mark$ in the exploded super-graph.

We could first build the graph of the IFDS problem, before building the exploded super-graph, and use the above result to get the correctness. However, the instantiation of context transformers cannot be coded in the framework of IFDS problems, whereas it can be coded with exploded super-graphs. That is why we consider directly the exploded super-graph associated with the considered problem. Using IFDS problems makes it easier to understand the ideas behind the use of the exploded super-graph.

5.1.2 Algorithm to build the exploded super-graph

The general principles of the previous section are now applied to build the exploded super-graph corresponding to analysis T.

The function $n = \text{NewNode}()$ initializes a new node of the exploded super-graph. Each node n of the exploded super-graph has a label $\text{Label}(n)$ which is an integer. This label will be used in the following, but it is not necessary for the first step of the algorithm. The label of a new node n is initialized to $\text{Label}(n) = \mathit{CurrentLabel}$.

For each node m of the exploded super-graph, we use a set $\text{Succ}(m)$ of successors of m, a set $\text{Callee}(m)$ of targets of call edges whose source is m. Let $\text{Ret}(e)$ be the set of return edges starting from m, and $\text{Callers}(i, s)$ the set of call nodes c, such that there exists an edge $c \overset{(_i}{\rightarrow} s$ pointing to s. The function call $\text{Link}(n, m)$, defined below, creates a new ordinary edge in *Edge* linking n to m by updating $\text{Succ}(n)$. $\text{Mark}(n)$ marks the node n as escaping. $\text{Call}(\mathit{listcall}, \mathit{listret})$ creates the edges corresponding to a new function call, by updating the sets Callee, Callers and Ret (*listcall* indicates the list of call edges to create, whereas *listret* indicates the list of return edges).

$\text{Link} : \mathit{Node} \times \mathit{Node} \rightarrow \text{unit}$
$\text{Link}(n, m) =$
if $m \notin \text{Succ}(n)$ **then**
    add m to $\text{Succ}(n)$

$\text{Call} : \mathcal{P}(\mathit{Node} \times \mathit{Node}) \times \mathcal{P}(\mathit{Node} \times \mathit{Node}) \rightarrow \text{unit}$
$\text{Call}(\mathit{listcall}, \mathit{listret}) =$
choose a new index i
foreach $(c, s) \in \mathit{listcall}$,
    add s to $\text{Callee}(c)$
    add c to $\text{Callers}(i, s)$
foreach $(r, e) \in \mathit{listret}$,
    add $e \overset{)_i}{\rightarrow} r$ to $\text{Ret}(e)$

$\text{Type}(\tau)$ builds the annotated type associated with type τ. Such an annotated type is in fact a copy of $\text{Gr}(\tau)$, in which each node is labelled by a node of the exploded super-graph. The set of exploded super-graph nodes labelling $D = \text{Type}(\tau)$ corresponds to the domain D_d associated with a piece of data of type τ. This set will be denoted by $\text{Nodes}(D)$. This defines a function $\text{Nodes} : \mathit{AnnotatedType} \rightarrow \mathcal{P}(\mathit{Node})$.

$\text{Type} : \mathit{Type} \rightarrow \mathit{AnnotatedType}$
$\text{Type}(\tau) =$

Create a copy of $\mathrm{Gr}(\tau)$. Each node of the copy is labelled by a new exploded super-graph node created by NewNode().

TopNode(D) yields the exploded super-graph node at the top of D. Accessors(D) yields the set of accessors of the type τ associated with D (i.e. the accessors labelling edges starting from the top node of $\mathrm{Gr}(\tau)$ in $\mathrm{Gr}(\tau)$). Let $n_0[\tau]$ be the top node of D. Let n_1 be the node such that $n_0[\tau] \xrightarrow{\delta} n_1$ is an edge of D (this has only a meaning if $\delta \in \mathrm{Accessors}(D)$). In this case, n_1 is said to be a son of n_0 in domain D. Then GetSon(D, δ) yields the domain associated with the subgraph of D reachable from n_1.

TopNode : *AnnotatedType* $\to$ *Node*

$\mathrm{TopNode}(n[\beta] \,|\, n[\mathtt{bool}] \,|\, n[\mathtt{int}]) = n$

$$\mathrm{TopNode}\left(\begin{array}{c} n[\tau_1 \to \tau_2] \\ \downarrow{\scriptstyle \mathrm{app}} \\ D_2 \end{array}\right) = n$$

$$\mathrm{TopNode}\left(\begin{array}{ccccc} & & n[\tau] & & \\ & \overset{\delta_1}{\swarrow} & & \overset{\delta_k}{\searrow} & \\ D_1 & & \cdots & & D_k \end{array}\right) = n$$

Accessors : *AnnotatedType* $\to \mathcal{P}($*Access*$)$

$\mathrm{Accessors}(n[\beta] \,|\, n[\mathtt{bool}] \,|\, n[\mathtt{int}]) = \emptyset$

$$\mathrm{Accessors}\left(\begin{array}{c} n[\tau_1 \to \tau_2] \\ \downarrow{\scriptstyle \mathrm{app}} \\ D_2 \end{array}\right) = \{\mathrm{app}\}$$

$$\mathrm{Accessors}\left(\begin{array}{ccccc} & & n[\tau] & & \\ & \overset{\delta_1}{\swarrow} & & \overset{\delta_k}{\searrow} & \\ D_1 & & \cdots & & D_k \end{array}\right) = \{\delta_1, ..., \delta_k\}$$

GetSon : *AnnotatedType* $\times$ *Access* $\to$ *AnnotatedType*

$$\mathrm{GetSon}\left(\begin{array}{c} n[\tau_1 \to \tau_2] \\ \downarrow{\scriptstyle \mathrm{app}} \\ D_2 \end{array}, \mathrm{app}\right) = D_2$$

$$\mathrm{GetSon}\left(\begin{array}{ccccc} & & n[\tau] & & \\ & \overset{\delta_1}{\swarrow} & & \overset{\delta_k}{\searrow} & \\ D_1 & & \cdots & & D_k \end{array}, \delta_j\right) = D_j$$

Build : *Access* $\times$ *Type* $\to$ *AnnotatedType* $\to$ *AnnotatedType*

$\mathrm{Build}(\mathrm{app}, \tau_1 \to \tau_2)(D_2) =$

if $\tau_1 \to \tau_2$ is the label of a node n of D_2 **then**
    return D_2 with top node n
else
    return $\begin{array}{c} \mathrm{NewNode}()[\tau_1 \to \tau_2] \\ \downarrow{\scriptstyle \mathrm{app}} \\ D_2 \end{array}$

The expression Build(δ, τ), where $\delta : \tau \to \tau'$, converts a domain D corresponding to type τ' in a domain corresponding to type τ, by creating the necessary nodes. It will only be useful for the app accessor, so we define it only in this case. Two cases have to be distinguished: either the top node of $\mathrm{Gr}(\tau)$ is alone in its strongly connected component, and a new exploded super-graph node corresponding to this top node has to be created (**else** part); or the top node is an element of a larger strongly connected component, then it already appears in the graph of τ', and only the top node of the graph changes (**then** part).

The function pType yields annotated types corresponding both to the parameters and to the result of type τ, $pAnnotatedType = AnnotatedType^* \times AnnotatedType$. $\mathrm{GetRes}(pD)$ returns the result part of pD.

pType : $Type \to pAnnotatedType$
$\mathrm{pType}(\tau) =$
let $\tau_1 \to ... \to \tau_n = \tau$ where τ_n is not a function type
return $([\mathrm{Type}(\tau_1), ..., \mathrm{Type}(\tau_{n-1})], \mathrm{Type}(\tau))$

GetRes : $pAnnotatedType \to AnnotatedType$
$\mathrm{GetRes}(D_p, D_r) = D_r$

The generalization function $\mathrm{Generalize}(D_i, D_g)$ links pairwise the nodes that correspond to the same occurrence in the annotated types D_i and D_g. D_i may correspond to a more instantiated type than D_g (i.e. some type variables β of the type of D_g may be substituted by type expressions $\sigma\beta$ in the type of D_i). In this case, the node associated with a type variable β of the type of D_g is linked to each node associated with the corresponding type expression $\sigma\beta$ in D_i. This function builds the exploded super-graph representation of the function G_T^τ.

Generalize : $AnnotatedType \times AnnotatedType \to$ unit
$\mathrm{Generalize}(D_i, D_g) =$
if the current call comes from a call to $\mathrm{Generalize}(x, D_g)$ (for any x) **then return**
match (D_i, D_g) **with**
 $(\_, n[\beta]) \to$
   foreach $m \in \mathrm{Nodes}(D_i), \mathrm{Link}(n, m)$
 $\mid (D_i, D_g) \to$
   $\mathrm{Link}(\mathrm{TopNode}(D_g), \mathrm{TopNode}(D_i))$
   foreach $\delta \in \mathrm{Accessors}(D_g), \mathrm{Generalize}(\mathrm{GetSon}(D_i, \delta), \mathrm{GetSon}(D_g, \delta))$

If a call to $\mathrm{Generalize}(D_i, D_g)$ leads to recursive calls of Generalize with the same domain D_g, we have to stop the traversal of the graph to avoid infinite loops. This is the meaning of the test "**if** the current call comes from a call to $\mathrm{Generalize}(x, D_g)$" on the second line. This can be implemented more precisely by marking the domains D_g on which $\mathrm{Generalize}(D_i, D_g)$ has been called (the marks have to be removed after a call to Generalize has been completed).

The function CallD inserts call and return edges corresponding to a function call in the exploded super-graph. It builds the exploded super-graph representation of the instantiation function.

Collect : $AnnotatedType \times AnnotatedType \times \{\mathrm{true}, \mathrm{false}\} \to \mathcal{P}(Node \times Node)$
$\mathrm{Collect}(D_i, D_g, inParam) =$
if the current call comes from a call to $\mathrm{Collect}(x, D_g, inParam)$ **then return** $\emptyset$
match (D_i, D_g) **with**
 $(n[\beta], m[\beta]) \to$
   return $\{(n, m)\}$

28 $\mid (\_, m[\beta]) \to$
29   **if** *inParam* **then**
30   add (m, D_i) to $\text{SubstPar}(\beta)$; **return** $\{(n,m) | n \in \text{Nodes}(D_i)\}$
31   **else**
32   add (m, D_i) to $\text{SubstRes}(\beta)$; **return** $\emptyset$
33 $\mid (D_i, D_g) \to$
34   **return** $\{(\text{TopNode}(D_i), \text{TopNode}(D_g))\} \cup$
35     $\cup_{\delta \in \text{Accessors}(D_g)} \text{Collect}(\text{GetSon}(D_i, \delta), \text{GetSon}(D_g, \delta), \mathit{inParam})$

CallD : $\mathit{pAnnotatedType} \times \mathit{pAnnotatedType} \to \text{unit}$
$\text{CallD}(([D_{i1}, ..., D_{ip}], D_{ir}), ([D_{g1}, ..., D_{gm}], D_{gr})) =$
36 **foreach** $\beta, \text{SubstRes}(\beta) = \{\}, \text{SubstPar}(\beta) = \{\}$
37 $\text{Call}(\cup_{k \in [\![1,m]\!]} \text{Collect}(D_{ik}, D_{gk}, \text{true}), \text{Collect}(D_{ir}, D_{gr}, \text{false}))$
38 **foreach** $k \in [\![m+1, p]\!]$, **foreach** $n \in \text{Nodes}(D_{ik}), \text{Mark}(n)$
39 **foreach** β,
40   **foreach** occurrence o in the graph of type of the first element of $\text{SubstRes}(\beta)$,
41   let $\mathit{par} = \{(c, s) | (s, D_i) \in \text{SubstPar}(\beta), c =$ the node labelling the occurrence o in $D_i\}$
42   let $\mathit{res} = \{(r, e) | (e, D_i) \in \text{SubstRes}(\beta), r =$ the node labelling the occurrence o in $D_i\}$
43     $\text{Call}(\mathit{par}, \mathit{res})$

The function Collect collects pairwise nodes that correspond to the same occurrence in the domains of the call site and of the callee. The domain D_i corresponds to the instantiated function (i.e. the call site). The domain D_g corresponds to the general function (i.e. the callee). The parameter *inParam* is true when the considered domains are the parameters of the function, and false when they are in its result. The test "**if** the current call comes from a call to $\text{Collect}(x, D_g, \mathit{inParam})$" has the same goal as for Generalize.

When nodes correspond pairwise, the mechanism of Collect is simple (first and last cases of the match, lines 26 and 33). If a type variable β has really been substituted by a (different) type $\sigma\beta$, the node of the type variable β in the general type D_g corresponds to all nodes of $\sigma\beta$ in D_i (second case of the match, line 28). The process is more complicated in this case: we remember the precise correspondence by storing it in SubstPar (for the parameters, line 30) or SubstRes (for the result, line 32). The edges are then added in the last five lines of CallD. An occurrence o in the graph of a type is represented by an access path in this graph. It is enough to consider one path leading to each node of the graph. We consider a pseudo function call (line 43) which creates edges $c \xrightarrow{(_i} s$ and $e \xrightarrow{)_i} r$ where s and e are nodes corresponding to a type variable β in the callee, and c and r are nodes corresponding to the same occurrence o in the type $\sigma\beta$ at the call site. These pseudo function calls take into account the contribution of the case when $\text{RemPath}_{\text{Gr}(\tau)}(p)$ corresponds to occurrence o ($\text{RemPath}_{\text{Gr}(\tau)}(p) \neq NV$) in the formula of instantiation (see page 60).

The function $\text{MarkParams}(D)$ marks as escaping all parameters of D.

MarkParams : $\mathit{AnnotatedType} \to \mathit{pAnnotatedType}$
$\text{MarkParams}(D) =$
44 let $\begin{array}{l} n[\tau_1 \to ... \to \tau_p \to \tau_{p+1}] \\ \qquad\qquad \big\downarrow \text{app} \\ \qquad\qquad D' \end{array} = D$ where τ_{p+1} is not a function
45 **foreach** $k \in [\![1, p]\!]$,

$\quad D_k = \mathrm{Type}(\tau_k)$, **foreach** $m \in \mathrm{Nodes}(D_k), \mathrm{Mark}(m)$
return $([D_1, ..., D_p], D)$

The function $\mathrm{Graph}(M, D_M, \rho)$ builds the exploded super-graph associated with expression M.

$\mathrm{Graph} : \mathit{Expression} \times \mathit{pAnnotatedType} \times (\mathit{Var} \to \mathit{pAnnotatedType}) \to \mathrm{unit}$
$\mathrm{Graph}(M, pD_M, \rho) =$
$\mathit{CurrentLabel} = \mathit{CurrentLabel} + 1$
match M **with**
$\quad x \to$
$\qquad \mathrm{Generalize}(\mathrm{GetRes}(pD_M), \mathrm{GetRes}(\rho(x)))$
$\qquad$ **if** x is a function **then** $\mathrm{CallD}(pD_M, \rho(x))$
$\quad |$ `let` x `=` N `in` $P \to$
$\qquad pD_N = \mathrm{pType}(\tau_x)$
$\qquad \mathrm{Graph}(P, pD_M, \rho[x \mapsto pD_N])$
$\qquad \mathrm{Graph}(N, pD_N, \rho)$
$\quad |$ `let rec` x `=` N `in` $P \to$
$\qquad pD_N = \mathrm{pType}(\tau_x)$
$\qquad \mathrm{Graph}(P, pD_M, \rho[x \mapsto pD_N])$
$\qquad \mathrm{Graph}(N, pD_N, \rho[x \mapsto pD_N])$
$\quad | \ N \ P \to$
$\qquad pD_P = \mathrm{pType}(\tau_P)$
$\qquad \mathrm{Graph}(P, pD_P, \rho)$
$\qquad pD_N = (\mathrm{GetRes}(pD_P) : D_{\mathrm{par}}, \mathrm{Build}(\mathrm{app}, \tau_P \to \tau_M)(D_M))$ where $(D_{\mathrm{par}}, D_M) = pD_M$
$\qquad \mathrm{Graph}(N, pD_N, \rho)$
$\quad |$ `fun` $x \to N \to$
$\qquad (D_x : D_{\mathrm{par}}, D_M) = pD_M$
$\qquad$ **foreach** $z \in FV($`fun` $x \to N)$, **foreach** $m \in \mathrm{Nodes}(\mathrm{GetRes}(\rho(z)))$,
$\qquad\quad \mathrm{Link}(m, \mathrm{TopNode}(D_M))$
$\qquad pD_N = (D_{\mathrm{par}}, \mathrm{GetSon}(D_M, \mathrm{app}))$
$\qquad \mathrm{Graph}(N, pD_N, \rho[x \mapsto \mathrm{MarkParams}(D_x)])$
$\quad | \ C(N_1, ..., N_k) \to$
$\qquad$ **foreach** $j \in [\![1, k]\!]$,
$\qquad\quad pD_{N_j} = ([], \mathrm{GetSon}(\mathrm{GetRes}(pD_M), \overline{C_j}))$
$\qquad\quad \mathrm{Graph}(N_j, pD_{N_j}, \rho)$
$\quad | \ \overline{C_i}(N) \leftarrow P \to$
$\qquad pD_N = \mathrm{pType}(\tau_N)$
$\qquad pD_P = \mathrm{pType}(\tau_P)$
$\qquad$ **foreach** $n \in \mathrm{Nodes}(\mathrm{GetRes}(pD_P)), \mathrm{Mark}(n)$
$\qquad \mathrm{Graph}(N, pD_N, \rho)$
$\qquad \mathrm{Graph}(P, pD_P, \rho)$
$\quad | \ N; P \to$
$\qquad pD_N = \mathrm{pType}(\tau_N)$
$\qquad \mathrm{Graph}(N, pD_N, \rho)$
$\qquad \mathrm{Graph}(P, pD_M, \rho)$
$\quad |$ `if` N `then` P `else` $Q \to$
$\qquad pD_N = \mathrm{pType}(\tau_N)$

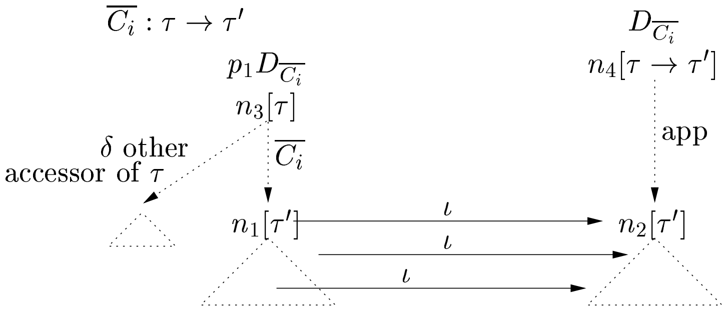


Figure 5.3: Graph associated with a destructor $\overline{C_i} : \tau \to \tau'$. If τ' is not a function type, the domain of $\overline{C_i}$ is $pD_{\overline{C_i}} = ([p_1 D_{\overline{C_i}}], D_{\overline{C_i}})$. If τ' is a function type, $pD_{\overline{C_i}} = ([p_1 D_{\overline{C_i}}, ..., p_n D_{\overline{C_i}}], D_{\overline{C_i}})$, where all nodes in $p_k D_{\overline{C_i}}$ $(k > 1)$ are marked as escaping. The graph associated with other functions is similar.

```
88    Graph(N, pD_N, ρ)
89    Graph(P, pD_M, ρ)
90    Graph(Q, pD_M, ρ)
```

$\rho_0 = \{\overline{C_i} \mapsto \text{graph of Figure 5.3}; ....\}$
$\mathit{CurrentLabel} = 1$
$pD_M = \text{pType}(\tau_M)$
$\text{Graph}(M, pD_M, \rho_0)$

The algorithm of Graph can be explained as follows. The value of *CurrentLabel* is first updated, to make sure that a node corresponding to an expression of greater scope has a smaller label.

In the variable case x, the function Generalize creates ordinary edges representing what escapes from x (line 51). If the variable is a function, the function CallD creates $(_i$ and $)_i$ edges representing a call to function x (line 52).

In the `let` case, a new domain is created corresponding to expression N, and the subexpressions N and P are analyzed. The domain corresponding to expression P is the same as the domain of the `let` expression. The domain of variable x in expression P is the domain of expression N, since N is bound to x.

The `let rec` case is similar to the `let` case, except that the variable x is also defined in expression N.

In the application case $N\ P$, a new domain is created for P, which is then analyzed. The domain of N is not created from scratch, but built from the domain of $M = N\ P$ (which corresponds to the result of N) and of P (which corresponds to the first parameter of N) at line 64. The expression N can then be analyzed.

In the function case, the domains for x and N are built from the domain of $M = \texttt{fun}\ x \to N$ (lines 67 and 70). Lines 68 and 69 take into account the fact that all free variables of N escape when the closure escapes. The subexpression N is finally analyzed.

Notice that this graph building algorithm is slightly optimized with respect to the technique described in the previous section. Indeed, if we follow strictly the exploded super-graph building algorithm, when a value d' of type τ' is extracted from a value d of type τ (we apply $\text{Restr}_T(\delta, \tau)$), we would build annotated types associated with the super-graph node corresponding to d': $D_{d'} = \text{Type}(\tau')$, and each node of $D_{d'}$ would be linked to the corresponding node of D_d. The more efficient technique used here is merging corresponding nodes of D_d and $D_{d'}$ by extracting directly $D_{d'}$ from D_d. For example, if d is a list and d' an element of this list, $D_{d'} = \text{GetSon}(D_d, \texttt{hd})$. This optimization is used in particular in the constructor case.

5.2 Computing summary edges

Function calls are represented in the original graph by call $c \xrightarrow{(_i} s$ and return $e \xrightarrow{)_i} r$ edges. To find the nodes reachable by realizable paths, the effect of a function call is first summarized by an edge $c \xrightarrow{\iota} r$. Such an edge is said to be a ***summary edge***. The summary edges are computed by the function Propagate which is detailed below. Traversing a summary edge corresponds to traversing an *SLRPEL* path in the exploded super-graph before calls to Propagate.

The edges labelled ι already present in the graph before the call to Propagate are said to be ordinary edges.

Propagate : $\mathit{Node} \times \mathit{Node} \to$ unit
Propagate(n, s) =
1 **if** $s \notin \text{PathEdge}(n)$ **then**
2 add s to PathEdge(n)
3 **foreach** $n \xrightarrow{)_i} r \in \text{Ret}(n)$
4 **foreach** $c \in \text{Callers}(i, s)$
5 **if** $r \notin \text{Succ}(c)$ **then**
6 add r to Succ(c)
7 **foreach** $s' \in \text{PathEdge}(c), \text{Propagate}(r, s')$
8 **if** no corresponding c has been found (i.e. lines 5–7 have not been executed during this call to Propagate) **then**
9 **if** Mark(n) **then** Mark(s) **else**
10 **foreach** $m \in \text{Succ}(n), \text{Propagate}(m, s)$
11 **foreach** $s' \in \text{Callee}(n)$,
12 Propagate(s', s')
13 **if** Mark(s') **then** Mark(s)

Essentially, Propagate(n, s) will be called for each pair (n, s) such that $\mathit{SLRP}(n, s) \neq \emptyset$. PathEdge is used to summarize *SLRP* paths: $s \in \text{PathEdge}(n)$ implies $\mathit{SLRP}(n, s) \neq \emptyset$, and it means that Propagate(n, s) has already been called. So line 1 first checks that Propagate(n, s) has not already been called. If not, line 2 updates PathEdge(n). Lines 3 to 7 add summary edges. Lines 3 and 4 make sure that we are in the situation of Figure 5.4. Since we have called Propagate(n, s), there is an *SLRP* path from s to n. There is a call edge $c \xrightarrow{(_i} s$ and a return edge $n \xrightarrow{)_i} r$. Then, there exists an *SLRPEL* path from c to r (5.2). Therefore, a summary edge $c \xrightarrow{\iota} r$ is added if it does not already exist (line 6). Adding this edge makes the algorithm discover new *SLRP* paths: if there was an *SLRP* path from s' to c, there is also an *SLRP* path from s' to r (by (5.4), since we prove in the correctness proof that there exists no path $m' \xrightarrow{(_i} s' : q_1 : c \xrightarrow{)_i} n'$). The goal of line 7 is to take these new paths into account.

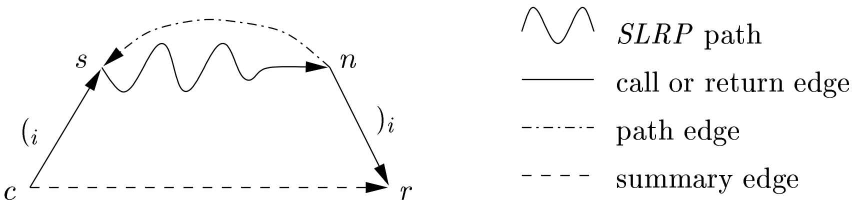


Figure 5.4: Adding summary edges (lines 5 to 7 of Propagate). The dashed edge from n to s is said to be a path edge. It indicates the fact that there exists a path from s to n symbolized by the wavy line. In this configuration, the summary edge represented by the dotted line from c to r is added (line 6).

Lines 8 to 13 then update the marked nodes, by marking nodes s such that there exists a path in $RP(s,n)$ and n is marked. The first case (line 9) is when $SLRP(s,n) \neq \emptyset$ (5.5). The condition "if no corresponding c has been found" checks that there exists no path $m' \xrightarrow{(_i} s : q : n \xrightarrow{)_i} n'$. If there is an $SLRP$ path from s to n and an edge $n \xrightarrow{\iota} m$, there exists also an $SLRP$ path from s to m (rule (5.4), knowing that there exists no path $m' \xrightarrow{(_i} s : q : n \xrightarrow{)_i} n'$). Line 10 takes these paths into account by calling Propagate on the appropriate nodes. Line 12 takes into account empty $SLRP$ paths. Line 13 marks node s if there is an $SLRP$ path from s to n and an edge $n \xrightarrow{(_i} s'$ with s' marked. Indeed, there is an RP path from s to s', this is a particular case of (5.6) (take q_2 equal to the empty path).

5.3 Computing escaping parts

5.3.1 Algorithm

The function Propagate$'$ will be used to compute the escape information. It is defined as follows:

Propagate$'(n)$ =
1 **if** Visited(n) **then return** ∞
2 Visited(n) = true
3 **if** Mark(n) **then return** 0
4 **foreach** $s \in$ Callee(n),
5   Propagate(s, s)
6   **if** Mark(s) **then return** 0
7 **return** $\min\{\text{Label}(n)\} \cup \{\text{Propagate}'(m) | m \in \text{Succ}(n)\}$

At the beginning, Visited(n) = false for every node n. Visited is reset to false after the call to Propagate$'$ is complete. The purpose of the Visited flag is simply to avoid visiting several times the same node during a call to Propagate$'$.

The function Propagate$'$ calls Propagate to compute the summary edges of the function calls (line 5). If a marked node is reachable from n, then the considered value escapes and Propagate$'(n)$ returns 0 (lines 3 and 6). Otherwise, the return value of Propagate$'(n)$ is the least label of a node reachable from n, which corresponds to the label of the expression of greatest scope in which the piece of data represented by node n may escape. Indeed, the function Graph initializes labels such that the label of a node of the domain of a subexpression of M is greater than the label of a node of the domain pD_M of M.

If the binder of the considered piece of data d has a label smaller than Propagate$'(n)$ then its scope is greater than the maximum scope in which d may escape, and d can be stack allocated.

Comparison with [RSH94] [RSH94] deals with the reachability from a single source. The result of Propagate$'(n)$ is 0 if and only if n is reachable from the source, and labels are not needed. In our case, we have to distinguish several sources (each source corresponds to a different scope). Labels will enable us to achieve this goal: the integer result of Propagate$'(n)$ indicates which is the largest reachable scope (corresponding to the maximum lifetime of the value corresponding to node n).

5.3.2 Correctness

Theorem 5.14 (The implementation of T is sound and complete) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let n_0 be the node at the top of the graph* $\mathrm{Gr}(\tau_x^{\mathrm{g}})$. *Let* $\phi = T[\![N]\!]\rho\, x$.

- `let rec` $x = $ `fun` $y \to M$ `in` N. *In this case, let ℓ be the location of the closure x. Let n_0 be the node at the top of the graph* $\mathrm{Gr}(\tau_x^{\mathrm{g}})$. *Let $\phi = R(x)$ where R is defined in the* `let rec` *case of Figure 3.6.*

In these two cases, assume that N has been analyzed by $\mathrm{Graph}(N, pD_N, \rho)$. *Let* $\rho[\![x]\!] = \mathrm{GetRes}(\rho(x))$ *and* $D_N = \mathrm{GetRes}(pD_N)$. $n_0 \notin \phi(\underline{\tau_N})$ *if and only if* $\mathrm{Propagate}'(\mathrm{TopNode}(\rho[\![x]\!])) > \mathrm{Label}(\mathrm{TopNode}(D_N))$. *If this condition is satisfied, ℓ can be stack allocated, and deallocated at the end of N.*

This theorem is proved in Appendix D. The result of $\mathrm{Propagate}'(\mathrm{TopNode}(\rho[\![x]\!]))$ is the label of the expression P of greatest scope in which x may escape. If it is greater than $\mathrm{Label}(\mathrm{TopNode}(D_N))$, P has a smaller scope than N (P is a subexpression of N). In this case, l can be stack allocated.

5.3.3 Optimized algorithm

The previous algorithm repeats part of the work if several calls to Propagate$'$ visit the same nodes. This can be avoided by slightly modifying the algorithm. We use Tarjan's algorithm to compute strongly connected components. Each node n visited by Propagate$'$ is marked as visited and labelled by the minimum of the labels of nodes reachable from n. The nodes reachable from n are the nodes of the strongly connected component of n and of components reachable from this one. During the following calls to Propagate$'$, if an already visited node is met, the result of Propagate$'$ is given by the already computed label without visiting nodes again.

Comparison with [RSH94] This new algorithm is similar to [RSH94, page 37] whereas the previous algorithm corresponded to page 34. This algorithm has the same-total-cost property [RSH94, page 36], i.e. "the total cost of any request sequence that places demands on all possible nodes [of the graph] is proportional to the cost of a single run of the algorithm [which computes all results at once]".

5.4 Complexity

We assume as in [RSH94] that adding an element to a set or checking whether an element is in a set can be done in constant time, which can be realized using hash tables (even if this is not the implementation choice that has been done in practice, because in most cases, the sets only have a small number of elements).

Then $\mathrm{Link}(n, m)$ runs in constant time, $\mathrm{Call}(\mathit{listcall}, \mathit{listret})$ in $\mathcal{O}(|\mathit{listcall}| + |\mathit{listret}|)$, $\mathrm{Type}(\tau)$ in $\mathcal{O}(|\mathrm{Gr}(\tau)|)$, $\mathrm{Generalize}(D_i, D_g)$, $\mathrm{Collect}(D_i, D_g)$, $\mathrm{CallD}(D_i, D_g)$ in $\mathcal{O}(|D_i|)$ and $\mathrm{Graph}(M, pD_M, \rho)$ in $\mathcal{O}(|M| \times D)$ where $D = \max\{|pD_P|, P \text{ sub-expression of } M\}$. The number of edges created by $\mathrm{Graph}(M, pD_M, \rho)$ is also $\mathcal{O}(|M| \times D)$.

The set of all calls to Propagate is performed in at most $\mathcal{O}(ED + \mathit{NCalls}\, D^3 + \mathit{NPropagate})$ where E is the number of edges of the exploded super-graph, *NCalls* the number of calls to CallD that have been performed, and *NPropagate* the number of calls to Propagate. Indeed, lines 1, 2, 8, 9 run in constant time. Only the cost of these lines is associated with each call to Propagate individually. The total cost of the other lines is computed for the set of all calls. Knowing that lines 2 and following are only executed once for each pair (n, s), the total cost of lines 2 and following for all calls is bounded by the total cost for all possible pairs (n, s). At line 3, there are E choices of $n \xrightarrow{)_i} r$, and for each of these choices, D choices of $c \xrightarrow{(_i} s$ at line 4, therefore lines 3, 4, 5 run in at most $\mathcal{O}(ED)$ time. At line 6, there are $\mathcal{O}(\mathit{NCalls}\, D^2)$ choices of r and c (there are at most D^2 summary edges for each call), and for each of these choices, at most D choices of s' at line 7 (the entry nodes of the function which contains c) therefore lines 6 and 7 run in time $\mathcal{O}(\mathit{NCalls}\, D^3)$. At line 10, there are $E + \mathit{NCalls}\, D^2$ choices of $n \xrightarrow{\iota} m$ and D choices of s (entry nodes of the function which contains n) therefore it runs in time $\mathcal{O}(ED + \mathit{NCalls}\, D^3)$. This is the same for lines 11, 12, 13. Adding the time of all the lines of Propagate yields the expected complexity.

In its optimized version, Propagate$'$ visits at most once each node, and it is called at most once for each node, therefore the total running time of all calls to Propagate$'$ is at most $\mathcal{O}(N)$, plus the cost of the calls to Propagate, where N is the number of nodes of the exploded super-graph. The number of calls to Propagate is at most $\mathcal{O}(N)$, hence the worst case running time of the calls to Propagate$'$: $\mathcal{O}(ED + \mathit{NCalls}\, D^3 + N) = \mathcal{O}(|M|D^2 + \mathit{NCalls}\, D^3)$, which dominates the time of building of the graph.

The complexity of analysis T is then $\mathcal{O}(|M|D^2 + \mathit{NCalls}\, D^3)$.

Remark. In [RSH94], the complexity result for the class of distributive functions is slightly different from the above result for the complexity of Propagate: $\mathcal{O}(\mathit{NCalls}\, B^2 D^2 + E' D^3)$ where B is such that each call or exit node of the exploded super-graph has at most B outgoing $(_i$ or $)_i$ edges, each entry or return node of the exploded super-graph has at most B incoming $(_i$ or $)_i$ edges for a given i. E' is the number of edges in the super-graph (not the exploded super-graph !).

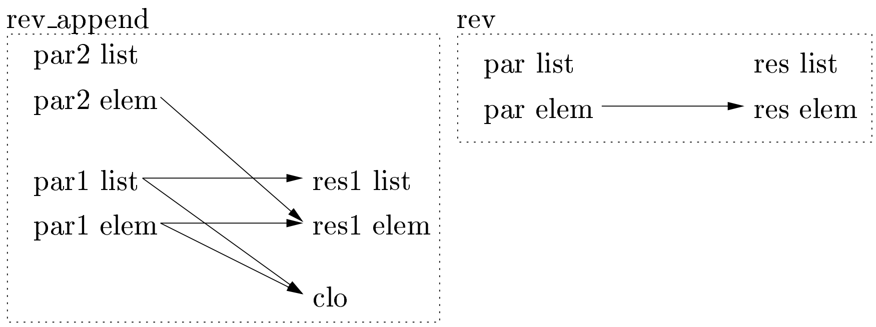


Figure 5.5: In Example 5.2, we get the above reduced graph.

In our case, each call node has at most one outgoing $(_i$ edge, each return node has at most one incoming $)_i$ edge for each i. But an entry node may have several incoming $(_i$ edges (if it represents a type variable which is instantiated to another type in the caller). The cost of installing summary edges is then bounded by $\mathcal{O}(ED + \mathit{NCalls}\, D^3)$ instead of $\mathcal{O}(\mathit{NCalls}\, B^2 D^2 + \mathit{NCalls}\, D^3)$ in [RSH94].

The cost of closure steps (equivalent to our line 10 of Propagate) is bounded by $\mathcal{O}(E'D^3)$ in [RSH94]. Our bound is $\mathcal{O}(ED + \mathit{NCalls}\, D^3)$ which yields the result of [RSH94] since $E = \mathcal{O}(E'D^2)$.

5.5 Optimizations

5.5.1 Reducing the graph

When the analysis of a block of recursive definitions has been completed, we simplify the graphs of the defined functions, to avoid using the relatively costly mechanism of summary edges too often.

We build a new graph for the function f, which contains the nodes of $\rho(f) = \mathrm{pType}(\tau_f)$ and edges linking the parameters to the result of the function. A node n_1 of a parameter of $\rho(f)$ is linked to a node n_2 in the result of $\rho(f)$ if and only if there exists a path from n_1 to n_2 in the exploded super-graph of f built by the analysis.

This graph reduction is done by a simple algorithm which visits the different paths. The complexity of this algorithm is the same as adding the summary edges corresponding to one call to f.

During the following of the analysis of the program, when a call to f has to be analyzed, we copy this simple graph (taking into account the possible instantiation in a way similar to CallD) between the nodes of pD_f. This analysis of this call is then faster than using Propagate to add the summary edges.

Figure 5.5 shows an example of a reduced graph.

5.5.2 Compacting the graph

For each program point, the graph has as many nodes as the size of the graph of the type of data manipulated at this program point. This may reach several thousands of nodes in large applications (for example, the Ocaml compiler itself). The algorithm becomes then too costly both in time and in memory.

This problem has been solved by taking into account the fact that contexts are downwards closed (i.e. if a piece of data escapes, all values reachable from it also escape). Then we can have a more compact representation of the graph.

A node denoted by n_D is said to be a compact node. It represents in a single node all nodes of the annotated type D. The edges starting from n_D may be of several kinds:

- Ordinary edges: they link only the top node of domain D: $n_D \xrightarrow{\iota} n'$ means that the top node of n_D is linked to n' (if n' is not a compact node) or to the top node of n' (if n' is a compact node). Ordinary edges are not transmitted to sons, and act as if the node n_D was an ordinary node.

- Compact edges: $n_D \xrightarrow{\iota,c} n'_{D'}$ replaces all edges $n \xrightarrow{\iota} n'$ where n and n' are nodes labelling respectively the annotated types D and D' at the same occurrences (D and D' must correspond to the same type, and therefore be isomorphic graphs). These edges correspond to edges created by the function Generalize when the instantiated and general type are the same.

- Closure edges: $n_D \xrightarrow{\iota,f} n'$ replaces all edges $n \xrightarrow{\iota} n'$ where n annotates D. These edges correspond to those created in the `fun` case of the function Graph.

- Compact call and return edges: $c_{Dc} \xrightarrow{(_i,c} s_{Ds}$ and $e_{De} \xrightarrow{)_i,c} r_{Dr}$ which replace respectively $c \xrightarrow{(_i} s$, c annotating Dc and s annotating Ds at the same occurrences, and $e \xrightarrow{)_i} r$, e annotating De and r annotating Dr at the same occurrences (Dc and Ds correspond to the same type, so do De and Dr). These edges correspond to edges created by CallD (when there is no instantiation).

Much compute time and memory can be saved up by replacing by a compact node every annotated type for which incoming and outgoing edges can be represented by compact or closure edges.

Figures 5.6 and 5.7 display two examples of compact graphs, automatically generated by the analyzer. Compact nodes are represented by rectangles, whereas other nodes are represented by ellipses. Compact edges can be recognized because of their label. The "iota" label replaces ι. Edges without label or labelled "pari" or "res" are not edges of the exploded super-graph, but edges linking a type to a type it contains (edges of the graphs of types). A function type is linked by a "par" edge to its parameter and by a "res" edge to its result. "par" edges are numbered with the number of the parameter of the function. Subgraphs corresponding to the parameters or the result of a function are boxed.

Compact nodes are "expanded" when we need to access their sons by the function GetSon. This means that they are replaced by an ordinary (not compact) node, whose sons are still compact nodes. The functions Generalize and CallD avoid expanding nodes when this is possible, by creating compact edges between compact nodes instead of creating ordinary edges between the expanded nodes.

5.5.3 Recursive types

We know that if a value escapes then all values contained in it also escape (contexts are downwards closed). In the case of a recursive type, this means that if a node is in a context,

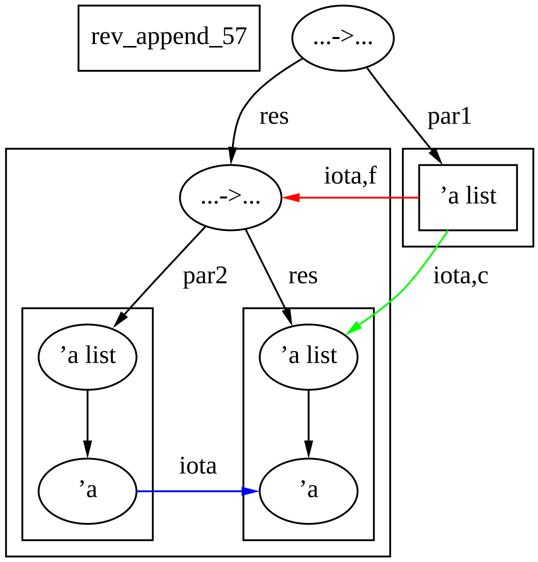


Figure 5.6: Reduced and compacted graph for function `rev_append`

then all nodes of the same strongly connected component are also in the context. Then one node of the exploded super-graph is enough to represent the whole strongly connected component.

Example 5.15 Let us consider the following type declaration:

```
type term =
    Var of int
  | Prop of head * term list
and head =
    { name: string;
      props: (term * term) list }
```

The types `term`, `term list`, `term * term`, `(term * term) list` and `head` belong to the same strongly connected component, therefore only one node of the exploded super-graph is associated with them.

In practise, this optimization is implemented in the following way. The function $\mathrm{Type}(\tau)$ returns a pair $D = (n_D, \mathrm{Gr}(\tau))$ containing a compact node n_D and the graph $\mathrm{Gr}(\tau)$. The function $\mathrm{GetSon}(D, \delta)$ where $D = (n_D, \mathrm{Gr}(\tau))$ returns $(n_D, \mathrm{Gr}(\tau'))$ if $\delta : \tau \to \tau'$ and τ' is in the same strongly connected component as τ. Otherwise, it returns $(n'_{D'}, \mathrm{Gr}(\tau'))$, where the node n_D has been expanded into an ordinary node n_o (i.e. if n_D was compact, it is not compact any more) and $n'_{D'}$ is the son of n_o such that there is an edge $n_o \xrightarrow{\delta} n'_{D'}$.

5.5.4 A difference with the real implementation

In our implementation of analysis T, we do not have a unique graph for each type. More precisely, the graph corresponding to a type declaration is the graph $\mathrm{Gr}((\beta_1, ..., \beta_p)\tau_{\mathrm{N}})$ but the graph of $(\sigma\beta_1, ..., \sigma\beta_p)\tau_{\mathrm{N}}$ is computed by substituting β_i with the graph of $\sigma\beta_i$ in

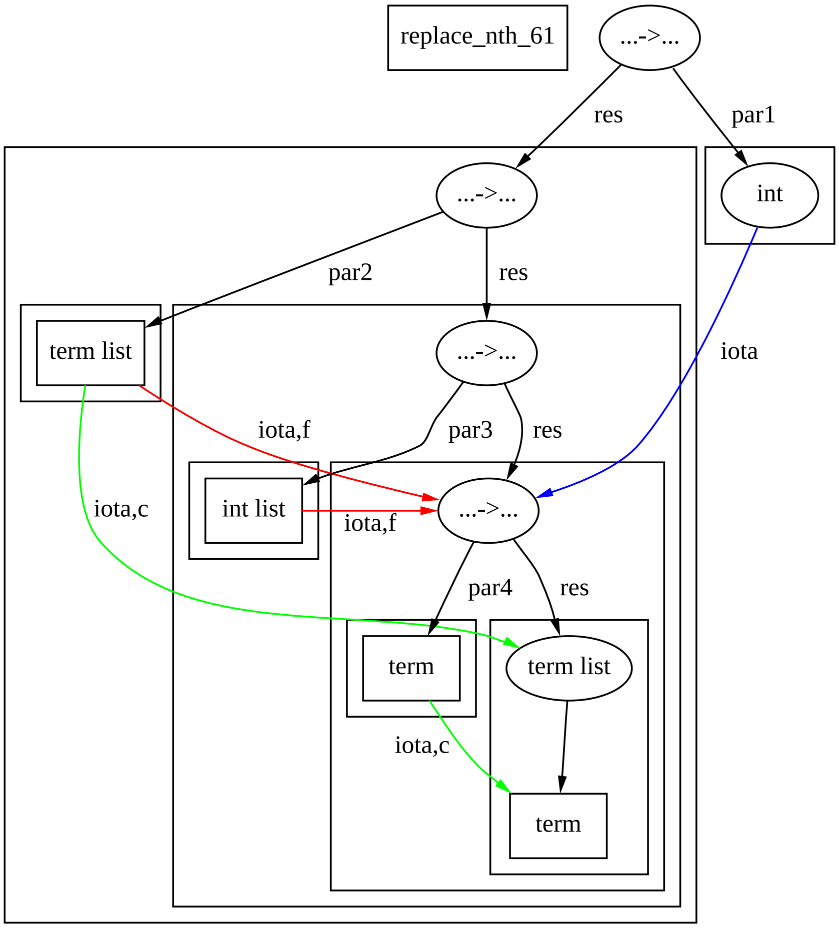


Figure 5.7: Reduced and compacted graph for function `replace_nth`. This function is taken from a Knuth Bendix completion program.

$\mathrm{Gr}((\beta_1, ..., \beta_p)\tau_{\mathrm{N}})$, without further merging nodes that are labelled by the same type and are linked by a path. This may be different from $\mathrm{Gr}((\sigma\beta_1, ..., \sigma\beta_p)\tau_{\mathrm{N}})$.

For example, consider

$$\begin{array}{rcl} \texttt{type t} & = & C \texttt{ of t list} \\ & | & C' \texttt{ of int} \end{array}$$

There are two possible graphs for the type `t list` represented in Figure 5.8. The graph on the left corresponds to the graph built by our implementation when calling Type. It is the graph of β `list` in which the variable β has been substituted by the graph of `t`. The graph on the right is Gr(`t list`) which can also be obtained by taking the $\overline{C_1}$ part of the graph of `t`. Therefore, these two graphs coexist in our implementation.

This does not change much the algorithm. The main difference is that Lemma 3.35 is no longer true. Therefore, in the function Generalize, we have to use the test "**if** the current call comes from a call to Generalize(D_i, D_g)" instead of "**if** the current call comes from a call to Generalize(x, D_g) (for any x)" to avoid infinite loops in our algorithm. This corresponds to the first version of the generalization algorithm instead of the second one (page 58). The situation is similar for Collect.

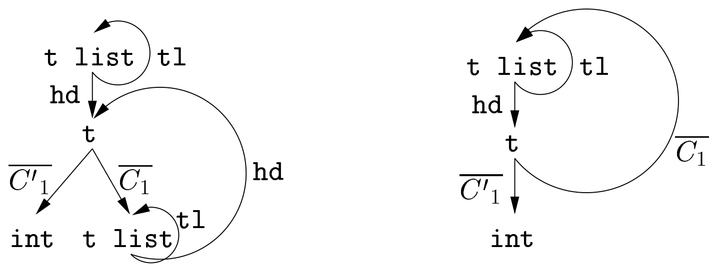


Figure 5.8: Two possible graphs for type `t list`.

The test "**if** the current call comes from a call to Generalize(D_i, D_g)" in Generalize(D_i, D_g) is slightly more difficult to implement efficiently. We use the following technique: we add a field otherType containing a list of nodes to the description of exploded super-graph nodes, and compute

Generalize(D_i, D_g) =
  if TopNode(D_i).otherType = [TopNode(D_g)]
    or TopNode(D_g).otherType = [TopNode(D_i)] **then return**
  TopNode(D_i).otherType = TopNode(D_g) : TopNode(D_i).otherType
  TopNode(D_g).otherType = TopNode(D_i) : TopNode(D_g).otherType
  .... do the generalization ....
  TopNode(D_i).otherType = tl(TopNode(D_i).otherType)
  TopNode(D_g).otherType = tl(TopNode(D_g).otherType)

If Generalize(D_i, D_g) has been called, TopNode(D_g).otherType contains TopNode(D_i) and TopNode(D_i).otherType contains TopNode(D_g). Therefore, the test "**if** the current call comes from a call to Generalize(D_i, D_g)" could be written TopNode$(D_i) \in$ TopNode(D_g).otherType, or if one prefers TopNode$(D_g) \in$ TopNode(D_i).otherType. But a key remark is to notice that one of the graphs D_i or D_g is an unfolding of the other, and therefore one of the lists TopNode(D_i).otherType or TopNode(D_g).otherType contains at most one element (for instance, if D_g is an unfolding of D_i, each node of D_g corresponds to at most one node of D_i, hence TopNode(D_g).otherType contains at most one element). This enables us to rewrite the test as above, thus giving a constant time implementation of the loop test.

5.5.5 Program transformation

We use exactly the same program transformation techniques as for analysis L (see Chapter 4). The only difference is the stack allocation criterion. For analysis L, we composed context transformers labelling edges of a tree. For analysis T, we compare the label of TopNode(D_N) (where N is the control expression or the expression outside of which we want to allocate x) to the result of Propagate$'$(TopNode$(\rho[\![x]\!])$), following Theorem 5.14.

Chapter 6

Benchmarks and comparisons

6.1 Our benchmark programs

Our analyses have been implemented in the optimizing compiler of Ocaml 1.05. The Ocaml compiler comes with several test programs which we have used as benchmarks for our analyzer. Among these benchmarks, the opportunities for stack allocation are negligible in fft (fast fourier transform), sieve (Erathostene's sieve), taku and quicksort. Therefore, we have not considered these programs in the following. We have also used as benchmarks the Latex-to-html translator hevea, the Ocaml optimizing compiler itself, and the theorem prover Coq. The benchmarks are summarized in Table 6.1.

Benchmark programs		Lines
reynolds2	Binary tree search, allocating closures	21
reynolds3	Binary tree search, allocating lists	26
boyer	Terms computations	903
kb	Knuth-Bendix completion algorithm	616
nucleic	Floating point computations	3231
hevea	Latex-to-html translator (version 1.02)	7268
ocamlopt	Ocaml compiler (version 1.05)	58810
coq	Theorem prover (version 6.1)	68256

Table 6.1: Benchmark programs

6.2 Comparison between different escape analyses

We have tested the benchmarks with both escape analyses F and T, in several configurations:

- Inlining may be allowed (for functions of size at most 500) or forbidden.
- When a stack allocation and a tail call optimization cannot be done simultaneously, several behaviors are possible:

  All Always choose to preserve the tail call optimization, and do not perform the stack allocation.

Rec. Perform the stack allocation if the tail call cannot be recursive. Otherwise, preserve the tail call optimization.

None Always perform the stack allocation. This last choice is unsafe because it may lead to stack overflow, even if it does not happen on our benchmarks.

These benchmarks have been performed on a Pentium MMX 233 MHz, 128 Mb RAM, primary cache 16 kb instructions and 16 kb data, secondary cache 512 kb instructions and data, under Linux 2.0.32 (RedHat 5.0). The runtime parameters have their default values: GC ratio 30% (percentage of free memory including dead objects not yet collected, with respect to the total size of the heap, when allocated data become garbage at a constant rate; it is shown in [Dol95, page 155] that at the start of a GC cycle, the volume of the free list is 2/3 of this percentage, i.e. 20% of the total heap size), minor heap size 32768 words.

The results of these benchmarks are summarized in Figures 6.1 and 6.2. In these figures, the X axis "Cost" corresponds to the geometric mean of the ratio compile time with escape analysis divided by compile time without escape analysis. This ratio is expressed as a percentage of increase, i.e. 0% means ratio 1, 100% ratio 2. In Figure 6.1, the Y axis is the arithmetic mean of the percentage of stack allocated data (in size, not in number of values), weighted by the number of lines of each benchmark. The geometric mean has no meaning in this case, since the ratio is can be very near from 0 (for reynolds2 and reynolds3 when preserving no tail call optimization), leading to a geometric mean of 0 (100% decrease) independently of the results for other programs. In Figure 6.2, the Y axis is the geometric mean of the ratio runtime with stack allocation divided by runtime without stack allocation, expressed as a percentage of decrease, i.e. 0% means ratio 1 (no speedup), 100% means ratio 0 (the program terminates instantly). Each color corresponds to a tail call configuration (red for "All", green for "Rec." and blue for "None"). The configuration with inlining is represented by a circle, and without inlining by a cross.

The cost, stack allocated size, and runtime decrease are also mentioned in Table 6.6, for the configuration with inlining and preserving recursive tail calls. This table is a quick summary of the main features of our escape analyses.

The brute results of the benchmarks are detailed in Tables 6.2 to 6.4. Table 6.2 presents the runtime decrease, computed by the formula $(T_{old} - T_{new})/T_{old}$ where T_{old} is the execution time without stack allocation and T_{new} the execution time with stack allocation. The column on the right indicates the runtime without optimization for reference. For small benchmarks, the uncertainty on time measures is about 1 to 2%, although we have run each benchmark 30 times and taken the mean to improve the precision. For ocamlopt, each benchmark has been run 3 times, and for coq 5 times. Table 6.3 indicates the percentage of stack allocated data (in size). The column on the right indicates the total size of allocated data without optimization. At last, Table 6.4 summarizes the cost of our analysis, as a percentage of the total compile time without escape analysis. The compile time is mentioned in the right column for reference. These tables contain many figures, so we have tried to summarize the results by computing means. These means have of course no statistical significance. We have considered the arithmetic mean weighted by the number of lines of each benchmark (in the "arith. mean" row) and the unweighted geometric mean (in the "geom. mean" row).

We can draw the following conclusions from these results:

- As we could expect, analysis T detects more stack allocation opportunities than analysis F. However, the difference between these analyses is small.

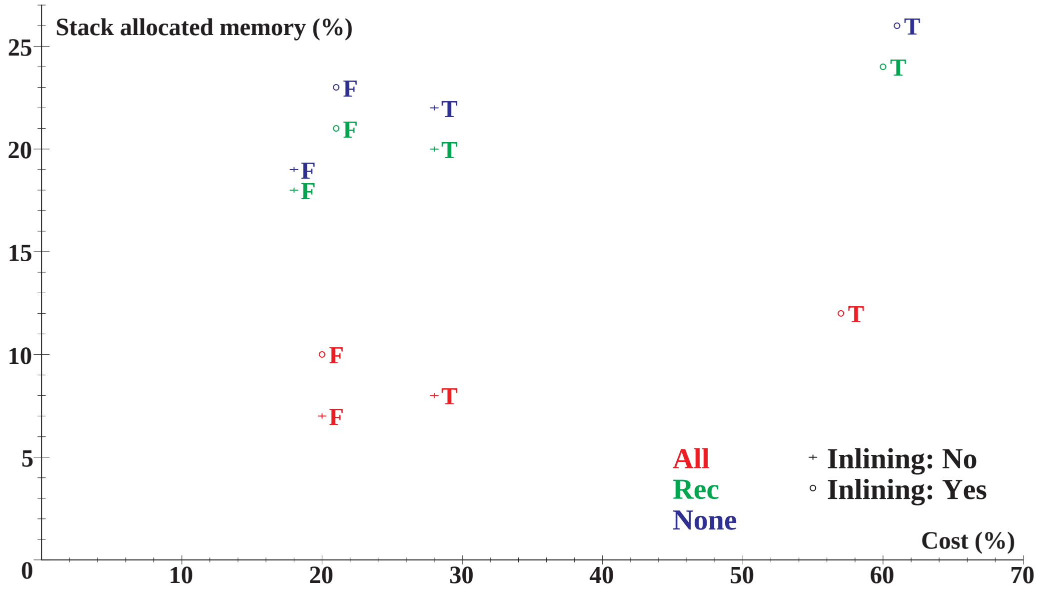


Figure 6.1: Stack allocated memory as a function of the cost of the escape analysis

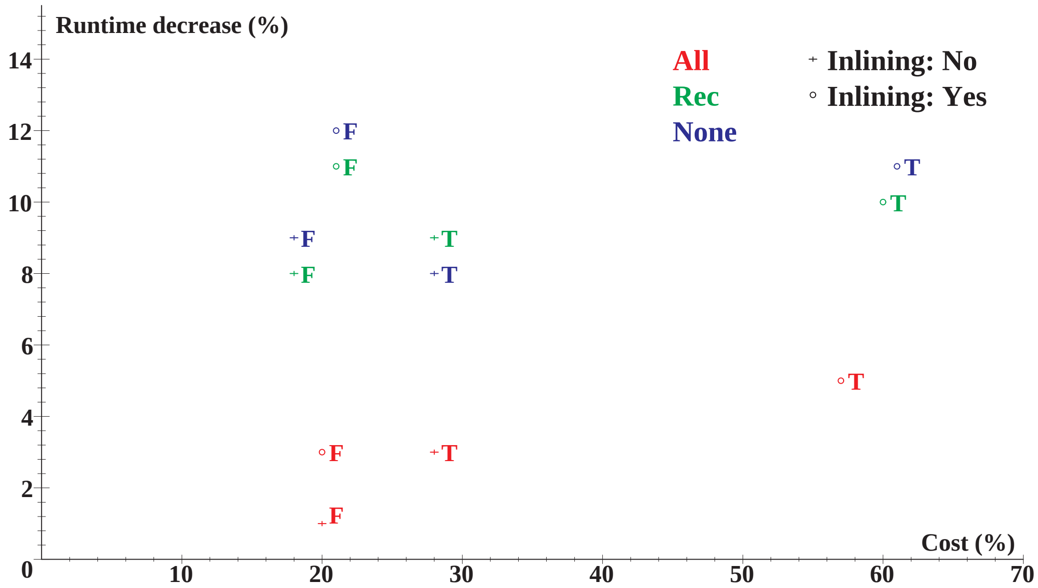


Figure 6.2: Runtime decrease as a function of the cost of the escape analysis

Runtime decrease (%)							Time (ms)
Inlining	No			Yes			
Preserve	All	Rec.	None	All	Rec.	None	
Analysis F							
reynolds2	4	4	8	4	4	8	5248
reynolds3	2	2	5	2	2	6	2201
boyer	0	11	11	0	12	12	476
kb	-9	18	16	-6	21	18	1353
nucleic	3	5	3	10	16	20	498
hevea	12	22	20	17	23	21	10735
ocamlopt	-1	4	4	0	3	6	81627
coq	1	0	0	-1	2	4	188676
arith. mean	0	3	3	0	4	6	
geom. mean	1	8	9	3	11	12	
Analysis T							
reynolds2	4	4	8	4	4	8	5248
reynolds3	2	2	6	2	3	6	2201
boyer	0	12	12	10	11	12	476
kb	0	23	20	0	23	21	1353
nucleic	4	6	3	11	16	20	498
hevea	18	20	15	16	11	10	10735
ocamlopt	-2	1	1	0	2	2	81627
coq	0	0	1	-1	4	5	188676
arith. mean	0	1	2	0	4	4	
geom. mean	3	9	8	5	10	11	

Table 6.2: Runtime decrease

Analysis T can be very fast for simple programs, especially when the size D of the graph of a type remains small. It is even faster than analysis F for boyer, kb and nucleic. However, the cost of analysis T increases quickly when the size of the benchmark increases. The cost of analysis F is much smaller for large benchmarks.

Therefore, even if analysis F is the least precise, it gives the best cost/precision compromise.

Table 6.5 gives some profiling results for analysis F. These figures correspond to the compilation of all our benchmarks. We can be surprised by the large amount of type spent for the computation of type heights. This part of the code has been optimized, but it remains more complex than what one could think. (In particular, polymorphic type declarations are difficult to handle.) Most of the time is spent for the generation of equations. This part of the escape analysis computes a complex information: for each expression, it computes whether it is in tail position or not, whether it performs a (recursive) tail call or not, and two escape contexts, one for `letstack`, and one for `letstack'`. As a comparison, on average, escape analysis takes less than 1.5 times the time of typing. The increase of the rest of the compile time mainly comes from inlining, which increases the size of the code to be compiled. (Even without escape analysis, the

Allocated size decrease (%)							Size (Mwords)
Inlining	No			Yes			
Preserve	All	Rec.	None	All	Rec.	None	
Analysis F							
reynolds2	49	49	99	49	49	99	10.5
reynolds3	49	49	99	49	49	99	3.1
boyer	0	16	16	0	16	16	1.2
kb	1	46	47	1	46	47	8.3
nucleic	11	12	13	34	53	54	3.8
hevea	21	23	29	22	24	31	9.2
ocamlopt	10	22	22	11	22	22	140.6
coq	3	14	16	7	19	21	562.4
arith. mean	7	18	19	10	21	23	
Analysis T							
reynolds2	49	49	99	49	49	99	10.5
reynolds3	49	49	99	49	49	99	3.1
boyer	0	16	16	13	16	16	1.2
kb	1	46	47	1	46	47	8.3
nucleic	11	12	13	34	53	54	3.8
hevea	22	24	31	24	26	32	9.2
ocamlopt	10	22	22	11	22	22	140.6
coq	5	18	20	11	24	27	562.4
arith. mean	8	20	22	12	24	26	

Table 6.3: Size of stack allocated data

program transformation already complicates the code a bit, by introducing `let`s.)

For boyer when preserving tail call optimizations, analysis T with inlining yields 13% allocated size decrease whereas other analyses yields 0% stack allocation. This comes from a subtle interaction between analysis T and inlining. Boyer contains the code fragment

```
let f x y =
   let rec frec x =
      ...
   in
   frec x
```

The closure `frec` cannot be stack allocated when preserving tail call optimizations because `frec x` is a tail call. But if `f` is inlined, the call to `frec` may not be in tail position any more. Due to technical differences in the implementations, analysis T recomputes the tail status when performing inlining, and can thus stack allocate `frec` when `f` is inlined. To save time, analysis F safely assumes that the call remains in tail position even after inlining. Therefore, it cannot perform this stack allocation.

- Inlining increases stack allocation opportunities for nucleic, coq and hevea. Indeed, a value can only be stack allocated if its allocator is syntactically present. The inlining of

Compile time increase (%)							Compile time (s)
Inlining	No			Yes			
Preserve	All	Rec.	None	All	Rec.	None	
Analysis F							
reynolds2	10	5	6	7	7	7	2.4
reynolds3	6	5	5	6	7	8	2.4
boyer	27	21	21	21	21	21	4.4
kb	18	17	16	17	16	17	4.6
nucleic	29	27	27	34	37	36	7.4
hevea	13	13	14	15	15	15	31.6
ocamlopt	31	31	31	34	34	34	99.2
coq	28	27	27	35	35	34	337.1
arith. mean	28	28	28	33	34	33	
geom. mean	20	18	18	20	21	21	
Analysis T							
reynolds2	6	6	6	6	6	6	2.4
reynolds3	6	6	6	6	7	6	2.4
boyer	13	14	14	13	13	13	4.4
kb	12	13	13	15	15	15	4.6
nucleic	22	21	21	29	33	33	7.4
hevea	15	15	15	17	17	19	31.6
ocamlopt	72	73	74	77	77	77	99.2
coq	109	107	106	848	934	988	337.1
arith. mean	85	84	84	450	493	519	
geom. mean	28	28	28	57	60	61	

Table 6.4: Analysis cost

functions which allocate their result exposes an allocator which was encapsulated in a function. It is logical that inlining only has an effect for large and modular programs.

Its cost can be very high for analysis T (in the coq benchmark). This comes from the fact that the graph representing the analysis of the inlined function has to be copied into the call site to be able to perform stack allocations inside the inlined function. Such copies are very costly when much inlining is performed.

- Preserving all tail call optimizations much reduces the stack allocation opportunities, and also the speedups. Our analysis of recursive calls, even if it is simple, is precise enough to detect that most tail calls are not recursive. Most stack allocations can then be performed while preserving recursive tail call optimizations (except on reynolds2 and reynolds3). Not preserving tail call optimizations (the None configuration) of course yields the maximum stack allocation, but it may lead to stack overflow. Moreover, the None configuration does not always yield the maximum speedup (on kb and hevea). Indeed, stack allocation slows down the program when it increases too much the stack size. In particular, data locality is worse, as we shall see in the following section. The best configuration is therefore Rec.

Computing type heights	5.1
Path compression	1.6
Rest of generation of equations	8.9
Total time of generation of equations	15.6
Solving of equations	0.8
Program transformation	1.6
Rest of escape analysis	2.5
Total time of escape analysis	20.5
Increase in the rest of compilation	8.1
Total compile time increase coming from escape analysis	28.6

Table 6.5: Profiling results, for analysis F, with inlining, in the Rec. configuration. All values are percentages of the compile time without escape analysis.

Analysis	Context transformers	Complexity
E	$\lambda c. \cup \{\phi(s) \mid s \in c\}$ where $c \in \mathcal{P}(Path)$	∞
T	$\lambda c. \cup \{\phi(s) \mid s \in c\}$ where $c \in \mathcal{P}(\mathrm{Nodes}(\mathrm{Gr}(\tau)))$	$\mathcal{O}(nD^2 + NCalls\, D^3)$
L	$\lambda c.(\text{if } c = \infty \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$	$\mathcal{O}(n^2 \log^2 n)$
F	$\lambda c.(\text{if } c = \infty \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$	$\mathcal{O}(n \log^2 n)$

n	size of the program
D	maximum size of the graph of a type
$NCalls$	number of function calls

Analysis	Cost (%)	Stack allocated size (%)	Runtime decrease (%)
T	60	24	10
F	21	21	11

Table 6.6: Summary of all our escape analyses

Moving `lets` and using `letstack`' are useful since they enable us to increase stack allocation opportunities, while preserving tail call optimizations. They also reduce the size of the stack by optimizing the lifetime of data (see [Bla96] for a detailed study of the effect of these two optimizations).

In the following benchmarks, we will only consider analysis F with inlining. We will consider two configurations Rec. and None, since the None configuration yields more stack allocation, but is unsafe.

6.3 The effects of stack allocation

Figure 6.3 represents the percentage of size of stack allocated data in our benchmarks. It gives an idea of how important the effect of stack allocation will be. We can expect important speedups for kb and nucleic, and the best results should be obtained for reynolds2 and reynolds3. By studying the source code of the benchmarks, we can define several patterns of code in which our analyzer performs stack allocation.

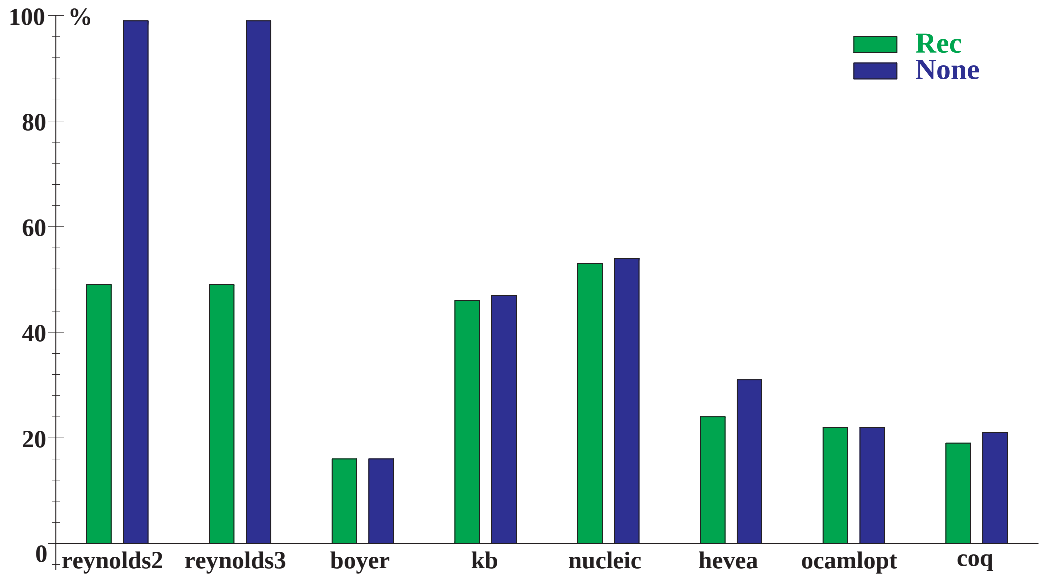


Figure 6.3: Percentage of stack allocated memory

- Stack allocation of closures in the definitions of recursive functions. The code is the following:

```
let f x y z =
   let rec frec x = ... frec x' ...
   in frec x
```

  where all occurrences of `frec` are fully applied. In such a case, `frec` can be stack allocated. This pattern of code is very important in boyer and kb. Notice that, in this case, stack allocation cannot be performed when preserving all tail call optimizations (because `frec x` is a tail call). It can be performed in configuration None. In most cases, our analyzer can detect that `frec x` is not a recursive call, and therefore the optimization is also performed in the Rec. configuration.

  This code could also be optimized by eta-expansion:

```
let rec f x y z = ... f x' y z ...
```

  This avoids creating a closure for `frec`, but leads to more parameter passing. Which optimization gives the best results depends on the number of iterations of the recursive call, and on the number of parameters.

- Stack allocation of parameters of higher-order functions. For example:

```
map (fun x -> x+1) l
```

  In this situation, the closure `fun x -> x+1` can be stack allocated. This could also be optimized by inlining the higher-order function `map` and avoiding the creation of the closure. However, this would lead to code duplication. This pattern appears many times in kb for example.

- Stack allocation of parameters of first-order functions. For example,

  ```
  let f { f1 = ...; f2 = ... } = ...

  f { f1 = ...; f2 = ... }
  ```

  Such a simple example could of course also be optimized by exploding the parameter. This is done by the compiler when the parameter is a tuple (this is the case in taku for example). However, when the parameter is not decomposed immediately by the callee, exploding becomes more difficult, and may even be impossible. This is the case in the reynolds2 and reynolds3 benchmarks. Let us consider this extract from reynolds3:

  ```
  let rec search p t =
     match t with
          Lf -> false
        | Br(x, t1, t2) ->
          if member x p then true
          else
             if (search (x::p) t1) then true
             else search (x::p) t2

  let _ = search [] (mk_tree 19)
  ```

  The list `p` can be stack allocated, but not exploded.

- Stack allocation of results of functions. For example:

  ```
  let f x = { f1 = ...; f2 = ... }

  let y = f x
  in ... y.f1 ... y.f2 ...
  ```

  The result of `f` can only be stack allocated by inlining the function `f`. This case is very important for nucleic. When the result of `f` is not passed to another function as a parameter, it could even be exploded (stored in several registers) instead of stack allocated. Preallocation (on the stack) would also be an alternative optimization for this case, to avoid inlining. Exploding objects [GS00] and preallocation are two other applications of escape analysis, which we have not yet considered in our study. We have chosen to focus on pure stack allocation, using inlining to increase stack allocation opportunities.

Of course, the optimizations performed by our analyzer are not limited to these simple patterns, but they represent an important part of all stack allocated data. Closures, in particular, represent an important part of the stack allocated data. For analysis F in the Rec. configuration with inlining, other stack allocated data represent at most 1% of data in reynolds2, boyer, kb, hevea and ocamlopt; stack allocated closures represent 10% of all data for nucleic, 13% for coq. Complex data structures are very often allocated and returned by recursive functions, and they cannot be stack allocated in this case. This limits the amount of possible stack allocation in ML programs. The above code patterns suggest that simpler and even faster escape analyses might yield good results. Such analyses have to satisfy certain conditions to give satisfactory results:

- They should be interprocedural, to detect when closures passed as parameters to higher-order functions can be stack allocated. (Otherwise, for closures, they would only detect that code with local functions but not higher-order functions, that is Pascal-like code, can be implemented without heap allocating closures. This is already what is done in the implementations of Pascal, but not in most current implementations of ML.)
- They should compute a fixed point for recursive definitions. Otherwise, the first parameter of `map` and similar recursive functions would escape. Indeed, this parameter is again passed to the recursive calls to `map`.

Several such analyses come to mind:

- An escape analysis targeted only towards stack allocation of closures would be a first idea. We would have only two type heights: 0 for types which do not contain closures, and ∞ for types which contain closures. The context transformers would then be simpler, and the analysis faster.
- We could use an escape analysis with only two contexts for each value. The escape context would indicate whether the top location of the value escapes (context $\top$) or not (context $\bot$). A value would be considered as escaping as soon as it is reachable from another data structure. This analysis would be similar to the escape analysis for Java of [GS00]. In practice, this analysis gives significantly less precise results than our escape analysis for Java, but the results might be different for ML. Only an implementation of this analysis could really show the results, but we can already notice that this simple analysis would not be able to allocate data of reynolds3 on the stack. It would probably detect most of the other stack allocations in the above code patterns. Notice that we cannot apply here the technique used in analysis F to avoid the enumeration of free variables of closures. If the result of an expression escapes, using the technique of analysis F, all free variables of the expression would escape. Therefore, we have to enumerate explicitly the free variables of closures, and the worst-case complexity of the analysis algorithm is at least quadratic. Hence, the improvement over analysis F is not clear.

Stack allocation obviously increases the maximum stack size. Table 6.7 indicates the percentage of stack size increase due to stack allocation. The column on the right indicates the maximum stack size without stack allocation for reference.

This table shows the interest of preserving tail recursion optimizations: for the kb program, it limits the stack size increase to less than 150%, instead of more than 300% otherwise. A 40% stack size increase is avoided for nucleic, and 120% for hevea. The stack size does not change for boyer, ocamlopt and coq, since even if data are stack allocated, this is not done when the stack size is greatest.

The graphs of Figures 6.4 and 6.5 display the decrease of the runtime and detail the reasons of the speedups. The "Time" bar represents the percentage of runtime decrease, computed by the formula $100 \times (T_{old} - T_{new})/T_{old}$ where T_{old} is the runtime without optimization and T_{new} the runtime with stack allocation. The "GC" bar represents the part of this percentage coming from a decrease of the GC time. The "let" bar represents the part coming only from

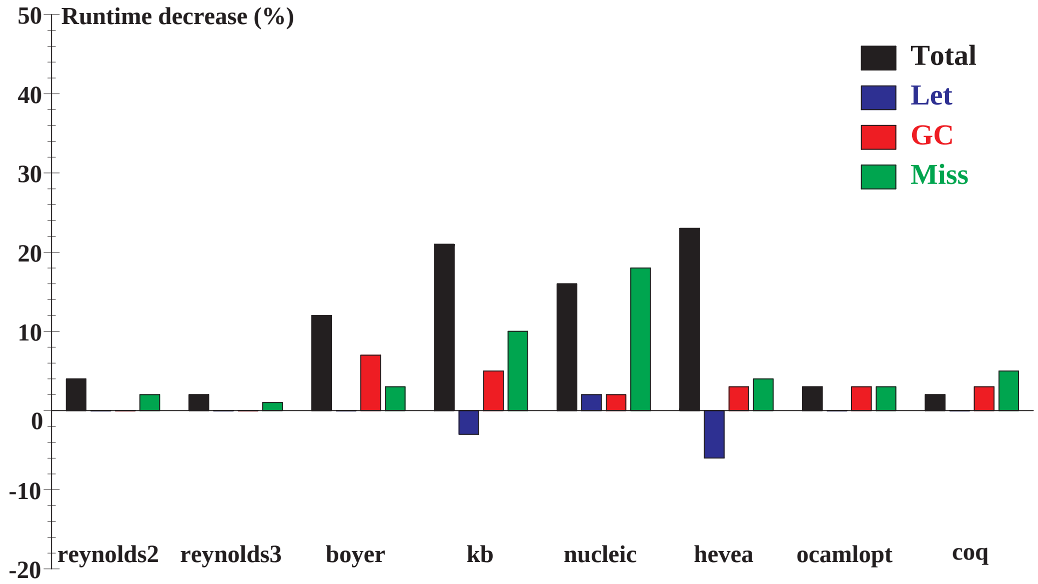


Figure 6.4: Tested on Pentium MMX, preserving tail recursion optimizations (configuration Rec.)

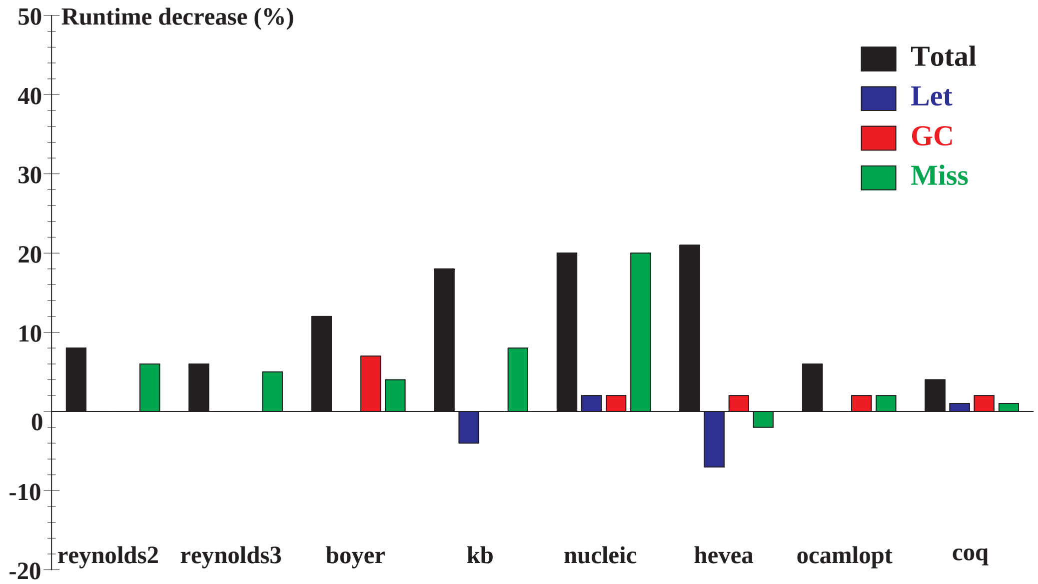


Figure 6.5: Tested on Pentium MMX, preserving no tail call optimization (configuration None)

Stack size increase (%)				Size
Preserve	All	Rec.	None	(bytes)
reynolds2	86	88	88	464
reynolds3	51	51	51	444
boyer	0	0	0	1480
kb	120	147	306	65516
nucleic	4	4	44	2068
hevea	0	0	120	472932
ocamlopt	0	0	0	214076
coq	0	0	0	36756

Table 6.7: Maximum stack size

adding `lets` and inlining (in this case, there is no stack allocation; `lets` have only been added where there are `letstacks` in the version with stack allocation). The "miss" bar is an estimate of the part coming from a better data locality. This estimate is based on the number of cycles the processor is stalled waiting for a read to memory or because the write buffer is full. The effects on data locality will be detailed in the following.

The effect of only adding `lets` and of inlining, as it appears on the graphs, is nearly negligible, as we could expect. The decrease of the number of function calls which could speed up the programs is balanced by the code size increase, which increases the number of instruction cache misses. Exceptions are hevea and kb for which inlining has a negative effect.

Table 6.8 indicates the part of the GC time on the runtime decrease. The last column indicates the runtime in milliseconds without stack allocation for reference. The difference with the previous tables comes from uncertainty of measures and the time taken by measures themselves.

Runtime decrease coming from the GC (%)				Time
Preserve	All	Rec.	None	(ms)
reynolds2	0	0	0	5269
reynolds3	0	0	0	2205
boyer	0	7	7	490
kb	-3	5	0	1387
nucleic	0	2	2	514
hevea	2	3	2	9378
ocamlopt	1	3	2	83123
coq	1	3	2	195740

Table 6.8: Runtime decrease coming from the GC

The speedup coming from the GC time is very small. This may be surprising at first sight. But in fact data that can be stack allocated are typically short lived data. These data are in general unreachable at the first minor GC after their allocation. Since the GC only scans reachable data, these data had no cost for the GC when they were in the heap. Nevertheless, the advantage of stack allocation is that it increases the time between two GC cycles, since there are less data allocated on the heap. For programs in which all data can be stack allocated, there is no GC with stack allocation, but it took very little time without stack allocation. For boyer, the runtime decrease is more important (7%). This comes probably from the fact that data stack allocated in boyer have a longer lifetime, and therefore a greater effect on the GC.

Table 6.9 gives the percentage of decrease of the GC time. The rightmost column mentions the GC time when there is no stack allocation (in ms).

GC time decrease (%)				GC time
Preserve	All	Rec.	None	(ms)
reynolds2	60	68	100	23
reynolds3	19	30	100	3
boyer	0	11	12	317
kb	-10	17	0	422
nucleic	2	46	41	32
hevea	17	18	16	1556
ocamlopt	2	6	5	42073
coq	5	11	10	51880

Table 6.9: GC time decrease

This table displays results which are obviously correlated with the previous one. The uncertainty of measure is relatively greater here because the total time measured are smaller, and fragmented. As we could expect, when nearly all values can be stack allocated, the GC time becomes zero (100% decrease). The decrease of the GC time is important for boyer and coq. It is surprisingly small on kb in the "None" configuration, whereas it is important in the "Rec." configuration. This is probably related to the increase of the stack size in the "None" configuration, which increases the time to scan the stack looking for roots (even if the minor GC does not scan stack allocated objects when looking for roots, the major GC has to scan the whole stack).

Tests show that the effect is nearly independent of the parameters of the GC (see [Bla96] for precise numerical results). This is also related with the fact that data that would be stack allocated have no cost for the GC. This does not depend on the size of the minor heap or on the GC ratio.

In a first implementation, kb presented a data retaining phenomenon which decreased the speedup. The lifetime of some data was overestimated, since we considered as roots all stack allocated data to maintain living data for the garbage collector. This behavior is safe, but a piece of data D may remain in the stack longer than its real lifetime. If a minor GC cycle happens when D is in the stack but unreachable, data pointed to by D in the minor heap will be copied uselessly, and this causes a loss of time. This problem has been solved by modifying the GC so that it follows pointers in the stack during the copying, as mentioned in Section 4.6. Also notice that "bad luck" may be responsible for a small increase of the major heap size as, when a minor GC takes place, there is always a small amount of short lived data that will be copied to the major heap because they are alive at this time. With stack allocation, we change the moment when the minor GC is done, so these data are not the same. Their size may be bigger when we stack allocate data than when we do not.

Table 6.10 indicates the decrease of the number of L1 data cache misses. The fifth and last columns indicate respectively the number of read and write L1 cache misses. These numbers

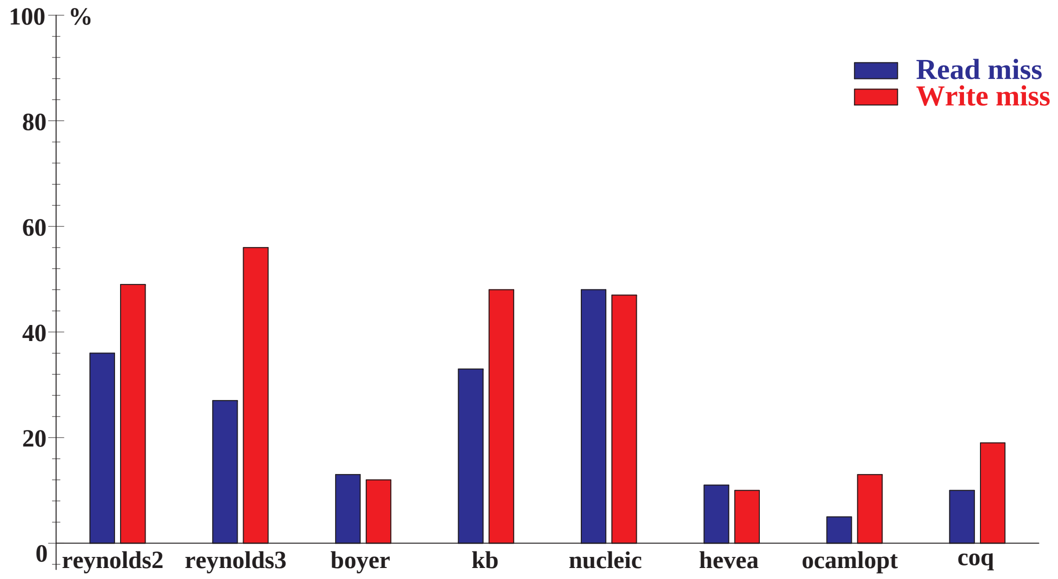


Figure 6.6: Decrease of the number of cache misses on Pentium MMX, when preserving recursive tail call optimizations (configuration Rec.).

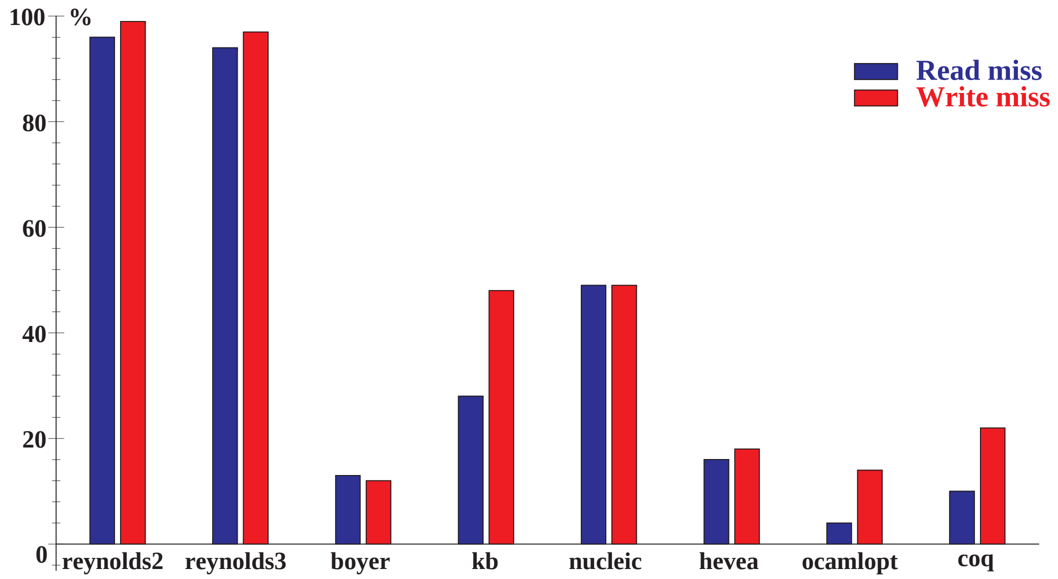


Figure 6.7: Decrease of the number of cache misses on Pentium MMX, when not preserving tail call optimizations (configuration None).

can be measured thanks to the performance-monitoring counters of the Pentium processor [Int99, Appendix A].

	Read miss				Write miss			
	Decrease (%)			Thousands	Decrease (%)			Thousands
Preserve	All	Rec.	None		All	Rec.	None	
reynolds2	35	35	97	1129	49	49	99	7957
reynolds3	31	27	94	463	47	56	97	2069
boyer	0	13	13	551	0	12	12	1600
kb	0	35	29	1893	1	48	48	5911
nucleic	32	49	50	666	32	47	49	3016
hevea	14	18	15	7572	22	22	19	14706
ocamlopt	1	5	5	109291	5	14	14	189007
coq	2	9	10	214042	5	21	23	454898

Table 6.10: Cache misses

We can notice that the number of cache misses is much reduced thanks to stack allocation. The percentage of decrease is nearly the percentage of stack allocated data. Stack allocation improves data locality [CMT94], since all useful values are near the top of the stack. This part of the stack remains in the cache since it is used very often. We can notice that the number of write misses is much higher than the number of read misses, and that it is more reduced by stack allocation than the number of read misses. This comes from the fact that nearly all allocations in the heap lead to a write miss: the newly allocated space is not in the cache. Indeed, the primary cache cannot contain the whole young generation (128 kb). Therefore, it contains in general only the most recently allocated objects. On architectures with non-blocking reads, or a specific prefetch instruction, this situation could be improved by explicitly ***prefetching*** the future allocation space (at each allocation, we would load in the cache memory for the next allocations).

The effect of this high number of write misses also heavily depends on the cache structure. When a write does not hit the cache, several techniques can be applied:

- ***Write around***: the value will be written directly to memory. To reduce the cost of this operation, the writes can be accumulated in a ***write buffer***, and do not block the execution of the following instructions. Write misses are followed by read misses when the newly allocated value is read. This increases the number and total cost of read misses. Write around is the technique used in the Pentium MMX.

- ***Fetch-on-write***: the cache line is loaded from memory, and the value written into the newly loaded cache line. Depending on the processor, the value may also be written to the memory. This leads to a very high penalty for a write miss.

- ***Write allocate***: a new cache line is allocated. It is marked invalid, except at the address just written which is stored in the cache. With such a technique, there is no penalty for a write miss. The rest of the cache line will be filled by subsequent allocations. Moreover, subsequent reads need not access the main memory since data just written are in the cache. Stack allocation would bring less improvement with this kind of cache, since the cost of cache misses is very small even with heap allocation. However, such an architecture complicates the design of the cache, and the cache coherence protocols in the case of multiprocessor architectures (there must be a valid bit for each word of a cache line).

These architectures have been discussed in [AS96, GA95].

Tables 6.11 and 6.12 translate the numbers of cache misses into a potential decrease of the runtime. For read misses, we have measured the number of cycles in which the processor is stalled because it is waiting for a read from memory to complete. This can be translated very precisely into a contribution in the runtime of the program. The results are summarized in Table 6.11. Columns 2 to 4 correspond to the decrease of the number of cycles during which the processor is stalled, waiting for a read from memory. Column 5 mentions this number of cycles without stack allocation for reference. Columns 6 to 8 translate the results into a percentage of runtime decrease, due to a decrease of read stalls.

	Number of data read stalls				Runtime decrease due to decrease of read stalls (%)			Time
	Decrease (%)			Millions				(ms)
Preserve	All	Rec.	None		All	Rec.	None	
reynolds2	37	37	96	75	2	2	5	5283
reynolds3	33	33	94	28	1	1	5	2222
boyer	0	13	14	31	0	3	4	471
kb	-1	34	28	97	0	9	7	1503
nucleic	29	43	50	49	10	16	18	571
hevea	6	28	-14	318	0	4	-2	9541
ocamlopt	11	11	6	6024	3	3	2	82260
coq	3	24	7	10384	0	5	1	195280

Table 6.11: Read stalls

The cost of a read miss contains the time necessary to load a cache line from memory, but also to flush the write buffer (since the Pentium MMX cannot read directly from the write buffer when the value is not yet written to the memory), write a cache line to memory if the loaded cache line replaces a dirty line (a line that has been modified in the cache, but not yet in memory). On average, the time of a read miss is about 60 cycles (250 ns).

Table 6.12 is similar to Table 6.11, but for write misses. It indicates the number of cycles while the pipeline is stalled due to full write buffers. A write miss has a very small cost in general, since data are simply written to a write buffer. The pipeline is only stalled when the write buffer is full. Moreover, the time spent waiting for full write buffers is very small. Hence the cost of write misses is small, even if there are many write misses.

Table 6.11 shows that a large part of the speedup comes from an improved data locality. For kb, hevea, ocamlopt, coq, the data locality is better when preserving recursive tail call optimizations. This comes from the fact that the stack size is much increased in the "None" configuration. Therefore, the locality of data in the stack is not so good in this configuration. For nucleic and coq, the speedup coming from data locality improvements is greater than the observed speedup. This comes probably from uncertainties of measures (for coq, the runtime decrease when measuring stalls was up to 13% instead of only 5% in the other benchmarks).

Remark. The floating point registers of the Pentium processor are organized as a stack, and the register allocation of the Ocaml compiler does not fit well with this organization. Adding

	Number of data write stalls				Runtime decrease due to decrease of write stalls (%)			Time (ms)
	Decrease (%)			Millions				
Preserve	All	Rec.	None		All	Rec.	None	
reynolds2	48	48	96	13	0	0	1	5283
reynolds3	3	4	49	0	0	0	0	2222
boyer	0	12	12	6	0	0	0	471
kb	-1	33	30	12	0	1	1	1503
nucleic	45	53	56	5	1	2	2	571
hevea	13	17	10	41	0	0	0	9541
ocamlopt	3	10	8	868	0	0	0	82260
coq	3	19	18	1175	0	0	0	195280

Table 6.12: Write stalls

lets on floats then introduces useless moves, leading to an important time loss. Therefore, we have modified our analyzer so that `letstack`s are not introduced on floats that the compiler can allocate in registers (unboxed floats). These `letstack`s were useless anyway. This avoids a 25% time loss on nucleic when not preserving tail recursion.

6.4 Comparison between different machines

Table 6.13 presents a collection of 6 different machines on which we have tested our analysis. The upper table indicates the sizes of caches of these machines. The primary data cache is fetch-on-write on Sparc 5 (data are also written to the main memory) [Sun97a, page 2], write around on UltraSparc [Sun97b, page 20], Alpha [Sam97, page 2-13] and Pentium MMX [Int99, page 9-5]. The Pentium Pro has a write allocate cache (several configurations are in fact programmable [Int99, page 9-6], write allocate is one of them). They are ordered by increasing SpecInt95. The two bottom tables present the speedups on these machines. The “All” and “None” columns indicate the runtime decrease in %, in the corresponding configurations of our analyzer. The “Time” column mentions the runtime in ms for reference.

The Alpha 3000 is the machine which benefits the most from stack allocation. The Sparc 5, Pentium MMX and Alpha 500 also benefit much from stack allocation. This optimization gives very few speedups on Pentium Pro and a bit more on UltraSparc (reynolds3 is strangely slowed down by stack allocation).

This can be explained by a different structure of the cache for these machines. The Pentium Pro has a very efficient cache. The L1 cache is small (only 8 kb) but the penalty for an access the L2 cache is very small (3 cycles, 15 ns) [SL96]. Since the minor heap will be most of the time in the L2 cache (its size is 128 kb, where the L2 cache is 256 kb), all accesses to the minor heap are very fast, even without stack allocation. Moreover, write allocate caches reduce the cost of cache misses. Therefore stack allocation does not yield important benefits. The cache is also fast on UltraSparc (latency 7 cycles).

On machines that really benefit from stack allocation, not preserving tail call optimizations increases the speedups, because the programs can then perform more stack allocations and the benefits of stack allocation are greater than the cost of the additional function call. On the contrary, on Pentium Pro and UltraSparc, the tail call optimizations sometimes yield

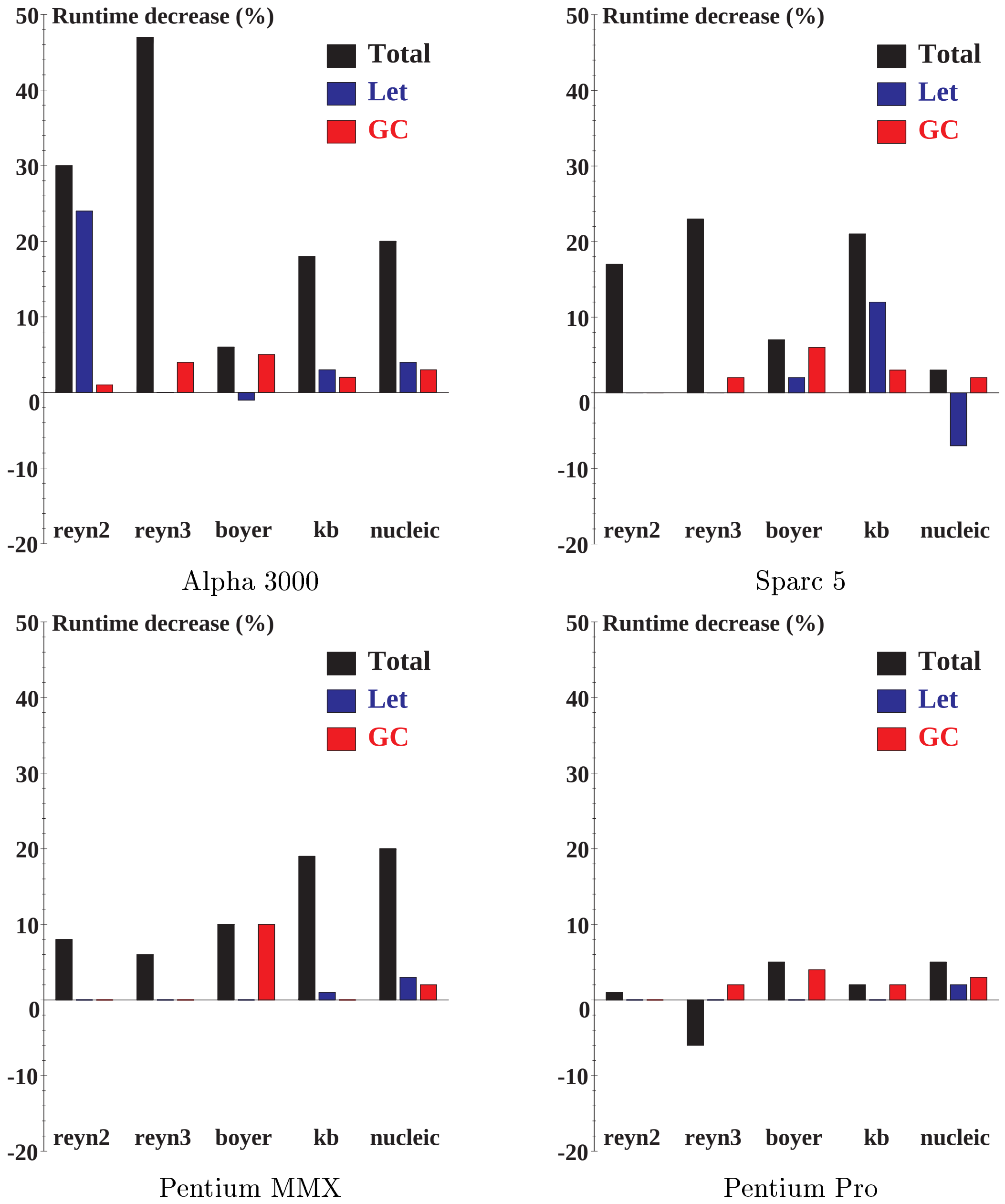


Figure 6.8: Effect of stack allocation on various machines, preserving no tail call optimization(1). The programs reyn2 and reyn3 correspond to reynolds2 and reynolds3 respectively.

Tested machines	Primary cache	Secondary cache	SPECint95
Alpha DEC 3000/300	8kb(I) + 8kb(D)	512kb(I+D)	Unknown
Sparc 5 110MHz	16kb(I) + 8kb(D)	None	1,59
Pentium MMX 233MHz	16kb(I) + 16kb(D)	512kb(I+D)	7,02
Pentium Pro 200MHz	8kb(I) + 8kb(D)	256kb(I+D)	8,20
Ultra 5 270MHz	16kb(I) + 16kb(D)	256kb(I+D)	9,16
Alpha DEC 500/400MHz	8kb(I) + 8kb(D)	96kb(I+D) + 2Mb(I+D)	12,3

I = instruction cache; D = data cache

	Alpha 3000			Sparc 5			Pentium MMX		
	All	None	Time	All	None	Time	All	None	Time
reynolds2	-13	30	15470	4	17	8681	4	8	5240
reynolds3	2	47	9459	3	23	5814	3	6	2202
boyer	0	6	1781	2	7	1014	0	10	429
kb	2	18	4793	8	21	3760	0	19	1320
nucleic	6	20	1313	-2	3	1635	10	20	498

	Pentium Pro			UltraSparc			Alpha 500		
	All	None	Time	All	None	Time	All	None	Time
reynolds2	1	1	6072	-8	7	2962	3	9	1930
reynolds3	2	-6	3070	-7	-13	1060	2	6	702
boyer	0	5	523	0	9	341	0	7	190
kb	0	2	1339	6	10	1138	0	8	587
nucleic	6	5	692	2	14	399	11	17	147

Table 6.13: Decrease of the runtime on different computers

greater benefits than stack allocation (for reynolds3 in particular). We can conjecture that a stack allocation yields more speedup when the size of the stack allocated value is greater (since more GC time is then saved and the cache behavior is improved for a greater number of accesses). Then, a size limit can exist above which stack allocation is more interesting than tail call optimization. However, this step depends on complex factors: the cache behavior depends on the rest of the program; the decrease of GC time depends not only on the size of the considered value, but also on other values (the cost of the GC cannot be precisely split into a cost for each value). This could however explain that reynolds2 is slightly improved by not preserving tail recursion, whereas reynolds3 becomes slower, since reynolds2 allocates closures which are larger than the list cells (pairs) allocated by reynolds3.

Figures 6.8 and 6.9 represent the speedups obtained on each of the tested machines, in the “All” configuration. They also indicate the causes of the speedups in the “Let” and “GC” bars. The decrease of the GC workload does not depend much on the machine. It is always very small, except for the boyer benchmark. The difference of runtime when adding lets and inlining functions can sometimes be important. Inlining has a negative effect for nucleic on Sparc 5, which leads to disappointing results in this case. This comes from an increased code size, which may increase the number of instruction cache misses, and from changes in register allocation (inlining may increase the register pressure). In contrast, kb is much faster after introducing lets (even without inlining) on Sparc 5. This comes probably from changes in

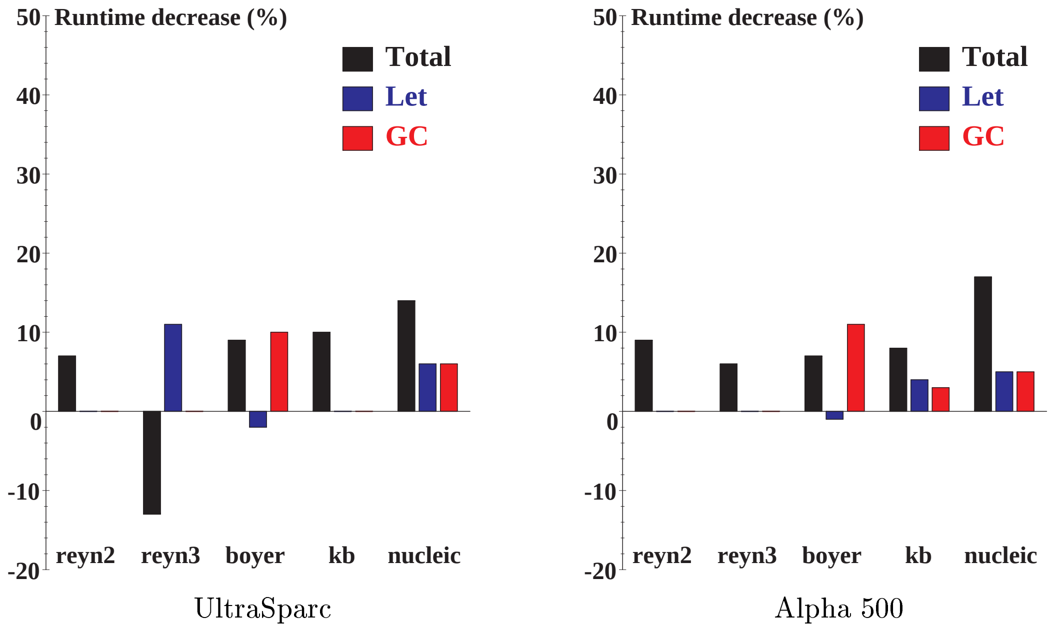


Figure 6.9: Effect of stack allocation on various machines, preserving no tail call optimization(2). The programs reyn2 and reyn3 correspond to reynolds2 and reynolds3 respectively.

the register allocation or code placement. Similarly, reynolds2 becomes faster on Alpha 3000 when introducing lets.

6.5 Conclusion

We can draw several conclusions from this study of escape analysis on Objective Caml:

- Using integers to represent the escaping part of values yields a very fast and still precise analysis. It is the best compromise among the analyses we have tested. The analysis using annotated types is a bit more precise but much more costly on large benchmarks, even though we have performed many optimizations.
- With the GC of Objective Caml, the speedups due to stack allocation do not come from a decrease of the GC workload, but from an improved data locality. The amount of speedup we obtain is limited by the high quality of the GC of Objective Caml.
- The speedups highly depend on the performance of the cache: the better the cache, the less important the speedups.
- The inlining of small functions is useful to increase stack allocation opportunities. It also increases to code size. Alone, it can slow down the execution of the programs, but the combination of inlining and stack allocation is in general beneficial (when it has an effect).
- Preserving recursive tail call optimizations is a good compromise. Most stack allocations can be performed in this configuration, and the stack size remains small. Thus, this configuration avoids stack overflows. It is safer than the None configuration.

Part II

Escape Analysis for Java

Chapter 7

Design of escape analysis for Java

7.1 Notations

The set of integers $\{a, ..., b\}$ is denoted by $[\![a, b]\!]$. The set of lists (or words or tuples) whose elements are in S is S^*. The empty list is $[]$. The list l at the head of which p_1 has been concatenated is $p_1 : l$. The function which maps x to $f(x)$ is $\{x \mapsto f(x)\}$. The extension of f which maps x to y is $f[x \mapsto y]$; if f was already defined at x, the new value replaces the old one. In a lattice, the join is $\sqcup$ and the meet is $\sqcap$.

7.2 Analyzed language

7.2.1 Syntax and informal semantics

Our analysis applies to the Java bytecode [LY96]. However, for clarity, we will represent the bytecodes by instructions listed in Figure 7.1, which correspond to a small but representative subset of the Java bytecodes. The syntax of these instructions is similar to the Java source code. Corresponding bytecodes are mentioned in the last column of the table. More information on the Java language can be found in [GJS96] for example, or at the JavaSoft home page (`java.sun.com`).

A class declaration of the form `class` C `extends` C' $\{...\}$ means that C ***inherits*** from C': all (non-static) fields and methods declared in C' also exist in C. The part `extends` C' is present in all classes except `Object`, since the `Object` class does not inherit from any other class. All other classes must inherit from `Object`, or from some other class. A class C is said to be a ***subclass*** of C' if C inherits from C_1 which inherits from C_2 ... which inherits from C'. A class C' is a superclass of C if C is a subclass of C'. We define a ***subtyping*** relation $\tau \leq \tau'$ by

$$\tau \leq \tau' \text{ if and only if } \begin{cases} \tau = \tau' \\ \text{or } \tau \text{ is a reference type and } \tau' = \texttt{Object} \\ \text{or } \tau = t\texttt{[]},\ \tau' = t'\texttt{[]} \text{ and } t \leq t' \\ \text{or } \tau \text{ and } \tau' \text{ are classes, and } \tau \text{ is a subclass of } \tau'. \end{cases}$$

If $\tau \leq \tau'$, a value of type τ may appear when a value of type τ' is expected.

Address	Address in the method ($[\![0, 65535]\!]$)
$v \in Var$	Name of a local variable
$C \in Class$	Name of a class
Name	Name of a field or a method
SimpleType	Simple type (we restrict ourselves to `int`)
RefType	Reference type (*Class* or array of t, denoted by t`[]` where $t \in Type$)
$t, \tau \in Type$	Type ($RefType \cup SimpleType$)
MethodType	Method type ($Type^* \times Type$)
$m \in Method$	Method ($Class \times Name \times MethodType$)
$f \in Field$	Field ($Class \times Name \times Type$)

Instructions (*Insn*):

$Var = Var$	local variable assignment	`astore`, `aload`, ...
$Var = (RefType)\, Var$	type cast	`checkcast`
$Var = \phi(Var, Var)$	ϕ function	
$Var.Field = Var$	assignment	`putfield`
$Var = Var.Field$	read	`getfield`
$Class.Field = Var$	static field assignment	`putstatic`
$Var = Class.Field$	static field read	`getstatic`
$Var[Var] = Var$	array assignment	`aastore`
$Var = Var[Var]$	array read	`aaload`
$Var =$ `null`	null constant	`aconst_null`
$Var =$ `new` *Class*	object creation	`new`
$Var =$ `new` $Type[Var]$	array creation	`newarray`, `anewarray`
$Var = Var.Method(Var, ..., Var)$	method call	`invokevirtual`
`return` *Var*	return	`areturn`
`if` (...) `goto` *Address*	branch	`ifnull`, ...

A program is a sequence of ***class declarations*** of the form

```
class Class [extends Class] {
  (Type Field;)*
  (static Type Field;)*
  (Type Method(Type Var, ..., Type Var) {(Insn;)*})*
  (static Type Method(Type Var, ..., Type Var) {(Insn;)*})*
}
```

Figure 7.1: Analyzed language syntax

A field declaration $t\ f$ or `static` $t\ f$ in class C means that field (C, f, t) exists. Fields declared `static` belong to the class C, whereas other fields belong to objects: they are contained in each object of class C or a subclass of C.

Each method declaration $t\ m(t_1\ v_1, ..., t_j\ v_j)\ \{i_1; ... i_n;\}$ means that method $(C, m, ((t_1, ..., t_j), t))$ exists, the names of its formal parameters are $v_1, ..., v_j$ and its code is $i_1; ... i_n$. We assume that all methods yield a result. The type t is the type of the result of m, $t_1, ..., t_j$ are the types of its parameters. The method m has a parameter, the receiver `this` of type C, that is not explicitly mentioned in the method declaration. The method m is said to be a virtual method. The static method declaration `static` $t\ m(t_1\ v_1, ..., t_j\ v_j)\ \{i_1; ... i_n;\}$ is similar except that static methods do not have a `this` parameter. Static method calls are not considered in our theoretical study, but they can be easily analyzed: the analysis is similar to the analysis of virtual calls, except that the called method is statically known. The only static method used in the theoretical study is `main`, which is called by the Java virtual machine at startup. Other static methods will be used in some examples.

Each object contains a pointer to its class, and a value for each non-static field of the class and its superclasses. Objects are represented in the implementation by their address. A class contains the information of its declaration, and a value for each static field of the class.

A variable can contain the address of an object, the special address `null` or an integer. The instruction $v_1 = v_2$ stores the contents of v_2 into v_1. The instruction $v_1 = (t)v_2$ has the same effect, except that it only applies when v_2 contains the address of an object. If the type of the object contained in v_2 is not a subtype of t, an exception is raised. (We do not consider this case in our theoretical study; we ignore exceptions for simplicity.)

We assume that the analyzed program is in Static Single Assignment (SSA) form[CFR+91]. In this form, there is a single assignment statement for each local variable of the methods. When the control-flow merges, the values of two different variables may have to be merged into a single one by a ϕ-function. The meaning of $v_1 = \phi(v_2, v_3)$ is equivalent to $v_1 = v_2$ if v_2 has been assigned after v_3 in the execution trace or v_3 has never been assigned. Otherwise, $v_1 = \phi(v_2, v_3)$ is equivalent to $v_1 = v_3$.

The field assignment $v_1.f = v_2$ stores the contents of v_2 in the f field of the object contained in v_1. If $f = (C, f', t)$, the class of the object in v_1 must be a subclass of C, and the type of the contents of v_2 must be a subtype of t. This is statically guaranteed by the bytecode verifier. The field read $v_1 = v_2.f$ stores in v_1 the contents of the f field of the object contained in v_2. The static field assignment $C.f = v_1$ stores the contents of v_1 in the static field f of class C. Similarly, $v_1 = C.f$ stores in v_1 the contents of the static field f of class C. (There are type constraints checked by the bytecode verifier for all these instructions.)

The array assignment $v_1[v_2] = v_3$ stores the contents of v_3 in the element of index v_2 of the array v_1. The variable v_2 must contain an integer. The index of the first element of an array is 0. If v_2 does not correpond to a valid index in the array ($v_2 < 0$ or v_2 greater than or equal to the size of the array), an exception is raised. An exception is also raised if the contents of v_1 is of type t`[]`, and the contents of v_3 is not of a subtype of t. Again, exceptions are ignored in our theoretical study. The array read $v_1 = v_2[v_3]$ stores in v_1 the contents of the element of index v_3 of the array v_2.

The instruction $v =$ `null` stores in v the null address. The new instruction $v =$ `new` C creates a new object of class C, and stores its address in v. Similarly, $v =$ `new` $t[v']$ creates a new array whose elements are of type t, whose size is the contents of v'. The address of the new array is stored in v.

The instruction $v = v_0.m(v_1, ..., v_j)$ calls a method. Assume that v_0 contains an object of class C, and that $m = (C', m', t')$. Then C must be a subclass of C' (this is guaranteed by the bytecode verifier), and the method (C', m', t') must be declared in class C'. $v = v_0.m(v_1, ..., v_j)$ looks for a method of name m' and type t' in the superclasses of C, starting with C itself. It calls the first method (C'', m', t') found with arguments $v_0, ..., v_j$ (v_0 is the receiver `this`). The result of the method is stored in v when returning from the call. The return instruction `return` v terminates the execution of the current method, and returns as its result the contents of v.

The branch instruction `if` (...) `goto` a first executes a test (a comparison between two integer variables, an equality test between to objects or arrays, ...) represented by the (...). If the test succeeds, it jumps to the address a, in the current method, and continues execution there. Otherwise, it executes the instruction following the branch.

The execution of a program starts by running the method `main`. This method takes as parameter an array of strings, containing the parameters of the command line. (Class initializers are not considered in the theoretical study, but they are of course handled in the implementation.)

For simplicity, we do not consider `jsr`, `ret`, and exceptions here, since they raise specific problems, with no connection with escape analysis. Our implementation correctly handles all these features. When there is no ambiguity, fields and methods will simply be designated by their name, without indicating their class and type.

7.2.2 Analysis hierarchy

To perform stack allocation or synchronization elimination, we need some kind of alias information. For synchronization elimination, we need to know whether an object is reachable from several threads, i.e. is it aliased to objects stored in static fields or in thread objects ? For stack allocation, we have to know whether an object is reachable after the end of the method in which it is allocated. That is, is it reachable from the parameters or result of the method, or from static fields ? This is also a kind of alias information. It is therefore natural to use an alias analysis for these purposes. However, we are in a particular case: the question is "is object o aliased with a *given* set of objects ?". We can use this fact to avoid performing a full and costly alias analysis: alias analysis will only be an intermediate step in the design of our escape analysis.

Moreover, using aliases will be very useful in the proof of correctness of our escape analysis. Indeed, for the Java language, we have to analyze assignments precisely, since they are very often used. We shall see that this goal is achieved by using two interdependent analyses: one forward (the escape information propagates from the parameters to the result of methods) and one backward (the escape information propagates from the result to the parameters). Even if the analyzed program contains no loop, this leads to iterations between these two analyses. The correctness of these iterations can be proved by mapping them to the iterations in the computation of the transitive closure of the alias relation. Using aliases is therefore a key point of the proof.

We start therefore from an alias analysis, then we define a very precise escape analysis $\mathbb{E}$ that represents escaping parts of objects by sets of access paths. It is still too complex to be implemented directly, so it is abstracted to give analyses $\mathbb{L}$ and $\mathbb{L}_1$ which use integers to represent escaping parts of objects (Figure 7.2). Analysis $\mathbb{L}_1$ is slightly more approximate than $\mathbb{L}$, to have a faster implementation.

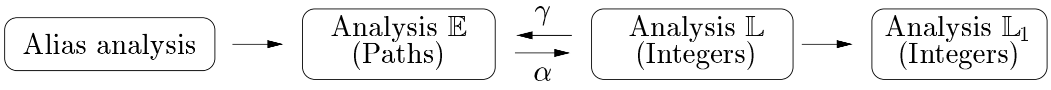


Figure 7.2: Analysis hierarchy

```
Vector {
  int count;
  Object[] elementData;
}
```

Let `v` be a `Vector` such that

```
static String s = "Yes"
v.count = 2
v.elementData = new Object[] { s, new Integer(3) }
```

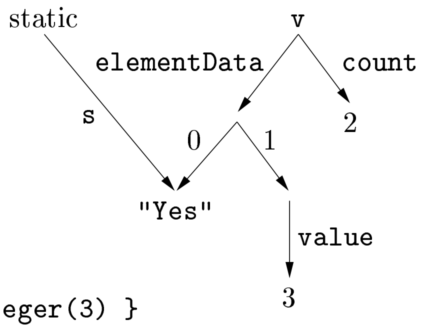


Figure 7.3: Data structure for examples of access paths

7.3 Alias Analysis

The correctness proof of our escape analysis could be performed using a wide variety of alias analyses. We have chosen a relatively simple alias analysis which does not remove aliases when objects are overwritten.

In our alias analysis, objects are represented by ***object names*** $N \in \mathit{OName}$, which are similar to locations, except that different names may represent the same location. An alias between two object names means precisely that they may represent the same location. Parts of objects are represented by ***access paths***:

$$\mathit{Path} = (\mathit{Field} \cup \mathbb{N})^*$$

where $(C, f, t) \in \mathit{Field}$ means that we access the (C, f, t) field of the considered object, and $n \in \mathbb{N}$ means that we access the nth element of the considered array. Let $\pi = N.p \in \mathit{OName} \times \mathit{Path}$ denote the object accessed from object of name N by path p. Let static.$f.p$ denote the object accessed from the static field f by path p. If π denotes object o, then $\pi.(C, f, t)$ where $(C, f, t) \in \mathit{Field}$ denotes the field (C, f, t) of the object o. If π denotes an array o, $\pi.n$ where $n \in \mathbb{N}$ denotes the nth element of o. The empty path is ϵ. An alias is represented by a pair (π_1, π_2) which is written $\pi_1 \sim \pi_2$ for readability. It means that π_1 and π_2 may represent the same object.

Example 7.1 Consider for example the structure represented in Figure 7.3. Then, if N is the object name representing the vector `v`, N.(`Vector`, `count`, `int`) (or simply N.`count`) represents the integer 2, N.`elementData` represents the array of objects, N.`elementData`.1.`value` represents the integer 3. static.`s` and N.`elementData`.0 both represent the string `"Yes"`, hence there is an alias static.`s` $\sim$ N.`elementData`.0.

Alias analysis is defined in Figure 7.4. Let $\mathit{NewName} \in \mathit{OName}$ be a new object name. Abstract values can either be object names $N \in \mathit{OName}$ for objects and arrays, or $\emptyset$ for variables of simple types such as integers.

$d \in Date = \mathbb{N}$	Date
$pc \in PC = Method \times Address$	Program counter
$\mathrm{P}(pc)$	Instruction at pc
$p \in Path = (Field \cup \mathbb{N})^*$	Access paths
$N \in OName$ (any countable set)	Object names
$L \in LocVar^\# = Var \to (OName \cup \{\emptyset\})$	Abstract local variables
$J \in JavaStack^\# = (LocVar^\#)^*$	Abstract java stack
$(L, J) \in State^\# = LocVar^\# \times JavaStack^\#$	Abstract state
$T^\# \in Trace^\# = Date \to State^\#$	Abstract trace
$Alias = \mathcal{P}(((OName \cup \mathrm{static}) \times Path)^2)$	Alias relation
$R \in DateAlias = \mathcal{P}(((OName \cup \mathrm{static}) \times Path)^2 \times Date)$	Dated alias relation
$\mathrm{Val}^\# : State^\# \times Var \to (OName \cup \{\emptyset\})$	Abstract value of a variable
$\mathrm{Val}^\#((L, J), v) = L(v)$	

$\mathrm{P}(\mathrm{pc}(T(d)))$	Abstract state transition $d \Rightarrow d+1$	Alias
entry of `main(args)`	$\Rightarrow(\{\texttt{args} \mapsto P'_0\}, []), P'_0 = NewName$	
$w = v_0.m'(v_1, ..., v_n)$	$(L, J) \Rightarrow (\{p_i \mapsto P'_i\}, L : J), P'_i = NewName$	$L(v_i) \sim P'_i$
The p_i are the names of formal parameters of the called method, $(i \in [\![0, n]\!])$		
`return` v	$(L, L' : J) \Rightarrow (L'[w \mapsto L(v)], J)$	
w is the variable in which we store the result of the corresponding method call.		

$\mathrm{P}(\mathrm{pc}(T(d)))$	Local variables transition $d \Rightarrow d+1$	Alias
$v_1 = v_2 / v_1 = (t)v_2$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2) \sim N$
$v_1 = \phi(v_2, v_3)$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2)$ or $L(v_3) \sim N$[a]
$v_1 = v_2.f$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2).f \sim N$
$v_1.f = v_2$	$L \Rightarrow L$	$L(v_1).f \sim L(v_2)$
$v_1 = C.f$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$N \sim \mathrm{static}.f$
$C.f = v_1$	$L \Rightarrow L$	$L(v_1) \sim \mathrm{static}.f$
$v_1 = v_2[v_3]$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2).\mathbb{N} \sim N$
$v_1[v_2] = v_3$	$L \Rightarrow L$	$L(v_1).\mathbb{N} \sim L(v_3)$
$v =$ `new` C/`new` $t[w]$/`null`	$L \Rightarrow L[v \mapsto NewName]$	
`if` (...) `goto` a		

[a]The created alias is $L(v_2) \sim N$ if v_2 has been defined after v_3 in the execution trace, or v_3 has never been defined. It is $L(v_3) \sim N$ otherwise.

Figure 7.4: Alias analysis. The java stack J is not modified by instructions from the bottom table. The "Alias" column indicates aliases added to relation R. We assume that all fields and arrays contain objects or arrays, and all parameters of methods are objects or arrays (no simple types). Data of simple types have no alias or escape effect. No alias is created for a value of simple type, and the corresponding abstract value is $\emptyset$ instead of an object name.

```
class BinTree {
  BinTree left, right; Object elem;

  BinTree(BinTree l, BinTree r, Object e) {
    left = l; right = r; elem = e;
  }
  static BinTree leaf(Object o) {
    return new BinTree(null, null, o);
  }
  static Object run() {
    BinTree local = new BinTree(null,
      leaf(new Integer(1)), new Integer(2));
    return local.elem;
  }
}
```

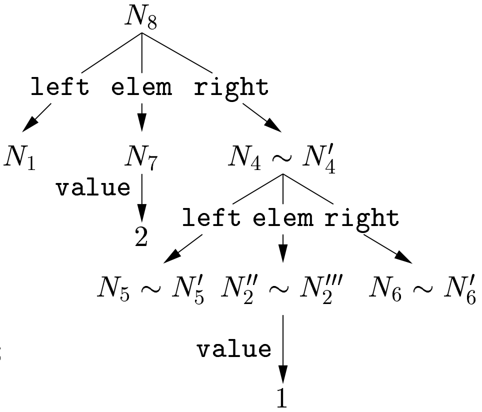


Figure 7.5: Code and data structure built by `run()` for Example 7.2. Some object names have been omitted for simplicity.

We define the ***date*** $d \in Date$ as the number of instructions executed in the analyzed program. An execution trace T of the analyzed program is then a function from dates to states: $T(d)$ is the state at date d. Consider one fixed execution trace T of the program. The program counter at date d in this execution trace is $\mathrm{pc}(T(d))$. We compute two pieces of alias information corresponding to the execution trace T:

1. An ***abstract trace*** $T^\#$ which gives the abstract state for each date d. The abstract state gives an abstract value for each variable. It contains two parts: $L \in LocVar^\#$ yields the abstract value of the local variables of the current method being executed m, whereas $J \in JavaStack^\#$ corresponds to the local variables of the callers of m. Notice that we have the abstract state for each date, and not for each program point. When there is a loop in the program, we simply follow the iterations of the execution trace (we come back to the same pc, but with a different date d, so we do not have fixed point equations here).

2. A ***dated alias relation*** $R \in DateAlias$ in which each alias is registered with its creation date. R summarizes all aliases created during the runtime of the program. The symmetric of an alias and its right-regular closure are always added at the same time as the alias itself, i.e. if the instruction executed at date d adds alias $\pi_1 \sim \pi_2$, R contains $\{(\pi_1.p, \pi_2.p, d+1), (\pi_2.p, \pi_1.p, d+1), p \in Path\}$ (the alias only exists *after* the instruction is executed, i.e. at date $d+1$). R may contain several times the same alias with different creation dates, if an instruction which adds an alias is executed when this alias already existed. The right-regular equivalence relation[1] $R(d)$ gives aliases at or before date d:

$$R(d) = \{\pi_1 \sim \pi_p | \forall i = 1, ..., p-1, (\pi_i \sim \pi_{i+1}, d_i) \in R, d_i \leq d\} \in Alias.$$

These two pieces of information could also be considered as an instrumentation of a small steps semantics of Java.

[1] Reflexive, symmetric and transitive relation, such that $\forall (\pi_1, \pi_2) \in R, \forall p \in Path, (\pi_1.p, \pi_2.p) \in R$.

d	Instruction	Abstract state $T^{\#}(d+1)$
0	entry	$(\{\mathtt{args} \mapsto P_0\}, [])$
1	`v0 = BinTree.run()`	$(\{\}, M)$
2	`  v1 = null`	$(\{\mathtt{v1} \mapsto N_1\}, M)$
3	`  v2 = new Integer`	$(\{\mathtt{v1} \mapsto N_1, \mathtt{v2} \mapsto N_2\}, M)$
4	`  v2.<init>(1)`	$(\{\mathtt{this} \mapsto N_2', \mathtt{x} \mapsto \emptyset\}, J_1)$
5	`    this.value = x`	$(\{\mathtt{this} \mapsto N_2', \mathtt{x} \mapsto \emptyset\}, J_1)$
6	`    return`	$(\{\mathtt{v1} \mapsto N_1, \mathtt{v2} \mapsto N_2\}, M)$
7	`  v3 = leaf(v2)`	$(\{\mathtt{o} \mapsto N_2''\}, J_1)$
8	`    v4 = new BinTree`	$(\{\mathtt{o} \mapsto N_2'', \mathtt{v4} \mapsto N_4\}, J_1)$
9	`    v5 = null`	$(\{\mathtt{o} \mapsto N_2'', \mathtt{v4} \mapsto N_4, \mathtt{v5} \mapsto N_5\}, J_1)$
10	`    v6 = null`	$(\{\mathtt{o} \mapsto N_2'', \mathtt{v4} \mapsto N_4, \mathtt{v5} \mapsto N_5, \mathtt{v6} \mapsto N_6\}, J_1)$
11	`    v4.<init>(v5,v6,o)`	$(\{\mathtt{this} \mapsto N_4', \mathtt{l} \mapsto N_5', \mathtt{r} \mapsto N_6', \mathtt{e} \mapsto N_2'''\}, J_2)$
12	`      this.left = l`	$(\{\mathtt{this} \mapsto N_4', \mathtt{l} \mapsto N_5', \mathtt{r} \mapsto N_6', \mathtt{e} \mapsto N_2'''\}, J_2)$
13	`      this.right = r`	$(\{\mathtt{this} \mapsto N_4', \mathtt{l} \mapsto N_5', \mathtt{r} \mapsto N_6', \mathtt{e} \mapsto N_2'''\}, J_2)$
14	`      this.elem = e`	$(\{\mathtt{this} \mapsto N_4', \mathtt{l} \mapsto N_5', \mathtt{r} \mapsto N_6', \mathtt{e} \mapsto N_2'''\}, J_2)$
15	`      return`	$(\{\mathtt{o} \mapsto N_2'', \mathtt{v4} \mapsto N_4, \mathtt{v5} \mapsto N_5, \mathtt{v6} \mapsto N_6\}, J_1)$

$M = [\{\mathtt{args} \mapsto P_0\}]$, $J_1 = \{\mathtt{v1} \mapsto N_1, \mathtt{v2} \mapsto N_2\} : M$,
$J_2 = \{\mathtt{o} \mapsto N_2'', \mathtt{v4} \mapsto N_4, \mathtt{v5} \mapsto N_5, \mathtt{v6} \mapsto N_6\} : J_1$
$R = \mathrm{clo}(\{(N_2 \sim N_2', 5), (N_2 \sim N_2'', 8), (N_4 \sim N_4', 12), (N_5 \sim N_5', 12), (N_6 \sim N_6', 12), (N_2'' \sim N_2''', 12), (N_4'.\mathtt{left} \sim N_5', 13), (N_4'.\mathtt{right} \sim N_6', 14), (N_4'.\mathtt{elem} \sim N_2''', 15), ...\})$
where $\mathrm{clo}(A) = \{(\pi_1.p \sim \pi_2.p, d), (\pi_2.p \sim \pi_1.p, d) | (\pi_1 \sim \pi_2, d) \in A, p \in Path\}$.

Figure 7.6: Example of alias analysis

On entry of `main`, the command line arguments `args` are stored in an array which is given a new name P_0. The aliases of this name are correctly represented by $R(0) = \{\pi \sim \pi\}$, since there is no sharing between command line arguments. We shall see in the following that the aliases created before the beginning of `main` do not intervene in the definition of escape analysis. So our escape analysis is correct even for other values of $R(0)$, if aliases are created by the initialization of the Java program.

When calling a method, new names for the parameters are created. They are aliased to the object names of the arguments, since they represent the same object. When returning from a method, the name of the returned object is transmitted to the caller.

When transferring objects from a variable to another by $v_1 = v_2$ or $v_1 = (t)v_2$, a new object name N is created, which is aliased to the old name. The analysis of $v_1 = \phi(v_2, v_3)$ is similar, except that we have to choose the right object, either in v_2 or in v_3.

When reading a field f, a new object name N is created for the read object, and it is aliased to the contents of the field f, since they correspond to the same object. When writing into a field by $v_1.f = v_2$, an alias is created between the stored object (in v_2) and the field f of the object in v_1. The process is similar for static fields and for arrays. (Different elements of the same array are not distinguished.)

When creating a new object with `new` C or `new` $t[w]$, a new object name is created corresponding to this object. This object name has no alias (at creation time) since it is the only one which represents the created object.

Example 7.2 Consider the binary tree class of Figure 7.5. The beginning of the abstract trace is represented in Figure 7.6. `<init>` is the name of constructor methods. The code is indented: when executing a method call, the code of the called method is two characters on the right with respect to the code of the caller. To make reading easier, we have chosen to use object names with the same index i: N_i, N'_i, N''_i, ... for object names which correspond to the same location. There will be aliases between these objects. The aliases are given in the dated alias relation R at the bottom of Figure 7.6. For instance, the fact that the `elem` field of the node `v4 = leaf(new Integer(1))` contains `new Integer(1)` is represented by the alias $N'_4.\texttt{elem} \sim N'''_2$, where N'_4 is an object name for the node in `v4` and N'''_2 an object name for the `new Integer(1)`. This alias is added by the instruction `this.elem = e` executed at date 14. It has therefore date 15. The aliases $N_4 \sim N'_4$, $N_5 \sim N'_5$, $N''_2 \sim N'''_2$, $N_6 \sim N'_6$ are created by the call to the `BinTree` constructor in `leaf`, executed at date 11 (they have date 12).

Theorem 7.3 (Stack allocation) *Consider one execution of method m, in execution trace T, and assume that this execution of m terminates at date r.*

Consider a `new` *in method m, allocating object o. Let Res be the object name associated with the result of m, P_l the object names associated with the parameters[2] of m (the names of these objects in the caller of m), N the object name of o. Let $Esc = \{\text{static}.p' | p' \in Path\} \cup \{Res.p' | p' \in Path\} \cup \cup_l \{P_l.p' | p' \in Path\}$.*

If $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, o can be stack allocated in m in the considered execution trace T and in the execution of m that ends at date r.

Intuitively, if $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, at the end of method m, the object o is not aliased to objects reachable from the parameters, the result of m, or from static fields. So it becomes unreachable at the end of m. A proof of this result can be found in Appendix E.1.

Example 7.4 Consider again Example 7.2. At the end of `BinTree.run`, the only aliases with elements of Esc are $N_7.p \sim Res.p$, where p is any path, and others due to aliases of N_7: $\pi' \sim Res.p$ where $\pi' \sim N_7.p$. In particular, since we do not have $N_8.\epsilon \sim N_7.p$, $\forall \pi \in Esc, (N_8.\epsilon \sim \pi) \notin R(r)$, so the `BinTree` node referenced by variable `local` can be stack allocated in `BinTree.run`.

7.4 Analysis $\mathbb{E}$: Escape Analysis using Access Paths

The following analysis is very precise, so it can be the basis for several less precise escape analyses, which can be derived by abstract interpretation. However, it cannot be implemented directly because the fixed point iteration may not terminate. We will therefore define a more approximate analysis in Section 7.5. Analysis $\mathbb{E}$ does not use types, so we do not have to deal with subtyping. Types will be used in Section 7.5.

7.4.1 Abstract lattice for values

An object is said to escape from method m if and only if it is reachable from the parameters or the result of m, or from static fields. If an object escapes from m, it will not be stack allocated in m. Having for each data structure only a binary information, "it may escape" or "it does not escape", would be far too coarse. A part of a data structure may escape, while

[2]The receiver `this` is considered as an ordinary parameter.

another part does not escape. For instance, in Example 7.2, `local.elem` escapes from `run`, but not `local` itself. Therefore, we have to represent parts of data structures, to be able to say "this part escapes". In analysis $\mathbb{E}$, we have chosen to use sets of paths to represent parts of objects. This is natural since access paths were already used in our alias analysis. The escaping part of an object is said to be the escape ***context*** of the object, so escape contexts for this analysis are sets of paths: $Ctx_{\mathbb{E}} = \mathcal{P}(Path)$. They are ordered by inclusion, $Ctx_{\mathbb{E}}$ is then a complete lattice.

Example 7.5 If we consider again the structure of Figure 7.3, the part of `v` corresponding to the array of objects can be represented by paths $\{\texttt{elementData}.p | p \in Path\}$. The part of `v` corresponding to the integer 3 is represented by $\{\texttt{elementData}.1.\texttt{value}\}$.

We have to define operations on these contexts. They are of two kinds:

1. ***constructions*** $f.c$ where $f \in Field$: if c is the context associated with the f field of an object o, $f.c$ gives the context associated with o. Let $f.c = \{f.p | p \in c\}$. Similarly, $\mathbb{N}.c = \{n.p | p \in c, n \in \mathbb{N}\}$ gives the context associated with the array a, where c is the context associated with an element of array a.

2. Conversely, $f^{-1}.c$ are ***restrictions***: if c is the context associated with an object o, $f^{-1}.c$ gives the context of the field $o.f$. Let $f^{-1}.c = \{p | f.p \in c\} \cup \{\epsilon \text{ if } \epsilon \in c\}$, $\mathbb{N}^{-1}.c = \{p | \exists n \in \mathbb{N}, n.p \in c\} \cup \{\epsilon \text{ if } \epsilon \in c\}$.

Example 7.6 Assume that the escaping part of variable `v` is represented by the context $c = \{\texttt{elementData}.1, \texttt{elementData}.1.\texttt{value}\}$, in the structure of Figure 7.3. Then the escaping part of the array of objects will be $\texttt{elementData}^{-1}.c = \{1, 1.\texttt{value}\}$. The escaping part of an element of the array of objects will be $\mathbb{N}^{-1}.\texttt{elementData}^{-1}.c = \{\epsilon, \texttt{value}\}$. (We have made an approximation here, since we do not distinguish any more the different elements of the array.)

If the escaping part of the array of objects is $c = Path$, then the escape context of the vector `v` is at least $\texttt{elementData}.c = \{\texttt{elementData}.p | p \in Path\}$.

To compute the escape information, we use a *bidirectional* propagation: $\mathbb{E}$ is a backward analysis, whereas $\mathbb{E}_S$ is a forward analysis. The analyses $\mathbb{E}$ and $\mathbb{E}_S$ depend on each other. First, what is read to build the result of a method escapes. (For example, if a method reads a field f with instruction $v_1 = v_2.f$, and returns this field as its result, then the field escapes.) This is computed by analysis $\mathbb{E}$, using a backward propagation (from the result to the parameters). The result of the method is first marked as escaping. Then, when a read instruction such as $v_1 = v_2.f$ is analyzed, and v_1 is marked as escaping, the corresponding part of v_2 is also marked as escaping. However, because of assignments, objects may also escape because they are stored in static fields, in parameters, or in the result of the method. Backward propagation cannot take into account the fact that an object o escapes because it is stored in a parameter o' of the method for instance, because at the point of the assignment, the analyzer would not know that o' is a parameter of the method. Therefore, we have to introduce a forward analysis $\mathbb{E}_S$ to cope with assignments (S for *store*). For example, in `BinTree.run()`, $\mathbb{E}$ can take into account that elements of `local` may be part of the result, but $\mathbb{E}_S$ is necessary to take into account the fact that the new `Integer` may be stored in `local`.

Contexts:	$Ctx_{\mathbb{E}} = \mathcal{P}(Path)$ ordered by inclusion
Abstract values:	$Val_{\mathbb{E}} = Ctx_{\mathbb{E}}{}^* \to Ctx_{\mathbb{E}}$
Order on abstract values:	pointwise extension of order on contexts.
Upper bound:	$E_1 \sqcup E_2 = \{(c_{-1}, ..., c_j) \mapsto E_1(c_{-1}, ..., c_j) \cup E_2(c_{-1}, ..., c_j)\}$
Constructions and restrictions:	$\mathbb{N}.E = \{(c_{-1}, ..., c_j) \mapsto \mathbb{N}.E(c_{-1}, ..., c_j)\}$, others similar.
Composition:	usual function composition $\circ$
Greatest element:	$\top_{\mathbb{E}}[f] = \{(c_{-1}, ..., c_j) \mapsto Path\}$[a]
Abstract value for the result:	$first_{\mathbb{E}} = \{(c_{-1}, ..., c_j) \mapsto c_{-1}\}$.
Abstract values for parameters:	$\forall i \leq j, P_{\mathbb{E}}(p_i) = \{(c_{-1}, ..., c_j) \mapsto c_i\}$ where the formal parameters of m are $p_0, ..., p_j$

[a]The parameter f is useless when defining analysis $\mathbb{E}$, but will be useful when defining $\mathbb{L}$. Mentioning this parameter makes it possible to use the same equations for $\mathbb{E}$ and $\mathbb{L}$.

Figure 7.7: Definitions for the abstract lattice for analysis $\mathbb{E}$

This bidirectional propagation is necessary to model precisely assignments. It introduces mutual dependencies between unknowns, even if there is no loop in the analyzed program. Therefore, the usual fixed point iteration which corresponds to iteration over the loops of the analyzed program is not enough. That is why we have started from an alias analysis, because it makes explicit a second form of iteration: the iteration to compute the transitive closure of the alias relation. This iteration can be mapped to the fixed point iteration used to compute the escape analysis. This will be used in the correctness proof (see Example 7.8).

If we want to know whether we can allocate an object o on the stack in method m, the reference scope is method m. Then the parameters and result of m escape. They have context $Path$. But if we want to perform stack allocation in a method m' which calls m, the reference scope is now m'. Then, the parameters and result of m do not always escape from m' (depending on what m' does). Therefore, we have to analyze m in several calling contexts. To avoid reanalyzing the method m for each calling context, our analysis is a function of the calling context, which is represented by the escape contexts of the parameters and of the result of m.

Therefore, abstract values are ***context transformers***, i.e. functions from contexts to contexts: knowing the contexts associated with the parameters (by analysis $\mathbb{E}_S$) $c_0, ..., c_j$ and with the result (by analysis $\mathbb{E}$) c_{-1} of the current method, they yield the context associated with the corresponding concrete value $\phi(c_{-1}, ..., c_j)$. Therefore $Val_{\mathbb{E}} = Ctx_{\mathbb{E}}{}^* \to Ctx_{\mathbb{E}}$. Notice that, in this definition, the receiver `this` is regarded as any other parameter of the methods. The order on abstract values is the pointwise extension of the order on contexts. The definitions are summarized in Figure 7.7.

Comparison with ML This analysis $\mathbb{E}$ is similar to analysis E for ML: both analyses use sets of access paths as contexts. However, the analysis of ML was only a backward analysis, and the context transformers only had one parameter: the escape context for the result.

7.4.2 The analysis

Analysis $\mathbb{E}$ is defined in Figure 7.8. Escape analyses $\mathbb{E}$ and $\mathbb{E}_S$ yield for each method m, for each local variable v of m, the corresponding escape abstract value $\mathbb{E}(m)(v)$ or $\mathbb{E}_S(m)(v)$. The

$i \in \mathit{Ind} = \mathbb{N}$	Parameter indices
$\rho, \rho' \in \mathit{Env}_{\mathbb{E}} = \mathit{Method} \to \mathit{Ind} \to \mathit{Val}_{\mathbb{E}}$	Environment ($\mathbb{E}$ contexts for the parameters)
$\rho_{\mathrm{S}}, \rho'_{\mathrm{S}} \in \mathit{Env}_{\mathbb{E}\mathrm{S}} = \mathit{Method} \to \mathit{Val}_{\mathbb{E}}$	Environment ($\mathbb{E}_{\mathrm{S}}$ context for the result)
$L, L_{\mathrm{S}} \in \mathit{LocVar}_{\mathbb{E}} = \mathit{Var} \to \mathit{Val}_{\mathbb{E}}$	Abstract local variables
$\mathbb{E}, \mathbb{E}_{\mathrm{S}} \in \mathit{Method} \to \mathit{LocVar}_{\mathbb{E}}$	Escape analyses

$\mathrm{P}(pc)$	Forward analysis $L_{\mathrm{S}} = \mathbb{E}_{\mathrm{S}}(m)$	Backward analysis $L = \mathbb{E}(m)$
entry of m	$L_{\mathrm{S}} = P_{\mathbb{E}}$	$\forall i \in [\![0, j]\!], \rho(m)(i) \geq L(p_i)$
$v_1 = v_2 / v_1 = (t)v_2$	$L_{\mathrm{S}}(v_1) = L_{\mathrm{S}}(v_2)$	$L(v_2) \geq L(v_1)$
$v_1 = \phi(v_2, v_3)$	$L_{\mathrm{S}}(v_1) = L_{\mathrm{S}}(v_2) \sqcup L_{\mathrm{S}}(v_3)$	$L(v_2) \geq L(v_1), L(v_3) \geq L(v_1)$
$v_1 = v_2.f$	$L_{\mathrm{S}}(v_1) = f^{-1}.L_{\mathrm{S}}(v_2)$	$L(v_2) \geq f.L(v_1)$
$v_1.f = v_2$		$L(v_2) \geq f^{-1}.L_{\mathrm{S}}(v_1), L(v_1) \geq f.L_{\mathrm{S}}(v_2)$
$v = C.f$	$L_{\mathrm{S}}(v) = \top_{\mathbb{E}}[f]$	
$C.f = v$		$L(v) \geq \top_{\mathbb{E}}[f]$
$v_1 = v_2[v_3]$	$L_{\mathrm{S}}(v_1) = \mathbb{N}^{-1}.L_{\mathrm{S}}(v_2)$	$L(v_2) \geq \mathbb{N}.L(v_1)$
$v_1[v_2] = v_3$		$L(v_3) \geq \mathbb{N}^{-1}.L_{\mathrm{S}}(v_1), L(v_1) \geq \mathbb{N}.L_{\mathrm{S}}(v_3)$
$v =$ `new` $C/$`new` $t[w]$	$L_{\mathrm{S}}(v) = L(v)$	
$v =$ `null`	$L_{\mathrm{S}}(v) = L(v)$	
`if` (...) `goto` a		
$w = v_0.m'(v_1, ..., v_n)$	$L_{\mathrm{S}}(w) = \rho'_{\mathrm{S}}(m') \circ (L(w), L_{\mathrm{S}}(v_0), ..., L_{\mathrm{S}}(v_n))$	$L(v_i) \geq \rho'(m')(i) \circ (L(w), L_{\mathrm{S}}(v_0), ..., L_{\mathrm{S}}(v_n)), i \in [\![0, n]\!]$
`return` v	$\rho_{\mathrm{S}}(m) \geq L_{\mathrm{S}}(v)$	$L(v) \geq \mathit{first}_{\mathbb{E}}$

Figure 7.8: Escape analysis. m is the current method, which has $j + 1$ parameters. In the above table, we assume that all fields and arrays contain objects or arrays. (Data of simple types have no escape effect, therefore no equation is generated for field or array accesses or assignments when they contain data of simple types.)

environment $\rho \in \mathit{Env}_{\mathbb{E}}$ yields for each method the escape information for its parameters (for analysis $\mathbb{E}$). Each parameter is represented by a parameter index $i \in \mathit{Ind}$. The environment $\rho_{\mathrm{S}} \in \mathit{Env}_{\mathbb{E}\mathrm{S}}$ yields for each method the escape information for its result (for analysis $\mathbb{E}_{\mathrm{S}}$).

On entry of method m, the abstract values of the parameters for the forward analysis $\mathbb{E}_{\mathrm{S}}$ are initialized. Since c_i represents the escape context of parameter p_i, it is natural to have $\mathbb{E}_{\mathrm{S}}(m)(p_i)(c_{-1}, ..., c_j) = c_i$, therefore $L_{\mathrm{S}}(p_i) = \{(c_{-1}, ..., c_j) \mapsto c_i\} = P_{\mathbb{E}}(p_i)$. For analysis $\mathbb{E}$, the computation of the backward escape contexts of the parameters is complete when the beginning of the method is reached, and the escape information for the parameters of method m can be updated in environment ρ.

For the local variable assignments $v_1 = v_2$ and $v_1 = (t)v_2$, the escape information is transmitted from v_2 to v_1 for analysis $\mathbb{E}_{\mathrm{S}}$ and from v_1 to v_2 for $\mathbb{E}$.

For $v_1 = \phi(v_2, v_3)$, we have to assume the worst case, that is the value in v_1 may be reachable as soon as the values in v_2 or v_3 are reachable, hence the upper bound for analysis $\mathbb{E}_{\mathrm{S}}$. For analysis $\mathbb{E}$, the values in v_2 and v_3 may be reachable as soon as the value in v_1 is reachable.

When reading a field by $v_1 = v_2.f$, the contents of v_1 is reachable as soon as the field f of v_2 is reachable, hence the escape information for v_1 is computed by restricting the information for v_2 to the field f, in analysis $\mathbb{E}_{\mathrm{S}}$. Conversely for analysis $\mathbb{E}$, if a part of v_1 escapes, the same part of the field f of v_2 escapes.

When writing into a field by $v_1.f = v_2$, intuitively, the part of the field f of v_1 that escapes is the part of v_2 that escapes: if the field f of v_1 escapes, v_2 may escape, and if v_2 escapes, the field f of v_1 may escape. The two equations update the escape information of v_2 and v_1 to take this fact into account.

The values reachable from static fields are considered to have an unbounded lifetime. Hence, when reading or writing static fields, the read or stored values escape entirely. The handling of reads and writes to arrays is similar to the one of objects (we consider that any element of the array may be accessed).

For the virtual method call $w = v_0.m'(v_1, ..., v_n)$ where $m' = (C, m'', t)$, we consider that all methods that have a correct signature (m'', t) and are defined in a subclass of C may be called. This corresponds to class hierarchy analysis [DGC95]. Therefore, the escape information used for virtual calls is defined by the following environments: $\rho'(C, m'', t)(i) = \sqcup_{C' \text{ subclass of } C}\rho(C', m'', t)(i)$ and $\rho'_{\mathrm{S}}(C, m'', t) = \sqcup_{C' \text{ subclass of } C}\rho_{\mathrm{S}}(C', m'', t)$. The escape contexts to be passed to these environments are the escaping parts of the result w for analysis $\mathbb{E}$ and of the parameters $v_0, ..., v_n$ for analysis $\mathbb{E}_{\mathrm{S}}$, hence the composition with $(L(w), L_{\mathrm{S}}(v_0), ..., L_{\mathrm{S}}(v_n))$.

On the return from method m, the escape information for the result of m is updated in environment ρ_{S}. For analysis $\mathbb{E}$, the escape context of the result is c_{-1}, therefore the abstract value is $\mathit{first}_{\mathbb{E}} = \{(c_{-1}, ..., c_j) \mapsto c_{-1}\}$.

Analysis $\mathbb{E}_{\mathrm{S}}$ is forward since the value of $\mathbb{E}_{\mathrm{S}}$ for each use of a variable v is computed from the value of $\mathbb{E}_{\mathrm{S}}(m)(v)$ at the definition of v. $\mathbb{E}_{\mathrm{S}}$ propagates from the definition to uses. On the contrary, $\mathbb{E}$ is backward since the value of $\mathbb{E}(m)(v)$ is in essence the maximum of the values of $\mathbb{E}$ at the uses of v. $\mathbb{E}$ propagates from the uses to the definition of v.

Example 7.7 Let m be the `BinTree` constructor from Example 7.2. Let $L_{\mathrm{S}} = \mathbb{E}_{\mathrm{S}}(m)$, $L = \mathbb{E}(m)$.

Analysis $\mathbb{E}_{\mathrm{S}}$	`BinTree(BinTree l,`	Analysis $\mathbb{E}$
$L_{\mathrm{S}}(p_i) = P_{\mathbb{E}}(p_i)$	`BinTree r, Object e) {`	
	`this.left = l;`	$L(\texttt{l}) \geq \texttt{left}^{-1}.L_{\mathrm{S}}(\texttt{this})$, $L(\texttt{this}) \geq \texttt{left}.L_{\mathrm{S}}(\texttt{l})$
	`this.right = r;`	$L(\texttt{r}) \geq \texttt{right}^{-1}.L_{\mathrm{S}}(\texttt{this})$, $L(\texttt{this}) \geq \texttt{right}.L_{\mathrm{S}}(\texttt{r})$
	`this.elem = e;`	$L(\texttt{e}) \geq \texttt{elem}^{-1}.L_{\mathrm{S}}(\texttt{this})$, $L(\texttt{this}) \geq \texttt{elem}.L_{\mathrm{S}}(\texttt{e})$
	`}`	

The middle column is the analyzed instruction. Abstract values for this method have four parameters c_0, c_1, c_2, c_3 which are respectively the $\mathbb{E}_{\mathrm{S}}$ escape contexts of `this`, `l`, `r` and `e` (there is no c_{-1} here since the method has no result). So the abstract values are of the form $\{(c_0, c_1, c_2, c_3) \mapsto \phi(c_0, ..., c_3)\}$. When analyzing another method m' that calls m, we need to know what happens if only a part of the parameters of m escapes from the caller m', that is why c_0, ..., c_3 are parameters and not constants.

The left column corresponds to the forward pass (analysis $\mathbb{E}_{\mathrm{S}}$). We say that an object o store-escapes when, if we store an object o' in o, o' escapes. Intuitively, analysis $\mathbb{E}_{\mathrm{S}}$ determines a superset of the set of objects which store-escape. This intuition yields the equation $L(v_2) \geq f^{-1}.L_{\mathrm{S}}(v_1)$ if $v_1.f = v_2$. However, we shall see in Example 7.16 that store-escaping is in fact very similar to simply escaping. In both cases, the object has a lifetime greater than the same limit. At the beginning of the method, the local variables are initialized with the parameters, so the corresponding escape information is $P_{\mathbb{E}}$, with $\mathbb{E}_{\mathrm{S}}(m)(\mathtt{this}) = P_{\mathbb{E}}(\mathtt{this}) = \{(c_0, c_1, c_2, c_3) \mapsto c_0\}$, which means that if the calling context is such that the implicit parameter of m store-escapes, then `this` store-escapes (they have the same escape context c_0). In the same way, $P_{\mathbb{E}}(\mathtt{l}) = \{(c_0, c_1, c_2, c_3) \mapsto c_1\}$ and so on.

The right column corresponds to the backward pass (analysis $\mathbb{E}$). The first inequation $L(\mathtt{l}) \geq \mathtt{left}^{-1}.L_{\mathrm{S}}(\mathtt{this})$ means that if the `left` field of `this` store-escapes, then `l` escapes. We obtain: $\mathbb{E}(m)(\mathtt{l}) = \{(c_0, c_1, c_2, c_3) \mapsto \mathtt{left}^{-1}.c_0\}$. The second inequation $L(\mathtt{this}) \geq \mathtt{left}.L_{\mathrm{S}}(\mathtt{l})$ means that if `l` store-escapes then the `left` field of `this` escapes. Therefore $\mathbb{E}(m)(\mathtt{this}) \geq \{(c_0, c_1, c_2, c_3) \mapsto \mathtt{left}.c_1\}$. We have similar inequations for other instructions.

The analysis yields:

$$\mathbb{E}(m)(\mathtt{this})(c_0, c_1, c_2, c_3) = \mathtt{left}.c_1 \cup \mathtt{right}.c_2 \cup \mathtt{elem}.c_3$$
$$\mathbb{E}(m)(\mathtt{l})(c_0, c_1, c_2, c_3) = \mathtt{left}^{-1}.c_0$$

The context transformers $\mathbb{E}(m)(\mathtt{r})$ and $\mathbb{E}(m)(\mathtt{e})$ are similar to $\mathbb{E}(m)(\mathtt{l})$. The context transformer $\mathbb{E}(m)(\mathtt{this})$ shows that the fields of `this` may escape if the parameters `l`, `r` and `e` escape. (Let us recall that c_1, c_2, c_3 are the $\mathbb{E}_{\mathrm{S}}$ escape contexts of `l`, `r`, `e` respectively.) Conversely, the values of $\mathbb{E}(m)(\mathtt{l})$, $\mathbb{E}(m)(\mathtt{r})$ and $\mathbb{E}(m)(\mathtt{e})$ show that the parameters `l`, `r` and `e` may escape if the corresponding field of `this` escapes. Analysis $\mathbb{E}$ is therefore able to express relations between escaping parts of different parameters, so it is very precise.

Example 7.8 Consider the following code (where $f = (\mathtt{List}, \mathtt{next}, \mathtt{List})$ is the `next` field, $\mathbb{E}(m)(\mathtt{a}) = E$, $\mathbb{E}_{\mathrm{S}}(m)(\mathtt{a}) = E_{\mathrm{S}}$):

```
class List {
  List next; Object elem;

  static List m() {
    a = new List();
    a.next = a;
    return a;
  }
}
```

Analysis $\mathbb{E}_{\mathrm{S}}$
$E_{\mathrm{S}} = E$

Analysis $\mathbb{E}$
$E \geq f^{-1}.E_{\mathrm{S}}, E \geq f.E_{\mathrm{S}}$
$E \geq \mathit{first}_{\mathbb{E}}$

The left column of the table is the forward analysis $\mathbb{E}_{\mathrm{S}}$ of method m. The right column is the backward analysis $\mathbb{E}$ of m. The equations can be simplified into:

$$E_{\mathrm{S}} \geq f^{-1}.E_{\mathrm{S}} \sqcup f.E_{\mathrm{S}} \sqcup \mathit{first}_{\mathbb{E}}.$$

The solution is $E_{\mathrm{S}} = E = \{c \mapsto f^*.(f^*)^{-1}.c\}$. This can only be found by iterating (an infinite number of times) the `a = new List(); a.next = a` part of the program, which contains no loop. So this iteration cannot be mapped to an iteration in the operational semantics, but corresponds to the iteration over the alias $N \sim N.f$ created by the instruction `a.next = a` (N is the object name of `a = new List()`). That is why we have introduced aliases in the preceding step.

7.4.3 Correctness proof

Definition 7.9 (Partial closure) Let $R \in \mathit{DateAlias}$, r, d, s dates, with $s \leq d \leq r$ (in the following, s will be the date of a call to method m, r at the end of the execution of m, and d during the execution of m), $k \in \mathbb{N} \cup \{\infty\}$.

$$\begin{aligned} R^k(r/d/s) = \{\pi_1 \sim \pi_p | \forall i = 1, ..., p-1, (\pi_i \sim \pi_{i+1}, d_i) \in R,\\ s < d_i \leq r, d_1 > d, p \leq k, \text{the } \pi_i \text{ are pairwise distinct}\} \end{aligned}$$

This relation only considers aliases created during the execution of m, the first alias being created after d. It uses at most $k-1$ aliases transitively. Limiting to $k-1$ aliases will be useful to make explicit the iteration used to compute the transitive closure of the alias relation. $R^k(r/d/s)$ is right-regular, but not transitive nor symmetric. The condition "π_i pairwise distinct" is necessary so that the condition $d_1 > d$ is really meaningful: otherwise, just take any alias $\pi_1 \sim \pi_2$ created after d and before r, and consider the chain $\pi_1 \sim \pi_2 \sim \pi_1 \sim ... \sim \pi$ then $(\pi_1 \sim \pi) \in R^{k+2}(r/d/s)$ for all $(\pi_1 \sim \pi) \in R^k(r/s/s)$. ($R^0(r/d/s) = \emptyset$, $R^1(r/d/s) = \{\pi \sim \pi\}$, $R^\infty(r/d/s) = \{\pi_1 \sim \pi_p | \forall i = 1, ..., p-1, (\pi_i \sim \pi_{i+1}, d_i) \in R, s < d_i \leq r, d_1 > d\}$).

Definition 7.10 (Correctness) Let m be the method executed at date d in execution trace T. We consider the execution of m that takes place at date d in trace T. Let Res be the object name associated with the result of this execution of m, P_l the object names associated with the parameters of m (the names of these objects in the caller of m, not their name in m), p_l the names of the formal parameters of m, r the date of the end of the execution of m, and s the date of the call to method m (s is just before the beginning of the execution of m; we have $s < d \leq r$). Assume that m has $j+1$ parameters.

$$\begin{aligned}
&\mathit{Esc}(c, c_0, ..., c_j) = \{\text{static}.p' | p' \in \mathit{Path}\} \cup \{\mathit{Res}.p' | p' \in c\} \cup \cup_l \{P_l.p' | p' \in c_l\}\\
&\text{escap}^k(r/d/s)(N)(c, c_0, ..., c_j) = \{p | \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N.p \sim \pi) \in R^k(r/d/s)\}\\
&\mathbb{E}^k(d)(v) = \text{escap}^k(r/d/s)(\text{Val}^\#(T^\#(d), v))\\
&\mathbb{E}^k_{\mathbb{S}}(d)(v) = \text{escap}^k(r/s/s)(\text{Val}^\#(T^\#(d), v))\\
&\text{corr}_{\mathbb{E}}(k, d) \Leftrightarrow \forall c, c_0, ..., c_j \in \mathit{Ctx}_{\mathbb{E}}, \text{ such that } \forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j),\\
&\quad \forall v \in \mathit{Var} \text{ such that } \text{Val}^\#(T^\#(d), v) \in \mathit{OName},\\
&\quad \mathbb{E}^k(d)(v)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)\\
&\text{corr}_{\mathbb{S}}(k, d) \Leftrightarrow \forall c, c_0, ..., c_j \in \mathit{Ctx}_{\mathbb{E}}, \text{ such that } \forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j),\\
&\quad \forall v \in \mathit{Var} \text{ such that } \text{Val}^\#(T^\#(d), v) \in \mathit{OName},\\
&\quad \mathbb{E}^k_{\mathbb{S}}(d)(v)(c, c_0, ..., c_j) \subseteq \mathbb{E}_{\mathbb{S}}(m)(v)(c, c_0, ..., c_j)
\end{aligned}$$

Intuitively, $\text{escap}^k(r/d/s)(N)$ corresponds to the paths representing escaping parts of the object of name N, when only aliases contained in the relation $R^k(r/d/s)$ are taken into account. The only considered alias chains are alias chains of length at most k, created during the execution of m, and whose first element dates from after d. The escaping parts are represented by the paths aliased to paths contained in $\mathit{Esc}(c, c_0, ..., c_j)$ (static fields or parts of the parameters or of the result that are considered as escaping).

Notice that if method m does not terminate, r is not defined. But objects stack allocated in m will never be deallocated and therefore any object can be stack allocated in m. The escape analysis can then yield any result; it will always be correct. Proving the correctness of escape analysis for synchronization elimination in the case of non-terminating programs is a bit more difficult. It will be detailed in the synchronization elimination proof (Appendix E.4).

The functions $\mathbb{E}^k(d)(v)$ and $\mathbb{E}^k_{\mathrm{S}}(d)(v)$ give the escape information for variables in the same way as $\mathrm{escap}^k(r/d/s)(N)$ gives the escape information for object names. If variable v contains an object of name N at date d, $\mathbb{E}^k(d)(v) = \mathrm{escap}^k(r/d/s)(N)$ and $\mathbb{E}^k_{\mathrm{S}}(d)(v) = \mathrm{escap}^k(r/s/s)(N)$. $\mathbb{E}^k(d)(v)$ corresponds to the backward analysis $\mathbb{E}$, and considers only aliases whose first element of the chain is created after date d. $\mathbb{E}^k_{\mathrm{S}}(d)(v)$ corresponds to the forward analysis $\mathbb{E}_{\mathrm{S}}$ and all aliases created during the execution of method m are considered. Also notice that s and r can be computed from d: they are the dates at the call and return of the method executed at date d in execution trace T.

The escape functions $\mathrm{escap}^k(r/d/s)$, $\mathbb{E}^k(d)$, $\mathbb{E}^k_{\mathrm{S}}(d)$ are in fact abstractions of $(T^{\#}, R)$ or even of the execution trace T.

A simplified version of $\mathrm{corr}_{\mathbb{E}}(k, d)$ would be $\forall v \in \mathit{Var}, \mathbb{E}^k(d)(v) \leq \mathbb{E}(m)(v)$, that is all objects that escape at date d according to $\mathbb{E}^k(d)$ (i.e. we consider only aliases of $R^k(r/d/s)$) are correctly taken into account by analysis $\mathbb{E}$. However, we have to check that the considered variables really contain objects (hence the condition $\mathrm{Val}^{\#}(T^{\#}(d), v) \in \mathit{OName}$). The condition $\forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j)$ will be necessary to prove that the escaping parts c_l for analysis $\mathbb{E}_{\mathrm{S}}$ at the entry point of method m are correct. The intuition for $\mathrm{corr}_{\mathrm{S}}$ is similar, but this time, we check that aliases in $R^k(r/s/s)$ are correctly taken into account by $\mathbb{E}_{\mathrm{S}}$.

Analyses $\mathbb{E}$ and $\mathbb{E}_{\mathrm{S}}$ are correct with respect to $R \in \mathit{DateAlias}$ if and only if for all d, $\mathbb{E}_{\mathrm{S}}$ and $\mathbb{E}$ are correct at date d when composing an unbounded number of aliases transitively: $\mathrm{corr}_{\mathbb{E}\mathrm{S}} \Leftrightarrow \forall d, \mathrm{corr}_{\mathrm{S}}(\infty, d)$ and $\forall d, \mathrm{corr}_{\mathbb{E}}(\infty, d)$.

Theorem 7.11 (Additivity) *The functions* $\phi = \mathrm{escap}^k(r/d/s)(N)$, $\mathbb{E}^k(d)(v)$ *and* $\mathbb{E}^k_{\mathrm{S}}(d)(v)$ *are additive, i.e.* $\forall c, c_0, ..., c_j, \forall c', c'_0, ..., c'_j \in \mathit{Ctx}_{\mathbb{E}}, \phi(c, c_0, ..., c_j) \sqcup \phi(c', c'_0, ..., c'_j) = \phi(c \sqcup c', c_0 \sqcup c'_0, ..., c_j \sqcup c'_j)$.

Proof: This comes immediately from the additivity of *Esc*. □

The escape analyses $\mathbb{E}(m)(v)$ and $\mathbb{E}_{\mathrm{S}}(m)(v)$ and their abstractions which will be defined later are also additive. This will simplify their general form, making the implementation much easier. This theorem shows that the additivity of $\mathbb{E}$ and $\mathbb{E}_{\mathrm{S}}$ does not come from approximations, but is an intrinsic property of the escape analysis as defined above.

Lemma 7.12 *Let* m *be a method,* $v \in \mathit{Var}$ *a local variable of* m *whose definition site is reachable (i.e. may be executed). Let* $c, c_0, ..., c_j \in \mathit{Ctx}_{\mathbb{E}}$ *such that*

$$\forall i \in [\![0, j]\!], c_i \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j) \tag{7.1}$$

where $p_0, ..., p_j$ *are the parameters of* m. *Then*

$$\mathbb{E}_{\mathrm{S}}(m)(v)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j). \tag{7.2}$$

This result is proved in Appendix E.2. The condition (7.1) $\forall i \in [\![0, j]\!], c_i \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ is necessary to have the above property (7.2). Indeed, $\mathbb{E}_{\mathrm{S}}(m)(p_i)(c, c_0, ..., c_j) = c_i \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ implies (7.1).

The condition (7.1) will also always be satisfied. If m is the current scope, $c = c_0 = ... = c_j = Path$, and the condition is satisfied. If the current scope is a caller m' of m: $w = v_0.m(v_1, ..., v_j)$, $c_i = \mathbb{E}_{\mathrm{S}}(m')(v_i)(Path, ..., Path) \supseteq \mathbb{E}(m')(v_i)(Path, ..., Path) \supseteq \rho(m)(i)(c, c_0, ..., c_j) = \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ (using the above lemma to prove $\mathbb{E}_{\mathrm{S}}(m')(v_i)(Path, ..., Path) \supseteq \mathbb{E}(m')(v_i)(Path, ..., Path)$). Otherwise, the current scope is a caller of a caller of m, and so on.

Slightly abusing notations, we could therefore write: $\forall m, \forall v \in Var, \mathbb{E}_{\mathrm{S}}(m)(v) \geq \mathbb{E}(m)(v)$.

Theorem 7.13 (Correctness) *$\mathbb{E}$ and $\mathbb{E}_{\mathrm{S}}$ are correct:* $\mathrm{corr}_{\mathbb{E}\mathrm{S}}$.

Proof: We only give a proof sketch here. A proof can be found in Appendix E.2. We show

1. $\forall d, \mathrm{corr}_{\mathrm{S}}(0, d), \mathrm{corr}_{\mathbb{E}}(0, d)$. Obvious, since $R^0(r/d/s) = \emptyset$.

2. $(\forall d, \mathrm{corr}_{\mathrm{S}}(k, d) \wedge \mathrm{corr}_{\mathbb{E}}(k, d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathbb{E}}(k+1, d))$. By backward induction on d.

3. $(\forall d, \mathrm{corr}_{\mathbb{E}}(k, d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathrm{S}}(k, d))$. By forward induction on d.

4. $\forall k, \mathrm{corr}_{\mathbb{E}}(k, d) \Rightarrow \mathrm{corr}_{\mathbb{E}}(\infty, d)$, $\forall k, \mathrm{corr}_{\mathrm{S}}(k, d) \Rightarrow \mathrm{corr}_{\mathrm{S}}(\infty, d)$. This is obvious since $\cup_{k \in \mathbb{N}} R^k(r/d/s) = R^\infty(r/d/s)$.

The result follows by induction on k. We find here two iterations: iteration over the number k of aliases transitively composed and over the date d. □

Theorem 7.14 (Stack allocation) *Consider $v = \texttt{new}\ C/\texttt{new}\ t[w]$ in method m, allocating object o. Then, if $\epsilon \notin \mathbb{E}(m)(v)(Path, Path, ..., Path)$, o can be stack allocated.*

Proof: Consider one execution of method m in trace T. Assume that this execution of m returns at date r. Let N be the name associated with object o in this execution of m. We show that $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$ thanks to the correctness of $\mathbb{E}$ with $c = c_0 = ... = c_j = Path$: $\{p | \exists \pi \in Esc, (N.p \sim \pi) \in R^\infty(r/d/s)\} \subseteq \mathbb{E}(m)(v)(Path, Path, ..., Path)$. Noticing that N is a new name at date d, the aliases of N are the same in $R^\infty(r/d/s)$ and in $R(r)$. Therefore $\epsilon \notin \{p | \exists \pi \in Esc, (N.p \sim \pi) \in R(r)\}$. Then Theorem 7.3 proves that o can be stack allocated in the execution of m that returns at date r in trace T. Since this is true for all executions of m in all traces, we have the required result. Intuitively, we assume that all parameters and the result of m escape (their context is $Path$), and object o does not escape (its context does not contain ϵ). □

Example 7.15 In Example 7.2, the analysis finds that `local` can be stack allocated:

$$\mathbb{E}(\texttt{run})(\texttt{local})(c) = \texttt{elem}.c, \text{ so } \epsilon \notin \mathbb{E}(\texttt{run})(\texttt{local})(Path).$$

Escape analysis is not control-flow sensitive. (Its result does not depend on the order of assignments to fields of objects, but it depends on the order of assignments to local variables; the control flow is only used to compute the SSA form.) Taking the control-flow more precisely into account would require a much more costly analysis such as [WR99, CGS+99]. Experiments have shown that this does not prevent our analysis from giving precise results.

Example 7.16 When there is an assignment `x.f = y` in method m, we have: $\mathbb{E}(m)(\mathtt{y}) \geq \mathtt{f}^{-1}.\mathbb{E}_S(m)(\mathtt{x})$ which expresses that when `x.f` escapes, `y` escapes, and $\mathbb{E}(m)(\mathtt{x}) \geq \mathtt{f}.\mathbb{E}_S(m)(\mathtt{y})$ which expresses that if `y` escapes, `x.f` escapes. The first equation seems fairly natural, but the second one may be surprising at first sight. It is however necessary, as the following example shows:

```
t x = new t(); t' y = new t'(); t'' z = new t''();
C.static_field = y;
x.f = y;
x.f.f' = z;
```

First notice that `y` escapes in the static field `C.static_field`. When executing the assignment `x.f = y`, without this second equation $\mathbb{E}(m)(\mathtt{x}) \geq \mathtt{f}.\mathbb{E}_S(m)(\mathtt{y})$, the $\mathbb{E}$ and $\mathbb{E}_S$-escaping parts of `x` would be empty. Therefore we would think that `z` does not escape, which is wrong. In effect, the above program stores `z` in `y` (it is equivalent to `y.f' = z`) and since `y` escapes, `z` also escapes.

Because of these two equations, as soon as an object `y` is stored in an object `x` by `x.f = y`, the escaping part $\mathbb{E}(m)(\mathtt{y})$ is equal to $\mathbb{E}_S(m)(\mathtt{y})$. Indeed, using notations of the assignment at the beginning of this example, and assuming that $\forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j)$,

$$
\begin{aligned}
\mathbb{E}(m)(\mathtt{y})(c, c_0, ..., c_j) &\supseteq \mathtt{f}^{-1}.\mathbb{E}_S(m)(\mathtt{x})(c, c_0, ..., c_j) && \\
&\supseteq \mathtt{f}^{-1}.\mathbb{E}(m)(\mathtt{x})(c, c_0, ..., c_j) && \text{by Lemma 7.12.}\\
&\supseteq \mathbb{E}_S(m)(\mathtt{y})(c, c_0, ..., c_j) && \\
\text{and } \mathbb{E}_S(m)(\mathtt{y})(c, c_0, ..., c_j) &\supseteq \mathbb{E}(m)(\mathtt{y})(c, c_0, ..., c_j) && \text{again by Lemma 7.12.}
\end{aligned}
$$

Similarly, for the object `x` in which we store, we have

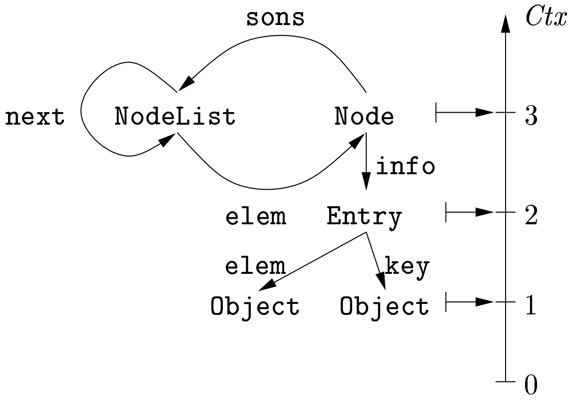
$$\mathtt{f}^{-1}.\mathbb{E}(m)(\mathtt{x})(c, c_0, ..., c_j) = \mathtt{f}^{-1}.\mathbb{E}_S(m)(\mathtt{x})(c, c_0, ..., c_j).$$

This suggests that we could let $\mathbb{E} = \mathbb{E}_S$ without losing much precision. But contrary to what one could think at first sight, this would not simplify the analysis. Indeed, in this case, we could not even express by a function the context transformers. There would be only relations between escaping parts, which would not be easier to simplify and to put into canonical form. (The function $\mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ would become the equation $c_i = \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ where the formal parameters of m are $p_0, ..., p_j$.)

7.5 Analysis $\mathbb{L}$: Escape Analysis using Integer Contexts

The escaping part of an object will be now represented by an integer. So the set of contexts for analysis $\mathbb{L}$ is: $Ctx_{\mathbb{L}} = \mathbb{N}$, ordered by the natural ordering. Integers can be manipulated much faster than sets of paths, therefore this will lead to an efficient escape analysis. The context of an object can be defined from the type of its escaping part, as follows.

For each object type $\tau \in RefType$, let $\mathrm{Cont}(\tau)$ be the set of the types that τ contains (types of the fields, or type of elements if τ is an array). The ***height*** $\top[\tau]$ of an object type τ




```
class Entry {
  Object elem; Object key;
}
class Node {
  NodeList sons; Entry info;
}
class NodeList {
  NodeList next; Node elem;
}
```

Figure 7.9: Type heights. An arrow $\tau \to \tau'$ means $\tau' \in \mathrm{Cont}(\tau)$ i.e. τ has a field of type τ'.

is the greatest context used for objects of type τ. Let $\top[\tau]$ be the smallest integer such that:

$$\top[\tau] \geq 1 \tag{7.3}$$
$$\text{if } \tau' \in \mathrm{Cont}(\tau), \top[\tau'] \leq \top[\tau] \tag{7.4}$$
$$\text{if } \tau' \text{ is a subtype of } \tau, \text{ and } \tau \text{ is not } \texttt{Object}, \top[\tau'] \leq \top[\tau] \tag{7.5}$$
$$\text{if that does not contradict (7.4) and (7.5), if } \tau' \in \mathrm{Cont}(\tau), 1 + \top[\tau'] \leq \top[\tau] \tag{7.6}$$

Only Rule (7.4) is necessary for the correctness of the analysis. Rules (7.5) and (7.6) are useful the get the best possible precision. Rule (7.5) avoids losing precision in type conversions (when we have a context represented with respect to one type, and we want to represent it with respect to another type). Indeed, type conversions occur in general between a type τ and one of its subtypes τ'. Thanks to Rule (7.5), these types have in general the same height: τ' is higher than τ by (7.4) and (7.6) since it has more fields than τ, and τ is higher than τ' by (7.5). The type conversion is then identity. Rule (7.6) distinguishes as many levels as possible.

Example 7.17 Consider the types of Figure 7.9, and assume that no other subtypes are defined. For `Entry`, there is no cycle in the Cont relation (i.e. no object may contain an object of the same type), so (7.6) always applies, and the height of a type is 1 plus the maximum height of the types it contains. For `Node`, there is a strongly connected component in the graph of the Cont relation, which contains `NodeList` and `Node` since $\mathrm{Cont}(\texttt{Node}) = \{\texttt{Entry}, \texttt{NodeList}\}$, and $\mathrm{Cont}(\texttt{NodeList}) = \{\texttt{Node}, \texttt{NodeList}\}$. According to (7.4), $\top[\texttt{Node}] \geq \top[\texttt{NodeList}] \geq \top[\texttt{Node}]$, so we cannot apply (7.6) between `NodeList` and `Node`. (7.6) yields $\top[\texttt{Node}] \geq \top[\texttt{Entry}] + 1 = 3$, so $\top[\texttt{Node}] = \top[\texttt{NodeList}] = 3$.

The escaping part of an object will be represented by the height of its type. (In Example 7.2 for instance, since `local.elem` escapes in `BinTree.run`, and has type `Object`, the context for `local` will be $\top[\texttt{Object}] = 1$.) We do not directly compute types, because computing integers will lead to a faster analysis. However, with integers we cannot in general distinguish different fields of the same object, therefore we lose some precision with this representation. The information for the top of data structures can however be precisely represented. This is very important for stack allocation, since the top of data structures is the part which can most often be stack allocated. The more we go inside complex data structures, the more we lose precision, but also the less stack allocation is probable anyway, even with a precise analysis. This representation is therefore well adapted to stack allocation. The experiments confirm that, even with such an approximation, the analysis is still precise.

Then, we define an abstraction from sets of paths to integers. The height of a path p in an object of type τ is $\top[\tau.p]$:

$$\top[t\,\texttt{[]}.n.p] = \top[t.p] \qquad (n \in \mathbb{N}) \tag{7.7}$$

$$\top[t.n.p] = \top[t.p] \text{ if } t \text{ is not an array } (n \in \mathbb{N}) \tag{7.8}$$

$$\top[t.(C,f,t').p] = \top[t] \sqcap \top[t'.p] \tag{7.9}$$

$$\top[t.\epsilon] = \top[t] \tag{7.10}$$

Equation (7.8) may seem strange, since we use an object of type t which is not array, as if it were one (data accessor $n \in \mathbb{N}$). It may indeed happen if we have type conversions: `Object l; ((int[] l)[n]`. `l` is an array at runtime, but its static type is $t =$ `Object`. For (7.9), a more intuitive definition would be $\top[t.(C,f,t').p] = \top[t'.p]$ but the previous definition is necessary to make sure that $\top[t.p] \leq \top[t]$. In the absence of type conversions, t is a subtype of C in this formula (the field (C,f,t') exists in class t, and thus has been declared in a superclass of t), but in general this is no longer true with type conversions.

The abstraction $\alpha^\tau : Ctx_{\mathbb{E}} \to Ctx_{\mathbb{L}}$ is $\alpha^\tau(c) = \sqcup\{\top[\tau.p] | p \in c\}$ and the concretization $\gamma^\tau : Ctx_{\mathbb{L}} \to Ctx_{\mathbb{E}}$ is $\gamma^\tau(n) = \{p | \top[\tau.p] \leq n\}$. $Ctx^\tau_{\mathbb{L}} = \alpha^\tau(Ctx_{\mathbb{E}}) = [\![0, \top[\tau]]\!]$. $Ctx_{\mathbb{E}} \underset{\alpha^\tau}{\overset{\gamma^\tau}{\leftrightarrows}} Ctx^\tau_{\mathbb{L}}$ is a Galois connection.

Remark. 1. Another abstraction would also have been possible: abstracting a set of access paths c by taking the length of the shortest path in c. This abstraction would have been simpler because it does not use types. Therefore, we would not have the additional complexity of type conversions. However, this abstraction does not yield very good results. For example, if there is a statement $x.f = x.f'.f''$, if a path p of length $n > 1$ escapes in x, a path of length $n-1$ may also escape (p may be $f'.f''.p'$, then $f.p'$ also escapes). Therefore, the escaping part of x is either nothing, or contains all paths of length at least 1. This is not a very precise escape information.

2. As in the ML part, we could define closed contexts by $c \in Ctx_{\mathbb{E}}$ is closed if and only if $\forall p \in c, \forall p' \in Path, p.p' \in c$. Intuitively, if a value escapes, all values contained in it also escape, and all contexts interesting for escape analysis are therefore closed. As in ML, all contexts that can be represented by integers are closed, and our analysis does not carry useless information.

As in analysis $\mathbb{E}$, abstract values are context transformers. But Java supports type casts and subtyping, so the static type of the same object may not be the same during the whole runtime, and we have to remember the assumed type of objects with each context transformer. Therefore, abstract values are now $Val_{\mathbb{L}} = \cup_{n\in\mathbb{N}}((Ctx^n_{\mathbb{L}} \to Ctx_{\mathbb{L}}) \times Type^n \times Type)$.

An abstract value $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau') \in Val_{\mathbb{L}}$ is correct with respect to $E \in Val_{\mathbb{E}}$ if and only if

$$\mathrm{corr}_{\mathbb{L}}(v, E) \Leftrightarrow \forall n_{-1}, ..., n_j, \phi(n_{-1}, ..., n_j) \geq \alpha^{\tau'}(E(\gamma^{\tau_{-1}}(n_{-1}), ..., \gamma^{\tau_j}(n_j))).$$

Let $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$, $v_k = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau'_k)$. We define the following abstract operations:

1. Conversion: The purpose of this operation is to convert a context computed for one type τ to another type τ'. Conversions appear explicitly when using $v_1 = (t)v_2$ to convert types (but in the analysis they are delayed until necessary), or implicitly when using a subtype as a supertype (in method calls or field accesses). $\text{convert}(\tau, \tau') : Ctx_{\mathbb{L}}^{\tau} \to Ctx_{\mathbb{L}}^{\tau'}$ is defined by $\text{convert}(\tau, \tau') = \alpha^{\tau'} \circ \gamma^{\tau}$.

2. Abstract values are ordered by the preorder relation[3] $(\phi, (\tau_{-1}, ..., \tau_j), \tau) \leq (\phi', (\tau'_{-1}, ..., \tau'_j), \tau') \Leftrightarrow \forall n_{-1} \in [\![0, \top[\tau'_{-1}]]\!], ..., \forall n_j \in [\![0, \top[\tau'_j]]\!], \text{convert}(\tau, \tau')(\phi(\text{convert}(\tau'_{-1}, \tau_{-1})(n_{-1}), ..., \text{convert}(\tau'_j, \tau_j)(n_j))) \leq \phi'(n_{-1}, ..., n_j)$. $v_1 \leq v_2$ if and only if v_1 gives a more precise escape information than v_2.

3. Upper bound: $v_1 \sqcup_{\mathbb{L}} v_2 = ((\text{convert}(\tau'_1, \tau'_2) \circ \phi_1) \sqcup \phi_2, (\tau_{-1}, ..., \tau_j), \tau'_2)$ if $\top[\tau_2] > \top[\tau_1]$. Otherwise, we swap the indices 1 and 2.

4. Construction: $(C, f, t)_{\mathbb{L}}.v = (\text{convert}(\tau', t) \circ \phi, (\tau_{-1}, ..., \tau_j), C)$.
   $\mathbb{N}_{\mathbb{L}}.v = (\phi, (\tau_{-1}, ..., \tau_j), \tau'\,\texttt{[]})$.

5. Restriction: $(C, f, t)_{\mathbb{L}}^{-1}.v = ((\text{convert}(\tau', C) \circ \phi) \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t)$.
   $\mathbb{N}_{\mathbb{L}}^{-1}.v = (\phi \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t)$ if $\tau' = t\,\texttt{[]}$. Otherwise, $\mathbb{N}_{\mathbb{L}}^{-1}.v = ((\text{convert}(\tau', t\,\texttt{[]}) \circ \phi) \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t)$ where t can be arbitrarily chosen.

6. Composition: $(\phi', (\tau''_{-1}, ..., \tau''_i), \tau''') \circ_{\mathbb{L}} (v_{-1}, ..., v_i) = (\phi' \circ (\text{convert}(\tau'_{-1}, \tau''_{-1}) \circ \phi_{-1}, ..., \text{convert}(\tau'_i, \tau''_i) \circ \phi_i), (\tau_{-1}, ..., \tau_j), \tau''')$.

If the analyzed method has j parameters of types $\tau_0, ..., \tau_j$ and a result of type τ_{-1}, $\top_{\mathbb{L}}[(C, f, t)] = (\{(c_{-1}, ..., c_j) \mapsto \top[t]\}, (\tau_{-1}, ..., \tau_j), t)$. $\mathit{first}_{\mathbb{L}} = (\{(c_{-1}, ..., c_j) \mapsto c_{-1}\}, (\tau_{-1}, ..., \tau_j), \tau_{-1})$. $\forall i \in [\![0, j]\!], P_{\mathbb{L}}(p_i) = (\{(c_{-1}, ..., c_j) \mapsto c_i\}, (\tau_{-1}, ..., \tau_j), \tau_i)$.

Lemma 7.18 *Let $t\,\texttt{[]}^k$ be the type $t\,\texttt{[]...[]}$ with k times $\texttt{[]}$. Assume that τ and τ' are array types with k and k' dimensions, $\tau = t\,\texttt{[]}^k$, $\tau' = t'\,\texttt{[]}^{k'}$, t and t' are not arrays.*

$$\text{convert}(\tau, \tau')(n) = (n \sqcap \top[\tau']) \sqcup \begin{cases} \top[t'\,\texttt{[]}^{k'-i}] & \text{if } i \geq 0 \text{ minimum such that } i \leq k, i \leq k' \text{ and } n = \top[t\,\texttt{[]}^{k-i}], \\ \top[t'] & \text{if } k > k' \text{ and } \top[t] \leq n \leq \top[t\,\texttt{[]}^{k-k'}], \\ 0 & \text{if } n < \top[t]. \end{cases}$$

(for integers, $\sqcap$ is the minimum, $\sqcup$ is the maximum).

Proof: If $n = \top[\tau]$, $\gamma^{\tau}(n) = \mathit{Path}$, therefore $\alpha^{\tau'} \circ \gamma^{\tau}(n) = \top[\tau']$.
If $n = 0$, $\alpha^{\tau'} \circ \gamma^{\tau}(n) = 0$.
Otherwise, if τ is not an array and $0 < n < \top[\tau]$,

$$\begin{aligned} \alpha^{\tau'} \circ \gamma^{\tau}(n) &= \sqcup\{\top[\tau'.n_1...n_j.(C, f, t).c'] | \top[\tau.n_1...n_j.(C, f, t).c'] \leq n, n_i \in \mathbb{N}\} \\ &= \sqcup\{\top[\tau'.n_1...n_j.(C, f, t).c'] | \top[\tau.(C, f, t).c'] \leq n, n_i \in \mathbb{N}\} \\ &= n \sqcap \top[\tau'] \end{aligned}$$

since $\top[\tau'.n_1...n_j.(C, f, t).c'] \leq \top[\tau'] \sqcap \top[t.c']$, this value being reached when there is no n_i $(j = 0)$.

[3] reflexive and transitive but not antisymmetric

If τ is an array and $0 < n < \top[\tau]$,

$$\begin{aligned}\alpha^{\tau'} \circ \gamma^{\tau}(n) &= \sqcup\{\top[\tau'.(C,f,t).c'] | \top[\tau.(C,f,t).c'] \leq n\} \\ &\quad \sqcup \sqcup\{\top[\tau'.n'.c'] | \top[\tau.n'.c'] \leq n, n' \in \mathbb{N}\} \\ &= (n \sqcap \top[\tau']) \sqcup \alpha^{t'} \circ \gamma^{t}(n)\end{aligned}$$

where $t\texttt{[]} = \tau$ and if τ' is an array $t'\texttt{[]} = \tau'$, otherwise $t' = \tau'$.

Then, an induction over the number of dimensions k of τ yields the expected result, using: If $\tau = t\texttt{[]}$ is an array,

$$\begin{cases}\text{If } n = \top[\tau], & \text{convert}(\tau,\tau')(n) = \top[\tau'] \\ \text{If } 0 < n < \top[\tau], & \text{convert}(\tau,\tau')(n) = (n \sqcap \top[\tau']) \sqcup \text{convert}(t,t')(n) \\ & \text{where } t' = \tau' \text{ if } \tau' \text{ is not an array, and } t'\texttt{[]} = \tau' \text{ otherwise} \\ \text{If } n = 0, & \text{convert}(\tau,\tau')(n) = 0\end{cases}$$

□

When τ and τ' are not array types, this reduces to:

$$\begin{cases}\text{If } \top[\tau'] \leq \top[\tau], & \text{convert}(\tau,\tau')(n) = n \sqcap \top[\tau']. \\ \text{Otherwise} & \text{convert}(\tau,\tau')(n) = \text{if } n \geq \top[\tau] \text{ then } \top[\tau'] \text{ else } n.\end{cases}$$

The difference between array and object types for the conversion comes from the fact that converting object types means adding or removing fields, whereas converting array types means converting the type of their elements.

Comparison with ML The type conversion takes a completely different form in ML and Java. ML uses Hindley-Milner polymorphism: the type of polymorphic values is represented by type variables. When a polymorphic type is instantiated, the type variable is replaced with the same type in all its occurrences. Our escape analysis uses this property to have a precise instantiation. For example, the identity function $\texttt{id} = \texttt{fun}\ x \to x$ has type $\beta \to \beta$. If it is used with a value of type `int list list`, its type becomes `int list list` $\to$ `int list list`.

On the contrary, the corresponding Java method is `Object id(Object` x`){return x;}`. If it is used with type `int[][]` for example, the type of its result remains `Object`.

Therefore, the ML type system gives a more precise information: the result of the function is of type `int list list`. This information can be used by our escape analysis. For example, if an element of type `int list` of the result of `id` may escape, our analysis finds that only elements of type `int list` may escape from the parameter. In Java, the escape information for the result is first converted to type `Object` which leads to a loss of precision: the result may escape entirely, and we do not find that the parameter cannot escape entirely. The more precise instantiation of ML cannot be adapted to Java because of its type system.

Remark. In some cases, when τ_1' or τ_2' are array types, our definition of the upper bound may not give exactly the *least* upper bound (using the above preorder). In fact, the least upper bound (with the same types as v_1 and v_2 for the input) does not always exist. For example, take $\tau_1 = t_1\texttt{[]}$, with $\top[t_1] = 3$, and $\tau_2 = t_2\texttt{[]}$ with $\top[t_2] = 2$. Take $v_1 = (\{(c_1, c_2) \mapsto c_1\}, (\tau_1, \tau_2), \tau_1)$, $v_2 = (\{(c_1, c_2) \mapsto c_2\}, (\tau_1, \tau_2), \tau_2)$, and assume that $v_3 = (\phi_3, (\tau_1, \tau_2), \tau_3)$ is the least upper bound of v_1 and v_2. We have $v_3 \geq v_1$ and $v_3 \geq v_2$, therefore $\forall c_1 \in [\![0, \top[\tau_1]]\!], \forall c_2 \in [\![0, \top[\tau_2]]\!], \phi_3(c_1, c_2) \geq \mathtt{convert}(\tau_1, \tau_3)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_3)(c_2)$. Since v_3 is the least upper bound, for all $v_4 = (\phi_4, (\tau_1, \tau_2), \tau_4)$ greater than v_1 and v_2, $v_4 \geq v_3$. Therefore, for all τ_4, we would have $\mathtt{convert}(\tau_1, \tau_4)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_4)(c_2) \geq \mathtt{convert}(\tau_3, \tau_4)(\mathtt{convert}(\tau_1, \tau_3)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_3)(c_2))$.

- If τ_3 is not an array type, take $c_1 = 3, c_2 = 0, \tau_4 = \tau_1$. Then $\mathtt{convert}(\tau_1, \tau_3)(c_1) = \top[\tau_3]$, hence $\mathtt{convert}(\tau_3, \tau_4)(\mathtt{convert}(\tau_1, \tau_3)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_3)(c_2)) = \top[\tau_4] = 4$, and we get $\mathtt{convert}(\tau_1, \tau_4)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_4)(c_2) = 3$. Contradiction.

- If $\tau_3 = t_3\,\mathtt{[]}^k$ with $\top[t_3] \leq 2$, take $c_1 = 2, c_2 = 0, \tau_4 = \tau_1$. $\mathtt{convert}(\tau_1, \tau_3)(c_1) \geq \top[t_3]$, hence $\mathtt{convert}(\tau_3, \tau_4)(\mathtt{convert}(\tau_1, \tau_3)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_3)(c_2)) \geq \top[t_1] = 3$, and $\mathtt{convert}(\tau_1, \tau_4)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_4)(c_2) = 2$. Contradiction.

- If $\tau_3 = t_3\,\mathtt{[]}^k$ with $\top[t_3] > 2$, take $c_1 = 0, c_2 = 2, \tau_4 = \tau_2$. $\mathtt{convert}(\tau_2, \tau_3)(c_2)) = \top[t_3]$ hence $\mathtt{convert}(\tau_3, \tau_4)(\mathtt{convert}(\tau_1, \tau_3)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_3)(c_2)) \geq \top[t_3] \sqcap \top[\tau_4] = 3$. $\mathtt{convert}(\tau_1, \tau_4)(c_1) \sqcup \mathtt{convert}(\tau_2, \tau_4)(c_2) = 2$. Contradiction.

In this case, we have proved that the least upper bound does not exist.

However, our definition of the upper bound is always correct (i.e. greater than v_1 and v_2). Choosing the highest type as the type of the result enables us to give the least upper bound when τ'_1 and τ'_2 are not array types, and to have a good approximation in the other cases.

Theorem 7.19 (Correctness) *Assume that* $\mathrm{corr}_{\mathbb{L}}(v_1, E_1)$ *and* $\mathrm{corr}_{\mathbb{L}}(v_2, E_2)$. *Then* $\mathrm{corr}_{\mathbb{L}}(v_1 \sqcup_{\mathbb{L}} v_2, E_1 \sqcup E_2)$, $\mathrm{corr}_{\mathbb{L}}((C, f, t)_{\mathbb{L}}^{-1}.v_1, (C, f, t)^{-1}.E_1)$, *and similarly for other primitive operations.*

Proof: 1. We have $\mathrm{corr}_{\mathbb{L}}((\alpha^{\tau'} \circ \gamma^{\tau}, (\tau), \tau'), \mathrm{Id})$ hence the correctness of the conversion: $\mathrm{corr}_{\mathbb{L}}((\mathtt{convert}(\tau, \tau'), (\tau), \tau'), \mathrm{Id})$.

2. Upper bound: Let us prove $\mathrm{corr}_{\mathbb{L}}((\{(n_1, n_2) \mapsto n_1 \sqcup n_2\}, (\tau, \tau), \tau), \{(c_1, c_2) \mapsto c_1 \cup c_2\})$. Since α^{τ} is additive,

$$\alpha^{\tau}(\gamma^{\tau}(n_1) \cup \gamma^{\tau}(n_2)) = \alpha^{\tau}(\gamma^{\tau}(n_1)) \sqcup \alpha^{\tau}(\gamma^{\tau}(n_2)) \leq n_1 \sqcup n_2.$$

This is the expected result.

3. Composition: Let $v_k = (\phi_k, (\tau_{-1}, ..., \tau_i), \tau'_k)$, $v' = (\phi', (\tau'_{-1}, ..., \tau'_j), \tau'')$. $v' \circ_{\mathbb{L}} (v_{-1}, ..., v_j) = (\phi' \circ (\phi_{-1}, ..., \phi_j), (\tau_{-1}, ..., \tau_i), \tau'')$. Assume $\mathrm{corr}_{\mathbb{L}}(v_k, E_k)$ and $\mathrm{corr}_{\mathbb{L}}(v', E')$. Then $\mathrm{corr}_{\mathbb{L}}(v' \circ_{\mathbb{L}} (v_{-1}, ..., v_j), E' \circ (E_{-1}, ..., E_j))$. Indeed

$$\begin{aligned}
&\alpha^{\tau''}((E' \circ (E_{-1}, ..., E_j))(\gamma^{\tau_{-1}}(n_{-1}), ..., \gamma^{\tau_i}(n_i))) \\
&\leq \alpha^{\tau''}(E'(\gamma^{\tau'_{-1}}(\alpha^{\tau'_{-1}}(E_{-1}(\gamma^{\tau_{-1}}(n_{-1}), ..., \gamma^{\tau_i}(n_i)))), ...)) \\
&\leq (\phi' \circ (\phi_{-1}, ..., \phi_j))(n_{-1}, ..., n_i)
\end{aligned}$$

by correctness of v', $v_{-1}, ..., v_j$.

4. Construction for objects: Assume $t \in \mathrm{Cont}(t')$ and t' is not an array.

$$\begin{aligned}\alpha^{t'}((C,f,t).\gamma^t(n)) &= \alpha^{t'}(\{(C,f,t).c|\top[t.c] \leq n\})\\ &= \sqcup\{\top[t'.(C,f,t).c]|\top[t.c] \leq n\}\\ &= \sqcup\{\top[t'] \sqcap \top[t.c]|\top[t.c] \leq n\}\\ &\leq n\end{aligned}$$

(and since $n \leq \top[t]$ and $\top[t] \leq \top[t']$ by the second rule of computation of $\top$, it is useless to write $n \sqcap \top[t']$). Hence, $\mathrm{corr}_{\mathbb{L}}((\mathrm{Id},(t),t'),\{c \mapsto (C,f,t).c\})$.

5. Construction for arrays:

$$\alpha^{t[]}(\mathbb{N}.\gamma^t(n)) = \sqcup\{\top[t[].n'.c]|\top[t.c] \leq n, n' \in \mathbb{N}\} \leq n$$

Hence $\mathrm{corr}_{\mathbb{L}}((\mathrm{Id},(t),t[]),\{c \mapsto \mathbb{N}.c\})$.

6. Restriction for objects: Assume $t \in \mathrm{Cont}(\tau)$ and τ is not an array. If $n \geq \top[\tau]$,

$$\alpha^t((C,f,t)^{-1}.\gamma^\tau(n)) = \alpha^t((C,f,t)^{-1}.Path) = \top[t] = n \sqcap \top[t]$$

(here, we use the second rule of computation of type heights, to make sure that $\top[\tau] \geq \top[t]$, and therefore if $n \geq \top[\tau]$, then $n \geq \top[t]$). Otherwise,

$$\alpha^t((C,f,t)^{-1}.\gamma^\tau(n)) = \sqcup\{\top[t.c]|\top[\tau.(C,f,t).c] \leq n\} \leq n \sqcap \top[t]$$

(since $\top[\tau.(C,f,t).c] = \top[\tau] \sqcap \top[t.c]$ and since $n < \top[\tau]$, $\top[\tau.(C,f,t).c] \leq n$ if and only if $\top[t.c] \leq n$. Moreover, $\forall t,c, \top[t.c] \leq \top[t]$). Hence $\mathrm{corr}_{\mathbb{L}}((\{n \mapsto n \sqcap \top[t]\},(\tau),t),\{c \mapsto (C,f,t)^{-1}.c\})$.

7. Restriction for arrays:

$$\alpha^t(\mathbb{N}^{-1}.\gamma^{t[]}(n)) = \sqcup\{\top[t.c]|\top[t[].\mathbb{N}.c] \leq n\} \leq n \sqcap \top[t]$$

since $\forall t,c, \top[t.c] \leq \top[t]$. Therefore $\mathrm{corr}_{\mathbb{L}}((\{n \mapsto n \sqcap \top[t]\},(t[]),t),\{c \mapsto \mathbb{N}^{-1}.c\})$.

Using the correctness of composition and the elementary correctness proofs for conversion, constructions and restrictions above, we get immediately the required result. □

All definitions of Section 7.4.2 are still correct for analysis $\mathbb{L}$, provided that we use the new abstract domain for values, with its associated abstract primitives. We get the correctness theorem from Theorems 7.14 and 7.19:

Theorem 7.20 (Stack allocation) *Consider* $v = \texttt{new}\ C/\texttt{new}\ t[w]$ *in method* m*, allocating object* o*. With* $\mathbb{L}(m)(v) = (\phi,(\tau_{-1},...,\tau_j),\tau')$*, if* $\phi(\top[\tau_{-1}],...,\top[\tau_j]) < \top[\tau']$*,* o *can be stack allocated.*

Proof: We have $\mathrm{corr}_{\mathbb{L}}(\mathbb{L}(m)(v),\mathbb{E}(m)(v))$ using the correctness theorem 7.19 at each step of the computation. Then by definition of the correctness,

$$\top[\tau'] > \phi(\top[\tau_{-1}],...,\top[\tau_j]) \geq \alpha^{\tau'}(\mathbb{E}(m)(v)(Path,...,Path))$$

therefore $\epsilon \notin \mathbb{E}(m)(v)(Path,...,Path)$ hence the result by Theorem 7.14. □

Theorem 7.21 (Additivity) *Let $\mathbb{L}(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$, then ϕ is additive. $\mathbb{L}_S$ has the same property.*

Proof: Additivity is preserved by composition, upper bound, construction and restriction. $first_{\mathbb{L}}$, $P_{\mathbb{L}}(p_i)$ and $\top_{\mathbb{L}}[f]$ are additive (more precisely, the function in their first component is additive). The conversion is monotone, hence additive (it has only one parameter). Since additivity is also preserved by fixed point, we have the expected result. □

Example 7.22 Let m be the `BinTree` constructor from Example 7.2. The analyzer yields

$$\begin{aligned}\mathbb{L}(m)(\texttt{this}) &= (\{(c_0, c_1, c_2, c_3) \mapsto c_1 \sqcup c_2 \sqcup c_3\},\\ &\qquad (\texttt{BinTree}, \texttt{BinTree}, \texttt{BinTree}, \texttt{Object}), \texttt{BinTree})\\ \mathbb{L}(m)(\texttt{1}) &= (\{(c_0, c_1, c_2, c_3) \mapsto c_0\}, (\texttt{BinTree}, \texttt{BinTree}, \texttt{BinTree}, \texttt{Object}), \texttt{BinTree})\end{aligned}$$

Analysis $\mathbb{L}$ yields the same relations between escaping parts of parameters as $\mathbb{E}$, so it is also very precise. Analysis $\mathbb{L}$ finds that `local` can be stack allocated: $\mathbb{L}(\texttt{BinTree.run()})(\texttt{local}) = (\phi = \{c_0 \mapsto c_0\}, (\texttt{Object}), \texttt{BinTree})$ and $\phi(\top[\texttt{Object}]) = 1 < \top[\texttt{BinTree}]$.

7.6 Representation of context transformers

Now, we wish to find a simple representation of the context transformers of analysis $\mathbb{L}$, to make the implementation easier.

Context transformers of analysis $\mathbb{L}$ are of the form $((Ctx_{\mathbb{L}}^n \to Ctx_{\mathbb{L}}) \times Type^n \times Type)$, $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$. Moreover, ϕ is additive (Theorem 7.21), so it can be split into $\phi(c_{-1}, ..., c_j) = g_{-1}(c_{-1}) \sqcup ... \sqcup g_j(c_j)$. Now we still have to represent the monotone functions $\mathbb{N} \to \mathbb{N}$ g_{-1}, ..., g_j.

The needed elementary operations are built by composition and upper bound from constants, identity, intersection: $n \mapsto n \sqcap \top[t]$, and step: $n \mapsto \text{if } n \geq \top[\tau] \text{ then } \top[\tau'] \text{ else } 0$ (the conversion between array types is the upper bound of several step functions). This suggests the general form

$$c \mapsto (\text{if } c \geq s \text{ then } s^+ \text{ else } 0) \sqcup (c \sqcap l) \sqcup u \qquad (s^+, u \in \mathbb{N}, s, l \in \mathbb{N} \cup \{\infty\})$$

(s stands for step, l for lower bound, u for upper bound). However, with this general form, we cannot represent all monotone functions. We have to use approximate operations for upper bound $\sqcup'$ and composition $\circ'$. If $\phi_1(c) = (\text{if } c \geq s_1 \text{ then } s_1^+ \text{ else } 0) \sqcup (c \sqcap l_1) \sqcup u_1$ and $\phi_2(c) = (\text{if } c \geq s_2 \text{ then } s_2^+ \text{ else } 0) \sqcup (c \sqcap l_2) \sqcup u_2$,

$$(\phi_1 \sqcup' \phi_2)(c) = (\text{if } c \geq s_1 \sqcap s_2 \text{ then } s_1^+ \sqcup s_2^+ \text{ else } 0) \sqcup (c \sqcap (l_1 \sqcup l_2)) \sqcup (u_1 \sqcup u_2)$$

$$\begin{aligned}(\phi_1 \circ' \phi_2)(c) = \text{if } c \geq{} & \underline{(\text{if } l_2 \geq s_1 \text{ then } s_1 \sqcap s_2 \text{ else } s_2)} \text{ then}\\ & \underline{[(\text{if } l_2 \sqcup s_2^+ \geq s_1 \text{ then } s_1^+ \text{ else } 0) \sqcup (s_2^+ \sqcap l_1)]} \text{ else } 0\\ \sqcup\, & (c \sqcap \underline{(l_2 \sqcap l_1)}) \sqcup \underline{((u_2 \sqcap l_1) \sqcup u_1 \sqcup (\text{if } u_2 \geq s_1 \text{ then } s_1^+ \text{ else } 0))}\end{aligned}$$

Notice that the approximate composition $\circ'$ is *not* associative.

Finally, we can represent function ϕ by $\phi(c_{-1}, ..., c_j) = (\text{if } c_{-1} \geq s_{-1} \text{ then } s_{-1}^+ \text{else } 0) \sqcup (c_{-1} \sqcap l_{-1}) \sqcup ... \sqcup (\text{if } c_j \geq s_j \text{ then } s_j^+ \text{ else } 0) \sqcup (c_j \sqcap l_j) \sqcup u$. Notice that there is not a unique representation for each function, and that the operations $\sqcup'$ and $\circ'$ depend not only on the functions but also on their representations. The choice of the values of s_i, s_i^+, l_i, u is therefore important. Then functions can be better represented by the tuple of integers $((s_{-1}, s_{-1}^+, l_{-1}), ..., (s_j, s_j^+, l_j), u)$ ordered by

$$((s_{-1}, s_{-1}^+, l_{-1}), ..., (s_j, s_j^+, l_j), u) \leq ((s'_{-1}, {s_{-1}^+}', l'_{-1}), ..., (s'_j, {s_j^+}', l'_j), u')$$
$$\Leftrightarrow u \leq u' \wedge \forall i, s_i \geq s'_i \wedge s_i^+ \leq {s_i^+}' \wedge l_i \leq l'_i$$

and we can define a concretization $\gamma_{\mathbb{L}_1}$, giving the meaning of this tuple of integers in terms of functions:

$$\gamma_{\mathbb{L}_1}((s_{-1}, s_{-1}^+, l_{-1}), ..., (s_j, s_j^+, l_j), u) = \lambda c_{-1}...c_j.(\text{if } c_{-1} \geq s_{-1} \text{ then } s_{-1}^+ \text{ else } 0)$$
$$\sqcup (c_{-1} \sqcap l_{-1}) \sqcup ... \sqcup (\text{if } c_j \geq s_j \text{ then } s_j^+ \text{ else } 0) \sqcup (c_j \sqcap l_j) \sqcup u$$

We can easily check that $\gamma_{\mathbb{L}_1}$ is monotone, which means that our ordering of tuples of integers is coherent with the ordering of functions. The best precision is achieved by choosing the smallest (according to the above ordering) tuple of integers representing a given function. This yields the following representations for identity: $\lambda c.(\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap \infty) \sqcup 0$, intersection: $\lambda c.(\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap \top[t]) \sqcup 0$. For type conversions from $\tau = t\texttt{[]}^k$ to $\tau' = t'\texttt{[]}^{k'}$, let $S = \{n | \text{convert}(\tau, \tau')(n) > n\}$. If $S = \emptyset$, $\text{convert}(\tau, \tau')(n) = n \sqcap \top[\tau']$ (as $\text{convert}(\tau, \tau')(n) \geq n \sqcap \top[\tau']$ and $\text{convert}(\tau, \tau')(n) \leq \top[\tau']$) which can be exactly represented by $\lambda n.(\text{if } n \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (n \sqcap \top[\tau']) \sqcup 0$. Otherwise let us define $s = \min S$, $s^+ = \text{convert}(\tau, \tau')(\max S)$. Then $\text{convert}(\tau, \tau')(n) \leq (\text{if } n \geq s \text{ then } s^+ \text{ else } 0) \sqcup (n \sqcap \top[\tau']) \sqcup 0$ which can be exactly represented.

The analysis with one-step context transformers will be denoted $\mathbb{L}_1$. Its correctness with respect to $\mathbb{L}$ comes from the fact that if $\mathbb{L}(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau)$, $\mathbb{L}_1(m)(v) = (\phi_1, (\tau_{-1}, ..., \tau_j), \tau)$ with $\phi_1 \geq \phi$. Theorem 7.20 immediately extends to $\mathbb{L}_1$.

Moreover, we use a sparse representation, where only the useful parameters appear, to save up both memory and compute time, since in general a context only depends on a few of the parameters of the method.

Comparison with ML Analysis $\mathbb{L}$ is very similar to analysis L' for ML, described in Appendix C: the context transformers have the same general form (except that they have several parameters for Java, and only one for ML). We use the same technique to simplify and speed up the implementation both in ML and Java: we restrict the general form of context transformers to one step.

7.7 Extensions

Type analysis The precision of the analysis of virtual calls is improved by a simple but very useful type analysis. When a parameter of a called method m' has type `Object`, and at the call to m' we have a more precise type information for this parameter (it is of type τ or a subtype of τ), we consider a specialized instance of m' with type τ instead of `Object`, and we reanalyze the code of m' with this more precise information. For example, consider `String.valueOf(Object obj)`

```
class String {
   static String valueOf(Object obj) {
      return (obj == null) ? "null" : obj.toString();
   }
}
```

When we can determine a more precise type for the object `obj`, we can determine more precisely which `toString` method will be called, and improve the precision of the escape information of `String.valueOf`. This way, we can often determine that `obj` does not escape. This is useful for example in string concatenation sequences `"s1" + o1 + "s2" + o2` to prove that objects `o1` and `o2` do not escape (if their `toString` methods do not make them escape, which is most often the case).

Moreover, when an object is created by `new`, its type is exactly known. We propagate this type information (intraprocedurally only). If a virtual method is called in this object, we can determine exactly which method will be called. We can therefore use the escape information of this method instead of taking the upper bound of the escape information of several methods.

Dynamic loading To be able to compute the environments $\rho'(m')$ and $\rho'_{\mathrm{S}}(m')$ used in the analysis of virtual calls, we have to know all subclasses that redefine m'. This is a problem (except for `final` classes, which cannot have subclasses) since Java may load new classes at runtime. We solve this by determining automatically as often as possible which classes will actually be loaded. This can be achieved when the name of the loaded class is constant, such as in `Class.forName("name")` or `name.class`. The expression `(`C`)Class.forname(`s`).newInstance()` loads a class of name s, creates an object of this class, and casts this object to class C. It can only run without error if the loaded class is a subclass of C. Therefore, we consider that all subclasses of C may be loaded by such an expression. In other cases, we rely on information provided by the user, or we consider that any method may be called: the parameters and result may escape when there is a virtual call to method m'.

Analysis in the presence of unknown code If some of the code remains unknown and the user does not provide information on it, we can still perform an escape analysis, but much precision is lost. In this case, when a virtual method m' is called, its parameters always escape (except when m' is final or the type of the receiver `this` can be statically determined). Nevertheless, this less precise analysis can be useful to analyze librairies, if we do not want to recompile them for each application.

Satisfactory precision and fast compilation can be obtained by the following steps. The Java standard library is first analyzed with the less precise analysis above, and it is compiled once and for all. For each program, the code of the program and the part of the library that the program uses are analyzed at once with the more precise analysis, assuming that all code is known. The program is then compiled. This is the process that has been used in our benchmarks.

In case of code changes, the entire program has to be reanalyzed and recompiled. (If a library has been analyzed with the analysis above, and the library has not been modified, then of course the library does not need to be recompiled, even if programs that use it have been modified.)

Native methods For native methods, the escape information cannot be determined automatically. This escape information can be provided by the user, or we consider that all parameters and the result of native methods escape.

Exceptions We consider that all exceptions objects escape when they are thrown by `throw`. This is not very precise, but in general exception objects cannot be stack allocated anyway. Exception handlers are analyzed by considering that there is a jump from each program point in the `try` block to the exception handler.

Finalization The programmer can define for each class a method that is called just before an object of the class in garbage collected. This method is called a finalizer. In our implementation, objects that have a non-empty finalizer are not stack allocated. This could be extended by running the finalizer at the end of the method in which the object is stack allocated.

Also notice that the finalizer may "resurrect" the object. (The object is unreachable when the finalizer is called, but the finalizer can store it again in a data structure.) When the finalizer makes the object escape (the escape context of `this` yields this information when analyzing the finalizer), we have to consider that all objects of the class escape. This is especially necessary for the correctness of synchronization elimination.

Chapter 8

Implementation

Our escape analysis has been implemented in the Java-to-C compiler turboJ [W$^+$98] produced by the Silicomp Research Institute. TurboJ takes Java bytecode as input and produces C code which is compiled by `gcc`. TurboJ also outputs a copy of the input Java class files in which all methods have been changed to be native methods. These class files can then be executed, and their execution will in fact call the compiled C code. Therefore, turboJ uses the runtime of the JDK. In particular, it uses its garbage collector. In our experiments, we have used the JDK 1.1.5, which has a mark-and-sweep garbage collector. Of course, to get satisfactory performance, the C code generated by turboJ accesses the internal structure of objects and calls other compiled methods directly, without calling the Java Virtual Machine (JVM). The generated C code is therefore specific to a particular JVM, to fit its internal structure of objects. The turboJ compiler is written in Java and so is our analyzer.

8.1 Computing type levels

The height of type τ, $\top[\tau]$, is computed using the rules (7.3) to (7.6) given in Section 7.5. To do this, we split the graph of the relation $\tau' \in \mathrm{Cont}(\tau) \vee (\tau'$ subtype of $\tau \wedge \tau \neq \mathtt{Object})$ into strongly connected components. Inside a strongly connected component, all types have the same height by rules (7.4) and (7.5). Between strongly connected components, we add one to the height, following rule (7.6).

Notice that the set of fields of a class is not given in a single class file. Super-classes also have to be read. Computing strongly connected components is linear in the size of the graph, so the total time to compute type heights is $\mathcal{O}(t)$ where t is the number of different types in the analyzed program.

8.2 The analysis algorithm

The analysis algorithm works in several passes, as follows:

For each method m, if it has not been analyzed yet, search the graph of methods that may be called from m (and that have not been analyzed yet), and compute the strongly connected components of this graph.

For each strongly connected component,

- Build the equations for each method:

  - Transform the code into SSA form.
  - Create unknowns $\mathbb{L}_1(v)$ and $\mathbb{L}_{1S}(v)$ for each variable v. Emit the equations mentioned in Figure 7.8, for each instruction, in any order.

- Solve the equations, by an iterative fixed point solver. To obtain satisfactory performance, we begin with splitting the dependence graph into strongly connected components, and we solve each component separately, using the algorithm of [HDT87]. The size of systems is therefore much reduced.

- Prepare the post-transformation, by building the structure giving for each allocation its escape information.

The post-transformation itself is done when generating the C code.

Remark. Technically, our analysis has been implemented directly on the bytecode, without first transforming it into SSA form. However, SSA form simplifies the theoretical presentation.

Another point of view would be that we transform the program into SSA form in a very naive way: we introduce ϕ-functions for all variables on meet points which are in loops (it is easy to determine the really useful ϕ-functions for meet points which are not in loops). This is of course not optimal, but correct and it gives reasonable results in practice.

We optimize the algorithm to reduce the number of introduced unknowns and generated equations. The algorithm we use for building equations is then more precisely:

- Sort the bytecode topologically by a depth first search. In case of cycle (loop), we cannot sort topologically. In this case, we mark specially the instruction at the end of the back edge during the depth first search. This phase also determines the strongly connected components of the control-flow graph of the program. This determines which instructions are in a loop. This will be useful for the extensions detailed in Section 8.3.2.

- Compute the equations for $\mathbb{L}_{1S}$ using a forward pass (in the order determined above). New unknowns are introduced at the end of back edges, since we need the analysis of the current bytecode to know the state before it.

- Compute the equations for $\mathbb{L}_1$ using a backward pass.

The escape information for the elements of the stack is represented by a list. For local variables, it is represented by a binary tree, each bit of the variable number indicates which branch of the tree should be visited to find the variable. This provides a sharing between data useful for the analysis of one instruction and the following one, and leads to a faster analyzer with smaller memory usage.

8.3 Program transformation

8.3.1 Stack allocation

Theorem 7.20 gives a criterion to decide stack allocation. Stack allocation is implemented by replacing an allocation `new`, `newarray`, `anewarray` or `multianewarray` by a stack allocation `alloca` when the allocated object does not escape from the method in which it is allocated.

If the size of the stack allocated value is statically known (i.e. it is an object or an array of constant number of elements), and the allocation is not in a loop, the stack allocation is done directly in the method frame, without using `alloca`.

Some extensions to this simple implementation are very important to get satisfying results. They will be detailed in the following sections.

8.3.2 Reuse allocated space in loops

If a stack allocation takes place in a loop, the allocation is done again at each iteration, which leads to increasing the stack size (this may lead to program failure because of stack overflow [Cha88]) whereas without stack allocation, the stack size remained constant. Furthermore, data referenced from the stack are considered as always alive by the GC (the GC is conservative for the stack, in the version of the JDK we have used), which may lead to keep important quantities of useless data in the heap.

Example 8.1 A typical example of this situation is the following:

```
String s = "";
for (int i = 0; i < 10000; i++)
  s += i + " ";
//i.e. s=new StringBuffer(s).append(i).append(" ").toString();
```

where all intermediate `StringBuffers` used to compute `s += i + " "` can be stack allocated, and so all intermediate char arrays are kept uselessly by the GC until the end of the method.

Comparison with ML Avoiding `alloca` in a loop is similar to preserving the tail recursion optimization in ML. Reusing the same allocated space corresponds to the `letstack`′ expression of ML. `letstack`′ deallocates the stack allocated value just before the tail call. In Java, the stack allocated value is virtually deallocated and immediately reallocated at the next iteration of the loop.

To solve this problem, we have extended our analysis so that, if an allocated object is always useless after one iteration, we do not use a new space for each allocation, but we reuse the already allocated space. This way, the stack does not grow. To determine whether the object will still be useful at the next iteration, we use the following criterion:

> Consider that all variables which are live just before the allocation escape. If the allocated object does not escape, then the allocated space can be reused after one iteration.

The intuitive idea behind this criterion is the following: consider for example the `while` loop

```
while (test) { ... x:  new C() ... }
```

and assume that we unroll that loop:

```
  if (!test) goto end;
  ... x1:  new C() ...
  if (!test) goto end;
  ... x2:  new C() ...
  ...
end:
```

Let `o` be the object allocated at `x1`. Assume that `o` does not escape when we consider that all variables live at `x2` escape. Then `o` is not reachable from a variable live at `x2`. Therefore, `o` is not live at `x2`. In this case, we can reuse at `x2` the memory space allocated at `x1`.

Technically, when analyzing a `new`, we emit, together with the equations already mentioned in the preceding section, the following equation for each variable v live before the `new`, t being the type of this variable:

$$\phi(p_{-1}, ..., p_n) \geq p_k \sqcap \top[t]$$

where $\mathbb{L}_1(m)(v) = (\phi, (\tau_{-1}, ..., \tau_n), t)$, $n \geq k$, p_k is a new parameter. For the normal escape analysis (the one which determines whether `alloca` should be introduced), p_k is 0. In this case, the above equation becomes $\phi(p_{-1}, ..., p_n) \geq 0$ and it can be removed: we did not change the results of the normal escape analysis. When determining if the allocation at pc can reuse the space allocated at the preceding iteration, $p_k = \infty$, and the above equation becomes $\phi(p_{-1}, ..., p_n) \geq \top[t]$ meaning that variable v escapes. Therefore, when $p_k = \infty$, all variables live before the allocation are regarded as escaping. If the allocated object does not escape with $p_k = \infty$, we can reuse the same stack allocated space for all iterations. A new p_k parameter is introduced for each allocation which occurs in a loop.

In the case of objects or constant size arrays, reusing the allocated space is relatively easy. It is more difficult for variable size arrays. In this case, we use the following algorithm:

- If the new size is smaller than the already allocated one, the space is simply reused.
- Otherwise, if the previously allocated value is still at the top of the stack, the allocated space is enlarged to be able to allocate the new array.
- Otherwise, the new array cannot be allocated in the old space. A new space is then allocated even if this may cause stack overflows if a large size is allocated. This did not cause problems during our experiments.

For very large arrays, even a single allocation could cause a stack overflow. A dynamic test could easily be performed at each stack allocation of arrays whose size is not statically known, to limit the size of the stack allocated arrays (if the allocated array is too large, it is heap allocated).

8.3.3 Inlining

The algorithm can still be improved to discover more stack allocation opportunities: it may happen that an object d cannot be stack allocated in method m' because it is still live at the end of m', but becomes dead at the end of method m which has called m'. In this case, we inline m' in m and allocate d on the stack in m. Our implementation also applies this technique to longer call chains (m calls m' which calls m'' ...).

However, we have to determine at the end of which method d will be dead. This is the subject of this section.

Stack allocation criterion with inlining

Assume that a method m' with $j+1$ parameters is called at pc in method m: $r = p_0.m'(p_1, ..., p_j)$. Then, we define

$$\mathbb{L}_{1C}(pc) = (\mathbb{L}_1(m)(r), \mathbb{L}_{1S}(m)(p_0), ..., \mathbb{L}_{1S}(m)(p_j))$$

the $(j+2)$-tuple containing the escaping part of the result and parameters of method m' in method m. $\mathbb{L}_{1C}(pc)$ is the transformer which converts escaping parts computed in the callee m' to escaping parts computed in the caller m, taking into account that the scope of m is larger than the scope of m'.

Theorem 8.2 (Inlining) *Assume that method m_0 calls m_1 at pc_1, which calls m_2 at pc_2, ..., which calls m_n at pc_n, and that in method m_n, at pc, an object o of type τ is allocated, and stored in variable v. Then, if $\phi(\top[\tau_{-1}], ..., \top[\tau_j]) < \top[\tau]$ where*

$$\mathbb{L}_1(m_n)(v) \circ \mathbb{L}_{1C}(pc_n) \circ ... \circ \mathbb{L}_{1C}(pc_1) = (\phi, (\tau_{-1}, ..., \tau_j), \tau)$$

the object o can be stack allocated if we inline m_1, ..., m_n in m_0.

This theorem is proved in Appendix E.3. Moreover, we wish to inline a call to method m' only if that allows more stack allocations, i.e. if the stack allocation criterion is true for some allocations in m' or in methods that m' calls, when inlining the call to m', but is wrong without the inlining of the call to m'.

Representing the stack allocation condition

We look for the condition on c_{-1}, ..., c_j such that: $\phi(c_{-1}, ..., c_j) < \top[\tau]$. As analysis $\mathbb{L}_1$ is additive (since only additive functions can be represented by the general form of Section 7.6), this is: $g_{-1}(c_{-1}) \sqcup ... \sqcup g_j(c_j) < \top[\tau]$ where g_i are monotone functions $\mathbb{N} \to \mathbb{N}$. That is $g_{-1}(c_{-1}) < \top[\tau] \wedge ... \wedge g_j(c_j) < \top[\tau]$, i.e. $c_{-1} \le c'_{-1}$, ..., $c_j \le c'_j$, where c'_i is the greatest integer such that $g_i(c'_i) < \top[\tau]$. As all c_i are zero or positive, this defines a right-angled parallelepiped in $\mathbb{R}^{j+2}$ whose sides are contained by the coordinate axes (see Figure 8.1) which is called a $j+2$-cell (this parallelepiped may be empty or infinitely long in certain directions, if $\forall c_i, g_i(c_i) < \top[\tau]$). The set of solutions is represented in the implementation by the null pointer for the empty set or an array of $j+2$ integers corresponding to the sides of the $j+2$-cell.

Practically, the sides of the $j+2$-cell can easily be computed. For the first step, we compute the condition on c_{-1}, ..., c_j such that $\phi(c_{-1}, ..., c_j) < \top[\tau]$ where $\mathbb{L}_1(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau)$. The set of solutions is called $\mathbb{L}_1(m)(v)^{-1}(\top[\tau])$. Knowing that

$$\phi(c_{-1}, ..., c_j) = i \sqcup \sqcup_k (\text{if } c_k \ge s_k \text{ then } i_k^+ \text{ else } 0) \sqcup (c_k \sqcap f_k),$$

we obtain: if $i \ge \top[\tau]$, the $j+2$-cell is empty; otherwise, the condition is (if $c_k \ge s_k$ then i_k^+ else 0) $\sqcup (c_k \sqcap f_k) < \top[\tau]$, i.e. if $f_k \ge \top[\tau]$, $c_k < \top[\tau]$ and if $i_k^+ \ge \top[\tau]$, $c_k < s_k$. Therefore, this is equivalent to $c_k < c'_k$ with $c'_k = \min\{\top[\tau] \text{ if } f_k \ge \top[\tau], s_k \text{ if } i_k^+ \ge \top[\tau]\}$.

For the following steps, we have to solve $(\phi_{-1}, ..., \phi_{j'})(c_{-1}, ..., c_j) \in S$ where S is the $j'+2$-cell computed at the preceding step, $\mathbb{L}_{1C}(pc) = (v_{-1}, ..., v_{j'})$ and $v_k = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau'_k)$. The set of solutions is called $\mathbb{L}_{1C}(pc)^{-1}(S)$. If S is not empty, we look for solutions of $(\phi_{-1}, ..., \phi_{j'})(c_{-1}, ..., c_j) < (c'_{-1}, ..., c'_{j'})$ where $(c'_{-1}, ..., c'_{j'})$ are the sides of S. This can be decomposed into the solving of $j'+2$ equations $\phi_k(c_{-1}, ..., c_j) < c'_k$, $k \in [\![-1, j']\!]$ which can be solved as above. The only remaining task is to compute the intersection of the sets of solutions of these $j'+2$ equations, which corresponds to taking the minimum of the sides of the $j+2$-cells solutions of these $j'+2$ equations.

```
class Memo {
  Vector memoVect;

  T memoT(int n, T[] memoArray) {
    if (memoArray.length > n) {
      if (memoArray[n] != null)
        return memoArray[n];
      T t1 = new T(n);
      memoArray[n] = t1;
      return t1;
    } else {
      T t = find(n);
      if (t != null) return t;
      T t2 = new T(n);
      memoVect.addElement(t2);
      return t2;
    }
  }
}
```

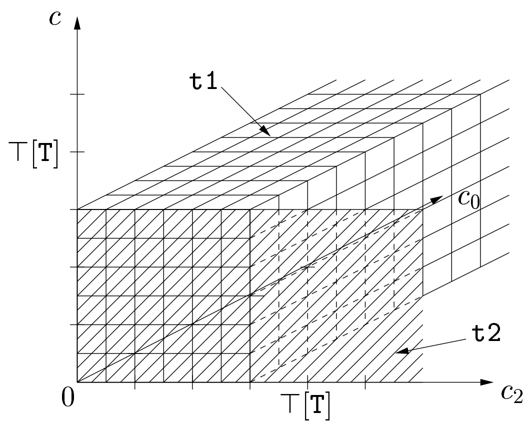


Figure 8.1: Example of inlining condition. The 3-cell of depth 0 corresponds to the stack allocation condition of `t2`, the other one of `t1`.

Remark. When there are no type conversions, that is when functions have no step terms, if the set of solutions is not empty, its sides are either the height of the allocated type or ∞.

Moreover, in practice, in the vast majority of cases, each stack allocation only depends on a single parameter (because an object is often stored in at most one other object). Even different objects allocated in the same method often depend on the same parameter. In this case, deciding inlining is easy, but our algorithm can handle all cases exactly.

Example 8.3 Consider the method from Figure 8.1. This method returns the result of `new T(n)`, memoizing it either in the array `memoArray` if it is large enough, or in the vector `memoVect`. Let c be the $\mathbb{L}_1$-escaping part of the result, c_0 and c_2 the $\mathbb{L}_{1S}$-escaping parts of the parameters `this` and `memoArray` respectively. The escaping part of `t1` is $c \sqcup c_2$ and that of `t2` is $c \sqcup (\text{if } c_0 > 0 \text{ then } \top[\mathtt{T}] \text{ else } 0)$. Then `t1` can be stack allocated if and only if $c < \top[\mathtt{T}]$ and $c_2 < \top[\mathtt{T}]$. `t2` can be stack allocated if and only if $c < \top[\mathtt{T}]$ and $c_0 = 0$. This is represented in Figure 8.1.

Consider now the method `display`:

```
static void display(Memo m, int n) {
  System.out.println(m.memoT(n, new T[10]).toString());
}
```

If the parameter of `T.toString()` does not escape, then `memoT` can be inlined to stack allocate `t1`. `t2` may be stack allocated in a caller of `display`, if $c_1 = 0$, where c_1 is the escape context of the parameter `Memo m` of `display`.

The algorithm

The above representation enables us to easily build the conditions such that a given chain of function inlining should be done. A first idea of algorithm would be to store for each method m' the list of all interesting chains of inlining. This leads to a memory space quadratic in the size of the program in the worst case (each method contains a copy of all stack allocation opportunities in all the methods it may call), which in the case of very large projects may lead to memory overflow.

So we have to keep only a summary of all stack allocation opportunities in the methods called from m. However, allocation conditions may be represented by complex sets and we cannot guarantee that they can be represented in a small enough space. To solve this problem, we use heuristics. When these heuristics do not give the answer to the question "should we inline ?", we make a complete and exact computation, at the cost of speed. In practice, this case will be rare enough for the system to be very efficient.

We store for each method m the following information:

- m.inter_cond is the $j+2$-cell such that all stack allocations can be done if the entry escaping parts are in it. (The entry escaping parts are the escaping parts of the parameters and the result of m, coming from the caller of m — $\mathbb{E}_{\mathrm{S}}$ for the parameters, $\mathbb{E}$ for the result.)

- m.maxvol_cond is a $j+2$-cell which corresponds to a stack allocation which cannot be done if we do not inline m, and of the greatest possible volume (so that it has the greatest probability to be realized if we inline m). The entry escaping parts of m are in m.maxvol_cond if and only if this stack allocation becomes possible if we inline m. The volume of a $j+2$-cell p will be denoted by $\mathrm{vol}(p)$.

- m.englob_cond is a $j+2$-cell which contains all $j+2$-cells corresponding to possible stack allocations that we are not sure to do without inlining m. If p_1 and p_2 are two $j+2$-cells, we call $\mathrm{contains}(p_1, p_2)$ the smallest $j+2$-cell which contains p_1 and p_2 (it can be computed by taking the maximum of the sides of p_1 and p_2).

Let us define ϕ_k and $\tau'_{k'}$ by $\mathbb{L}_{1\mathrm{C}}(pc_i) \circ ... \circ \mathbb{L}_{1\mathrm{C}}(pc_1) = (v_{-1}, ..., v_j)$ and $v_k = (\phi_k, (\tau'_{-1}, ..., \tau'_{j'}), \tau_k)$. Let $C = (\phi_{-1}, ..., \phi_j)(\top[\tau'_{-1}], ..., \top[\tau'_{j'}])$ be the entry escaping parts of method $m = m_i$. If $C \in m$.maxvol_cond, we know that we should inline the call to m. If $C \notin m$.englob_cond, we know that it is useless to inline m. When $C \in m$.englob_cond $- m$.maxvol_cond, we still do not have the answer, and we have to compute it by looking at all possible stack allocations.

Those $j+2$-cells can be computed by the algorithm of Figure 8.2 (the current method is m, its parameters are of types $\tau_0, ..., \tau_j$, its result is of type τ_{-1}; the parameters of m' are of types $\tau'_0, ..., \tau'_{j'}$, its result is of type τ'_{-1}).

Example 8.4 Let us consider again Example 8.3. `memoT`.englob_cond is the 3-cell $c < \top[\mathtt{T}]$. `memoT`.maxvol_cond is $c < \top[\mathtt{T}], c_2 < \top[\mathtt{T}]$ (this is the stack allocation condition of `t2`). `memoT`.inter_cond is $c < \top[\mathtt{T}], c_0 = 0, c_2 < \top[\mathtt{T}]$.

The entry context of `memoT` when analyzing `display` will be $c = 0, c_0 = \top[\mathtt{Vector}], c_2 = 0$, so our algorithm correctly detects that `memoT` should be inlined, since this point is in `memoT`.maxvol_cond.

Initialization: m.englob_cond $= m$.maxvol_cond $= \emptyset$, m.inter_cond $= \mathbb{N}^{j+2}$.

AddCond(pc, $englob\_cond$, $maxvol\_cond$, $inter\_cond$) =
  if $(0,...,0) \in englob\_cond$ **then**
    m.inter_cond $= m$.inter_cond $\cap$ $inter\_cond$
    if $(\top[\tau_{-1}],...,\top[\tau_j]) \notin maxvol\_cond$
    and vol($maxvol\_cond$) $>$ vol(m.maxvol_cond) **then**
      m.maxvol_cond $= maxvol\_cond$
    m.englob_cond $=$ contains(m.englob_cond, $englob\_cond$)
    Add (pc, $englob\_cond$, $maxvol\_cond$, $inter\_cond$) to m.inline_info

foreach call to a method m' at pc,
  if $(0,...,0) \in m'$.englob_cond
  and $(\top[\tau'_{-1}],...,\top[\tau'_{j'}]) \notin m'$.inter_cond **then**
    AddCond(pc,$\mathbb{L}_{1C}(pc)^{-1}$(m'.englob_cond),
      $\mathbb{L}_{1C}(pc)^{-1}$(m'.maxvol_cond),
      $\mathbb{L}_{1C}(pc)^{-1}$(m'.inter_cond))
foreach $v =$ `new` C/`new` $t[w]$ allocating an object of type τ in m,
  $cond = \mathbb{L}_1(m)(v)^{-1}(\top[\tau])$
  AddCond(pc, $cond$, $cond$, $cond$)

Figure 8.2: Computing inlining conditions

8.3.4 Synchronization elimination

A method only needs to be synchronized if the object on which it is called can be accessed by several threads. Objects accessible from several threads are objects reachable from static fields and from `Thread` objects. Objects reachable from static fields are considered as escaping by our analysis, and the thread creation method `Thread.start()` makes its parameter escape. Therefore, if our analysis finds that an object does not escape from a method m, it can be accessed only by one thread, the thread in which it is allocated. So there is no need to synchronize methods called on this object.[1] This is an important optimization since synchronization is a costly operation in the JDK. We perform two transformations.

Global synchronization analysis

Assume that a synchronized method m' is called from method m. If for all calls to m, we can prove that the object on which we call m' will be local to one thread, then the synchronization can be removed.

More precisely, we define $\mathbb{L}_{1\text{sync}}(m) \in Ctx_{\mathbb{L}}^{j+2}$ where method m has $j+1$ parameters. $\mathbb{L}_{1\text{sync}}(m)$ indicates for each parameter and for the result of m the part that may be shared by several threads. $\mathbb{L}_{1\text{sync}}$ is defined by the equations:

1. Assume that method m has $j+1$ parameters of types $\tau_0,...,\tau_j$ and a result of type τ_{-1}. Also assume that, at pc, m can call method m' which has j' parameters of types

[1] On some architectures, there is a memory barrier at each synchronization. We have to leave this barrier even if we remove the locking or unlocking operation.

$\tau'_0, ..., \tau'_{j'}$ and a result of type τ'_{-1}. Let $\mathbb{L}_{1C}(pc) = (v_{-1}, ..., v_{j'})$ with $v_k = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau'_k)$. In this case, $\mathbb{L}_{1\text{sync}}(m') \geq (\phi_{-1}, ..., \phi_{j'})(\mathbb{L}_{1\text{sync}}(m))$.

2. If $\rho_S(m) = (\phi_{-1}, (\tau_{-1}, ..., \tau_j), \tau_{-1})$ and $\rho(m)(k) = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau_k)$, $\mathbb{L}_{1\text{sync}}(m) \geq (\phi_{-1}, ..., \phi_j)(\mathbb{L}_{1\text{sync}}(m))$.

3. $\mathbb{L}_{1\text{sync}}(\texttt{void Thread.start()}) = (0, \top[\texttt{Thread}])$.

The value of $\mathbb{L}_{1\text{sync}}$ is computed by fixed point iteration of these equations on the whole program. Notice that we need to have enough information on the call graph (which methods may be called from which program point) to be able to perform this analysis. Therefore, we assume the worst when we are not able to determine which classes will be loaded at runtime, or when analyzing a library: in this case, $\mathbb{L}_{1\text{sync}}(m) = (\top[\tau_{-1}], ..., \top[\tau_j])$ where the parameters of m have type $\tau_0, ..., \tau_j$ and its result τ_{-1}.

We have the following result:

Theorem 8.5 *Assume that in method m, a synchronized method m' is called: $w = v_0.m'(v_1, ..., v_j)$. Let $\mathbb{L}_{1S}(m)(v_0) = (\phi, (\tau_{-1}, ..., \tau_j), \tau'_0)$. If $\phi(\mathbb{L}_{1\text{sync}}(m)) < \top[\tau'_0]$, then the object in v_0, on which m' is called, is thread local, and the synchronization can be eliminated.*

This theorem is proved in Appendix E.4. When the conditions of the above theorem are satisfied, we call a copy of m' without synchronization.

Example 8.6 Consider the following code:

```
class Test {
   Object elem;

   Vector createVector() {
     Vector v = new Vector();
     v.addElement(this.elem);
     return v;
   }
   void displayVector(Vector v) {
     for (int i = 0; i < v.size(); i++) {
       System.out.println(v.elementAt(i));
     }
   }
   void run() {
     Vector v = createVector();
     displayVector(v);
   }
}
static Test t = new Test();
static void main(String[] args) { t.run(); }
```

The `Vector v` cannot be stack allocated (except if inlining `createVector`, but we will assume that `createVector` is too large to be inlined). However, it is local to its allocating thread, because it is never stored in static fields or `Thread` objects. Only its elements, which are reachable from the static field `t` may be shared between several threads:

$$\mathbb{L}_{1S}(\texttt{run})(\texttt{this}) = (\{(c_0) \mapsto c_0\}, (\texttt{Test}), \texttt{Test})$$
$$\mathbb{L}_{1S}(\texttt{run})(\texttt{v}) = (\{(c_0) \mapsto c_0 \sqcap 1\}, (\texttt{Test}), \texttt{Vector})$$
$$\mathbb{L}_1(\texttt{createVector})(\texttt{this}) = (\{(c_{-1}, c_0) \mapsto (c_{-1} \sqcap 1) \sqcup (c_0 \sqcap 1), (\texttt{Vector}, \texttt{Test}), \texttt{Test})$$
$$\mathbb{L}_{1S}(\texttt{createVector})(\texttt{v}) = (\{(c_{-1}, c_0) \mapsto c_{-1} \sqcup (c_0 \sqcap 1), (\texttt{Vector}, \texttt{Test}), \texttt{Vector})$$
$$\mathbb{L}_1(\texttt{displayVector})(\texttt{this}) = (\{(c_0, c_1) \mapsto 0\}, (\texttt{Test}, \texttt{Vector}), \texttt{Test})$$
$$\mathbb{L}_1(\texttt{displayVector})(\texttt{v}) = (\{(c_0, c_1) \mapsto c_1 \sqcap 1\}, (\texttt{Test}, \texttt{Vector}), \texttt{Vector})$$

then we can compute

$$\mathbb{L}_{1\text{sync}}(\texttt{createVector}) = (1, \top[\texttt{Test}]), \quad \mathbb{L}_{1\text{sync}}(\texttt{displayVector}) = (\top[\texttt{Test}], 1).$$

The synchronizations of methods `addElement` and `elementAt` can therefore be removed. A copy of `addElement` and `elementAt` without synchronization is created, and this copy is called instead of the original method.

No synchronization on stack allocated objects

Before acquiring a lock on an object o, we test at runtime whether o is on the stack, and if it is, we skip the synchronization. Contrary to the previous analysis, this is a local analysis: it can be performed when only part of the program is known. Stack allocation in method m can be performed as soon as we know all the code that can be called from m.

Example 8.7 Consider for example:

```
void m() {
  DataInputStream f = new DataInputStream(new BufferedInputStream(
     new FileInputStream("Foo.class")));
  System.out.println(f.readInt());
  f.close();
}
```

In this program, the `DataInputStream` and the `BufferedInputStream` can be stack allocated. `f.readInt()` calls `BufferedInputStream.read()` which is synchronized. The dynamic test detects that the `BufferedInputStream` is stack allocated, and skips the synchronization.

This synchronization cannot be removed by the global synchronization analysis because the call to `BufferedInputStream.read()` is a virtual call. Therefore removing the synchronization would require specializing all `read` methods that may be called. Another reason is that, elsewhere in the program, there may exist a `DataInputStream` whose underlying input stream is shared between several threads.

This technique could be extended by marking at allocation time objects that are detected to be local to one thread (by the global analysis above), and testing this mark before acquiring a lock, as in [CGS+99]. However this requires at least one free bit in the representation of each object.

Another way of performing this is by defining a subclass C_{nosync} for each class C which has at least one synchronized method, as in [BH99]. The class C_{nosync} defines the same methods

Size of the SSA form	n
Number of parameters of a method	p
Number of parameters of a context	p'
Maximum type height	H

Equations building	$\mathcal{O}(npp')$
Number of equations	$n_e = \mathcal{O}(n)$
Number of unknowns	$n_u = \mathcal{O}(n)$
Number of iterations	$n_i = \mathcal{O}(n_u p' H) = \mathcal{O}(np'H)$
Equations solving	$\mathcal{O}(n_e pp' n_i) = \mathcal{O}(npp' n_i)$
Posttransformation	$\mathcal{O}(np)$

Figure 8.3: Notations and results on the complexity of escape analysis

as C but without synchronization. When allocating an object of class C detected as thread local, we in fact allocate an object of class C_{nosync}. This way, all synchronizations on the object are also eliminated. This replaces the runtime check of the preceding methods with a dynamic dispatch (which may be more costly, when there was no dynamic dispatch before synchronization elimination).

The idea behind these three techniques is that sometimes, at the call site, the object may be thread local or not depending on what happens at runtime. Therefore, the synchronizations cannot be optimized by the global synchronization analysis. But it may happen that at the allocation site, we can prove that the allocated object is always thread local. The synchronizations may then be eliminated by one of these transformations.

Combining both the global synchronization analysis and the elimination of synchronizations on stack allocated objects is needed to get the best results.

Remark. Event notication methods `Object.wait`, `Object.notify` and `Object.notifyAll` require that their argument is locked. In our analysis, we consider that their argument always escapes, therefore we never eliminate synchronizations on objects that are passed to these methods, thus preserving a correct behaviour in this case.

8.4 Complexity

Notations are summarized in Figure 8.3. The size of each instruction is 1, except for $v_0.m(v_1, ..., v_p)$ which has size p. When out-of-loop stack allocation is activated, context transformers may have more parameters than methods, hence the distinction between p' and p. If out-of-loop stack allocation is disabled, $p' = p$.

The analysis time of a bytecode depends on the considered bytecode: the analysis of a method call ($v_0.m(v_1, ..., v_p)$) takes $\mathcal{O}(pp')$ which is $\mathcal{O}(p')$ times its size; the analysis of an upper bound, conversion or minimum on contexts takes $\mathcal{O}(p')$. The total complexity of equation building is therefore $\mathcal{O}(np')$. The number of generated equations or unknowns is $\mathcal{O}(n)$ (each operation generates a constant number of equations, except the method call which generates as many equations as there are parameters).

If the system contains n_u unknowns, the number of iterations n_i is less than the height of the lattice of n_u-tuples of context transformers, so $n_i = \mathcal{O}(n_u p' H)$. In practice, the number of iterations is very small. (It is at most 17 in our benchmarks. 44.8% of equations are iterated at most twice, 91.3% at most 7 times, 98.4% at most 10 times. On average, each equation is iterated 3.9 times.) This is possible thanks to the splitting of equations systems into strongly connected components. On the other hand, one iteration costs the compute time of equations, which is at most $\mathcal{O}(n_e pp')$. The complexity of the solving is therefore $\mathcal{O}(n_e pp' n_i)$. This complexity, which sounds large, is much reduced in practice by using a sparse representation for context transformers, which ensures that the number of parameters of a context is in fact small compared to p in most cases.

The post-treatment of a method call (when inlining is activated) or of an allocation is in $\mathcal{O}(p)$ if we can decide to inline or not with our fast algorithm. In this case, the post-treatment is dominated by the generation of equations. If our fast algorithm does not give the answer, the post-treatment of a call may take time up to $\mathcal{O}(n'p)$ where n' is the number of allocations in all methods called from the program point, following any call chain. This case is very rare and can be neglected (it happens only 21 times in all the tested benchmarks, which represent more than 2 Mb of classes).

Finally, the complexity is therefore $\mathcal{O}(npp'n_i) = \mathcal{O}(n^2 pp'^2 H)$, with n_i small in practice.

Chapter 9

Benchmarks

9.1 Benchmark programs

Benchmarks have been conducted on a 233 MHz Pentium MMX, 128 Mb RAM, primary cache 16 kb instructions and 16 kb data, secondary cache 512 kb instructions and data, under Linux 2.0.32 (RedHat 5.0). Benchmark programs are listed in Table 9.1.

Benchmark programs		Size(kb)
dhry	Dhrystone	73
Symantec	Set of small benchmarks	76
javac	Java compiler (JDK 1.1.5) compiling jBYTEmark	242
turboJ	Java to C compiler from Silicomp RI compiling jBYTEmark	311
JLex	Lexer generator (v. 1.2.3) running `sample.lex`	97
jglSort	Sorting benchmarks of JGL 3.1.0	95
jess	Expert system (v. 4.1), solving `fullmab.clp`	363
javacc	Java parser generator (v. 0.8pre2) generating a Java parser	254

Table 9.1: Benchmark programs. The size is the total size of bytecode, including the useful part of the libraries. The considered size contains only the bytecode of the useful methods itself, not the constant pool and other structures of the class files.

9.2 Results

Each program has been tested in two stack allocation configurations: in the first configuration called "No loops" all stack allocations done in a loop must reuse the same memory space. In the second configuration "All", `alloca` is allowed even in a loop. The last configuration may stack allocate more objects, but may lead to stack overflows. In the following benchmarks, inlining is allowed for methods of less than 150 bytes.

Table 9.2 gives the percentage of stack allocated data in each program. The left table deals with stack allocated size. Its last column represents the total size of allocated data in the program. The right table is similar but deals with numbers of objects instead of size. Table 9.3 gives the percentage of eliminated synchronizations at runtime. The left column represents the part of eliminated synchronizations that we would get without testing at runtime whether objects are in the stack or not. The middle column gives the part of

Memory size decrease (%)			Size
`alloca`	No loops	All	(Mb)
dhry	95	95	5.1
Symantec	84	84	40.2
javac	13	13	6.5
turboJ	27	29	23.2
JLex	25	26	1.7
jglSort	18	66	18.2
jess	21	21	1.9
javacc	20	43	7.5

Stack allocated objects (%)			Total number
`alloca`	No loops	All	of objects (K)
dhry	99	99	600
Symantec	99	99	1199
javac	19	19	169
turboJ	36	36	468
JLex	29	30	39
jglSort	28	71	508
jess	27	27	43
javacc	14	18	187

Table 9.2: Stack allocation

Eliminated synchro. (%)			Thousands
	(1)	(2)	of synchro.
dhry	–	–	0
Symantec	99	99	2520
javac	5	31	430
turboJ	21	46	587
JLex	78	94	1989
jglSort	28	71	9200
jess	13	21	158
javacc	2	3	1075

(1) Eliminated synchronizations without dynamic “is in stack” test.
(2) Eliminated synchronizations with dynamic “is in stack” test.

Table 9.3: Eliminated synchronizations

eliminated synchronizations with this dynamic test. The right column is the total number of synchronizations (without elimination) at runtime. Figure 9.1 also represents the percentage of allocated size decrease (black bar), and the percentage of eliminated synchronizations. The red bar corresponds to global synchronization analysis only (without dynamic tests), and the blue bar corresponds to the improvement observed thanks to dynamic tests.

In Dhrystone, we mainly stack allocate arrays of one integer, which are used as references to integers. Symantec is a set of small benchmarks which in particular, contains a version of Dhrystone, in which we do the same optimizations. It also contains a function which uselessly allocates many nodes of a tree, which can be stack allocated. We also manage to eliminate all synchronizations corresponding to synchronized calls to a random number generator.

In turboJ and javacc, we mainly stack allocate strings and arrays of characters[1]. Our inlining algorithm is useful to allocate such data on the stack, since they are often allocated in `StringBuffer.ensureCapacity` and `StringBuffer.toString`, and of course still live at

[1]TurboJ detects string concatenation sequences, and replaces them by an optimized code that computes the length of the final string and directly allocates an array of characters of the required size [W+98]. This is much more efficient than the code generated by javac which uses `StringBuffer.append`. This also reduces the amount of stack allocated data in our benchmarks because the `StringBuffer`s and the intermediate arrays of characters would be stack allocated. Our analyzer takes the bytecode with `StringBuffer`s as input, before transformation by TurboJ.

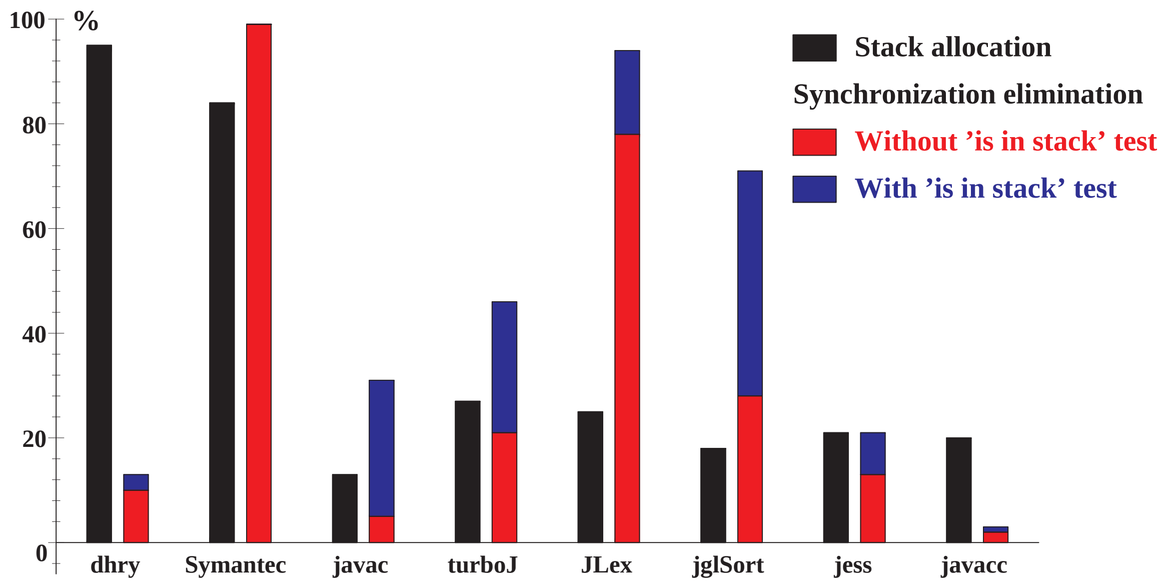


Figure 9.1: Stack allocation and synchronization elimination without `alloca` in loops.

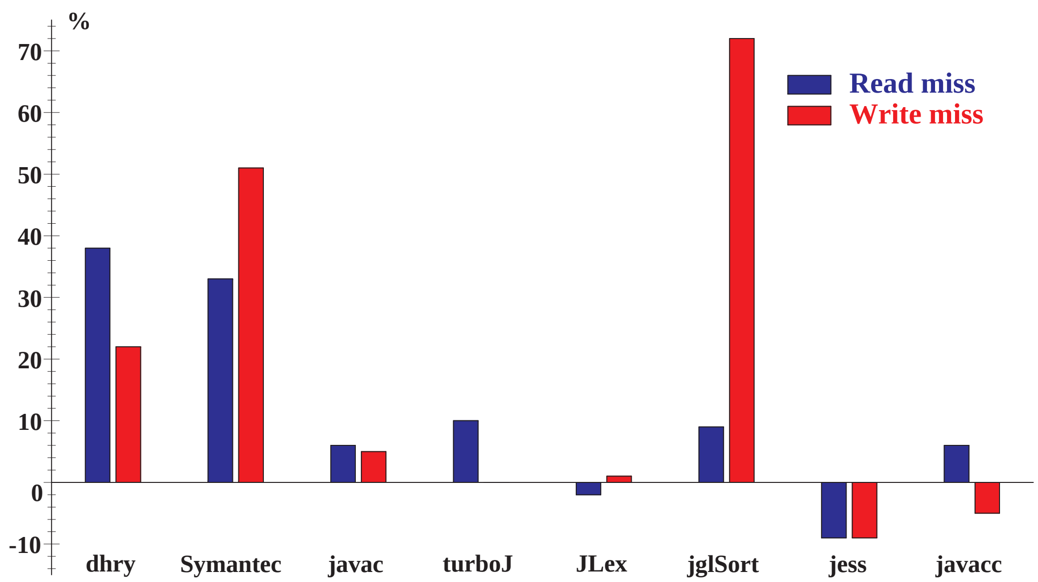


Figure 9.2: Decrease of the number of cache misses without `alloca` in loops.

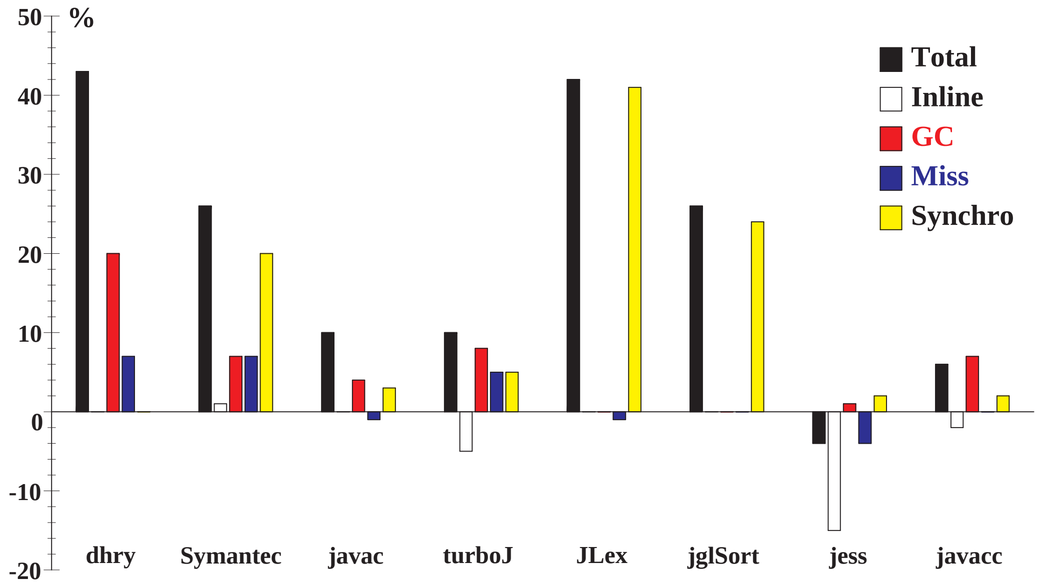


Figure 9.3: Runtime decrease without `alloca` in loops.

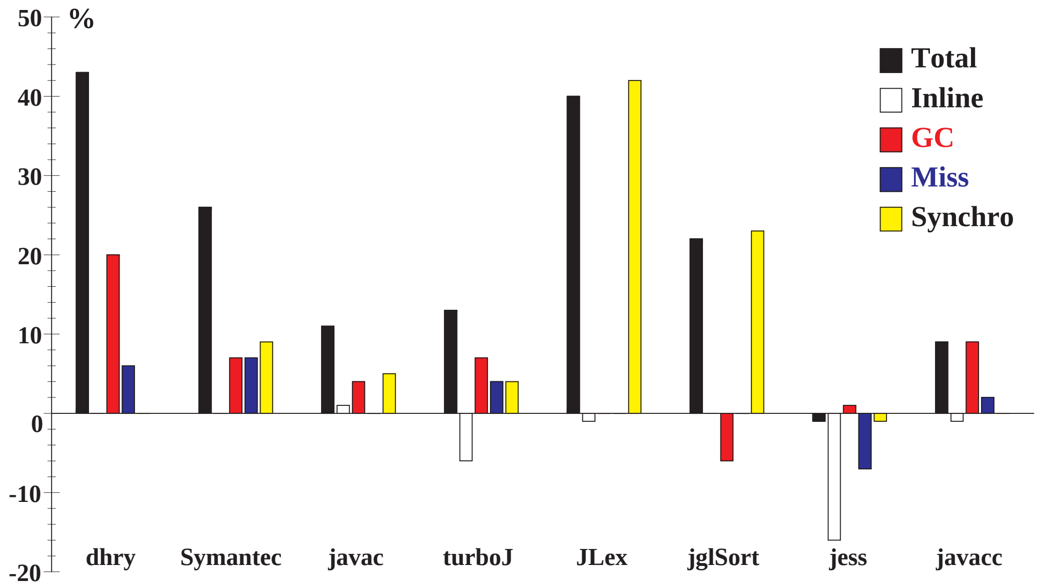


Figure 9.4: Runtime decrease with all possible stack allocations, even in loops

Runtime decrease (%)			Time
alloca	No loops	All	(ms)
dhry	43	43	2710
Symantec	26	26	19327
javac	10	11	4109
turboJ	10	13	19678
JLex	42	40	3367
jglSort	26	22	20344
jess	-4	-1	3643
javacc	6	9	7695
mean	22	22	

Table 9.4: Runtime decrease

GC time decrease (%)			GC time	Runtime decrease due to the GC (%)			Time
alloca	No loops	All	(ms)	alloca	No loops	All	(ms)
dhry	100	100	567	dhry	20	20	2716
Symantec	97	97	1543	Symantec	7	7	19209
javac	100	100	197	javac	4	4	4014
turboJ	20	17	8511	turboJ	8	7	19579
JLex	-1	1	55	JLex	-0	0	3341
jglSort	3	-88	1439	jglSort	0	-6	20444
jess	28	29	203	jess	1	1	3537
javacc	31	37	1852	javacc	7	9	7607

Table 9.5: Effect on the GC

the end of these methods (char arrays stack allocated thanks to inlining represent 4% of data for javac, 20% for turboJ, 29% for javacc). The difference between the “No loops” and the “All” configurations is only important for jglSort and javacc.

Table 9.4 indicates the percentage of runtime decrease, computed by the formula $100 \times (T_{old} - T_{new})/T_{old}$ where T_{old} is the runtime without optimization, and T_{new} the runtime with optimization. The last column is the runtime without any optimization, for reference. The mean in the bottom row is the geometric mean. Figures 9.3 and 9.4 also give the percentage of runtime decrease (Total bar), and they detail the reasons of the speedup. The GC bar gives the part of the runtime decrease that comes from a decrease of GC time. The Inline bar gives the part that comes from inlining of methods (inlining is done as if there were stack allocation, but all allocations are done in the heap). The Miss bar gives the contribution of the improvement on data locality to the speedup. This is based on the number of cycles during which the processor is stalled waiting for a read from memory, or because the write buffer is full, measured experimentally. The Synchro bar gives the part of the runtime decrease that comes from synchronization elimination. A synchronization costs 790 ns when the monitor is in the monitor cache, and 1450 ns when it is not.

The speedups correspond to what could be expected knowing the percentage of stack allocated data and of synchronization elimination. We get high speedups for dhry (thanks to stack allocation), for JLex (thanks to synchronization elimination), for Symantec and jglSort. The runtime decreases by about 10% for javac, turboJ and javacc. Inlining has negative effects for turboJ and jess. This may come from an increase of the code size, which leads to more traffic for loading the code from memory. Inlining also leads to changes in register allocation which may affect performance. This negative effect of inlining explains the slowdown that we obtain for jess. For JLex, stack allocating data gives negligible speedups, because the GC time was very small anyway.

There are several different kinds of GC in the various Java implementations. In the version of Java we have used (JDK 1.1.5 on Linux) and in Sun JDK until version 1.2, the GC is a ***mark and sweep collector***. (Notice that turboJ uses the runtime of a JVM. TurboJ is described in more detail page 171.) This GC works in two phases: the first phase is the marking phase. The GC follows pointers from the roots (registers of the CPU and all values in call frames, even integers if their value could be the value of a pointer — this GC is conservative for the stack), and marks the values it can reach by changing a bit in their header. The second phase is sweeping: the whole heap is scanned. Unmarked values are added to a list of free blocks, and marked values are left unchanged (except that their mark is removed). The GC is exact for the heap: it knows the type and hence the structure of heap allocated objects. It can therefore follow only pointers in the heap.

The speedups that come from stack allocation mainly have three causes: reduced allocation time, better data locality, decrease of the GC workload. The graphs show that the part of the speedup coming from the GC is important (about half of the speedup not coming from synchronization elimination). This is logical since the mark and sweep garbage collector of the JDK is not efficient for short lived data, which can precisely be stack allocated. Indeed, even data which are already dead at the first GC have to be scanned during the sweeping phase, which has an important cost. The better data locality is responsible for about a quarter of the speedup. The rest, i.e. about a quarter comes from allocation time. Stack allocating an object just requires moving the stack pointer, whereas heap allocating it requires scanning the list of free blocks, and acquiring a lock, which is much less efficient.

Comparison with ML In Ocaml, the speedups mainly come from a better data locality, because the GC is already very efficient for short lived data. It has two generations, and uses a copy GC for the young generation. With such a GC, data which are already unreachable at the time of the GC are not scanned and have no cost. The allocation is also very fast (just moving the allocation pointer).

For javacc, the speedup is more important in the “All” configuration, in which more allocations can be done in the stack. In the “All” configuration, the stack size increases by a factor 10 for javacc and by a factor 1000 for jglSort (see Table 9.6), whereas in the “No loops” configuration the increase is limited to only 77% for javacc and a factor 13 for jglSort. This shows that it may be useful to limit ourselves to the “No loops” configuration to avoid stack overflows. (For jglSort, we had to increase the maximum stack size, otherwise a stack overflow happens in the “All” configuration.) The stack size increase is 129% for JLex in the “All” configuration. It is at most 68% in all other cases.

The left part of Table 9.5 gives the percentage of decrease of the GC time, and in the last column the GC time for reference. In general, the decrease of the GC time is similar to the percentage of stack allocated data. javac does a single GC without stack allocation, and none with stack allocation, so its GC time becomes null. The GC time is more reduced than what we could expect for jess, and less for turboJ and JLex. This comes from the fact that stack allocated data are mainly short lived data which are scanned only once by the GC, and

Stack size increase (%)			Size
`alloca`	No loops	All	(bytes)
dhry	0	0	4960
Symantec	1	2	6392
javac	62	68	10336
turboJ	24	29	17468
JLex	-12	129	6536
jglSort	1235	106832	9636
jess	10	10	9884
javacc	77	1023	12188

Table 9.6: Stack size increase

	Read miss			Write miss		
	Decrease (%)		Thousands	Decrease (%)		Thousands
`alloca`	No loops	All	of misses	No loops	All	of misses
dhry	38	40	2905	22	20	3228
Symantec	33	33	15011	51	50	11241
javac	6	8	6448	5	6	8139
turboJ	10	15	27212	0	5	13284
JLex	-2	-3	2161	1	-7	2485
jglSort	9	2	14232	72	35	11799
jess	-9	-8	6931	-9	-21	4886
javacc	6	13	10856	-5	-14	5706

Table 9.7: Effects on the cache.

so have a smaller cost for the GC than data which cannot be stack allocated (they may be scanned in several GC cycles if they remain alive). Data allocated in turboJ and JLex may live longer than in the other benchmarks. The GC time is increased for jglSort in the “All” configuration. This comes from the very large stack size increase: the GC has to scan the stack. Since the stack allocated values are mainly arrays of characters, the GC does not scan them when they are allocated in the heap (it knows that they contain no pointers).

Table 9.7 indicates the percentage of decrease of the number of L1 read and write misses due to our optimizations, and the number of misses without optimization for reference. These values are represented in Figure 9.2. They show that stack allocation reduces the number of read misses (except for jess and JLex). Stack allocation improves data locality because the top of the stack, which is the most useful part, is very often in the cache. However, in some cases, stack allocation may also have a negative effect on data locality. For example, in javacc, when `alloca` is allowed even in a loop, arrays of characters are stack allocated inside loops. These arrays of characters become in fact unreachable at the next iteration of the loop, but our analysis cannot detect this fact. Therefore, there are gaps in the stack which will not be reused until the end of the method. On the contrary, when these arrays of characters are allocated in the heap, they may be reclaimed earlier by a GC and reused. This negative effect on data locality explains the increase of the number of misses that can be noticed for several programs. This also explains that the number of misses is greater in the “All” configuration

	Number of data read stalls			Runtime decrease due to decrease of read stalls (%)		Time (ms)
	Decrease (%)		Millions			
`alloca`	No loops	All		No loops	All	
dhry	19	17	212	6	5	2733
Symantec	38	39	835	7	7	19441
javac	-4	0	482	-1	0	4216
turboJ	16	14	1617	5	4	19831
JLex	-6	1	175	-1	0	3376
jglSort	4	2	760	0	0	20439
jess	-7	-11	558	-4	-7	3743
javacc	0	7	637	0	2	7842

	Number of data write stalls			Runtime decrease due to decrease of write stalls (%)		Time (ms)
	Decrease (%)		Millions			
`alloca`	No loops	All		No loops	All	
dhry	39	38	19	1	1	2733
Symantec	32	32	93	0	0	19441
javac	4	5	57	0	0	4216
turboJ	11	8	98	0	0	19831
JLex	-2	0	16	0	0	3376
jglSort	4	-87	37	0	0	20439
jess	-6	-6	28	0	0	3743
javacc	0	0	45	0	0	7842

Table 9.8: Time effect of cache misses

than in the "No loops" configuration for jglSort.

Table 9.8 enables us to translate the decrease of the number of cache misses into decrease of the runtime. The first part of this table deals with the cost of read misses. Columns 2 to 4 correspond to the decrease of the number of cycles during which the processor is stalled, waiting for a read from memory. Column 5 mentions this number of cycles without stack allocation for reference. Columns 6 to 8 translate the results into a percentage of runtime decrease, due to a decrease of read stalls. In most cases, the time spent because of read misses decreases thanks to stack allocation, and this has an important effect on the whole runtime of the program.

The cost of write misses is small because data can be written directly into the write buffer without accessing memory. The pipeline is only stalled due to a write miss when the write buffer is full. The second part of Table 9.8 is similar to the first one, but it deals with the number of cycles during which the pipeline is stalled because of full write buffer. We can notice that this time decreases in most cases, but it is too small to have a real effect on the execution time of the program.

The previous tables dealt with the execution time of the program itself. The loading time has to be added. The loading time is relatively long since Java loads dynamic libraries. This loading time increases slightly due to our optimizations, since the code size increases because of inlining and synchronization elimination. Table 9.9 details this increase. It also mentions the loading time without optimization in the last column, for reference.

Loading time increase (%)			Time
alloca	No loops	All	(ms)
dhry	9	11	1064
Symantec	10	13	971
javac	3	2	1101
turboJ	-2	-1	1111
JLex	0	7	1029
jglSort	15	9	974
jess	2	0	1039
javacc	4	8	1020

Code size increase (%)		Size (kb)
dhry	0	16
Symantec	7	48
javac	5	1897
turboJ	16	2516
JLex	13	191
jglSort	3	161
jess	14	4470
javacc	10	1956

Table 9.9: Loading time and code size increase due to our optimizations

The increase of the loading time is about 10% which is acceptable since the loading time is in general very short with respect to the total runtime of a program. This increase could be reduced by limiting inlining to smaller methods or by using other techniques to replace inlining (for example, allocating objects on a stack different from the C call stack).

To quantify the code size increase, we have measured the size in kilo-bytes of the dynamic library created by turboJ for each of our benchmark programs. This size without escape analysis, and the increase percentage due to our optimizations (in the "All" configuration) are summarized in Table 9.9.

A code size increase of a bit more than 10% on average is acceptable, but we could still try to reduce it. The increase of the loading time for dhry and Symantec comes from an increase of the size of the standard Java library, not the size of the program code.

We have performed a few complementary benchmarks on a Pentium III 500 MHz, with the JDK 1.1.7v3 to compare with the results on Pentium MMX[2].

Runtime decrease (%)			Time
alloca	No loops	All	(ms)
dhry	42	42	1002
Symantec	24	25	7694
javac	-0	1	967
JLex	42	43	1156
jglSort	25	16	7879
jess	3	3	864
javacc	8	14	2831

Table 9.10: Runtime decrease on Pentium III

The speedups are displayed in Table 9.10. They are very similar to what is obtained on Pentium MMX. We can only notice the decrease of the speedup on javac. This comes from the fact that javac does not perform any GC on Pentium III, since it allocates a few less objects.

The event counters of the Pentium III processor are slightly different from those of the

[2]We had to change the JDK version because the Linux version installed on P.III is newer than on P. MMX. TurboJ has not been tested on Pentium III because its code is confidential.

Runtime decrease due to the GC (%)			Time	GC time decrease (%)			GC time
alloca	No loops	All	(ms)	alloca	No loops	All	(ms)
dhry	18	18	1004	dhry	100	100	187
Symantec	7	7	7703	Symantec	97	97	615
javac	0	0	965	javac	0	0	0
JLex	0	0	1158	JLex	0	3	28
jglSort	0	-11	7872	jglSort	1	-126	693
jess	2	2	827	jess	24	23	100
javacc	3	9	2790	javacc	12	32	828

Table 9.11: Effect on the GC on Pentium III

	L2 reads			L2 writes		
	Decrease (%)		Thousands	Decrease (%)		Thousands
alloca	No loops	All		No loops	All	
dhry	53	54	1533	48	49	53
Symantec	27	27	11942	40	41	400
javac	-0	0	3299	2	2	280
JLex	-1	-1	1037	14	10	69
jglSort	10	2	12399	21	29	313
jess	-0	0	3679	4	3	225
javacc	7	17	9178	3	-1	523

Table 9.12: Effect on the cache, on Pentium III

Pentium MMX: we can measure the number of reads from the L2 cache (which corresponds to the number of L1 read misses), and the number of writes to L2 cache. This does not correspond to the number of write misses. The Pentium III has a write allocate cache: on a write miss, a new cache line is allocated, and marked invalid except for the written location. This way, a write miss does not immediately mean a write to the L2 cache. The L2 cache is only written when the cache line is replaced by another one in the L1 cache. The number of writes to the L2 cache is therefore much smaller than on Pentium MMX, as shown in Table 9.12. The percentage of decrease, on the contrary, is very similar to the one observed on Pentium MMX.

9.3 Analysis speed

We have measured the analysis time and compilation overhead, as a percentage of the total compilation time without stack allocation (Table 9.13). The compilation overhead comes from inlining and synchronization elimination, which lead to more code generation. In these tests, the whole program and the useful part of the standard library have been analyzed and compiled. (In the preceding benchmarks, the library has been analyzed with each benchmark program, but it has been compiled only once for all benchmark programs. However, it does not make sense to compare the analysis time of a large amount of code with the compilation time of a small program.) These tests have been run with TurboJ bootstrapped.

These results show that our analysis takes about 10% of the C compilation time, and the

	An (%)	*Gen* (%)	*Comp* (%)	*Time* (s)
dhry	9	6	2	83.0
Symantec	8	5	3	106.2
javac	11	19	6	671.5
turboJ	11	27	12	907.8
JLex	6	8	9	143.1
jglSort	8	8	10	172.4
jess	9	21	10	1429.2
javacc	9	23	9	538.2

Time is the total compilation time without stack allocation.
An is the escape analysis time, as a percentage of *Time*.
Gen is the C code generation overhead, as a percentage of *Time*. *Gen* includes *An*.
Comp is the C compilation overhead, as a percentage of *Time*.

Table 9.13: Analysis speed

total compilation overhead generated by our optimizations is about 28% (arithmetic mean weighted by the size of the code). The compilation algorithm of turboJ is not linear (in particular, the common subexpression elimination). This is why the compile time increases faster than the code size.

9.4 Conclusion

Our escape analysis for Java yields precise enough results thanks to its precise analysis of assignments. The use of integers yields an efficient analysis (the analysis takes about 10% of the compile time), which is therefore a very good compromise between cost and precision.

Thanks to stack allocation and synchronization elimination, escape analysis gives high speedups for Java programs (geometric mean 22%). The speedups due to stack allocation mainly come from a decrease of the GC time, because the JDK 1.1.5 uses a mark and sweep collector. The other causes of speedups are improvements on data locality and decrease of the allocation time.

The analysis could still be improved to eliminate more synchronizations. One idea to achieve this goal is the following: even if an object is reachable from a static field, it may happen that only one thread accesses this static field, and in this case, the synchronizations can also be removed.

Chapter 10

Related and future work

10.1 Comparison with related work

The work on escape analysis contains two nearly independent parts: the analysis, to infer lifetime and/or alias properties of objects, and the program transformation which can use these properties for various purposes. We first compare the different analyses, and then focus on the program transformations.

10.1.1 Various escape analyses and similar analyses

Figure 10.1 summarizes the different analyses (in chronological order). The analyzed language Fun is a higher-order functional language, Fun1 a first-order functional language, FunLists a functional language with lists as only data type, FunPar a parallel functional language. The "Assignments" column corresponds to the way assignments are handled: N/A (not applicable) when the language does not contain assignments, Imp. (imprecise) when all stored data escape, Pre. (precise) when the analysis is more precise. The "Levels" column indicates the number of distinguished levels: the maximum number of pointer dereferences that may lead to a value that does not escape. The "Fields" column corresponds to the maximum number of distinguished fields. One could see "Levels" as an evaluation of "vertical" precision, and "Fields" as an evaluation of "horizontal" precision, assuming that data structures are represented such that pointer dereferences go "down" into the data structure.

The rightmost column is the complexity of the analysis. The size of the analyzed program is n, the number of calls *NCalls*, the number of parameters of a function p, the number of functions F, the size of the graph of a type D, the maximum height of a type H. N/C means that the complexity was not mentioned by the authors. The complexity is not always easy to determine in these cases. Nevertheless, we can guess that store-based analyses [Hud86, Hed88, RM88, Hic93] should have at least complexity $\mathcal{O}(n^4)$. Indeed, the number of unknowns is $\mathcal{O}(n)$, each unknown is a set of abstract locations, and the number of abstract locations is at least $\mathcal{O}(n)$. Therefore, the height of the lattice of all unknowns is at least $\mathcal{O}(n^2)$, so $\mathcal{O}(n^2)$ iterations are needed. Each iteration involves searching and updating $\mathcal{O}(n)$ sets of $\mathcal{O}(n)$ elements, therefore an iteration costs $\mathcal{O}(n^2)$. This yields the announced $\mathcal{O}(n^4)$ complexity. Analyses that manipulate graphs [CGS+99, WR99] or alias sets [ACSE99, BH99] should also have a high complexity. Our integer escape analyses are very efficient analyses: they are quasi-linear (our escape analysis for Java has complexity $\mathcal{O}(np^2n_i)$ and the number of iterations n_i is very small in practice). They are the only analyses which at the same time have a

	Analyzed language	Assignments	Levels	Fields	Complexity
[Hud86]	Fun1	N/A	0	1	N/C
[Hed88]	Intermediate lang.	Imp.	0	1	N/C
[RM88]	Object-oriented	Pre.	∞	∞	N/C
[ISY88]	FunLists	N/A	0	N/A	$\mathcal{O}(n)$
[Hug92]	FunLists	N/A	∞	N/A	N/C
[PG92]	FunLists	N/A	∞	N/A	exponential
[TT93]	SML	Pre.	∞	∞	N/C
[Hic93]	FunPar	?[a]	∞	∞	N/C
[Han95]	Fun	N/A	0	N/A	no algorithm
[Moh95b]	Fun	N/A	∞	∞	$\mathcal{O}(D^3p^2Fn)$
[SF96]	Fun1	N/A	∞	∞	$\mathcal{O}(n^4)$
[Deu97]	Fun1Lists	Imp.	∞	1	$\mathcal{O}(n \log^2 n)$
F [Bla98]	Ocaml	Imp.	∞	1	$\mathcal{O}(n \log^2 n)$
[BS98]	λ-calculus	N/A	∞	N/A	N/C
[Liu98]	Fun1	N/A	∞	∞	at least $\mathcal{O}(n^3)$
T	Ocaml	Imp.	∞	∞	$\mathcal{O}(nD^2 + NCallsD^3)$
[ACSE99]	Java	Pre.	∞	∞	N/C
$\mathbb{L}_1$ [Bla99]	Java	Pre.	∞	1	$\mathcal{O}(n^2p^3H)$
[BH99]	Java	Pre.	1	∞	N/C
[CGS+99]	Java	Pre.	∞	∞ or 3	N/C
[WR99]	Java	Pre.	∞	∞	N/C
[GS00]	Java	Imp.	0	1	$\mathcal{O}(n)$
[Ruf00]	Java	Pre.	∞	∞	exponential

[a] Assignments are not mentioned in the paper, but the implementation supports arrays

Figure 10.1: Comparison between different escape analyses and similar analyses

very small complexity and distinguish infinitely many levels (they are extensions of Deutsch's analysis which has the same properties).

Several choices are possible to represent approximately sets of access paths (which correspond to escaping part of data). We have used annotated types and integers in our analyses. In the following paragraphs, we describe the different escape analyses considered in the literature, classified according to the representation they use.

Domain with two escape contexts

Gay and Steensgaard [GS00] have considered a very fast escape analysis (linear in the size of the SSA form of the program) for Java, with only two abstract values: either the object does not escape at all, or it may escape entirely. As soon as an object is included into another data structure, it is considered as escaping. This is the right level of precision for exploding objects (i.e. transforming an object into several variables, one for each element of the object), but a more precise analysis can discover more stack allocation opportunities.

Integers

Escape analysis on lists has been introduced by Park and Goldberg [PG92]. They consider a purely functional higher-order polymorphic language with lists as only non-atomic data type.

The lattice they use contains in fact pairs of integers, but it is isomorphic to the lattice of integers. Their analysis is a forward analysis. It is exponential for first-order programs, and the complexity is much greater in the higher-order case.

Deutsch [Deu97] has much improved the complexity of their analysis, reducing it to $\mathcal{O}(n \log^2 n)$, with exactly the same results for first-order expressions, by using a backward analysis and exploiting the specific properties of the predicate transformers of escape analysis. He has proved that Park and Goldberg's escape analysis for higher-order functions is DEXPTIME-complete. He has also suggested many extensions (cartesian products, polymorphism, imperative operations, and a less complex and less precise extension to higher-order functions).

Our work on escape analysis using integers is in the line of the work of Deutsch. We have fully implemented his $\mathcal{O}(n \log^2 n)$ algorithm in the Objective Caml compiler. We have also implemented further extensions to declared recursive types. We have improved the program transformation techniques by designing a more precise analysis of recursive calls (useful to preserve tail recursion optimizations), being able to move `lets` to have the minimal lifetime for stack allocated data, and using `letstack'` to be able to perform a stack allocation while preserving a tail call optimization. We use inlining to increase stack allocation opportunities. We have given a correctness proof from a semantics of the language. Moreover, we have adapted this technique to allow a precise analysis of assignments in the Java language, while keeping a fast analysis.

Hughes [Hug92] already introduces integer levels to represent the escaping part of data. He considers a purely functional higher-order language with lists. His analysis is split into two parts: generation analysis and inheritance analysis. A location is generated by an expression M if it is allocated by M and reachable from the result of M. A location reachable from the result of a subexpression N of M is inherited by M if that location is reachable from the result of M. Locations generated by the subexpression N of M but not inherited by M can be deallocated after evaluation of M. He does not perform stack allocation, but explicit deallocation: he keeps in memory addresses of data to be deallocated in order to avoid using the GC. The work closest to Hughes' is [ISY88] by Inoue, Seki and Yagi, who only deallocate the top spine of lists, but give experimental results.

Type systems

Hannan [Han95, Han98] uses a type system in which types are annotated to describe escape information in a purely functional language. He only gives inference rules and no algorithm to compute annotated types. The results are not as precise as ours. For example, in $f = \lambda x.\lambda y.xy$, only the parameter passed to y can ever be stack allocated, because x escapes in the partial application fx.

Annotated types

Mohnen [Moh95a, Moh95b] describes an analysis which has a similar precision as our analysis T. We have extended this work since we handle imperative operations and polymorphism. Our higher-order extension is less precise than Mohnen's, but dramatically faster (Mohnen's analysis is exponential in the higher-order case[1]). Mohnen uses a forward analysis, whereas our analysis is backward, thus avoiding analyzing the same function several times. Our

[1] His analysis is not additive any more in the higher-order case, and his optimizations cannot be applied.

analysis is therefore less complex even in the first-order case: $\mathcal{O}(nD^2 + \mathit{NCalls}\,D^3)$ instead of $\mathcal{O}(D^3P^2Fn)$, where D is the maximum size of the graph of a type, n the size of the program, NCalls the number of function calls, P the maximum number of parameters of a function and F the number of functions. We have further optimized our analysis by using compact nodes, therefore reducing the value of D (this value may be very large in complex programs: there are thousands of nodes in some cases).

Graphs

Other escape analyses for Java are based on graphs: [WR99] uses points-to escape graphs, and [CGS$^+$99] uses connection graphs. Connection graphs are similar to alias graphs and points-to graphs but can be easier summarized to avoid recomputing the escape information when a method is called in different escape contexts. These analyses are applied to stack allocation and synchronization elimination. They are more precise than our integer escape analysis for Java because they distinguish different fields of objects and are flow sensitive. Our analysis has a much lower cost and appears to be precise enough on our benchmarks. In these graph escape analyses, objects are considered as escaping when certain nodes are reachable in a graph. The assignments are precisely analyzed, which is necessary to have a precise escape analysis for Java. [CGS$^+$99] considers a `new` statement as stack allocatable only when it is so in all calling contexts of the method. Therefore, they cannot use inlining to increase stack allocation opportunities.

Regular tree grammars

Sets of paths could also be represented by regular tree grammars, as in [Liu98]. Regular tree grammars are quadruples $(\mathcal{T}, \mathcal{N}, S, \mathcal{P})$ where $\mathcal{T}$ is a set of terminal symbols containing ID, AB and T_C for each constructor C, $\mathcal{N}$ is a set of non-terminal symbols N, S is a start symbol, $\mathcal{P}$ is a set of production rules of the form

$$N \Rightarrow ID, \quad N \Rightarrow AB, \quad N \Rightarrow T_C(N_1, ..., N_n).$$

In our formalism, ID represents the full set of paths Path, AB represents the empty set of paths. The rule $N \Rightarrow T_C(N_1, ..., N_n)$ means that $\mathit{Paths}(N) \supseteq \cup_i \overline{C_i}.\mathit{Paths}(N_i)$ where $\mathit{Paths}(N)$ is the set of path corresponding to non-terminal N.

However, representing transformers for regular tree grammars is a complex task. In [Liu98], there is no such representation. The applications of transformers of regular tree grammars are computed on demand. Moreover, approximation operations (widenings) have to be used to make sure that the iteration terminates.

Regular tree grammars are in fact relatively close to the annotated types we use in analysis T. The main difference is that the cycles are fixed by the type in our analysis, whereas they can be chosen dynamically in regular tree grammars, to fit the set of paths we have to represent. Regular tree grammars lead to a slightly more precise analysis. However, they also make the implementation of the analysis more complex and more difficult.

Other analyses

Alias analysis [Deu94], reference counting [Hud86, Hed88], storage use analysis [SF96] which is similar to [JM82, Har89, Deu90, Shi88] can be applied to stack allocation though at a much higher cost.

Escape analysis in fact determines particular kinds of alias properties (the aliases with the result of functions, or more generally, the aliases with a known value). Therefore, alias analysis can also be used in place of escape analysis. However, an alias analysis is more costly than escape analysis, since it does not take into account the particular properties of the considered problem to improve efficiency.

Bogda and Hölzle [BH99] use an escape analysis based on a limited form of alias information (only access paths of length at most 2 are considered). They apply it to synchronization elimination for Java. This analysis is more precise than our integer escape analysis for Java on some examples since it can distinguish different fields of objects. It is however less precise on other examples because it cannot detect objects that are transitively reachable only from local objects if two or more dereferences are needed to reach the object. This analysis is also more costly than ours.

Other analysis techniques are store-based analyses: locations in the store are abstracted, in general by their allocation site. Properties of these abstract locations are then determined. Such techniques have been used in [Hud86, Hed88, RM88, Hic93, SF96]. In general, they suffer from a high complexity. Moreover, the simplest versions of these analyses are not context sensitive, and context sensitive versions have an even higher complexity. In contrast, our escape analyses are context sensitive.

The reference count of a value v is the number of pointers towards v. The idea of abstract reference counting [Hud86, Hed88] is to statically compute an approximation of these reference counts. Data can be deallocated when their reference count becomes 0. Only values which are never reachable from other data structures can be deallocated by such techniques.

Storage use analysis [SF96] records for each variable the set of allocation sites that may create data that will be put in that variable. It also computes for each constructor the allocation sites from which the parameters may come. This analysis is used for both stack allocation and unboxing. Important speedups are reported. Even if speedups coming from stack allocation are not clearly separated, the improvement seems to come mostly from unboxing.

Hicks [Hic93] analyses a parallel functional language. He also uses a store semantics in which objects are abstracted by their allocation site, plus possibly a set of call strings to the allocation site, represented by a regular expression. He computes inherited sets (abstract locations that can be passed to an expression e from the context, i.e the free variables of e) and reachable sets (abstract locations reachable from the result of expression e). He can then infer a deallocation safety criterion. Deallocation commands can be inserted by hand and verified, or automatically inserted at safe positions.

Ruggieri and Murtagh [RM88] perform lifetime analysis in an object-oriented language, using again a similar abstraction, giving for each variable the set of possible allocation sites of its contents. One originality is that they use static typing to have a minimal interprocedural alias information before the intraprocedural phase. The intraprocedural analysis can then be performed once for all, before the interprocedural analysis.

Guider [Gui00] determines alias properties with an analysis based on Wilhelm, Sagiv and Reps' shape analysis [WSR00]. There are some similarities with our analyses: both determine alias properties; Guider's abstract garbage collection operation eliminates unreachable data, and data eliminated by this operation could also be deallocated at runtime. However, Guider's goals, automatic parallelisation and distribution of Java programs, are different from ours. These goals justify the use of a more complex and precise analysis. Guider has also used turboJ as an implementation platform.

Banerjee and Schmidt [BS98] use an analysis based on Sestoft's closure analysis [Ses91] to perform stack allocation in the simply-typed λ-calculus. They obtain a very precise criterion to determine statically whether an expression can be evaluated with a single global stack (no heap). Their goal is therefore slightly different from ours (we still use a heap). Their analysis is also more complex than our integer escape analyses.

10.1.2 Program transformations

Several program transformations have been considered in the literature. Stack allocation has been considered in many papers [PG92, Han95, Moh95a, Moh95b, SF96, Deu97, Bla98, Bla99, WR99, CGS$^+$99, GS00], and it is the main transformation considered in this thesis, so we are not going to comment more on it.

Explicit deallocation

Explicit deallocation has been studied in [ISY88, Hug92, Hic93]. The program analysis is used to determine the maximum lifetime of objects, and explicit deallocation statements are inserted in the program. This technique recovers the explicit memory management mechanism we have in C with `malloc` and `free`. This is more general than stack allocation, since objects need not be deallocated in the reverse order of the allocation. However, this may also lead to less speedup, since the cost of heap allocation and deallocation is greater than for a stack.

Preallocation

Preallocation has been used for example in [WR99, GS00]. When an object is allocated in a function f, and escapes from f, we cannot directly perform stack allocation. If the object does not escape from a caller f' of f, we can stack allocate it in f', and pass the allocated space to f. The object is then preallocated, before the call to f. This technique is an alternative to inlining, which we have used in similar situations to perform stack allocation. The advantage of preallocation is that it avoids code duplication (the called function has to be specialized for preallocation, but not duplicated every time it is called). The limit of this technique is that we must be sure that an object will be allocated in f, and we must know its size before the call to f. Depending on the kind of GC that is used, preallocation may also cause some problems: with the GC of Objective Caml, all data must be initialized when the GC occurs. Initializing the preallocated space before the call to f would lead to a time loss (it would be initialized twice: before the function call and when the function builds its result). This is a reason why we have preferred inlining to preallocation.

Exploding objects

Exploding objects [GS00] means replacing an object with several local variables, thus completely avoiding the allocation. This can lead to greater speedups than stack allocation, since local variables can be put in registers. However, there are less opportunities for object exploding than for stack allocation, since exploded objects must not be part of a data structure. Moreover, when the exploded value is passed as a parameter to a function, we have to inline or specialize the function. This leads to an important code growth. Value exploding can be performed by the simple escape analysis with only two escape contexts mentioned above: $\perp$ when the value is not included in any other data structure, $\top$ otherwise.

Region allocation

Another allocation optimization has been suggested in [TT93, AFL95, BTV96]: region allocation. All objects are allocated in heap regions whose size is not known statically in general, but for which we know when they can be deallocated. Regions can therefore be deallocated without GC. This analysis solves a more general problem than ours, but at the cost of a much increased complexity. In fact, on many programs, opportunities for stack allocation outnumber opportunities for region allocation, as noticed in [BTV96].

[AFL95] extends the system by inferring creation and deallocation points of regions such that the data have minimal lifetime. This decreases the memory usage.

Escape analysis can also be extended to allow region allocation. According to the maximal scope at which a value may escape, it is allocated in a different region, and the whole region is deallocated when we are sure that all the values it contains are unreachable.

Synchronization elimination in Java

Escape analysis has been applied to synchronization elimination in the Java programming language [Bla99, BH99, CGS+99, WR99]. The general idea is that when an object is reachable from only one thread, there is no need to synchronize on it. This can lead to several techniques to remove synchronizations: the synchronization can be removed at the call site [Bla99, WR99], synchronizations can be removed on stack allocated objects [Bla99], objects can be marked thread-local at allocation site [CGS+99], or a subclass without synchronization can be defined for each class C that contains a synchronized method and used instead of C when the object is detected as thread-local [BH99]. These techniques have been compared in Section 8.3.4. In all the papers mentioned above, as soon as an object is reachable from a static field, it is considered as shared between several threads. Ruf [Ruf00] goes one step further in this direction by marking static fields either as shared fields, if several threads may access them, or unshared if they are accessed by one single thread. Synchronizations can then be eliminated on objects reachable from unshared static fields. His analysis seems reasonably fast in practice, even if it is at least exponential in the worst case.

Aldrich et al. [ACSE99] eliminate synchronizations on thread local objects with an analysis based on a preliminary alias analysis. However, their results are limited by the lack of specialization. They also eliminate synchronizations when a monitor is always protected from concurrent access by another monitor. We did not investigate this approach here, since eliminating synchronizations on thread local objects already deals with most cases.

Other techniques have been investigated to reduce the cost of synchronization not by eliminating synchronizations, but by designing a faster synchronization primitive [BKMS98, ADG+99, OK99]. Both approaches could be combined to reduce even more the synchronization cost.

Diniz and Rinard [DR98] reduce the cost of synchronization by coarsening the granularity at which the computation acquires and releases locks. First, they allow objects to share locks. Second, when a lock is repeatedly acquired and released in a region of code, they move synchronization constructs outside this region. However, this can increase the contention on locks. They reduce the cost of synchronization for objects shared by several threads, while our goal is to eliminate synchronizations on thread local objects.

All synchronizations can be eliminated in single threaded programs by simply detecting that no second thread is created [MMBC97, Bac97, FKR+00]. However, such an approach cannot optimize multithreaded programs, which much reduces its scope.

10.1.3 Study of the impact of stack allocation

Appel [App87] claims that *garbage collection can be faster than stack allocation.* Indeed, copy garbage collection can deallocate data at no cost (dead data are not scanned by the GC), whereas stack allocation requires at least one instruction to deallocate values. However, the cost of garbage collection becomes very small only when the amount of available memory is much larger than necessary. This is in general not the case. Indeed, even if the cost of memory decreases quickly, and the amount of available memory increases, typical programs need more and more memory.

Appel and Shao [AS96] argue even in favor of heap allocating function frames. This has been implemented in SML. Their analysis mostly coincides with our constatations: heap allocation leads to high write-miss rates, using stack allocation or heap allocation does not change much the load of the GC with a copy collector. Appel and Shao assume a write allocate cache, whereas most of our tests have been performed on Pentium MMX, with a write around cache. Stack allocation leads to a larger decrease of the time of read misses with a write around cache than with a write allocate cache. It is therefore logical that we find stack allocation beneficial, whereas they find heap allocation as efficient as stack allocation.

Other studies of the effect of stack allocation or compile time garbage collection [ISY88, JW90, Hic93, McD98, CGS+99, WR99, GS00] report improvements, as in our benchmarks.

10.2 Future work

We have seen that escape analysis can be regarded as a particular kind of alias analysis. Escape analysis determines whether each value is aliased with a fixed set of known values (with the results of functions and the stored values for stack allocation in ML, with static fields for synchronization elimination in Java). Escape analysis can therefore have many more applications than those studied in this thesis:

- Other representation optimizations: our escape analysis could also be applied to preallocation [GS00], region allocation [TT93, AFL95, BTV96], exploding objects [GS00].
- Stack allocation has another important advantage that we have not yet mentioned: it runs in constant time, and avoids the pauses due to the GC. It would therefore be interesting to use stack allocation in real-time programs. In this case, all data in the time critical parts must be stack allocated. Escape analysis would then be used to verify that the stack allocations are correct. (This has been suggested by Xavier Leroy.) A more precise and more costly analysis could then be useful, to be able to prove that all stack allocations are correct. It would also be more usable, since only the time-critical parts would be analyzed.

  Stack allocation could also be implemented in a just-in-time compiler. In this case, we would not analyze the whole program, but only the part that we want to compile because most of the time is spent in this part. We may have to analyze some functions that we do not compile because some values escape in these functions. Even with these additional analyzes, escape analysis could most probably be done at a much smaller cost than the analysis of the whole program. (There is some not yet published work in this area, by Frédéric Vivien and Martin Rinard.)

- Synchronization: Applying the ideas of [Ruf00], our escape analysis could be extended to remove more synchronizations: we would mark static fields that are accessed by several threads; other static fields are thread local. Synchronizations may be removed on objects even if they are stored in thread-local static fields.

  We could also check statically that a program contains all the necessary synchronizations. This means checking that each time an object o shared between several threads is accessed, a lock has been acquired. This lock need not be on the object o itself, but for each object o, the lock must be acquired on the same object o'. For example, when accessing elements of a `Vector v`, the array `v.elementData` is read. The synchronizations are not on `v.elementData` itself, but on `v`. The techniques of [ACSE99] to detect enclosed monitors could be useful (combined with escape analysis) to handle the case when o' is different from o. Notice that this would not completely check that the synchronizations are correct in the program, but it could detect numerous mistakes.

- Security properties: properties such as confinement [BV99], or the fact that objects cannot be modified by methods outside their class could also be checked by (a slightly modified) escape analysis.

  Confinement is defined as follows. Let a protection domain be given (for example a package). A type is said to be confined in this domain if and only if "all references to instances of that type originate from objects in the domain" [BV99].

We plan to study some of these applications in the future.

Chapter 11

Conclusion

We have studied and proved correct escape analyses for two realistic and quite different languages. A higher-order functional language with imperative operations, polymorphism and recursive data structures has been the basis of the implementation in Objective Caml. On the other hand, we have considered a realistic subset of Java, containing in particular subtyping and virtual method calls.

In the design of our analyses, we have used abstract interpretation techniques, which enable us to derive our analyses. One important advantage of these techniques is that they can be used to define a hierarchy of analyses of different levels of precision. The correctness of the less precise analyses is a consequence of the correctness of the more precise ones, thus we do not have to prove each analysis from scratch. Proving the correctness of analyses on languages that combine many different features is technically much more difficult than for simple toy languages. Difficulties arise not from each feature individually, but from the combination of several features. For example, the presence of imperative operations (together with polymorphism) makes the formula and proof of instantiation more complex (using closed contexts would not be necessary without imperative operations).

We have particularly focused on designing efficient implementations of these analyses. We have used efficient algorithms such as the graph reachability algorithm of Horwitz, Reps and Sagiv, Tarjan's path compression algorithm and Knuth's solving algorithm. Moreover, the design of our analyses has taken into account the particular properties of the language to which they are applied. Our escape analysis for Java is different from the escape analysis for ML, because the important features of the two languages are different. The escape analysis for Java has to take care precisely of assignments, whereas the analysis for ML can analyze them very approximately. To have a fast analysis, we have to accept some loss of precision, but a careful design can choose on which aspects a loss of precision will not be too important.

Our implementation has demonstrated the feasability and the interest of escape analysis. The integer escape analyses yield high speedups at a reasonable cost. For Ocaml, speedups are around 10%, whereas for Java, they reach a geometric mean of 22%. For Ocaml, they mainly come from improvements on data locality, whereas Java benefits both from a decrease of the GC workload and of the synchronization elimination. The compile time increase is about 20% for Ocaml and 30% for Java (the analysis itself takes only 10% of the compile time for Java). Moreover, the analysis scales very well on large benchmarks.

A fast analysis, even if not too precise, seems to be a very good compromise for optimization purposes. This is illustrated in ML: our precise analysis T has a much higher cost than the less precise analysis L, but does not bring much more improvement in practice. In this case, detecting a few more optimizations opportunities is not worth the additional cost. Moreover, implementing a costly analysis is much more difficult than implementing a fast analysis: the analyzer has to be carefully optimized to avoid memory overflows, and to run in a reasonable amount of time.

On the contrary, a more precise and more costly analysis may be worthwhile for verification purposes. The user has to hand-check each case which is not handled by the analyzer. It is therefore very important that there are not too many remaining cases.

From a more general point of view, escape analysis is useful to determine some kinds of alias properties: is a value aliased with a given set of values ? Its advantage over usual alias analyses is that it is a very fast analysis. We have studied two applications of escape analysis, stack allocation and synchronization elimination, but it is likely to be applied to many other purposes in the future, since alias analysis is useful for a wide range of subjects. We plan to investigate some of these applications, such as verification of security properties, of synchronizations.

Appendix A

Proofs for analysis E for ML

A.1 Correctness proof

As we have seen in Section 3.3, escape analysis gives a kind of alias information. Therefore, the most natural way of performing the correctness proof would probably be to use an alias relation, and to give a definition of escape analysis such as (this is only informal, and this does not take into account assignments)

> Let R be the result of M. Let $R.c$ be the part represented by context c in the result of M. Let $c' = E[\![M]\!]\rho x c$. c' is correct if and only if it satisfies the following condition for all access paths p: if $x.p$ is aliased with $R.c$ by aliases created during the execution of M, then p is in c'.

However, there is a difficulty for applying this definition here: the access paths that we use contain the accessor app which has no meaning in an alias relation: app cannot be an element of a path in the alias relation.

So we have chosen a different idea: we define which locations correspond to a given context c for a given value v. The definition of correctness is then informally:

> Let R be the result of M. $c' = E[\![M]\!]\rho x c$ is correct if and only if it satisfies the following condition. Assume that ℓ is a location of the part of R represented by c (that is, ℓ escapes). If ℓ is also in x, then ℓ is in the part of x represented by c'.

The main difficulty here is to distinguish where the locations come from: several variables, and parameters of functions may share the same locations. For example, if an expression M is a variable `x`, we expect that `x` will escape from M, but no other variable will escape from M. However, if `x` has been defined by `let` as follows:

```
let x = y in x
```

then, with the above definition, `y` also escapes. Indeed, with usual semantics, the above example of expression is equivalent to

```
let x = y in y
```

or even simply `y`. Therefore, we have to instrument the semantics so that the above expressions have different semantics. We want to be able to find out from which variable a location has been obtained. The idea is then to associate a label with each variable. All values obtained by accessing this variable will have the corresponding label.

However, simple labels are not enough when a variable is inside another, as in

```
N = let y = [x] in
      hd y
```

Let λ_1 be the label of `x`, and λ_2 be the label of `y`. If we consider the expression `hd y`, only variable `y` escapes, then the result of this expression should have label λ_2. But if we consider the whole `N` expression, then variable `x` escapes, and the result of `N` should have label λ_1. How can we reconcile these contradictory requirements ? This can be achieved by concatenating the labels: the label of the result of `hd y` will be the concatenation of the labels of the various variables accessed to build this result: $[\lambda_2 : \lambda_1]$ (the concatenation is computed by $\mathrm{getval}(s, v)$ below). The first label of the list λ_2 corresponds to the first accessed variable, `y`. It is also the label of the escaping variable for the innermost expression `hd y`. When considering the outermost expression `N`, we remove labels that did not exist at the beginning of the evaluation of this expression, i.e. we remove λ_2, and the remaining label $[\lambda_1]$, is the label of the escaping variable `x`. Indeed, when analyzing an expression N, we are interested in the escaping part of the free variables of N (and of the parameters of N if it is a function), therefore, we remove all labels which do not correspond to such variables. This is done by what we call label filtering (see Definition A.8).

A.1.1 Semantics

We first define our semantics instrumented with labels. We have chosen to use a denotational semantics with store and non-termination. The advantage of a denotational semantics is that it is defined similarly to our analysis: both are defined by an explicit fixed point in the `let rec` case and they are expressed by induction on the syntax of the expressions. This will simplify the proof. The store is necessary to represent allocated values, which is a key point to be able to deal with heap and stack allocation.

- ***Locations*** (any countable set): $\ell \in \mathit{Loc}$.
- ***Values***: $\mathit{Val} = \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z} \cup \mathit{Loc}$. true and false are boolean values, NIL is the value of type `unit`, and the elements of $\mathbb{Z}$ are used to represent integers. Locations $\ell \in \mathit{Loc}$ are used to represent more complex values (functions and constructed types) whose structure is represented in the store defined below.
- ***Labels***: $\lambda \in \mathit{Label} = \mathit{Lbl}^* \cup \{U\}$. Lbl is any countable set, which corresponds to labels of variables. The definition of Label enables us to concatenate such labels. The special label U is the label of updated values (values that have been modified by an assignment). This label is not concatenated with others, it has priority over other labels, and makes any other label disappear.
- ***Labelled values***: $\mathit{LabelVal} = \mathit{Val} \times \mathit{Label}$.
- ***Storable values***: $SV = (\mathit{Constr} \times \mathit{LabelVal}^*) \cup (\{\mathtt{fun}\} \times (\mathit{Env} \to \mathit{Exp}) \times \mathit{Env} \times \mathit{Var})$. Storable values are the possible contents of each location of the store. The contents may either be a constructed value which is represented by a tuple containing the constructor

$[\![M]\!] : Env \rightarrow Exp$

$[\![x]\!]es = \text{up}(e[\![x]\!], s)$

$[\![M\ N]\!]es = \text{let } [(f', s')] = [\![M]\!]es \text{ in let } [((v, \lambda_1), s'')] = [\![N]\!]es' \text{ in let } (\texttt{fun}, m, e', x) =$
$\quad \text{getval}(s'', f') \text{ in let } (\lambda_2, s''') = \text{newlbl}(s'') \text{ in } m(e'[x \mapsto (s''', (v, \lambda_2 : \lambda_1))])s'''$

$[\![\texttt{fun}\ x \rightarrow M]\!]es = \text{newloc}(s, (\texttt{fun}, [\![M]\!], e_{|FV(\texttt{fun}\ x \rightarrow M)}, x))$ [a]

$[\![\texttt{let}\ x = M\ \texttt{in}\ N]\!]es = \text{let } [((v, \lambda_1), s')] = [\![M]\!]es \text{ in let } (\lambda_2, s'') = \text{newlbl}(s') \text{ in}$
$\quad [\![N]\!](e[x \mapsto (s'', (v, \lambda_2 : \lambda_1))])s''$

$[\![\texttt{let rec}\ f = \texttt{fun}\ x \rightarrow M\ \texttt{in}\ N]\!]es = \text{let } (\lambda, s'') = \text{newlbl}(s) \text{ in } [\![N]\!](e[f \mapsto (s', (\ell, [\lambda]))])s'$
where $\ell \notin \text{Dom}(s)$, $s' = (s_0[\ell \mapsto (\texttt{fun}, \phi, e_{|FV(\texttt{fun}\ x \rightarrow M)}, x)], a, \Lambda)$ if $s'' = (s_0, a, \Lambda)$ and ϕ is the least function such that $\phi = \lambda e_1.\lambda s_1.[\![M]\!](e_1[f \mapsto (s_1, (\ell, [\lambda])])s_1$.

$[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]es = \text{let } [(x, s')] = [\![M]\!]es \text{ in if } x \text{ then } [\![N]\!]es' \text{ else } [\![P]\!]es'$

$[\![M; N]\!]es = \text{let } [(l, s')] = [\![M]\!]es \text{ in } [\![N]\!]es'$

$[\![C(M_1, ..., M_k)]\!]es = \text{let } [(x_1, s_1)] = [\![M_1]\!]es \text{ in ... let } [(x_k, s_k)] = [\![M_k]\!]es_{k-1} \text{ in}$
$\quad \text{newloc}(s_k, (C, x_1, ..., x_k))$

$[\![\overline{C_i}(M) \leftarrow N]\!]es = \text{let } [((x, \lambda_x), s')] = [\![M]\!]es \text{ in let } [((y, \lambda_y), (s'', a'', \Lambda''))] = [\![N]\!]es' \text{ in}$
$\quad \text{let } (C, v_1, ..., v_k) = s''(x) \text{ in up}((\text{NIL}, []),$
$\quad (s''[x \mapsto (C, v_1, ..., v_{i-1}, (y, U), v_{i+1}, ..., v_k)], a'' \cup (y, \lambda_y) \downarrow_{s''}, \Lambda''))$

Initial store: $s_0 = (\{\overline{C_i}\text{clo} \mapsto (\texttt{fun}, \lambda e.\lambda s.\text{let } (C, v_1, ..., v_k) = \text{getval}(s, e[\![x]\!]) \text{ in up}(v_i, s), \emptyset, x),$
$\quad C?\text{clo} \mapsto (\texttt{fun}, \lambda e.\lambda s.(\text{getval}(s, e[\![x]\!]) = ((C, \_, ..., \_), \_)), \emptyset, x)\}, \emptyset, \emptyset)$

Initial environment: $e_0 = \{\overline{C_i} \mapsto (s_0, (\overline{C_i}\text{clo}, [])), C? \mapsto (s_0, (C?\text{clo}, [])), \texttt{true} \mapsto (s_0, (\text{true}, [])),$
$\quad \texttt{false} \mapsto (s_0, (\text{false}, [])), n \mapsto (s_0, (n, [])) \text{ if } n \in \mathbb{Z}\}$.

[a] We create minimal closures, therefore we only store free variables of $\texttt{fun}\ x \rightarrow M$ in the environment of the closure.

Figure A.1: Semantics

and the values of the parameters of the constructor, or a closure which represents a function. A closure is represented by a tuple containing the `fun` tag to distinguish closures from constructed types, the semantics of the function, the environment in which it has been defined and the bound variable of the function.

- ***Store***: $s \in Store = (Loc \rightarrow SV) \times \mathcal{P}(LabelVal) \times \mathcal{P}(Lbl)$. In a usual denotational semantics, the store would only be $Store = Loc \rightarrow SV$, indicating for each location the storable value it contains. Here, we add two more pieces of information that will be useful in the correctness proof of our analysis. The second component of the store s is the set of labelled locations that have been stored by assignments during program execution, until store s is reached. All these locations will be considered as escaping in our analysis. The third component of the store s is the set of labels that have been created until store s is reached. Let $\text{Dom}(s)$ be the set of locations existing in store $s = (s_0, a_0, \Lambda_0)$, i.e. $\text{Dom}(s_0, a_0, \Lambda_0) = \text{Dom}(s_0)$.

- Semantics of expressions: $Exp = Store \rightarrow (LabelVal \times Store)_\perp$. The semantics of an expression takes as parameter the store in which it is executed. If the evaluation of the expression terminates, it yields a pair containing a value which is the result of the expression, and the new store at the end of the execution of the expression. When the execution of the expression does not terminate, it yields $\perp$.

- ***Environments***: $e \in Env = Var \to (Store \times LabelVal)$. The environment gives for each variable its value. However, we do not only mention the value, but also the store in which it has been created. We define $e[\![x]\!] = v$ if $e(x) = (s, v)$. By α-conversion (renaming of variables) we can assume that all variables in any environment have distinct names.

The reader can notice that this is a recursive definition (*Exp* depends on *Store* which depends on SV which itself depends on *Exp*). The semantics of such a definition is the same as the semantics of a recursive type. It can be given a precise meaning by domain theory.

The semantic function is $[\![M]\!] : Env \to Exp$. Let us define the following abbreviations:

- up is the injection from $LabelVal \times Store$ to $(LabelVal \times Store)_\perp$.
- The expression
$$\text{let } [(x, s)] = h \text{ in } g$$
is an abbreviation of $\begin{cases} \perp & \text{if } h = \perp, \\ g[x'/x, s'/s] & \text{if } h = \text{up}(x', s'). \end{cases}$.
- $\text{newloc}(s, v) \in (LabelVal \times Store)_\perp$ where $s \in Store$ and $v \in SV$ is defined by $\text{newloc}((s, a, \Lambda), v) = \text{up}((\ell, []), (s[\ell \mapsto v], a, \Lambda))$ where $\ell \notin \text{Dom}(s)$ is a new location, which is initialized to storable value v. The label of the new location is the empty label $[]$.
- $\lambda_1 @ \lambda_2$ is the concatenation of the lists of labels λ_1 and λ_2 if λ_1 and λ_2 are in Lbl^*. If one of λ_1 or λ_2 is U, $\lambda_1 @ \lambda_2 = U$. $\lambda_1 : \lambda_2$ is the list λ_2 at the head of which λ_1 has been added. $\lambda_1 : U = U$. With these definitions, U will not be changed when concatenating labels, and all locations reachable from an updated location will have label U (the definition of $\text{getval}(s, v)$ below shows that the label of a value v_k reachable from v is obtained by concatenating the label of v with the label stored in s for v_k).
- Let $\text{getval}(s, v) \in SV$ where $s \in Store$ and $v \in LabelVal$ be defined by: $\text{getval}((s, a, \Lambda), (v, \lambda)) = (C, (v_1, \lambda @ \lambda_1), ..., (v_k, \lambda @ \lambda_k))$ if we have $s(v) = (C, (v_1, \lambda_1), ..., (v_k, \lambda_k))$, and $\text{getval}((s, a, \Lambda), (v, \lambda)) = (\texttt{fun}, \phi, e', x)$ if $s(v) = (\texttt{fun}, \phi, e, x)$ where e' is defined by $e'(x) = (s_x, (v_x, \lambda @ \lambda_x))$ if $e(x) = (s_x, (v_x, \lambda_x))$. $\text{getval}(s, v)$ replaces the usual $s(l)$, and is useful to take into account our instrumentation with labels.
- Let $v \downarrow_s$ be the set of labelled locations reachable from the labelled value v in store s. Updated locations (locations labelled U) are ignored. This set can be precisely defined by fixed point iteration:
$$\begin{aligned}
&(v, \lambda) \downarrow_s \; = \emptyset \text{ if } v \in \{\text{true}, \text{false}, \text{NIL}\} \cup \mathbb{Z} \text{ or } \lambda = U \\
&(\ell, \lambda) \in (\ell, \lambda) \downarrow_s \; \text{ if } \ell \in Loc \text{ and } \lambda \neq U \\
&w \in v \downarrow_s \; \Rightarrow v_i \downarrow_s \; \subseteq v \downarrow_s \; \text{ if } \text{getval}(s, w) = (C, v_1, ..., v_k) \\
&w \in v \downarrow_s \; \Rightarrow e \downarrow_s \; \subseteq v \downarrow_s \; \text{ if } \text{getval}(s, w) = (\texttt{fun}, \phi, e, x) \\
&e \downarrow_s \; = \cup \{ e[\![x]\!] \downarrow_s \; \mid x \in \text{Dom}(e) \}
\end{aligned}$$
- $\text{newlbl}(s) \in Lbl \times Store$ with $s \in Store$ creates a new label, and adds it to the set of existing labels: $\text{newlbl}((s, a, \Lambda)) = (\lambda, (s, a, \Lambda \cup \{\lambda\}))$ where $\lambda \notin \Lambda$.

- The expression

$$\text{let } (C, v_1, ..., v_k) = h \text{ in } g$$

is an abbreviation of

$$g[v'_1/v_1, ..., v'_k/v_k] \text{ if } h = (C, v'_1, ..., v'_k).$$

If h is not a constructed value of constructor C, then a runtime error occurs. However, for simplicity, we do not consider this error case. The semantics is undefined in this case.

The semantic equations are summarized in Figure A.1. The first key property of this semantics is that the instrumentation can be removed without changing the results. Therefore, we have not changed the real meaning of the language by adding our instrumentation.

In the x case, if $x \notin \mathrm{Dom}(e)$, an error occurs. However, this error can never happen in a statically typed language as Objective Caml. We leave the semantics undefined in this case.

In the application case, the function M is evaluated first, then the argument N, and then the function is applied to its argument. A new label λ is created with newlbl(s) each time we add a value in the environment, to be able to distinguish locations that come from different variables. This happens in the application, the `let` and `let rec` cases. The new label is added to the set of labels existing in the store (the third component of the store) by newlbl(s).

In the `fun` case, a closure is created, containing the values of the free variables of the function.

In the `let` case, M is evaluated first, then a new environment is created in which x is bound to the value v of M, and N is evaluated in this new environment.

In the `let rec` case, the reader may be surprised by the fact that no fixed point appears. The definition of ϕ by

$$\phi = \lambda e_1.\lambda s_1.[\![M]\!](e_1[f \mapsto (s_1, (\ell, [\lambda]))])s_1$$

is in fact an equation, since the semantics of M depends on the contents of $\mathrm{getval}(s_1, (\ell, [\lambda])) = \mathrm{getval}(s', (\ell, [\lambda])) = (\mathtt{fun}, \phi, e_{|FV(\mathtt{fun}\ x \to M)}, x)$ (since a function is not mutable, the contents of location ℓ remains the same from its creation to the end of the program), therefore the semantics of M depends on ϕ. This equation can be solved by fixed point iteration:

$$\phi = \mathrm{lfp}(\lambda\phi_1.\lambda e_1.\lambda s_1.[\![M]\!](e_1[f \mapsto (s'_1, (\ell, [\lambda]))])s'_1)$$

where $s'_1 = (s[\ell \mapsto (\mathtt{fun}, \phi_1, e_1, x)], a, \Lambda)$ if $s_1 = (s, a, \Lambda)$. We get the expected fixed point.

In the assignment case, $\overline{C_i}(M) \leftarrow N$, we first evaluate M then N, and then we update the ith component of the result of M. The new component has label U, the special label for updated values, and the stored locations are added to the set of stored locations (the second component of the store).

Other semantic equations are relatively straightforward.

A.1.2 Correctness

The first step in the definition of correctness is to define the set of labelled locations which correspond to a given context c for a given value v. This is easy for values which do not contain closures. Indeed, the set of locations is simply given by following the paths contained in the context. For functions, we have the difficulty mentioned above in the alias definition of the correctness: we have to define the meaning of the app accessor in terms of locations, to be able to give the locations corresponding to paths such as app, app.app, and so on, and this is a complex task. The idea is to simulate the application of the value v to a given set of parameters (which will be stored in what we call the parameter environment, see Definition A.3), and to see which locations appear in the result. The formalization of all this is in Definition A.6.

We begin with a few preliminary definitions and lemmas.

Lemma A.1 *All labels in* $(v', \lambda') \downarrow_s$ *begin with* λ'*:* $\forall (v, \lambda) \in (v', \lambda') \downarrow_s, \exists \lambda'', \lambda = \lambda' @ \lambda''$.

Proof: This comes immediately from the definitions of $(v', \lambda') \downarrow_s$ and of $\mathrm{getval}(v, s)$. □

Definition A.2 (Store order) A store s' is said to be after s if s' can be obtained after s during the execution of a program. If s and s' are in the same execution trace, s' is after s if and only if sA^*s' where A^* is the reflexive and transitive closure of the relation A defined as follows:

- $(s, a, \Lambda) A (s', a, \Lambda')$ if $s' = s[\ell \mapsto v]$ where $\ell \in \mathit{Loc}$ and $\ell \notin \mathrm{Dom}(s)$ (creation of a new location);
- $(s, a, \Lambda) A (s', a', \Lambda')$ if $s(\ell) = (C, v_1, ..., v_k)$, $s' = s[\ell \mapsto (C, v_1, ..., v_{i-1}, (y, U), v_{i+1}, ...v_k)]$, $a' = a \cup (y, \lambda_y) \downarrow_s$, $\ell \in \mathrm{Dom}(s)$ (the store modification is an assignment); (y, λ_y) is a labelled value.

with $\Lambda \subseteq \Lambda'$.

Definition A.3 (Parameter environment)

The parameter environment is $e_\mathrm{p} \in \mathit{Env}_\mathrm{p} = (\mathit{Store} \times \mathit{Val})^*$. The parameter environment is useful to define which parameters will be given to functions in the following definitions. We do not only specify the value of each parameter but also the store in which it will be applied to a function. All elements of parameter environments must have distinct first labels, i.e. if $e_\mathrm{p}(i) = (s_i, (v_i, \lambda_i : \lambda'_i))$, then the λ_i are pairwise distinct, and different from labels that may appear in the environment. The stores are in the order of creation: s_1 is before s_2, which is before s_3 and so on. The label λ_i has been created just before computing s_i (we can notice in the semantics of application that the label associated with the parameter given to the function is new).

Let $\mathrm{loc}(e_\mathrm{p}) = \cup_{i, e_\mathrm{p}(i) = (s_\mathrm{p}, v_\mathrm{p})} v_\mathrm{p} \downarrow_{s_\mathrm{p}}$ be the set of labelled locations of the parameter environment.

We use the approximation that all locations stored during the computation of the application escape. Thus we first have to define the set of stored locations.

Definition A.4 (Stored locations) Let $a : \mathit{Store} \to \mathcal{P}(\mathit{Loc} \times \mathit{Label})$ such that $a(s)$ is the set of locations that have appeared in the right-hand side of an assignment since the beginning of the execution of the program, until the state corresponding to store s has been reached. More precisely, with our instrumentation $a(s) = a'$ if $s = (s', a', \Lambda)$

Let $a(s, s') = a(s') - a(s)$ be the set of locations stored between stores s and s'.

Lemma A.5 $a(s,s'') = a(s,s') \cup a(s',s'')$ *if* s'' *is after* s' *which is after* s.

Proof: Obvious from the definition of stored locations. $\square$

Definition A.6 (Context concretization)
Let $\mathrm{loca} : \mathit{Store} \times \mathit{Val} \times \mathit{Ctx}_E \times \mathit{Env}_\mathrm{p} \to \mathcal{P}(\mathit{Loc} \times \mathit{Label})$ such that $\mathrm{loca}(s, v, c, e_\mathrm{p})$ represents the set of locations which escape when we keep the locations of context c of value v in store s. The parameter environment e_p determines the parameters that will be given to function v (if v is a function).

Let $\mathrm{loca}_\mathrm{e} : \mathit{Store} \times \mathit{Exp} \times \mathit{Ctx}_E \times \mathit{Env}_\mathrm{p} \to \mathcal{P}(\mathit{Loc} \times \mathit{Label})$ such that $\mathrm{loca}(s, f, c, e_\mathrm{p})$ represents the set of locations which escape when an expression of semantics f is evaluated in store s, and we keep the locations of context c in its result.

We define loca by: $\mathrm{loca}(s, v, c, e_\mathrm{p})$ is the smallest set such that the following equations are true:

- If $v \in \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z}$, $\mathrm{loca}(s, (v, \lambda), c, e_\mathrm{p}) = \emptyset$.

- If $\mathrm{getval}(s, v) = (C, v_1, ..., v_k)$ (constructed types),

$$\mathrm{loca}(s, v, c, []) = (\{v \text{ if } \epsilon \in c \text{ and } v \neq (\_, U)\}) \cup \cup_{i \in [\![1,k]\!]} \mathrm{loca}(s, v_i, \overline{C_i}^{-1}.c, []), \quad \text{(A.1)}$$
$$\mathrm{loca}(s, v, c, e_\mathrm{p}) = \emptyset \text{ if } e_\mathrm{p} \neq []. \quad \text{(A.2)}$$

  If $\epsilon \in c$ and the parameter environment is empty, the location v at the top of the considered value is marked.

- If $\mathrm{getval}(s, v) = (\mathtt{fun}, m, e', x)$ (closures),

$$\mathrm{loca}(s, v, c, (s', p_1) : e_\mathrm{p}) = \mathrm{loca}_\mathrm{e}(s', m(e'[x \mapsto (s', p_1)]), \mathrm{app}^{-1}.c, e_\mathrm{p}) \text{ if } s' \text{ is after } s, \quad \text{(A.3)}$$
$$\mathrm{loca}(s, v, c, (s', p_1) : e_\mathrm{p}) = \emptyset \text{ if } s' \text{ is not after } s,$$
$$\mathrm{loca}(s, v, c, []) = v \downarrow_s \text{ if } \epsilon \in c, \emptyset \text{ otherwise.} \quad \text{(A.4)}$$

  In the first case, we simply run the application of the function to the given parameter. The escaping locations, given by loca_e, will be defined below. If $\epsilon \in c$ and the parameter environment is empty, the function is not applied, the closure escapes, so all values in its environment are marked.

- The following equations define loca_e:

$$\text{if } f\ s = \bot,\ \mathrm{loca}_\mathrm{e}(s, f, c, e_\mathrm{p}) = \emptyset, \quad \text{(A.5)}$$
$$\text{if } f\ s = \mathrm{up}(v, s'),\ \mathrm{loca}_\mathrm{e}(s, f, c, e_\mathrm{p}) = a(s, s') \cup \mathrm{loca}(s', v, c, e_\mathrm{p}). \quad \text{(A.6)}$$

  When the expression does not terminate, we consider that nothing escapes. When the expression terminates, the stored locations $a(s, s')$ escape, as well as the part of the result corresponding to context c.

Intuitively, we follow the data structure of the value to look for marked locations. When the value is of a constructed type, it is enough to analyze the data structure to find escaping locations. For functions, on the contrary, we have to apply them.

The above definition of loca is in fact given by equations since data structures may be cyclic and therefore we may come back to the same value inside itself. The result can be computed by making explicit the fixed point iteration used to solve the equations as follows:

If $n = 0$, $\mathrm{loca}(n, s, v, c, e_\mathrm{p}) = \emptyset$. (A.7)

If $n > 0$, $\mathrm{getval}(s, v) = (C, v_1, ..., v_k)$, and $e_\mathrm{p} = []$ then $\mathrm{loca}(n, s, \ell, c, e_\mathrm{p}) =$

$(\{v \text{ if } \epsilon \in c \text{ and } v \neq (\_, U)\}) \cup \cup_{i \in [\![1,k]\!]} \mathrm{loca}(n-1, s, v_i, \overline{C_i}^{-1}.c, e_\mathrm{p})$. (A.8)

If $n > 0$, $\mathrm{getval}(s, v) = (\mathtt{fun}, m, e', x)$, and s' is after s then

$\mathrm{loca}(n, s, v, c, (s', p_1) : e_\mathrm{p}) = \mathrm{loca_e}(n-1, s', m(e'[x \mapsto (s', p_1)]), \mathrm{app}^{-1}.c, e_\mathrm{p})$. (A.9)

If $n > 0$, $\mathrm{getval}(s, v) = (\mathtt{fun}, m, e', x)$, and $e_\mathrm{p} = []$ then

$\mathrm{loca}(n, s, v, c, e_\mathrm{p}) = v \downarrow_s$ if $\epsilon \in c, \emptyset$ otherwise. (A.10)

If $n > 0$, in all other cases, $\mathrm{loca}(n, s, v, c, e_\mathrm{p}) = \emptyset$. (A.11)

If $f\ s = \perp$, $\mathrm{loca_e}(n, s, f, c, e_\mathrm{p}) = \emptyset$. (A.12)

If $f\ s = \mathrm{up}(v, s')$, $\mathrm{loca_e}(n, s, f, c, e_\mathrm{p}) = a(s, s') \cup \mathrm{loca}(n, s', v, c, e_\mathrm{p})$. (A.13)

$\mathrm{loca}(s, v, c, e_\mathrm{p}) = \cup_{n \in \mathbb{N}} \mathrm{loca}(n, s, v, c, e_\mathrm{p}), \mathrm{loca_e}(s, f, c, e_\mathrm{p}) = \cup_{n \in \mathbb{N}} \mathrm{loca_e}(n, s, f, c, e_\mathrm{p})$. (A.14)

With this definition, it is easy to prove that loca satisfies the above equations.

Lemma A.7 loca *is additive with respect to the context:*

$$\cup_i \mathrm{loca}(s, v, c_i, e_\mathrm{p}) = \mathrm{loca}(s, v, \cup_i c_i, e_\mathrm{p}).$$

Proof: The proof is an easy fixed point induction, following the definition of loca. □

Definition A.8 (Label filtering) Let $\mathrm{lbl}(s)$ be the set of labels that exist in store s:

$$\mathrm{lbl}((s', a, \Lambda)) = \Lambda.$$

We define label filtering by

$$\mathrm{filt}_{(s,e_\mathrm{p})}([]) = [], \mathrm{filt}_{(s,e_\mathrm{p})}(U) = U,$$

$$\mathrm{filt}_{(s,e_\mathrm{p})}(\lambda_2 : \lambda_1) = \begin{cases} \lambda_2 : \mathrm{filt}_{(s,e_\mathrm{p})}(\lambda_1) & \text{if } \lambda_2 \in \mathrm{lbl}(s), \\ \lambda_2 : \mathrm{filt}_{(s_\mathrm{p},[])}(\lambda_1) & \text{if } \exists i \in \mathrm{Dom}(e_\mathrm{p}), e_\mathrm{p}(i) = (s_\mathrm{p}, (v_\mathrm{p}, \lambda_2 : \lambda_3)), \\ \mathrm{filt}_{(s,e_\mathrm{p})}(\lambda_1) & \text{otherwise.} \end{cases}$$

$$\mathrm{filt}_{(s,e_\mathrm{p})}(X) = \{(\ell, \lambda') | (\ell, \lambda) \in X, \lambda' = \mathrm{filt}_{(s,e_\mathrm{p})}(\lambda)\}$$

The goal of $\mathrm{filt}_{(s,e_\mathrm{p})}(X)$ is to eliminate all labels which did not exist in store s or in the parameter environment e_p, from the set X of labelled locations (computed after store s).

Definition A.9 Let $X \in \mathcal{P}(Loc \times Label)$. Let $X_{|Labelled}$ be the set of labelled locations in X whose label is not empty. Let $X_{|\overline{e_\mathrm{p}}}$ be the set of labelled locations of X whose label does not begin with the label of an element of e_p. More precisely:

$$X_{|Labelled} = \{(\ell, \lambda) \in X | \lambda \neq []\}$$
$$X_{|\overline{e_\mathrm{p}}} = \{(\ell, \lambda : \lambda') \in X | \forall i \in \mathrm{Dom}(e_\mathrm{p}), e_\mathrm{p}(i) \neq (\_, (\_, \lambda : \_))\}$$

Lemma A.10 *If* $[\![M]\!]es = \mathrm{up}(\ell, s')$, $\mathrm{filt}_{(s,[])}(a(s,s') \cup \ell\downarrow_{s'})_{|Labelled} \subseteq e\downarrow_s$. *This means that only locations reachable from the environment can be reached during any computation.*

Proof: Looking at the semantics, we notice that we can ignore new labels created during the evaluation of M, since they will be removed by $\mathrm{filt}_{(s,[])}$. Let us define $L(e,s) = e\downarrow_s \cup \{(\ell,[])|\ell \in Loc - \mathrm{Dom}(s)\}$. $SV(e,s) = Constr \times V(e,s)^* \cup \{\mathtt{fun}\} \times (Env \to Exp) \times Env(e,s) \times Var$ and $V(e,s) = L(e,s) \cup \{(\ell,U)|\ell \in Loc\} \cup ((\{\mathrm{NIL}, \mathrm{true}, \mathrm{false}\} \cup \mathbb{Z}) \times Label)$. $Env(e,s) = \{e' \in Env | e'[\![x]\!] \in V(e,s)\}$. We can notice that all labelled values appearing in a computation are the ones yielded by $e[\![x]\!]$ where $x \in Var$, $\mathrm{getval}(s,v)$ where v is a labelled value and $\mathrm{newloc}(s,x)$. We prove, by induction on the execution trace, that for all labelled values v appearing in the computation, for all stores s' after s, we have $v\downarrow_{s'} \subseteq L(e,s)$. Then $v \in V(e,s)$. We also prove that for all environments e' appearing in the computation, $e'\downarrow_{s'} \subseteq L(e,s)$. These two proofs are performed together by induction on the execution trace of the evaluation of $[\![M]\!]es$, and by cases on the elementary steps of this execution trace.

- Initial step: for the environment e, $e\downarrow_s \subseteq L(e,s)$.
- Case 1: a new environment is created by $e' = e[x \mapsto (s,v)]$ with $x \in Var$, $v \in V(e,s)$. Then $v\downarrow_s \subseteq L(e,s)$, and $e'\downarrow_s \subseteq L(e,s)$.
- Case 2: the store is changed by an assignment, $s' = (s_2, a_2, \Lambda) = (s_1[x \mapsto (C, v_1, ..., v_{i-1}, (y,U), v_{i+1}, ..., v_k)], a_1 \cup (y,\lambda_y)\downarrow_s, \Lambda)$ where $s = (s_1, a_1, \Lambda)$, $(y,\lambda_y) \in V(e,s)$, $(x,\lambda_x) \in L(e,s)$, $(C, v_1, ..., v_k) = s_1(x)$. Let v be any value. If $v\downarrow_{s'}$ is different from $v\downarrow_s$, the stored locations have label U, and they are therefore not included in $v\downarrow_{s'}$. Therefore $v\downarrow_{s'} \subseteq v\downarrow_s$. Assuming by induction that $v\downarrow_s \subseteq L(e,s)$, we have $v\downarrow_{s'} \subseteq L(e,s)$.
- Case 3: a value is obtained from the environment e' by $e'[\![x]\!]$. Assuming (by induction) $e'\downarrow_s \subseteq L(e,s)$, $e'[\![x]\!]\downarrow_s \subseteq L(e,s)$.
- Case 4: a value is obtained by $\mathrm{getval}(v,s)$. By induction hypothesis, $v\downarrow_s \subseteq L(e,s)$. If $(C, v_1, ..., v_k) = \mathrm{getval}(v,s)$, then $v_i\downarrow_s \subseteq L(e,s)$. If $(\mathtt{fun}, m, e', x) = \mathrm{getval}(v,s)$, then $e'\downarrow_s \subseteq L(e,s)$ (obvious by definition of $\downarrow_s$).
- Case 5: a value is obtained by $\mathrm{newloc}(s,v)$. If $\mathrm{up}(v',s') = \mathrm{newloc}(s,v)$ with $v \in SV(e,s)$ implies $v' = (\ell,[])$ with $s' = (s_1[\ell \mapsto v], a)$, $s = (s_1, a)$. We have $\ell \in Loc - \mathrm{Dom}(s)$ therefore $(\ell,[]) \in L(e,s)$. Since $v \in SV(e,s)$, if a value w appears in v then $w \in V(e,s)$ and $w\downarrow_{s'} \subseteq L(e,s)$. $v'\downarrow_{s'} \subseteq L(e,s)$. Of course, $x\downarrow_{s'} = x\downarrow_s \subseteq L(e,s)$ for all other values x.

Since for all appearing labelled values v, $v\downarrow_s \subseteq L(e,s)$,

$$\mathrm{filt}_{(s,[])}(a(s,s') \cup \ell\downarrow_s) \subseteq \mathrm{filt}_{(s,[])}(L(e,s)) = e\downarrow_s \cup \{(\ell,[])|\ell \in Loc - \mathrm{Dom}(s)\}.$$

$\square$

Lemma A.11 $\mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s,v,c,e_\mathrm{p}))_{|Labelled} \subseteq v\downarrow_s \cup \mathrm{loc}(e_\mathrm{p})$.

Proof: By fixed point induction following the definition of loca. The result is obvious at iteration $n = 0$ since $\mathrm{loca}(0,s,v,c,e_\mathrm{p}) = \emptyset$ by (A.7). Let us assume that the result is true at iteration n.

The only non-trivial case is the one of closures when $e_{\mathrm{p}} = (s', p_1) : e'_{\mathrm{p}}$, and s' is after s. Let $\mathrm{getval}(s, v) = (\mathtt{fun}, m, e', x)$, and $m(e'[x \mapsto (s', p_1)])s = \mathrm{up}(v', s')$

$$\begin{aligned} X &= \mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{loca}(n, s, v, c, e_{\mathrm{p}}))_{|Labelled} \\ &= \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s', s'') \cup \mathrm{loca}(n-1, s'', v', \mathrm{app}^{-1}.c, e'_{\mathrm{p}}))_{|Labelled} && \text{by (A.9) and (A.13)} \\ &= \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s', s'') \cup \mathrm{filt}_{(s'',e'_{\mathrm{p}})}(\mathrm{loca}(n-1, s'', v', \mathrm{app}^{-1}.c, e'_{\mathrm{p}})))_{|Labelled} \end{aligned}$$

because all labels preserved by $\mathrm{filt}_{(s,e_{\mathrm{p}})}$ are also preserved by $\mathrm{filt}_{(s'',e'_{\mathrm{p}})}$ because s'' is after s and s'

$$\begin{aligned} X &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s', s'') \cup v' \downarrow_{s''} \cup \mathrm{loc}(e'_{\mathrm{p}}))_{|Labelled} && \text{by induction hypothesis} \\ &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{filt}_{(s',[])}(a(s', s'') \cup v' \downarrow_{s''}) \cup \mathrm{loc}(e'_{\mathrm{p}}))_{|Labelled} \end{aligned}$$

for a similar reason (No $\lambda \in Lbl$, first label of an element of e'_{p}, may appear in $a(s', s'') \cup v' \downarrow_{s''}$ because first labels of elements of e'_{p} have been created after s''. Labels of $a(s', s'') \cup v' \downarrow_{s''}$ which appear in s or in the store s' of the first parameter of e_{p} therefore also appear in s', hence we can add $\mathrm{filt}_{(s',[])}$ without changing the result.)

$$\begin{aligned} X &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}((e'[x \mapsto (s', p_1)]) \downarrow_{s'}) \cup \mathrm{loc}(e'_{\mathrm{p}}) && \text{by Lemma A.10} \\ &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(e' \downarrow_{s'}) \cup \mathrm{filt}_{(s,e_{\mathrm{p}})}(p_1 \downarrow_{s'}) \cup \mathrm{loc}(e'_{\mathrm{p}}) \\ &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(e' \downarrow_{s'}) \cup \mathrm{filt}_{(s',[])}(p_1 \downarrow_{s'}) \cup \mathrm{loc}(e'_{\mathrm{p}}) \end{aligned}$$

since all labels in $p_1 \downarrow_{s'}$ begin with the label of p_1, and are therefore filtered in store s' by definition of $\mathrm{filt}_{(s,e_{\mathrm{p}})}$.

$$\begin{aligned} X &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(v \downarrow_{s'} \cup \mathrm{loc}((s', p_1) : e'_{\mathrm{p}})) \\ &\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(v \downarrow_{s} \cup \mathrm{loc}(e_{\mathrm{p}})) \end{aligned}$$

since, if $v \downarrow_{s'}$ is different from $v \downarrow_{s}$, this comes from an assignment between s and s', and in this case, the stored locations have label U, and they are therefore not included in $v \downarrow_{s'}$. Therefore $v \downarrow_{s'} \subseteq v \downarrow_{s}$.

We have therefore $\mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{loca}(n, s, v, c, e_{\mathrm{p}}))_{|Labelled} \subseteq v \downarrow_{s} \cup \mathrm{loc}(e_{\mathrm{p}})$. Taking the upper bound on $n \in \mathbb{N}$, we get the expected result. □

Lemma A.12 *If s' is after s in the execution trace,* $\mathrm{loca}(s', v, c, e_{\mathrm{p}}) \subseteq \mathrm{loca}(s, v, c, e_{\mathrm{p}})$. *This is useful to make sure that if the escaping part is estimated in a given store, the analysis will remain correct in a future store (in particular, when the analyzed expression will be used).*

Proof: We distinguish two cases.

- **Case** $e_{\mathrm{p}} \neq []$. By definition of loca, if $e_{\mathrm{p}} = (s'', p_1) : e'_{\mathrm{p}}$,

$$\mathrm{loca}(s, v, c, e_{\mathrm{p}}) = \begin{cases} \mathrm{loca}_{\mathrm{e}}(s'', m(e[x \mapsto (s'', p_1)]), \mathrm{app}^{-1}.c, e'_{\mathrm{p}}) & \text{if } \mathrm{getval}(s, v) = (\mathtt{fun}, m, e, x) \text{ and } s'' \text{ is after } s. \\ \emptyset & \text{otherwise.} \end{cases}$$

Closures are not mutable, therefore $\mathrm{getval}(s, v)$ does not change between s and s'. Therefore if s and s' are both before s'', $\mathrm{loca}(s', v, c, e_\mathrm{p}) = \mathrm{loca}(s, v, c, e_\mathrm{p})$. Otherwise, s' is after s'' and $\mathrm{loca}(s', v, c, e_\mathrm{p}) = \emptyset \subseteq \mathrm{loca}(s, v, c, e_\mathrm{p})$. Then if $e_\mathrm{p} \neq []$, $\mathrm{loca}(s', v, c, e_\mathrm{p}) \subseteq \mathrm{loca}(s, v, c, e_\mathrm{p})$.

- **Case** $e_\mathrm{p} = []$. We perform a fixed point induction.
  For $n = 0$, $\mathrm{loca}(0, s', v, c, []) = \emptyset = \mathrm{loca}(0, s, v, c, [])$.
  If $v \in \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z}$, $\mathrm{loca}(n, s', v, c, []) = \emptyset = \mathrm{loca}(n, s, v, c, [])$.
  If $\mathrm{getval}(s', v) = (C, v'_1, ..., v'_k)$ and $\mathrm{getval}(s, v) = (C, v_1, ..., v_k)$,

$$\mathrm{loca}(n, s', v, c, []) = (\{v \text{ if } \epsilon \in c \text{ and } v \neq (\_, U)\}) \cup \cup_{i \in [\![1,k]\!]} \mathrm{loca}(n-1, s', v'_i, \overline{C_i}^{-1}.c, []).$$

If v has been modified between s and s', and $v_i \neq v'_i$, then $\mathrm{loca}(s', v'_i, \overline{C_i}^{-1}.c, []) \subseteq v'_i \downarrow'_s = \emptyset$ by Lemma A.11 (v'_i has label U). If $v_i = v'_i$,

$$\mathrm{loca}(n-1, s', v'_i, \overline{C_i}^{-1}.c, []) = \mathrm{loca}(n-1, s', v_i, \overline{C_i}^{-1}.c, []) \subseteq \mathrm{loca}(n-1, s, v_i, \overline{C_i}^{-1}.c, [])$$

by induction hypothesis. Hence, we have the result:

$$\begin{aligned}\mathrm{loca}(n, s', v, c, []) &\subseteq (\{v \text{ if } \epsilon \in c \text{ and } v \neq (\_, U)\}) \cup \cup_{i \in [\![1,k]\!]} \mathrm{loca}(n-1, s, v_i, \overline{C_i}^{-1}.c, []) \\ &= \mathrm{loca}(n, s, v, c, []).\end{aligned}$$

If $\mathrm{getval}(s', v) = (\mathtt{fun}, m, e', x)$, $\mathrm{getval}(s, v) = (\mathtt{fun}, m, e', x)$ since closures are not mutable,

$$\mathrm{loca}(n, s', v, c, []) = v \downarrow_{s'} \text{ if } \epsilon \in c, \emptyset \text{ otherwise.}$$

As we have noticed in the proof of Lemma A.11, $v \downarrow_{s'} \subseteq v \downarrow_s$ hence the result. □

Lemma A.13 $\mathrm{loca}(s, v, \mathit{Path}, []) = v \downarrow_s$. *Then* $\mathrm{filt}_{(s,[])}(\mathrm{loca}(s, v, \mathit{Path}, [])) = v \downarrow_s$.

Proof: Let us define $G(s, v) = \mathrm{loca}(s, v, \mathit{Path}, [])$. The definition of loca simplifies into:

If $v \in \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z}$, $G(s, v) = \emptyset$
If $\mathrm{getval}(s, \ell) = (C, \ell_1, ..., \ell_k)$, $G(s, \ell) = \{\ell \text{ if } \ell \neq (\_, U)\} \cup \cup_{i \in [\![1,k]\!]} G(s, \ell_i)$
If $\mathrm{getval}(s, \ell) = (\mathtt{fun}, m, e', x)$, $G(s, \ell) = \ell \downarrow_s$

We can easily see that $G(s, v) \subseteq v \downarrow_s$. Moreover $v \downarrow_s \subseteq G(s, v)$ since $G(s, v)$ satisfies the equations defining $v \downarrow_s$. Then $v \downarrow_s = G(s, v) = \mathrm{loca}(s, v, \mathit{Path}, [])$ which gives the required result. □

We now define another concretization:

Definition A.14 Let dest : $(\mathit{Store} \times \mathit{LabelVal}) \times \mathit{Ctx}_E \to \mathcal{P}(\mathit{Loc} \times \mathit{Label})$ be defined by

$$\mathrm{dest}((s, v), c) = \cup_{e_\mathrm{p} \in \mathit{Env}_\mathrm{p}} \mathrm{filt}_{(s, e_\mathrm{p})}(\mathrm{loca}(s, v, c, e_\mathrm{p}))_{|\overline{e_\mathrm{p}}}.$$

Notice that by definition of loca, it is enough to consider parameter environments e_p such that the store s_p of the first element of e_p is after s (otherwise, $\mathrm{loca}(s, v, c, e_\mathrm{p}) = \emptyset$).

Remark. If v is not a function, $\mathrm{dest}((s,v),c) = \mathrm{loca}(s,v,c,[])_{|Labelled}$. Indeed, we have obviously $\mathrm{dest}((s,v),c) \supseteq \mathrm{filt}_{(s,[])}(\mathrm{loca}(s,v,c,[]))_{|Labelled}$ (just consider the term $e_\mathrm{p} = []$ in the definition of dest). Conversely, $\mathrm{dest}((s,v),c) \subseteq \mathrm{filt}_{(s,[])}(\mathrm{loca}(s,v,c,[]))_{|Labelled}$ since we can see on the definition of loca that $\mathrm{loca}(s,v,c,e_\mathrm{p}) \subseteq \mathrm{loca}(s,v,c,[])$ for values which are not functions.

Lemma A.15 $\mathrm{dest}((s,v),c) \subseteq v\downarrow_s$.

Proof: The result immediately follows from Lemma A.11, since all labels of locations in $\mathrm{loc}(e_\mathrm{p})$ begin with the label of an element $e_\mathrm{p}(i)$ of e_p and these labels are excluded by definition of dest. □

Such a property can be expected of a reasonable concretization: the locations represented by a marking of a part of the value are locations reachable from the value. It is however not true for loca because it may also contain locations of e_p.

Lemma A.16 $\mathrm{dest}((s,v),Path) = v\downarrow_s$.

Proof: By Lemma A.15, $\mathrm{dest}((s,v),c) \subseteq v\downarrow_s$. The converse inclusion comes from Lemma A.13. □

Lemma A.17

$$\begin{aligned}\mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s,v,c,e_\mathrm{p}))_{|Labelled} &\subseteq \mathrm{dest}((s,v),c) \cup \mathrm{loc}(e_\mathrm{p})\\ &\subseteq \mathrm{dest}((s,v),c) \cup \cup_{i\in\mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i),Path)\end{aligned}$$

Proof: If a store in e_p is before s, then $\mathrm{loca}(s,v,c,e_\mathrm{p}) = \emptyset$, and the result is true. Otherwise, by Lemma A.11,

$$X = \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s,v,c,e_\mathrm{p}))_{|Labelled} \subseteq v\downarrow_s \cup \mathrm{loc}(e_\mathrm{p}).$$

Moreover, no locations in $v\downarrow_s$ have a label beginning with the first label of an element of e_p since such labels do not exist yet in store s. Then

$$\{(v,\lambda:\lambda_2) \in X | \exists i, e_\mathrm{p}(i) = (s_\mathrm{p},(v_\mathrm{p},\lambda:\lambda_3))\} \subseteq \mathrm{loc}(e_\mathrm{p}).$$

By definition of dest, $\{(v,\lambda:\lambda_2) \in X | \forall i, e_\mathrm{p}(i) \neq (s_\mathrm{p},(v_\mathrm{p},\lambda:\lambda_3))\} \subseteq \mathrm{dest}((s,v),c)$. Taking the upper bound of these equations, we have: $X \subseteq \mathrm{dest}((s,v),c) \cup \mathrm{loc}(e_\mathrm{p})$.

The second result comes immediately from the first one, since we have $\mathrm{loc}(e_\mathrm{p}) = \cup_{i\in\mathrm{Dom}(e_\mathrm{p}),e_\mathrm{p}(i)=(s_\mathrm{p},v_\mathrm{p})} v_\mathrm{p}\downarrow_{s_\mathrm{p}}$, and $\mathrm{dest}(e_\mathrm{p}(i),Path) = v_\mathrm{p}\downarrow_{s_\mathrm{p}}$ by Lemma A.16. □

Lemma A.18 $\cup_i \mathrm{dest}((s,v),c_i) = \mathrm{dest}((s,v),\cup_i c_i)$.

Proof: This comes immediately from the additivity of loca (Lemma A.7). □

Theorem A.19 *Let us consider an instance of variable x in an expression of the form:* `let` x `=` M `in` N *or* `let rec` x = `fun` $y \to M$ `in` N. *Let s_x be the store at the beginning of the evaluation of N, e the environment given to N. Let us assume that the location ℓ at the top of x is new when x is defined (i.e. the creation of this location is the last operation of M in the* `let` *case, and ℓ is the location of the closure created by* `let rec` *in the* `let rec` *case). Let λ be the label created during the evaluation of the* `let` *or* `let rec`. *If $\forall \lambda_1 \in Lbl^*, (\ell,\lambda:\lambda_1) \notin \mathrm{filt}_{(s_x,[])}(\mathrm{loca_e}(s_x,[\![N]\!]e,Path,[]))$, then ℓ can be stack allocated.*

Proof: If $[\![N]\!]es_x = \mathrm{up}(r, s'_x)$, we have:

$$\begin{aligned}\mathrm{filt}_{(s_x,[])}(\mathrm{loca_e}(s_x, [\![N]\!]e, \mathit{Path}, [])) &= \mathrm{filt}_{(s_x,[])}(a(s_x, s'_x) \cup \mathrm{loca}(s'_x, r, \mathit{Path}, [])) && \text{by (A.6)}\\ &= \mathrm{filt}_{(s_x,[])}(a(s_x, s'_x) \cup r \downarrow_{s'_x}) && \text{by Lemma A.13.}\end{aligned}$$

Assume that $(\ell, \lambda') \in \mathrm{filt}_{(s_x,[])}(a(s_x, s'_x) \cup r \downarrow_{s'_x})$. By Lemma A.10, $(\ell, \lambda') \in \mathrm{filt}_{(s_x,[])}(e \downarrow_{s_x})$. For all $z \neq x$, $(\ell, \lambda') \notin \mathrm{filt}_{(s_x,[])}(e[\![z]\!] \downarrow_{s_x})$ (ℓ is not reachable from any variable other than x). Then $(\ell, \lambda') \in \mathrm{filt}_{(s_x,[])}(e[\![x]\!] \downarrow_{s_x}) = \mathrm{filt}_{(s_x,[])}((\ell, [\lambda]) \downarrow_{s_x})$. During the computation of $\downarrow_{s_x}$, the function $\mathrm{getval}(s, v)$ always appends labels. Therefore $\exists \lambda_1 \in \mathit{Lbl}^*$ such that $\lambda' = \lambda : \lambda_1$ (Lemma A.1). By hypothesis, $(\ell, \lambda : \lambda_1) \notin \mathrm{filt}_{(s_x,[])}(\mathrm{loca_e}(s_x, [\![N]\!]e, \mathit{Path}, []))$. Therefore $(\ell, \lambda') \notin \mathrm{filt}_{(s_x,[])}(a(s_x, s'_x) \cup r \downarrow_{s'_x})$

If ℓ was reachable from the result r, then either ℓ has label U, or $(\ell, \lambda') \in r \downarrow_{s'_x}$. As we have just seen, the second case is impossible. For the first case, if ℓ has label U, it has been stored by an assignment. The first time ℓ has been stored by an assignment, it had not yet a U label, then it has been added to $a(s_x, s'_x)$. Therefore, the first case is also impossible.

If ℓ is reachable from another variable in the environment, then it has been stored in it by an assignment, and we have just seen that it is impossible.

Therefore ℓ is only reachable from x in store s'_x. When x is discarded, ℓ is not reachable any more and can be deallocated. □

Definition A.20 (Correctness) The correctness is defined in several steps:

- **For a value v:** ϕ is a correct abstraction of the semantics of a value v in store s if $\mathrm{corr_v}(s, v, \phi)$.

$$\begin{aligned}&\mathrm{corr_v} : \mathit{Store} \times \mathit{Val} \times \mathit{Val}_E \to \{\mathrm{true}, \mathrm{false}\}\\ &\mathrm{corr_v}(s, v, \phi) \Leftrightarrow \forall c \in \mathit{Ctx}_E, \forall e_\mathrm{p} \in \mathit{Env}_\mathrm{p},\\ &\quad \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s, v, c, e_\mathrm{p}))_{|\mathit{Labelled}} \subseteq \mathrm{dest}((s, v), c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c).\end{aligned} \tag{A.15}$$

  This correctness criterion only deals with the parameters of functions: the context c for value v is always correct (since we precisely mark locations corresponding to context c in $\mathrm{loca}(s, v, c, e_\mathrm{p})$, see Lemma A.17). This formula expresses that if we apply function v to parameters in e_p, then the escaping locations (those in $\mathrm{loca}(s, v, c, e_\mathrm{p})$ that are either in e_p or existed in store s) are correctly taken into account by the escape function ϕ.

- **For an expression M:** ϕ is a correct abstraction of the semantics $f = [\![M]\!]e$ of an expression M in environment e if $\mathrm{corr_e}(e, s, f, \phi)$.

$$\begin{aligned}&\mathrm{corr_e} : \mathit{Env} \times \mathit{Store} \times \mathit{Exp} \times \mathit{Exp}_E \to \{\mathrm{true}, \mathrm{false}\}\\ &\mathrm{corr_e}(e, s, f, \phi) \Leftrightarrow \forall c \in \mathit{Ctx}_E, \forall e_\mathrm{p} \in \mathit{Env}_\mathrm{p},\\ &\quad \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca_e}(s, f, c, e_\mathrm{p}))_{|\mathit{Labelled}}\\ &\qquad \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c).\end{aligned} \tag{A.16}$$

  Let us recall that $e(x)$ is a pair (store when x has been defined, value of x). Our semantics is instrumented by replacing $e[x \mapsto \ell]$ with $e[x \mapsto (s, \ell)]$ to also memorize the store when the variable has been created.

  Intuitively, we mark the part $\mathrm{loca_e}(s, f, c, e_\mathrm{p})$ of the result, represented by context c, and the stored locations, which are regarded as escaping. We look for marked locations in variables and parameters of f.

- **For the whole analysis:** The analysis is correct if it gives correct results when it is given a correct environment, that is:

$$\begin{aligned}
&\mathrm{corr} : (\mathit{Env} \to \mathit{Exp}) \times (\mathit{Env}_E \to \mathit{Exp}_E) \to \{\mathrm{vrai}, \mathrm{faux}\} \\
&\mathrm{corr}(f, \phi) \Leftrightarrow \forall s, \forall e \in \mathit{Env}, \forall \rho \in \mathit{Env}_E, \forall y \in \mathrm{Dom}(e), \mathrm{corr_v}(s, e[\![y]\!], \rho[\![y]\!]) \\
&\quad \Rightarrow \mathrm{corr_e}(e, s, fe, \phi\rho)
\end{aligned} \tag{A.17}$$

where $\phi\rho$ is defined on $\mathrm{Dom}(e) \cup \mathit{Ind}$. The goal is to prove that $\mathrm{corr}([\![M]\!], E[\![M]\!])$.

Example A.21 For example,

```
let x = (true :: []) :: [] in
let y = hd x in
  y
```

Let e and s be the environment and the store during the evaluation of `y`. Let x_1 and $x_2 = y$ be the locations such that $e[\![\mathtt{x}]\!] = x_1$, $e[\![\mathtt{y}]\!] = y$, $s(x_1) = (::, x_2, x_3)$, $s(x_2) = (::, \mathrm{true}, x_4)$. $s(x_3) = s(x_4) = ([])$ (empty list). We are going to compute $\phi\mathtt{x}\mathit{Path}$ such that $\mathrm{corr_v}(s, y, \phi)$.

We have $\mathrm{loca}(s, y, \mathit{Path}, []) = \{x_2, x_4\}$, $\mathrm{dest}((s, x_1), \mathtt{hd}.\mathit{Path}) = \{x_2, x_4\}$ and $\mathrm{dest}((s, x_1), \mathtt{hd}.\mathtt{hd}.\mathit{Path}) = \emptyset$ therefore $\phi\mathtt{x}\mathit{Path} = \mathtt{hd}.\mathit{Path}$.

Remark. The existence of the least element of the set of correct contexts is not obvious. For example, if we have a pair $x = (a, a)$, and a escapes, both $\mathtt{fst}.\mathit{Path}$ and $\mathtt{snd}.\mathit{Path}$ are incomparable correct contexts. However, if we look at the expression using x, for a value of x different from (a, a), we will know which part of the pair really escapes. In particular, the least element exists when there is initially no alias (several paths leading to the same location). The difficulty is to convince oneself that escaping paths found in this particular case are also correct in the general case. That is why we have not defined an abstraction $\alpha : \mathcal{P}(\mathit{Exp}) \to \mathit{Exp}_E$. We use the correctness relation framework of [CC92], is which the existence of a least correct element is not necessary.

The following lemma intuitively means that, if ϕ is a correct escape function for an expression M, it is also a correct escape function for the result of M (considered as a value). Notice that we have to add a label λ_2 to get this result: $\mathrm{corr_e}(e, s, [\![M]\!]e, \phi) \Rightarrow \mathrm{corr_v}(s'', (v, \lambda_2 : \lambda_1), \phi)$ where $[\![M]\!]es = \mathrm{up}((v, \lambda_1), s')$ and s'' is after s'. This result will be useful in the $\mathtt{let}\ x = M\ \mathtt{in}\ N$ case of the proof, to show that a correct escape function for M is also a correct escape function for x.

Lemma A.22 *If*

$$\mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s', (v, \lambda_1), c, e_\mathrm{p}))_{|\mathit{Labelled}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi xc) \cup \cup_{i \in \mathrm{Dom}(e)} \mathrm{dest}(e(i), \phi ic),$$

then

$$\begin{aligned}
&\mathrm{filt}_{(s'',e_\mathrm{p})}(\mathrm{loca}(s'', (v, \lambda_2 : \lambda_1), c, e_\mathrm{p}))_{|\mathit{Labelled}} \\
&\quad \subseteq \mathrm{dest}((s'', (v, \lambda_2 : \lambda_1)), c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi ic)
\end{aligned}$$

where s'' is after s' which is after s, λ_2 is a label created after s and before s'', variables in e have been bound before s.

Proof: If the first element of e_p, $e_p(1) = (s_p, v_p)$ has a store s_p which has been created before s'', then the result is obvious, since $\mathrm{loca}(s'', (v, \lambda_2 : \lambda_1), c, e_p) = \emptyset$. We assume now that the store s_p is after s''.

Let $X = \mathrm{filt}_{(s'',e_p)}(\mathrm{loca}(s'', (v, \lambda_2 : \lambda_1), c, e_p))$. Let $Y_{|e_p} = \{(v, \lambda' : \lambda'') \in Y | \exists i \in \mathrm{Dom}(e_p), e_p(i) = (s_p, (v_p, \lambda' : \lambda'''))\}$. By Lemma A.11,

$$\begin{aligned} X_{|e_p} &= \mathrm{filt}_{(s'',e_p)}(\mathrm{loca}(s'', (v, \lambda_2 : \lambda_1), c, e_p))_{|e_p} \\ &\subseteq ((v, \lambda_2 : \lambda_1) \downarrow_{s''} \cup \mathrm{loc}(e_p))_{|e_p} \\ &\subseteq \mathrm{loc}(e_p) \end{aligned}$$

since λ_2 is not the first label of an element of e_p. (The first labels of elements of e_p are created after s''.) Then all labels in $X_{|e_p}$ are preserved by $\mathrm{filt}_{(s,e_p)}$, then

$$X_{|e_p} = \mathrm{filt}_{(s,e_p)}(X_{|e_p}) = \mathrm{filt}_{(s,e_p)}(X_{|e_p})_{|e_p}.$$

Then

$$\begin{aligned} X_{|e_p} &\subseteq \mathrm{filt}_{(s,e_p)}(\mathrm{loca}(s'', (v, \lambda_2 : \lambda_1), c, e_p))_{|e_p} \\ &\subseteq \mathrm{filt}_{(s,e_p)}(\mathrm{loca}(s'', (v, \lambda_1), c, e_p))_{|e_p} && \text{since } \lambda_2 \text{ is removed by } \mathrm{filt}_{(s,e_p)} \\ &\subseteq \mathrm{filt}_{(s,e_p)}(\mathrm{loca}(s', (v, \lambda_1), c, e_p))_{|e_p} && \text{by Lemma A.12} \\ &\subseteq (\cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \cup_{i \in \mathrm{Dom}(e_p)} \mathrm{dest}(e_p(i), \phi i c))_{|e_p} && \text{by hypothesis} \\ &\subseteq \cup_{i \in \mathrm{Dom}(e_p)} \mathrm{dest}(e_p(i), \phi i c) \end{aligned}$$

since all labels in $\cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c)$ have been created before binding the variables in e (i.e. before s), and therefore before the stores of elements of e_p (which are after s''). Therefore, they do not contain first labels of elements of e_p.

Now, let $Y_{|\lambda_2} = \{(v, \lambda_2 : \lambda) \in Y\}$. We have

$$X_{|\lambda_2} \subseteq X_{|\overline{e_p}} \subseteq \mathrm{dest}((s'', (v, \lambda_2 : \lambda_1)), c)$$

by definition of dest. Then by Lemma A.11, $X \subseteq (v, \lambda_2 : \lambda_1) \downarrow_{s''} \cup \mathrm{loc}(e_p)$, therefore labels in X begin either by λ_2 (first term) or by the first label of an element of e_p (second term). Then

$$X \subseteq X_{|\lambda_2} \cup X_{|e_p} \subseteq \mathrm{dest}((s'', (v, \lambda_2 : \lambda_1)), c) \cup \cup_{i \in \mathrm{Dom}(e_p)} \mathrm{dest}(e_p(i), \phi i c).$$

This is the expected result. □

Lemma A.23 *If* $\mathrm{corr_e}(e, s, f, \phi)$ *and* $f\ s = \mathrm{up}(v, s')$, *then*

$$\mathrm{filt}_{(s,[])}(a(s, s'))_{|\mathit{Labelled}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x \emptyset), \tag{A.18}$$

$$\mathrm{filt}_{(s,[])}(a(s, s') \cup v \downarrow_{s'})_{|\mathit{Labelled}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x \mathit{Path}), \tag{A.19}$$

$$\mathrm{filt}_{(s,[])}(a(s, s') \cup \mathrm{dest}((s', v), c))_{|\mathit{Labelled}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c). \tag{A.20}$$

If $f\ s' = \mathrm{up}(v, s'')$, *and* s *is after all states in the environment* e, *and before* s',

$$\begin{aligned} &\mathrm{filt}_{(s,e_p)}(a(s', s'') \cup \mathrm{loca}(s'', v, c, e_p))_{|\mathit{Labelled}} \\ &\quad \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \cup_{i \in \mathrm{Dom}(e_p)} \mathrm{dest}(e_p(i), \phi i c). \end{aligned} \tag{A.21}$$

Proof: Applying corr_e with $c = \emptyset$ and $e_\mathrm{p} = []$, and taking into account the definition of loca_e (A.6), we easily get (A.18).

Applying corr_e with $c = \mathit{Path}$ and $e_\mathrm{p} = []$, and taking into account the definition of loca_e (A.6), we get: $\mathrm{filt}_{(s,[])}(a(s,s') \cup \mathrm{loca}(s', v, \mathit{Path}, []))_{|\mathit{Labelled}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x \mathit{Path})$. By Lemma A.13, we get (A.19).

By definition of corr_e, $\forall e_\mathrm{p}$,

$$\begin{aligned}&\mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s') \cup \mathrm{loca}(s', v, c, e_\mathrm{p}))_{|\mathit{Labelled}}\\ &\subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c).\end{aligned}$$

All labels of locations in $\cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c)$ begin with the first label of an element of e_p. Therefore,

$$\mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s') \cup \mathrm{loca}(s', v, c, e_\mathrm{p}))_{|\overline{e_\mathrm{p}}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c).$$

By definition of dest, considering only parameter environments such that the store of the first element of e_p is after s',

$$\cup_{e_\mathrm{p}} \mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s') \cup \mathrm{dest}((s', v), c))_{|\overline{e_\mathrm{p}}} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c).$$

All first labels of elements of e_p are created after s', therefore they do not appear in $a(s,s')$ or $\mathrm{dest}((s',v),c)$, and the formula simplifies into:

$$\mathrm{filt}_{(s,[])}(a(s,s'))_{|\mathit{Labelled}} \cup \mathrm{dest}((s',v),c) \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c).$$

For (A.21), since s is before s', all labels removed by $\mathrm{filt}_{(s',e_\mathrm{p})}$ were already removed by $\mathrm{filt}_{(s,e_\mathrm{p})}$, and $\mathrm{filt}_{(s,e_\mathrm{p})}(X) = \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{filt}_{(s',e_\mathrm{p})}(X))$. Then

$$\begin{aligned}&\mathrm{filt}_{(s,e_\mathrm{p})}(a(s',s'') \cup \mathrm{loca}(s'', v, c, e_\mathrm{p}))_{|\mathit{Labelled}}\\ &= \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{filt}_{(s',e_\mathrm{p})}(a(s',s'') \cup \mathrm{loca}(s'', v, c, e_\mathrm{p})))_{|\mathit{Labelled}}\\ &\subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(\cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c))_{|\mathit{Labelled}}\\ &\qquad\qquad \text{by correctness and (A.6)}\\ &\subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c)\end{aligned}$$

because, since s is after all states in e, $\mathrm{filt}_{(s,e_\mathrm{p})}$ does not change anything to $\cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x x)$ (labels in $\mathrm{dest}(e(x), \phi x c)$ are labels that existed in the state s_x of $e(x) = (s_x, v_x)$). Moreover, because we filter with respect to e_p, labels in $\cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \phi i c)$ do not change either. □

Lemma A.24 $\mathrm{corr}_\mathrm{v}(s, v, \phi) \Rightarrow \mathrm{corr}_\mathrm{v}(s', v, \phi)$ *if s' is after s in the execution trace.*

Proof: By Lemma A.12, $\mathrm{loca}(s', v, c, e_\mathrm{p}) \subseteq \mathrm{loca}(s, v, c, e_\mathrm{p})$. Therefore $\mathrm{filt}_{(s',e_\mathrm{p})}(\mathrm{loca}(s', v, c, e_\mathrm{p})) \subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s, v, c, e_\mathrm{p}))$ because, by Lemma A.11, labels of labelled locations in $\mathrm{filt}_{(s',e_\mathrm{p})}(\mathrm{loca}(s, v, c, e_\mathrm{p}))$ are either in $\mathrm{filt}_{(s',e_\mathrm{p})}(v \downarrow_{s'})$ (in which case, since v already exists in store s, $v \downarrow_{s'}$ contains no labels created between s and s' and therefore the labels are filtered in the same way by $\mathrm{filt}_{(s',e_\mathrm{p})}$ and $\mathrm{filt}_{(s,e_\mathrm{p})}$) or in $\mathrm{loc}(e_\mathrm{p})$ (in which case they are also filtered in the same way by $\mathrm{filt}_{(s',e_\mathrm{p})}$ and $\mathrm{filt}_{(s,e_\mathrm{p})}$). This gives the required implication on corr_v using the definition of corr_v. □

Theorem A.25 *Analysis E is correct* $\mathrm{corr}([\![M]\!], E[\![M]\!])$, *and the initial environment* ρ_0 *is correct with respect to* e_0*:* $\forall y \in \{\overline{C_i}, C?, \mathtt{true}, \mathtt{false}\} \cup \mathbb{Z}, \mathrm{corr_v}(s, e_0[\![y]\!], \rho_0[\![y]\!])$. *Moreover, we will have* $\forall y \in \mathrm{Dom}(e), \mathrm{corr_v}(s, e[\![y]\!], \rho[\![y]\!])$. *Then we have* $\mathrm{corr_e}(e, s, [\![M]\!]e, E[\![M]\!]\rho)$.

Proof: Let us first prove $\mathrm{corr}([\![M]\!], E[\![M]\!])$. We assume $\forall y \in \mathrm{Dom}(e), \mathrm{corr_v}(s, e[\![y]\!], \rho[\![y]\!])$ and prove $\mathrm{corr_e}(e, s, [\![M]\!]e, E[\![M]\!]\rho)$. The proof is obviously by induction of the syntax and by cases. Non-terminating cases are omitted since they are obvious: nothing escapes in these cases. Let $A = \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca_e}(s, [\![M]\!]e, c, e_\mathrm{p}))_{|Labelled}$.

- **Case** y**.**

$$[\![y]\!]es = \mathrm{up}(e[\![y]\!], s)$$

$$\begin{aligned} A &= \mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s) \cup \mathrm{loca}(s, e[\![y]\!], c, e_\mathrm{p}))_{|Labelled} && \text{by (A.6)} \\ &\subseteq \mathrm{dest}((s,v),c) \cup \cup_{i \in \mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), \rho[\![y]\!]ic) && \text{by hypothesis on } \rho \text{ (A.15)} \end{aligned}$$

therefore

$$\mathrm{corr_e}(e, s, [\![y]\!]e, \{y \mapsto \lambda c.c, i \mapsto \rho[\![y]\!]i \text{ if } i \in Ind\})$$

- **Case** $M\ N$**.**

$$\begin{aligned} [\![M\ N]\!]es = {} & \text{let } [(f', s')] = [\![M]\!]es \text{ in let } [((v, \lambda_1), s'')] = [\![N]\!]es' \text{ in} \\ & \text{let } (\mathtt{fun}, m, e', y) = \mathrm{getval}(s'', f') \text{ in let } (\lambda_2, s''') = \mathrm{newlbl}(s'') \text{ in} \\ & m(e'[y \mapsto (s''', (v, \lambda_2 : \lambda_1))])s''' \\ = {} & \mathrm{up}(\ell, s_4) \end{aligned}$$

$$\begin{aligned} A &= \mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s_4) \cup \mathrm{loca}(s_4, \ell, c, e_\mathrm{p}))_{|Labelled} && \text{by (A.6)} \\ A &= \mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s') \cup a(s',s'') \cup a(s''',s_4) \cup \mathrm{loca}(s_4, \ell, c, e_\mathrm{p}))_{|Labelled} && \text{by Lemma A.5} \\ &\subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(a(s,s') \cup a(s',s'') \cup \mathrm{loca}(s''', f', \mathrm{app}.c, (s''', (v, \lambda_2 : \lambda_1)) : e_\mathrm{p}))_{|Labelled} \\ & && \text{by (A.6) and (A.3)} \end{aligned}$$

Let $e'_\mathrm{p} = (s''', (v, \lambda_2 : \lambda_1)) : e_\mathrm{p}$. By induction hypothesis on M, we have:

$$\begin{aligned} & \mathrm{filt}_{(s,e'_\mathrm{p})}(a(s,s') \cup \mathrm{loca}(s''', f', \mathrm{app}.c, e'_\mathrm{p}))_{|Labelled} \\ & \quad \subseteq \mathrm{filt}_{(s,e'_\mathrm{p})}(a(s,s') \cup \mathrm{loca}(s', f', \mathrm{app}.c, e'_\mathrm{p}))_{|Labelled} && \text{using Lemma A.12} \\ & \quad \subseteq \mathrm{filt}_{(s,e'_\mathrm{p})}(\mathrm{loca_e}(s, [\![M]\!]e, \mathrm{app}.c, e'_\mathrm{p}))_{|Labelled} && \text{by (A.6)} \\ & \quad \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c)) \cup \cup_{i \in \mathrm{Dom}(e'_\mathrm{p})} \mathrm{dest}(e'_\mathrm{p}(i), E[\![M]\!]\rho i \mathrm{app}.c) \\ & && \text{by induction hypothesis on } M \\ & \quad \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c)) \cup \cup_{i>1} \mathrm{dest}(e_\mathrm{p}(i-1), E[\![M]\!]\rho i(\mathrm{app}.c)) \\ & \qquad \cup\, \mathrm{dest}((s''', (v, \lambda_2 : \lambda_1)), E[\![M]\!]\rho 1(\mathrm{app}.c))\} && \text{by definition of } e'_\mathrm{p} \end{aligned}$$

The label λ_2 does not appear in $a(s,s')$ because it has been created after s'. It does not appear in $\cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c))$ either because the state of $e(x)$ is before s (x

has been bound before MN is evaluated). Therefore filtering with respect to e_{p} or e'_{p} is the same for all these terms. For $i > 1$, $e_{\mathrm{p}}(i)$ is included in both e_{p} and e'_{p} therefore filtering with respect to e_{p} or e'_{p} is the same for $\cup_{i>1} \mathrm{dest}(e_{\mathrm{p}}(i-1), E[\![M]\!]\rho i(\mathrm{app}.c))$, and we have:

$$\begin{aligned}
A &\subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c)) \cup \cup_{i>1} \mathrm{dest}(e_{\mathrm{p}}(i-1), E[\![M]\!]\rho i(\mathrm{app}.c)) \\
&\quad \cup \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s',s'') \cup \mathrm{dest}((s'',(v,\lambda_2:\lambda_1)), E[\![M]\!]\rho 1(\mathrm{app}.c)))_{|Labelled} \\
&\subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c)) \cup \cup_{i>1} \mathrm{dest}(e_{\mathrm{p}}(i-1), E[\![M]\!]\rho i(\mathrm{app}.c)) \\
&\quad \cup \mathrm{filt}_{(s,[])}(a(s',s'') \cup \mathrm{dest}((s'',(v,\lambda_1)), E[\![M]\!]\rho 1(\mathrm{app}.c)))_{|Labelled}
\end{aligned}$$

since first labels of elements of e_{p} do not appear in $a(s',s'') \cup \mathrm{dest}((s'',(v,\lambda_1))$, as they are created after s''

$$\begin{aligned}
A &\subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c)) \cup \cup_{i>1} \mathrm{dest}(e_{\mathrm{p}}(i-1), E[\![M]\!]\rho i(\mathrm{app}.c)) \\
&\quad \cup \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho x(E[\![M]\!]\rho 1(\mathrm{app}.c)))
\end{aligned}$$

since by induction hypothesis on N (with $c' = E[\![M]\!]\rho 1(\mathrm{app}.c)$), using (A.20),

$$\mathrm{filt}_{(s,[])}((a(s',s'') \cup \mathrm{dest}((s'',(v,\lambda_1)),c')))_{|Labelled} \subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho x c')$$

Then, by Lemma A.18,

$$\begin{aligned}
A &\subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x(\mathrm{app}.c) \cup E[\![N]\!]\rho x(E[\![M]\!]\rho 1(\mathrm{app}.c))) \\
&\quad \cup \cup_{i\in \mathrm{Dom}(e_{\mathrm{p}})} \mathrm{dest}(e_{\mathrm{p}}(i), E[\![M]\!]\rho(i+1)(\mathrm{app}.c))
\end{aligned}$$

This proves that:

$$\mathrm{corr_e}(e, s, [\![M\ \ N]\!]e, \lambda x.\lambda c. \begin{cases} E[\![M]\!]\rho x(\mathrm{app}.c) \cup E[\![N]\!]\rho x(E[\![M]\!]\rho 1(\mathrm{app}.c)) & \text{if } x \in Var \\ E[\![M]\!]\rho(x+1)(\mathrm{app}.c) & \text{if } x \in Ind \end{cases})$$

- **Case** $\mathtt{fun}\ y \to M$**.**

$$[\![\mathtt{fun}\ y \to M]\!]es = \mathrm{newloc}(s, (\mathtt{fun}, [\![M]\!], e_{|FV(\mathtt{fun}\ y\to M)}, y)) = \mathrm{up}(\ell, s')$$

Subcase $e_{\mathrm{p}} = []$**.**

$$A = \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s,s') \cup \mathrm{loca}(s',\ell,c,[]))_{|Labelled}$$

If $\epsilon \notin c$, $A = \emptyset$. If $\epsilon \in c$,

$$\begin{aligned}
A &= \mathrm{filt}_{(s,e_{\mathrm{p}})}(\ell \downarrow_{s'})_{|Labelled} && \text{by (A.4)} \\
&= \bigcup_{z\in FV(\mathtt{fun}\ y\to M)} \mathrm{filt}_{(s,e_{\mathrm{p}})}(e[\![z]\!] \downarrow_s)_{|Labelled} \\
&= \bigcup_{z\in FV(\mathtt{fun}\ y\to M)} e[\![z]\!] \downarrow_{s\,|Labelled} && \text{since all labels of } e[\![z]\!] \downarrow_s \text{ exist in state } s \\
&\subseteq \bigcup_{z\in FV(\mathtt{fun}\ y\to M), e(z)=(s_z,v_z)} v_z \downarrow_{s_z\,|Labelled} \\
&\subseteq \bigcup_{z\in FV(\mathtt{fun}\ y\to M)} \mathrm{dest}(e(z), Path) && \text{by Lemma A.16}
\end{aligned}$$

Subcase $e_{\mathrm{p}} \neq []$. Let $e_{\mathrm{p}} = (s'', p_1) : e'_{\mathrm{p}}$.

$$\begin{aligned}
A &= \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s,s') \cup \mathrm{loca}(s', \ell, c, (s'', p_1) : e'_{\mathrm{p}}))_{|Labelled} \\
&\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{loca_e}(s'', [\![M]\!](e_{|FV(\mathtt{fun}\ y \to M)}[y \mapsto p_1]), \mathrm{app}^{-1}.c, e'_{\mathrm{p}}))_{|Labelled} && \text{by (A.3)} \\
&\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{filt}_{(s'',e'_{\mathrm{p}})}(\mathrm{loca_e}(s'', [\![M]\!](e_{|FV(\mathtt{fun}\ y \to M)}[y \mapsto p_1]), \mathrm{app}^{-1}.c, e'_{\mathrm{p}})))_{|Labelled}
\end{aligned}$$

where $\mathrm{getval}(s'', \ell) = (\mathtt{fun}, [\![M]\!], e_{|FV(\mathtt{fun}\ y \to M)}, y)$. We have the coherence between p_1 and $\rho_\top$. Indeed, by Lemma A.17, for all e''_{p},

$$\mathrm{filt}_{(s'',e''_{\mathrm{p}})}(\mathrm{loca}(s'', p_1, c, e''_{\mathrm{p}}))_{|Labelled} \subseteq \mathrm{dest}((s'', p_1), c) \cup \cup_{i \in \mathrm{Dom}(e''_{\mathrm{p}})} \mathrm{dest}(e''_{\mathrm{p}}(i), Path)$$

Therefore we can apply the induction hypothesis to M

$$\begin{aligned}
A \subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(&\cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!](\rho[y \mapsto \rho_\top])x(\mathrm{app}^{-1}.c)) \\
&\cup \mathrm{dest}((s'', p_1), E[\![M]\!](\rho[y \mapsto \rho_\top])y(\mathrm{app}^{-1}.c)) \\
&\cup \cup_{i \in \mathrm{Dom}(e'_{\mathrm{p}})} \mathrm{dest}(e'_{\mathrm{p}}(i), E[\![M]\!](\rho[y \mapsto \rho_\top])i(\mathrm{app}^{-1}.c)))_{|Labelled}
\end{aligned}$$

All labels of these terms are preserved by $\mathrm{filt}_{(s,e_{\mathrm{p}})}$ because variables in the environment have been created before s and other terms are in e_{p}. Therefore, we can remove $\mathrm{filt}_{(s,e_{\mathrm{p}})}$. Let $\phi(x) = E[\![M]\!]\rho[y \mapsto \rho_\top]x \circ (\lambda c.\mathrm{app}^{-1}.c)$ then

$$A \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x c) \cup \mathrm{dest}((s'', p_1), \phi y c) \cup \cup_{i \in \mathrm{Dom}(e'_{\mathrm{p}})} \mathrm{dest}(e'_{\mathrm{p}}(i), \phi i c)$$

$$\begin{aligned}
&\mathrm{corr_e}(e, s, [\![\mathtt{fun}\ y \to M]\!]e, \{x \mapsto \lambda c.\mathrm{if}\ \epsilon \in c \text{ and } x \in FV(\mathtt{fun}\ y \to M) \text{ then } Path \text{ else } \emptyset\} \\
&\quad \sqcup \{x \mapsto \phi(x) \text{ if } x \in Var, 1 \mapsto \phi(y), i \mapsto \phi(i-1) \text{ if } i > 1\})
\end{aligned}$$

- **Case** $\mathtt{let}\ y = M\ \mathtt{in}\ N$.

$$\begin{aligned}
[\![\mathtt{let}\ y = M\ \mathtt{in}\ N]\!]es = \mathrm{let}\ [((v, \lambda_1), s')] = [\![M]\!]es \text{ in let } (\lambda_2, s'') = \mathrm{newlbl}(s') \\
\text{in } [\![N]\!](e[y \mapsto (s'', (v, \lambda_2 : \lambda_1))])s'' = \mathrm{up}(\ell'', s''')
\end{aligned}$$

$$\begin{aligned}
A &= \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s, s''') \cup \mathrm{loca}(s''', \ell'', c, e_{\mathrm{p}}))_{|Labelled} && \text{by (A.6)} \\
&= \mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s, s') \cup a(s'', s''') \cup \mathrm{loca}(s''', \ell'', c, e_{\mathrm{p}}))_{|Labelled} && \text{by Lemma A.5}
\end{aligned}$$

By induction hypothesis on M (A.18),

$$\mathrm{filt}_{(s,[])}(a(s, s'))_{|Labelled} \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x \emptyset)$$

Let $\rho' = \rho[y \mapsto E[\![M]\!]\rho \sqcup \{y \mapsto \lambda c.c\}]$ and $e' = e[y \mapsto (s'', (v, \lambda_2 : \lambda_1))]$. Let us now prove that the environment ρ' used for the analysis of N is correct, i.e. $\mathrm{corr_v}(s'', e'[\![y]\!], \rho'[\![y]\!])$ which is clearly implied by:

$$\begin{aligned}
&\mathrm{filt}_{(s'',e'_{\mathrm{p}})}(\mathrm{loca}(s'', (v, \lambda_2 : \lambda_1), c, e'_{\mathrm{p}}))_{|Labelled} \\
&\quad \subseteq \mathrm{dest}((s'', (v, \lambda_2 : \lambda_1)), c) \cup \cup_{i \in \mathrm{Dom}(e'_{\mathrm{p}})} \mathrm{dest}(e'_{\mathrm{p}}(i), E[\![M]\!]\rho i c)
\end{aligned}$$

By induction hypothesis applied to M,

$$\mathrm{filt}_{(s,e'_{\mathrm{p}})}(\mathrm{loca}(s',(v,\lambda_1),c,e'_{\mathrm{p}}))_{|\mathit{Labelled}} \subseteq \cup_{x\in\mathrm{Dom}(e)}\mathrm{dest}(e(x),E[\![M]\!]\rho xc)\cup\cup_{i\in\mathrm{Dom}(e'_{\mathrm{p}})}\mathrm{dest}(e'_{\mathrm{p}}(i),E[\![M]\!]\rho ic)$$

Then Lemma A.22 gives the expected correctness of ρ'.

Then we can apply the induction hypothesis on N, which gives:

$$\mathrm{filt}_{(s'',e_{\mathrm{p}})}(a(s'',s''')\cup\mathrm{loca}(s''',\ell'',c,e_{\mathrm{p}}))_{|\mathit{Labelled}} \subseteq \cup_{x\in\mathrm{Dom}(e')}\mathrm{dest}(e'(x),E[\![N]\!]\rho' xc)\cup\cup_{i\in\mathrm{Dom}(e_{\mathrm{p}})}\mathrm{dest}(e_{\mathrm{p}}(i),E[\![N]\!]\rho' ic)$$

Then

$$\begin{aligned}&\mathrm{filt}_{(s,e_{\mathrm{p}})}(a(s'',s''')\cup\mathrm{loca}(s''',\ell'',c,e_{\mathrm{p}}))_{|\mathit{Labelled}}\\ &\quad\subseteq \mathrm{filt}_{(s,e_{\mathrm{p}})}(\cup_{x\in\mathrm{Dom}(e')}\mathrm{dest}(e'(x),E[\![N]\!]\rho' xc)\cup\cup_{i\in\mathrm{Dom}(e_{\mathrm{p}})}\mathrm{dest}(e_{\mathrm{p}}(i),E[\![N]\!]\rho' ic))_{|\mathit{Labelled}}\\ &\quad\subseteq \cup_{x\in\mathrm{Dom}(e)}\mathrm{dest}(e(x),E[\![N]\!]\rho' xc)\cup\mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{dest}((s',(v,\lambda_1)),c'))_{|\mathit{Labelled}}\\ &\qquad\cup\cup_{i\in\mathrm{Dom}(e_{\mathrm{p}})}\mathrm{dest}(e_{\mathrm{p}}(i),E[\![N]\!]\rho' ic)\end{aligned}$$

with $c' = E[\![N]\!]\rho' yc$. By correctness of $E[\![M]\!]$ (A.20),

$$\mathrm{filt}_{(s,e_{\mathrm{p}})}(\mathrm{dest}((s',(v,\lambda_1)),c'))_{|\mathit{Labelled}} \subseteq \cup_{x\in\mathrm{Dom}(e)}\mathrm{dest}(e(x),E[\![M]\!]\rho xc').$$

This yields

$$\begin{aligned}A &\subseteq \cup_{x\in\mathrm{Dom}(e)}\mathrm{dest}(e(x),E[\![M]\!]\rho x(E[\![N]\!]\rho' yc))\cup\cup_{x\in\mathrm{Dom}(e')}\mathrm{dest}(e'(x),E[\![N]\!]\rho' xc)\\ &\quad\cup\cup_{i\in\mathrm{Dom}(e_{\mathrm{p}})}\mathrm{dest}(e_{\mathrm{p}}(i),E[\![N]\!]\rho' ic)\end{aligned}$$

Then, by Lemma A.18,

$$\begin{aligned}\mathrm{corr_e}(e,s,[\![\texttt{let}\ y=M\ \texttt{in}\ N]\!]e,\{&x\mapsto E[\![N]\!]\rho' x\sqcup(E[\![M]\!]\rho x\circ E[\![N]\!]\rho' y)\text{ if }x\in \mathit{Var},\\ &i\mapsto E[\![N]\!]\rho' i\text{ if }i\in\mathit{Ind}\})\end{aligned}$$

- **Case** $\texttt{let rec}\ f=\texttt{fun}\ y\to M\ \texttt{in}\ N$. Let $P=\texttt{fun}\ y\to M$.

$$[\![\texttt{let rec}\ f=\texttt{fun}\ y\to M\ \texttt{in}\ N]\!]es = \text{let }(\lambda,s'')=\mathrm{newlbl}(s)\text{ in }[\![N]\!](e[f\mapsto(s',(\ell,[\lambda]))])s'$$

where $\ell\notin\mathrm{Dom}(s)$, $s'=(s_0[\ell\mapsto(\texttt{fun},\phi^\infty,e_{|FV(P)},y)],a,\Lambda)$ if $s''=(s_0,a,\Lambda)$ and $\phi^\infty = \mathrm{lfp}(\lambda\phi_1.\lambda e_1.\lambda s_1.[\![M]\!](e_1[f\mapsto(s'_1,(\ell,[\lambda]))])s'_1)$ where $s'_1=(s_2[\ell\mapsto(\texttt{fun},\phi_1,e_1,y)],a_2,\Lambda_2)$ if $s_1=(s_2,a_2,\Lambda_2)$.

We define

$$\begin{aligned}&\phi^0=\lambda e_1.\lambda s_1.\bot,\ \phi^{n+1}=\lambda e_1.\lambda s_1.\text{let }(s_2,a_2,\Lambda_2)=s_1\text{ in}\\ &\quad\text{let }s_3=(s_2[\ell\mapsto(\texttt{fun},\phi^n,e,y)],a_2\Lambda_2)\text{ in }[\![M]\!](e_1[f\mapsto(s_3,(\ell,[\lambda]))])s_3)\\ &s^n=(s_0[\ell\mapsto(\texttt{fun},\phi^n,e_{|FV(P)},y)],a,\Lambda),\ s^\infty=s'\\ &e_1^n=e[f\mapsto(s^n,(\ell,[\lambda]))],\ e_1^\infty=e[f\mapsto(s^\infty,(\ell,[\lambda]))]\\ &\rho_2^\infty=\mathrm{lfp}(\lambda\rho_1.\rho[f\mapsto E[\![P]\!]\rho_{1|\mathit{Ind}}])\\ &\rho_2^0=\rho[f\mapsto\{\}],\ \rho_2^{n+1}=\rho[f\mapsto E[\![P]\!]\rho^n_{2|\mathit{Ind}}]\end{aligned}$$

Let $R_1^0 = E[\![P]\!]\rho_2^0$. Let $R_1^{n+1} = \{x \mapsto E[\![P]\!]\rho_2^n x \sqcup R_1^n(x) \circ E[\![P]\!]\rho_2^n f\}$. $R_1^\infty = \{x \mapsto (E[\![P]\!]\rho_2^\infty x) \circ \sqcup_{n\in\mathbb{N}}(E[\![P]\!]\rho_2^\infty f)^n\}$ where a function ϕ with the exponent n in this formula means ϕ composed n times by itself.

We prove that:

$$\begin{aligned}&\mathrm{filt}_{(s,e'_{\mathrm{p}})}(\mathrm{loca}(s', (\ell, []), c, e'_{\mathrm{p}}))_{|\mathit{Labelled}}\\ &\quad\subseteq \cup_{x\in\mathrm{Dom}(e)}\,\mathrm{dest}(e(x), R_1^\infty xc) \cup \cup_{i\in\mathrm{Dom}(e_{\mathrm{p}})}\,\mathrm{dest}(e_{\mathrm{p}}(i), \rho_2^\infty[\![f]\!]ic)\end{aligned} \tag{A.22}$$

We perform a fixed point induction, considering imaginary executions in which the fixed point computation is stopped at the nth iteration, and then taking the limit.

- Initial case: $f_1^0 = \lambda e_1.\lambda s.\bot$, $s^0 = (s[\ell \mapsto (\mathtt{fun}, f_1^0, e, y)], a, \Lambda)$, then

  $$\mathrm{filt}_{(s,e'_{\mathrm{p}})}(\mathrm{loca}(s^0, (\ell, []), c, e'_{\mathrm{p}}))_{|\mathit{Labelled}} = \emptyset$$

  if $\epsilon \notin c$ or $e'_{\mathrm{p}} \neq []$, since the function never terminates. If $\epsilon \in c$ and $e'_{\mathrm{p}} = []$,

  $$\begin{aligned}\mathrm{filt}_{(s,e'_{\mathrm{p}})}(\mathrm{loca}(s^0, (\ell, []), c, e'_{\mathrm{p}}))_{|\mathit{Labelled}} &= \cup_{z\in FV(P)} e[\![z]\!]\downarrow_{s^0}\\ &\subseteq \cup_{x\in\mathrm{Dom}(e)}\,\mathrm{dest}(e(x), R_1^0 xc)\end{aligned}$$

  by the proof of the `fun` case.

- Induction: we prove by induction that

  $$\begin{aligned}&\mathrm{filt}_{(s,e'_{\mathrm{p}})}(\mathrm{loca}(s^n, (\ell, []), c, e'_{\mathrm{p}}))_{|\mathit{Labelled}}\\ &\quad\subseteq \cup_{x\in\mathrm{Dom}(e)}\,\mathrm{dest}(e(x), R_1^n xc) \cup \cup_{i\in\mathrm{Dom}(e'_{\mathrm{p}})}\,\mathrm{dest}(e'_{\mathrm{p}}(i), \rho_2^n[\![f]\!]ic).\end{aligned}$$

  Indeed, by Lemma A.22 and the induction hypothesis

  $$\begin{aligned}&\mathrm{filt}_{(s^n,e'_{\mathrm{p}})}(\mathrm{loca}(s^n, (\ell, [\lambda]), c, e'_{\mathrm{p}}))_{|\mathit{Labelled}}\\ &\quad\subseteq \mathrm{dest}((s^n, (\ell, [\lambda])), c) \cup \cup_{i\in\mathrm{Dom}(e'_{\mathrm{p}})}\,\mathrm{dest}(e'_{\mathrm{p}}(i), \rho_2^n[\![f]\!]ic).\end{aligned}$$

  This gives the correctness of $\rho_2^n[\![f]\!]$ with respect to $e_1^n(f)$ using formula (A.15), therefore the induction hypothesis on the expression can be applied to $P = \mathtt{fun}\ y \to M$. It yields:

  $$\begin{aligned}&\mathrm{filt}_{(s',e'_{\mathrm{p}})}(a(s, s^{n+1}) \cup \mathrm{loca}(s^{n+1}, (\ell, []), c, e'_{\mathrm{p}}))_{|\mathit{Labelled}}\\ &\subseteq \cup_{x\in\mathrm{Dom}(e_1^n)}\,\mathrm{dest}(e_1^n(x), E[\![P]\!]\rho_2^n xc) \cup \cup_{i\in\mathrm{Dom}(e'_{\mathrm{p}})}\,\mathrm{dest}(e'_{\mathrm{p}}(i), E[\![P]\!]\rho_2^n ic)\\ &\subseteq \cup_{x\in\mathrm{Dom}(e)}\,\mathrm{dest}(e(x), E[\![P]\!]\rho_2^n xc)\\ &\quad \cup \mathrm{dest}((s^n, (\ell, [\lambda])), c') \cup \cup_{i\in\mathrm{Dom}(e'_{\mathrm{p}})}\,\mathrm{dest}(e'_{\mathrm{p}}(i), E[\![P]\!]\rho_2^n ic)\end{aligned}$$

  with $c' = E[\![P]\!]\rho_2^n fc$. Applying (A.20) to the induction hypothesis, we get

  $$\begin{aligned}\mathrm{filt}_{(s,[])}(\mathrm{dest}((s^n, (\ell, [])), c'))_{|\mathit{Labelled}} &\subseteq \cup_{x\in\mathrm{Dom}(e)}\,\mathrm{dest}(e(x), R_1^n xc')\\ &\subseteq \cup_{x\in\mathrm{Dom}(e)}\,\mathrm{dest}(e(x), R_1^{n+1} xc)\end{aligned}$$

Noticing that $\rho_2^{n+1}[\![f]\!] = E[\![P]\!]\rho_2^n{}_{|Ind}$, and that $E[\![P]\!]\rho_2^n xc \subseteq R^{n+1}xc$,

$$
\begin{aligned}
&\text{filt}_{(s,e'_\text{p})}(\text{loca}(s^{n+1},(\ell,[]),c,e'_\text{p}))_{|Labelled}\\
&\quad\subseteq \text{filt}_{(s,e'_\text{p})}(\cup_{x\in\text{Dom}(e)}\text{dest}(e(x),E[\![P]\!]\rho_2^n xc)\\
&\qquad\cup\,\text{dest}((s^{n+1},(\ell,[\lambda])),c')\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),E[\![P]\!]\rho_2^n ic))_{|Labelled}\\
&\quad\subseteq \cup_{x\in\text{Dom}(e)}\text{dest}(e(x),E[\![P]\!]\rho_2^n xc)\\
&\qquad\cup\,\text{filt}_{(s,e'_\text{p})}(\text{dest}((s^{n+1},(\ell,[\lambda])),c'))_{|Labelled}\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),E[\![P]\!]\rho_2^n ic)\\
&\quad\subseteq \cup_{x\in\text{Dom}(e)}\text{dest}(e(x),E[\![P]\!]\rho_2^n xc)\\
&\qquad\cup\cup_{x\in\text{Dom}(e)}\text{dest}(e(x),R_1^{n+1}xc)\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),E[\![P]\!]\rho_2^n ic)\\
&\quad\subseteq \cup_{x\in\text{Dom}(e)}\text{dest}(e(x),R_1^{n+1}xc)\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),\rho_2^{n+1}[\![f]\!]ic)
\end{aligned}
$$

- Limit:

$$
\begin{aligned}
&\text{filt}_{(s,e'_\text{p})}(\text{loca}(s',(\ell,[]),c,e'_\text{p}))_{|Labelled}\\
&\quad= \cup_n\text{filt}_{(s,e'_\text{p})}(\text{loca}(s^n,(\ell,[]),c,e'_\text{p}))_{|Labelled}\\
&\quad\subseteq \cup_n\cup_{x\in\text{Dom}(e)}\text{dest}(e(x),R_1^n xc)\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),\rho_2^n[\![f]\!]ic)\\
&\quad\subseteq \cup_{x\in\text{Dom}(e)}\text{dest}(e(x),\cup_n R_1^n xc)\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),\cup_n\rho_2^n[\![f]\!]ic)\\
&\hspace{22em}\text{by Lemma A.18}\\
&\quad\subseteq \cup_{x\in\text{Dom}(e)}\text{dest}(e(x),R_1^\infty xc)\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),\rho_2^\infty[\![f]\!]ic)
\end{aligned}
$$

By Lemma A.22,

$$
\begin{aligned}
&\text{filt}_{(s',e'_\text{p})}(\text{loca}(s',(\ell,[\lambda]),c,e'_\text{p}))_{|Labelled}\\
&\quad\subseteq \text{dest}((s',(\ell,[\lambda])),c)\cup\cup_{i\in\text{Dom}(e'_\text{p})}\text{dest}(e'_\text{p}(i),\rho_2^\infty[\![f]\!]ic)
\end{aligned}
$$

We have then the correctness of $\rho_2^\infty[\![f]\!]$ with respect to ℓ (A.15), therefore the induction hypothesis can be applied to N.

$$
\begin{aligned}
A &= \text{filt}_{(s,e_\text{p})}(a(s,s'')\cup\text{loca}(s'',\ell',c,e_\text{p}))_{|Labelled}\\
&= \text{filt}_{(s,e_\text{p})}(\text{filt}_{(s',e_\text{p})}(a(s',s'')\cup\text{loca}(s'',\ell',c,e_\text{p})))_{|Labelled}
\end{aligned}
$$

since all labels removed by $\text{filt}_{(s',e_\text{p})}$ were already removed by $\text{filt}_{(s,e_\text{p})}$ because s' is after s

$$
\subseteq \text{filt}_{(s,e_\text{p})}(\cup_{x\in\text{Dom}(e_1^\infty)}\text{dest}(e_1^\infty(x),E[\![N]\!]\rho_2^\infty xc)\cup\cup_{i\in\text{Dom}(e_\text{p})}\text{dest}(e_\text{p}(i),E[\![N]\!]\rho_2^\infty ic))_{|Labelled}
$$

As in the `let` case, the $x = f$ term can be handled as follows (with $c' = E[\![N]\!]\rho_2 fc$):

$$
\begin{aligned}
\text{filt}_{(s,e_\text{p})}(\text{dest}(e_1^\infty(f),c'))_{|Labelled} &= \text{filt}_{(s,[])}(\text{dest}((s',(\ell,[\lambda])),c'))_{|Labelled}\\
&= \text{filt}_{(s,[])}(\text{dest}((s',(\ell,[])),c'))_{|Labelled}\\
&\subseteq \cup_{x\in\text{Dom}(e)}\text{dest}(e(x),R_1^\infty xc')\\
&\hspace{6em}\text{by applying (A.20) to (A.22)}
\end{aligned}
$$

Then $A \subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho_2^\infty xc) \cup \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), R_1^\infty x(E[\![N]\!]\rho_2^\infty fc)) \cup \cup_{i\in\mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), E[\![N]\!]\rho_2 ic)$, and

$$\begin{aligned} R(x) &= E[\![N]\!]\rho_2^\infty x \sqcup (E[\![P]\!]\rho_2^\infty x) \circ \mathrm{lfp}(\lambda R_2.E[\![N]\!]\rho_2^\infty f \sqcup (E[\![P]\!]\rho_2^\infty f) \circ R_2) \\ &= E[\![N]\!]\rho_2^\infty x \sqcup (E[\![P]\!]\rho_2^\infty x) \circ \sqcup_{n\in\mathbb{N}} (E[\![P]\!]\rho_2^\infty f)^n \circ E[\![N]\!]\rho_2^\infty f \\ &= E[\![N]\!]\rho_2^\infty x \sqcup R_1^\infty(x) \circ E[\![N]\!]\rho_2^\infty f \end{aligned}$$

therefore

$$\begin{aligned} \mathrm{corr_e}(e, s, [\![\texttt{let rec } f = \texttt{fun } y \to M \texttt{ in } N]\!]e, \{&x \mapsto R(x) \text{ if } x \in \mathit{Var} - \{f\}, \\ &i \mapsto E[\![N]\!]\rho' i \text{ if } i \in \mathit{Ind}\}) \end{aligned}$$

- **Case** $\texttt{if } M \texttt{ then } N \texttt{ else } P$**.**

$$\begin{aligned} [\![\texttt{if } M \texttt{ then } N \texttt{ else } P]\!]es &= \text{let } [(x, s')] = [\![M]\!]es \text{ in if } x \text{ then } [\![N]\!]es' \text{ else } [\![P]\!]es' \\ &= \mathrm{up}(\ell'', s'') \end{aligned}$$

$$\begin{aligned} A &= \mathrm{filt}_{(s,e_\mathrm{p})}(a(s, s'') \cup \mathrm{loca}(s'', \ell'', c, e_\mathrm{p}))_{|\mathit{Labelled}} \\ &\subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(\mathrm{loca}(s'', \ell'', c, e_\mathrm{p}) \cup a(s, s') \cup a(s', s''))_{|\mathit{Labelled}} && \text{by Lemma A.5} \\ &\subseteq [\cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho xc) \cup \cup_{i\in\mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), E[\![N]\!]\rho ic) \\ &\quad \cup \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![P]\!]\rho xc) \cup \cup_{i\in\mathrm{Dom}(e_\mathrm{p})} \mathrm{dest}(e_\mathrm{p}(i), E[\![P]\!]\rho ic)] \\ &\quad \cup \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho x\emptyset) \end{aligned}$$

by induction hypothesis applied to M (A.18), N and P (A.21). By Lemma A.18,

$$\mathrm{corr_e}(e, s, [\![\texttt{if } M \texttt{ then } N \texttt{ else } P]\!]e, \lambda x.\lambda c.E[\![M]\!]\rho x\emptyset \cup E[\![N]\!]\rho xc \cup E[\![P]\!]\rho xc)$$

- **Case** $M; N$**.** This case is left to the reader (it is very similar to the `if` case, and even simpler).

- **Case** $C(M_1, ..., M_n)$**.**

$$\begin{aligned} [\![C(M_1, ..., M_n)]\!]es_0 &= \text{let } [(x_1, s_1)] = [\![M_1]\!]es_0 \text{ in ...let } [(x_n, s_n)] = [\![M_n]\!]es_{n-1} \text{ in} \\ &\qquad \mathrm{newloc}(s_n, (C, x_1, ..., x_n)) \\ &= \mathrm{up}(\ell, s''') \\ A &= (a(s_0, s''') \cup \mathrm{loca}(s''', \ell, c, e_\mathrm{p}))_{|\mathit{Labelled}} \end{aligned}$$

If $e_\mathrm{p} \neq []$, $\mathrm{loca}(s''', \ell, c, e_\mathrm{p}) = \emptyset$. Therefore,

$$\begin{aligned} A &\subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(a(s_0, s''') \cup \mathrm{loca}(s''', \ell, c, []))_{|\mathit{Labelled}} \\ &\subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(a(s_0, s''') \cup \cup_k \mathrm{loca}(s''', x_k, \overline{C_k}^{-1}.c, []))_{|\mathit{Labelled}} && \text{by (A.1) since } \ell \text{ has label } [] \\ &\subseteq \mathrm{filt}_{(s,e_\mathrm{p})}(a(s_0, s''') \cup \cup_k \mathrm{loca}(s_k, x_k, \overline{C_k}^{-1}.c, []))_{|\mathit{Labelled}} && \text{by Lemma A.12} \end{aligned}$$

$$A \subseteq \text{filt}_{(s,e_\text{p})}(\cup_k a(s_{k-1}, s_k) \cup a(s_n, s''') \cup \cup_k \text{loca}(s_k, x_k, \overline{C_k}^{-1}.c, []))_{|Labelled} \quad \text{by Lemma A.5}$$
$$\subseteq \cup_k \text{filt}_{(s,e_\text{p})}(a(s_{k-1}, s_k) \cup \text{loca}(s_k, x_k, \overline{C_k}^{-1}.c, []))_{|Labelled} \quad \text{since } a(s_n, s''') = \emptyset$$
$$\subseteq \cup_k \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M_k]\!]\rho x(\overline{C_k}^{-1}.c)) \cup \cup_{i \in \text{Dom}(e_\text{p})} \text{dest}(e_\text{p}(i), E[\![M_k]\!]\rho i(\overline{C_k}^{-1}.c))$$

by induction hypothesis on M_1, ..., M_n (A.21). By Lemma A.18,

$$\text{corr}_\text{e}(e, s, [\![C(M_1, ..., M_n)]\!]e, \lambda x.\lambda c. \cup_k E[\![M_k]\!]\rho x(\overline{C_k}^{-1}.c))$$

- **Case $\overline{C_i}(M) \leftarrow N$.**

$$\begin{aligned}[\![\overline{C_i}(M) \leftarrow N]\!]es &= \text{let } [((x, \lambda_x), s')] = [\![M]\!]es \text{ in let } [((y, \lambda_y), (s'', a'', \Lambda''))] = [\![N]\!]es' \text{ in} \\ &\quad \text{let } (C, v_1, ..., v_k) = s''(x) \text{ in up}((\text{NIL}, []), \\ &\quad (s''[x \mapsto (C, v_1, ..., v_{i-1}, (y, U), v_{i+1}, ..., v_k)], a'' \cup (y, \lambda_y) \downarrow_{s''}, \Lambda'')) \\ &= \text{up}((\text{NIL}, []), s''')\end{aligned}$$

$$\begin{aligned}A &= \text{filt}_{(s,e_\text{p})}(a(s, s''') \cup \text{loca}(s''', (\text{NIL}, []), c, e_\text{p}))_{|Labelled} && \text{by (A.6)} \\ &= \text{filt}_{(s,e_\text{p})}(a(s, s') \cup a(s', (s'', a'', \Lambda'')) \cup a((s'', a'', \Lambda''), s'''))_{|Labelled} && \text{by Lemma A.5} \\ &= \text{filt}_{(s,e_\text{p})}(a(s, s') \cup a(s', (s'', a'', \Lambda'')) \cup (y, \lambda_y) \downarrow_{(s'', a'', \Lambda'')})_{|Labelled} \\ &\subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!]\rho x\emptyset) \cup \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![N]\!]\rho x Path)\end{aligned}$$

by induction hypothesis applied to M (A.18) and N (A.21). By Lemma A.18,

$$\text{corr}_\text{e}(e, s, [\![\overline{C_i}(M) \leftarrow N]\!]e, \lambda x.\lambda c.E[\![M]\!]\rho x\emptyset \cup E[\![N]\!]\rho x Path).$$

We now prove $\text{corr}_\text{v}(s, e_0[\![y]\!], \rho_0[\![y]\!])$. Let $A = \text{filt}_{(s,e_\text{p})}(\text{loca}(s, v, c, e_\text{p}))_{|Labelled}$.

- **Case $\overline{C_i}$.**

$$e[\![\overline{C_i}]\!] = (\overline{C_i}\text{clo}, []) \text{ where}$$
$$\text{getval}(s, (\overline{C_i}\text{clo}, [])) = (\lambda e.\lambda s'.\text{let } (C, v_1, ..., v_k) = \text{getval}(s', e[\![x]\!]) \text{ in up}(v_i, s'), \{\}, x)$$

  - Subcase $e_\text{p} = []$.

$$A = \text{filt}_{(s,e_\text{p})}(\text{loca}(s, (\overline{C_i}\text{clo}, []), c, []))_{|Labelled} \subseteq \text{filt}_{(s,e_\text{p})}(\{(\overline{C_i}\text{clo}, [])\})_{|Labelled} = \emptyset$$

  - Subcase $e_\text{p} \neq []$. Let $s'(p_1) = (C, v_1, ..., v_k)$.

$$\begin{aligned}A &= \text{filt}_{(s,(s',p_1):e_\text{p})}(\text{loca}(s, (\overline{C_i}\text{clo}, []), c, (s', p_1) : e_\text{p}))_{|Labelled} \\ &\subseteq \text{filt}_{(s,(s',p_1):e_\text{p})}(\text{loca}(s', v_i, \text{app}^{-1}.c, e_\text{p}))_{|Labelled} && \text{by (A.3)} \\ &\subseteq \text{filt}_{(s,(s',p_1):e_\text{p})}(\text{loca}(s', p_1, \overline{C_i}.\text{app}^{-1}.c, e_\text{p}))_{|Labelled} && \text{by (A.1)} \\ &\subseteq \text{dest}((s', p_1), \overline{C_i}.\text{app}^{-1}.c) \cup \cup_{k \in \text{Dom}(e_\text{p})} \text{dest}(e_\text{p}(k), Path) && \text{by Lemma A.17.}\end{aligned}$$

Then,

$$\text{corr}_\text{v}(s, \overline{C_i}\text{clo}, \{1 \mapsto \lambda c.\overline{C_i}.\text{app}^{-1}.c, k \mapsto \lambda c.Path \text{ if } k > 1\})$$

- **Case** $C?$. Similar to $\overline{C_i}$ (in fact simpler: $A = \emptyset$ in all cases).

- **Case true.** $e[\![\texttt{true}]\!] = (\text{true}, [])$ and $\text{loca}(s, \text{true}, c, []) = \emptyset$ therefore $\text{corr}_\text{v}(s, \text{true}, \{\})$

- **Cases false and $\mathbb{Z}$.** similar to `true`.

Finally, notice that $\text{corr}_\text{v}(s, e[\![y]\!], \rho[\![y]\!])$ will always be true for the values of ρ that will be used. Indeed, ρ is computed as defined in Definition 3.13. Let ρ_1 be the environment just before bounding y. The environment ρ_2 with y bound is given by: $\rho_2 = \rho_1[y \mapsto \rho_\top]$ if y is bound by `fun`, $\rho_2 = \rho_1[y \mapsto (E[\![N]\!]\rho_1)_{|Ind}]$ if y is bound by `let` $y = N$, and $\rho_2 = \text{lfp}(\lambda\rho_2.\rho_1[y \mapsto (E[\![\texttt{fun}\ z \to N]\!]\rho_2)_{|Ind}])$ if y is bound by `let rec` y = `fun` $z \to N$ `in` .... These values of $\rho[\![y]\!]$ have been proved correct in the `fun`, `let` and `let rec` cases respectively. Using $\text{corr}([\![M]\!], E[\![M]\!])$, we have $\text{corr}_\text{e}(e, s, [\![M]\!]e, E[\![M]\!]\rho)$ for every expression M. □

We can now prove the expected correctness theorem:

Theorem 3.18 (Correctness of analysis E) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let us assume that $\epsilon \notin E[\![N]\!]\rho\, x\, Path$.*

- `let rec` x = `fun` $y \to M$ `in` N. *In this case, let ℓ be the location of the closure x. We assume $\epsilon \notin R(x)Path$ where R is as in the definition of analysis E.*

In these two cases, ℓ can be stack allocated, and deallocated at the end of N.

Notice that $\epsilon \notin E[\![N]\!]\rho\, x\, Path$ would not be enough in the `let rec` case. For example:

```
let rec f x = ... z := f ...
in f(3)
```

The closure f escapes, even if $\epsilon \notin \text{app}.Path = E[\![\texttt{f(3)}]\!]\rho\texttt{f}\,Path$. The same problem happens when recursive data structures can be defined by `let rec`: the top location may be accessed by a path different from ϵ.

Proof: Let s_x be the store before evaluating N, and e be the environment in which N is evaluated.

If x is defined by `let`, we assume that $\epsilon \notin c$, and prove that $(\ell, \lambda : \lambda') \notin \text{dest}((s_x, (\ell, \lambda : \lambda'')), c)$. If $\text{getval}(s_x, (\ell, \lambda : \lambda'')) = (C, v_1, ..., v_n)$, M is $C(M_1, ..., M_k)$.

$$\text{dest}((s_x, (\ell, \lambda : \lambda'')), c) \subseteq \text{filt}_{(s_x, [])}((\cup_k \text{loca}(s_x, v_k, \overline{C_i}^{-1}.c, [])))_{|Labelled} \subseteq \cup_k v_k \downarrow_{s_x}$$

Then $(\ell, \lambda) \notin \text{dest}((s_x, (\ell, \lambda)), c)$ since ℓ has been created by $C(v_1, ..., v_k)$ and is therefore not reachable from $v_1, ..., v_k$.

If $\text{getval}(s_x, (\ell, \lambda : \lambda'')) = (\texttt{fun}, \phi, e, x)$, M is `fun` $y \to P$.

$$\text{dest}((s_x, (\ell, \lambda : \lambda'')), c) \subseteq \cup_{e_\text{p}} \text{filt}_{(s_x, e_\text{p})}(\text{loca}_\text{e}(s_1, \phi(e[x \mapsto (s_1, p_1)]), \text{app}^{-1}.c, e'_\text{p}))_{|\overline{e_\text{p}}}$$

where $e_\mathrm{p} = (s_1, p_1) : e'_\mathrm{p}$. $\mathrm{dest}((s_x, (\ell, \lambda : \lambda'')), c) \subseteq \mathrm{filt}_{(s_x, e_\mathrm{p})}(a(s_1, s'_1) \cup r' \downarrow_{s'_1})_{|Labelled}$ where $\phi(e[x \mapsto p_1])s_1 = \mathrm{up}(r', s'_1)$. Then, by Lemma A.10,

$$\begin{aligned}\mathrm{dest}((s_x, (\ell, \lambda : \lambda'')), c) &\subseteq \mathrm{filt}_{(s_x, e_\mathrm{p})}((e[x \mapsto (s_1, p_1)]) \downarrow_{s_1})_{|Labelled} \\ &= \mathrm{filt}_{(s_x, e_\mathrm{p})}(e \downarrow_{s_1} \cup p_1 \downarrow_{s_1})_{|Labelled}.\end{aligned}$$

We have seen in the proof of Lemma A.11 that $v \downarrow_{s_1} \subseteq v \downarrow_{s_0}$ if s_1 is after s_0. Therefore $e \downarrow_{s_1} \subseteq e \downarrow_{s_0}$ where s_0 is the store at the beginning of the `let`. Then $e \downarrow_{s_1}$ does not contain ℓ (since ℓ is the location of the closure, created after s_0). In $p_1 \downarrow_{s_1}$, ℓ may appear, but with the label of p_1, which does not begin with λ. Therefore $(\ell, \lambda : \lambda') \notin \mathrm{dest}((s_x, (\ell, \lambda : \lambda'')), c)$

If x is defined by `let rec`, x itself may refer to ℓ. We have

$$\epsilon \notin R(x)Path = \cup_{n \in \mathbb{N}}(E[\![\texttt{fun } y \to M]\!]\rho' x)^n (E[\![N]\!]\rho' \, x \, Path).$$

Let $c = E[\![N]\!]\rho \, x \, Path$. Therefore $\epsilon \notin \cup_{n \in \mathbb{N}}(E[\![\texttt{fun } y \to M]\!]\rho' x)^n (c)$.

In a way similar to the correctness proof of the `let rec` case, let us consider imaginary executions where only n iterations are done in the fixed point computation, to prove by induction that $(\ell, \lambda : \lambda') \notin \mathrm{dest}((s_x^n, (\ell, [\lambda])), c)$, if $c = E[\![N]\!]\rho \, x \, Path$.

- **If $n = 0$,** x is the function which never terminates, then nothing escapes: $\mathrm{dest}((s_x^0, (\ell, [\lambda])), c) = \emptyset$ since $\epsilon \notin c$.

- **For $n+1$,**

$$\begin{aligned}
&\mathrm{dest}((s^{n+1}, (\ell, \lambda : \lambda'')), c) \\
&\quad = \cup_{e_\mathrm{p}} \mathrm{filt}_{(s^{n+1}, e_\mathrm{p})}(\mathrm{loca}(s^{n+1}, (\ell, \lambda : \lambda''), c, e_\mathrm{p}))_{|\overline{e_\mathrm{p}}} && \text{by definition of dest} \\
&\quad = \cup_{e_\mathrm{p}} \mathrm{filt}_{(s^{n+1}, e_\mathrm{p})}(\mathrm{loca_e}(s', \phi^{n+1}(e^{n+1}[y \mapsto (s', p_1)]), \mathrm{app}^{-1}.c, e'_\mathrm{p}))_{|\overline{e_\mathrm{p}}} \\
&\qquad\qquad \text{where } \mathrm{getval}(s^{n+1}, (\ell, \lambda : \lambda'')) = (\texttt{fun}, \phi^{n+1}, e, y),\ e_\mathrm{p} = (s', p_1) : e'_\mathrm{p}, \text{ by (A.3)} \\
&\quad = \cup_{e_\mathrm{p}} \mathrm{filt}_{(s^{n+1}, e_\mathrm{p})}(a(s', s'') \cup \mathrm{loca}(s'', v, \mathrm{app}^{-1}.c, e'_\mathrm{p}))_{|\overline{e_\mathrm{p}}} && \text{by (A.6)} \\
&\qquad \text{where } \mathrm{up}(v, s'') = \phi^{n+1}(e[y \mapsto (s', p_1)])s' = [\![M]\!](e[y \mapsto (s', p_1), x \mapsto (s^n, (\ell, [\lambda]))])s' \\
&\quad \subseteq \cup_{e_\mathrm{p}} \mathrm{filt}_{(s^{n+1}, e_\mathrm{p})}(\cup_{z \in \mathrm{Dom}(e)} \mathrm{dest}(e(z), E[\![M]\!](\rho'[y \mapsto \rho_\top])z(\mathrm{app}^{-1}.c)) \cup \mathrm{dest}((s', p_1), C) \\
&\qquad \cup \mathrm{dest}((s^n, (\ell, [\lambda])), E[\![M]\!](\rho'[y \mapsto \rho_\top])x(\mathrm{app}^{-1}.c)) \cup \cup_{i \in \mathrm{Dom}(e'_\mathrm{p})} \mathrm{dest}(e'_\mathrm{p}(i), C' i c))_{|\overline{e_\mathrm{p}}} \\
&&& \text{by correctness of analysis of } M \\
&\quad \subseteq \cup_{e_\mathrm{p}} \mathrm{filt}_{(s^{n+1}, e_\mathrm{p})}(\cup_{z \in \mathrm{Dom}(e)} \mathrm{dest}(e(z), E[\![M]\!](\rho'[y \mapsto \rho_\top])z(\mathrm{app}^{-1}.c)) \\
&\qquad \cup \mathrm{dest}((s^n, (\ell, [\lambda])), E[\![\texttt{fun } y \to M]\!]\rho' x c))_{|\overline{e_\mathrm{p}}}
\end{aligned}$$

Assume that $(\ell, \lambda : \lambda') \in \mathrm{dest}((s^{n+1}, (\ell, [\lambda])), c)$. Since ℓ is not reachable from e then $\exists e_\mathrm{p}$, $(\ell, \lambda : \lambda_2) \in \mathrm{dest}((s^n, (\ell, [\lambda])), E[\![\texttt{fun } y \to M]\!]\rho' x c)$. $\epsilon \notin E[\![\texttt{fun } y \to M]\!]\rho' x c$, then by induction hypothesis, this is impossible. $(\ell, \lambda : \lambda') \notin \mathrm{dest}((s^{n+1}, (\ell, [\lambda])), c)$.

- **Limit:** We have $\mathrm{dest}((s_x, (\ell, [\lambda])), c) = \cup_n \mathrm{dest}((s^n, (\ell, [\lambda])), c)$, therefore $(\ell, \lambda : \lambda') \notin \mathrm{dest}((s_x, (\ell, [\lambda])), c)$.

Then we have $(\ell, \lambda : \lambda') \notin \mathrm{dest}((s_x, (\ell, [\lambda])), c)$ if $c \in E[\![N]\!]\rho \, x \, Path$.

Then $(\ell, \lambda : \lambda') \notin \mathrm{dest}(e(x), E[\![N]\!]\rho\, x\, \mathit{Path})$. Moreover, for all $y \neq x$, $(\ell, \lambda : \lambda') \notin \mathrm{dest}(e(y)$, $E[\![N]\!]\rho\, y\, \mathit{Path})$ since ℓ has been created when adding x to the environment. Therefore ℓ is not reachable from variables of greater scope than x. By correctness of analysis E (Theorem A.25),

$$\mathrm{filt}_{(s,[])}(\mathrm{loca}_{\mathrm{e}}(s_x, [\![N]\!]e, \mathit{Path}, []))_{|\mathit{Labelled}} \subseteq \cup_{y \in \mathit{Var}}\, \mathrm{dest}(e(y), E[\![N]\!]\rho\, y\, \mathit{Path}).$$

Then $(\ell, \lambda : \lambda') \notin \mathrm{filt}_{(s,[])}(\mathrm{loca}_{\mathrm{e}}(s_x, [\![N]\!]e, \mathit{Path}, []))$. By Theorem A.19, ℓ can be stack allocated.
$\square$

A.2 Adapting analysis E to types

We prove the correctness of analysis $\tilde{E}$, described in Section 3.4.1. One way to do this would be to prove that if v is of type τ, $\mathrm{loca}(s, v, c, e_p) = \mathrm{loca}(s, v, c \cap \mathit{Path}(\tau), e_p)$. We could then intersect all contexts with $\mathit{Path}(\tau)$ which would lead to analysis $\tilde{E}$. However, the equality $\mathrm{loca}(s, v, c, e_p) = \mathrm{loca}(s, v, c \cap \mathit{Path}(\tau), e_p)$ is only true when the parameters e_p are of the type expected for parameters of v. Therefore, we would have to alter significantly the previous proof of analysis E to make all the typing constraints explicit. This would complicate again this proof. We have therefore chosen another way of performing the proof, using only the result of correctness of E and properties of the typing.

Let us define $\mathit{Path}_{\mathrm{s}}(\forall \beta_1 ... \beta_k . \tau) = \mathit{Path}_{\beta_1 ... \beta_k}(\tau)$ in the same way as $\mathit{Path}(\tau)$ except that $\mathit{Path}_{\beta_1 ... \beta_k}(\beta_i) = \{\epsilon\}$ (where $i \in [\![1, k]\!]$). τ_x^{s} is the type scheme of x in the typing environment Γ.

Lemma A.27 *Assume that*

$$\forall \sigma, \forall x \in \mathit{Var}, \forall i \in \mathit{Ind}, \rho[\![x]\!]ic \cap \mathit{Path}(\sigma\tau_{x,i}^{\mathrm{g}}) \subseteq \tilde{\rho}[\![x]\!]i(c \cap \mathit{Path}(\sigma\tau_x^{\mathrm{g}})).$$

Then

$$\forall \sigma, \forall x \in \mathit{Var}, E[\![M]\!]\rho x c \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} x(c \cap \mathit{Path}(\sigma\tau_M)),$$
$$\forall \sigma, \forall i \in \mathit{Ind}, E[\![M]\!]\rho i c \cap \mathit{Path}(\sigma\tau_{M,i}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} i(c \cap \mathit{Path}(\sigma\tau_M)).$$

The goal of the substitution σ is to prove directly the result for all instances of the expressions, not only for the most general instance.

Proof: By induction on M.

- **Case** y**.** If $y \neq x$, $E[\![M]\!]\rho x c = \emptyset$, hence we have the result.

If $y = x$, $E[\![M]\!]\rho x c \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) = c \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}})$ and $\tilde{E}[\![M]\!]\tilde{\rho} x(c \cap \mathit{Path}(\sigma\tau_x)) = c \cap \mathit{Path}(\sigma\tau_x)$. We have to prove that $\mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \mathit{Path}(\sigma\tau_x)$ where $\tau_x = \sigma'\tau_x^{\mathrm{g}}$ and $\tau_x^{\mathrm{s}} = \forall \beta_1 ... \beta_k . \tau_x^{\mathrm{g}}$, $\mathrm{Dom}(\sigma') = \{\beta_1, ..., \beta_k\}$. We first rename $\beta_1 ... \beta_k$ to make sure that they are not captured by σ. Then $\mathit{Path}_{\beta_1 ... \beta_k}(\sigma\tau_x^{\mathrm{g}}) \subseteq \mathit{Path}(\sigma\sigma'\tau_x^{\mathrm{g}})$. Indeed, if τ_x^{g} is a type variable β_i in $\beta_1, ..., \beta_k$, $\mathit{Path}_{\beta_1 ... \beta_k}(\sigma\beta_i) = \mathit{Path}_{\beta_1 ... \beta_k}(\beta_i) = \{\epsilon\} \subseteq \mathit{Path}(\sigma\sigma'\beta_i)$. If τ_x^{g} is a type variable β not in $\beta_1, ..., \beta_k$, $\mathit{Path}_{\beta_1 ... \beta_k}(\sigma\beta) = \mathit{Path}(\sigma\beta) = \mathit{Path}(\sigma\sigma'\beta)$. This can be extended to all types by fixed point induction, following the definition of $\mathit{Path}(\tau)$.

The key property of typing we have used is that the substitutions σ (corresponding to a future instantiation) and σ' (corresponding to the generalization) cannot substitute the same type variables: σ substitutes variables which are not bound in the type scheme, whereas σ' substitutes only variables which are bound in the type scheme.

Assume that $i < p$ where p is the number of parameters of y: $\tau_y^{\mathrm{g}} = \tau_1 \to \ldots \to \tau_p$ and τ_p is not a function type. $E[\![M]\!]\rho i c \cap \mathit{Path}(\sigma\tau_{M,i}) = \rho[\![y]\!] i c \mathit{Path}(\sigma\tau_{M,i})$ and $\tilde{E}[\![M]\!]\tilde{\rho} i(c \cap \mathit{Path}(\sigma\tau_M)) = \tilde{\rho}[\![y]\!] i(c \cap \mathit{Path}(\sigma\tau_M)) \cap \mathit{Path}(\tau_{y,i})$, where $\tau_{M,i} = \sigma'\tau_{y,i}^{\mathrm{g}}$ and $\tau_M = \sigma'\tau_y^{\mathrm{g}}$. The inclusion is true by hypothesis on ρ and $\tilde{\rho}$.

Otherwise, $\tilde{E}[\![M]\!]\tilde{\rho} i(c \cap \mathit{Path}(\sigma\tau_M)) = \mathit{Path}(\tau_{M,i})$ hence we have the result.

• **Case** $M\ N$**.** Let $i \in \mathit{Ind}$. By induction hypothesis on M:

$$E[\![M]\!]\rho i c' \cap \mathit{Path}(\sigma\tau_{M,i}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} i(c' \cap \mathit{Path}(\sigma\tau_M))$$

hence

$$E[\![M]\!]\rho(i+1)(\mathrm{app}.c) \cap \mathit{Path}(\sigma\tau_{M,i+1}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho}(i+1)(\mathrm{app}.(c \cap \mathit{Path}(\sigma\tau_{M\ N})))$$

since $\mathit{Path}(\sigma\tau_M) \subseteq \mathrm{app}.\mathit{Path}(\sigma\tau_{M\ N})$. This yields the expected result.

Let $x \in \mathit{Var}$. By induction hypothesis on M,

$$E[\![M]\!]\rho x c' \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} x(c' \cap \mathit{Path}(\sigma\tau_M))$$

hence

$$E[\![M]\!]\rho x(\mathrm{app}.c) \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} x(\mathrm{app}.(c \cap \mathit{Path}(\sigma\tau_{M\ N}))).$$

By induction hypothesis on N,

$$E[\![N]\!]\rho x c'' \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![N]\!]\tilde{\rho} x(c'' \cap \mathit{Path}(\sigma\tau_N))$$

and by induction on M,

$$E[\![M]\!]\rho 1(\mathrm{app}.c) \cap \mathit{Path}(\sigma\tau_{M,1}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} 1(\mathrm{app}.(c \cap \mathit{Path}(\sigma\tau_{M\ N}))).$$

Taking $c'' = E[\![M]\!]\rho 1(\mathrm{app}.c)$, and noticing that $\tau_N = \tau_{M,1}$,

$$E[\![N]\!]\rho x(E[\![M]\!]\rho 1(\mathrm{app}.c)) \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![N]\!]\tilde{\rho} x(\tilde{E}[\![M]\!]\tilde{\rho} 1(\mathrm{app}.(c \cap \mathit{Path}(\sigma\tau_{M\ N})))).$$

Taking the upper bound of both equations

$$E[\![M\ N]\!]\rho x c \cap \mathit{Path}_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M\ N]\!]\tilde{\rho} x(c \cap \mathit{Path}(\sigma\tau_{M\ N}))$$

which is the expected result.

• **Case** `fun` $y \to M$**.** The environment $\rho' = \rho[y \mapsto \rho_\top]$ satisfies the hypothesis, since $\rho[\![y]\!] i c \cap \mathit{Path}(\sigma\tau_{y,i}^{\mathrm{g}}) = \mathit{Path} \cap \mathit{Path}(\sigma\tau_{y,i}^{\mathrm{g}}) \subseteq \mathit{Path}(\tau_{y,i}^{\mathrm{g}}) = \tilde{\rho}[\![y]\!] i(c \cap \mathit{Path}(\sigma\tau_y^{\mathrm{g}}))$.

Let $i > 1$. By induction on M,

$$\forall i \in \mathit{Ind}, E[\![M]\!]\rho'(i-1)c' \cap \mathit{Path}(\sigma\tau_{M,i-1}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho}'(i-1)(c' \cap \mathit{Path}(\sigma\tau_M)).$$

Taking $c' = \mathrm{app}^{-1}.c$ and noticing that $\mathit{Path}(\sigma\tau_M) = \mathrm{app}^{-1}.\mathit{Path}(\sigma\tau_{\mathtt{fun}\ y \to M})$, this yields the result.

Let $i = 1$. By induction on M,

$$E[\![M]\!]\rho y c' \cap Path_{\mathrm{s}}(\sigma\tau_y^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} y(c' \cap Path(\sigma\tau_M)).$$

There is no bound type variable in the type scheme τ_y^{s}, hence

$$E[\![M]\!]\rho y c' \cap Path(\sigma\tau_y) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} y(c' \cap Path(\sigma\tau_M)).$$

Taking $c' = \mathrm{app}^{-1}.c$, this yields the result.

Let $x \in Var - \{y\}$. By induction on M,

$$E[\![M]\!]\rho x c' \cap Path_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} x(c' \cap Path(\sigma\tau_M))$$

where $c' = \mathrm{app}^{-1}.c$. On the other hand, $Path \cap Path_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq Path(\tau_x^{\mathrm{g}})$. Therefore,

$$E[\![\texttt{fun}\ y \rightarrow M]\!]\rho x c \cap Path_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![\texttt{fun}\ y \rightarrow M]\!]\tilde{\rho} x(c \cap Path(\sigma\tau_{\texttt{fun}\ y\rightarrow M})).$$

• **Case** `let` $y = M$ `in` N. The environment $\rho' = \rho[y \mapsto E[\![M]\!]\rho_{|Ind}]$ satisfies the hypothesis. This is an obvious consequence of the induction hypothesis on M.

Let $i \in Ind$. The result is an obvious consequence of the induction hypothesis on N.

Let $x \in Var - \{y\}$. By induction hypothesis on N,

$$E[\![N]\!]\rho' x c \cap Path_{\mathrm{s}}(\sigma\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![N]\!]\tilde{\rho}' x(c \cap Path(\sigma\tau_N))$$

and

$$E[\![N]\!]\rho' y c \cap Path_{\mathrm{s}}(\sigma\tau_y^{\mathrm{s}}) \subseteq \tilde{E}[\![N]\!]\tilde{\rho}' y(c \cap Path(\sigma\tau_N)).$$

By induction hypothesis on M,

$$E[\![M]\!]\rho x c' \cap Path_{\mathrm{s}}(\sigma'\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} x(c' \cap Path(\sigma'\tau_M)).$$

Take $c' = E[\![N]\!]\rho' y c$, $\sigma' = \sigma[\beta_1 \mapsto \texttt{int}, ..., \beta_k \mapsto \texttt{int}]$ where $\tau_y^{\mathrm{s}} = \forall\beta_1...\beta_k.\tau_M$. Then $Path(\sigma'\tau_M) = Path_{\mathrm{s}}(\sigma\tau_y^{\mathrm{s}})$. Therefore

$$E[\![M]\!]\rho x(E[\![N]\!]\rho' y c) \cap Path_{\mathrm{s}}(\sigma'\tau_x^{\mathrm{s}}) \subseteq \tilde{E}[\![M]\!]\tilde{\rho} x(\tilde{E}[\![N]\!]\tilde{\rho}' y(c \cap Path(\sigma\tau_N))).$$

Notice that $\sigma'\tau_x^{\mathrm{s}} = \sigma\tau_x^{\mathrm{s}}$ since the β_i, $i \in [\![1, k]\!]$ do not appear free in τ_x^{s}. Indeed, x is a variable which is in the environment when y is generalized, hence the generalization of y cannot generalize the free variables of the type of x. Taking the upper bound with the first inequation, we get the required result.

• **The other cases** are left to the reader.

□

This lemma easily leads to the correctness of $\tilde{E}$. Indeed, if $\epsilon \notin \tilde{E}[\![M]\!]\tilde{\rho} x(Path(\tau_M))$, $\epsilon \notin E[\![M]\!]\rho x Path \cap Path_{\mathrm{s}}(\tau_x^{\mathrm{s}})$ (applying the lemma with $\sigma = \mathrm{Id}$ and $c = Path$). Therefore $\epsilon \notin E[\![M]\!]\rho x Path$.

A.3 From monomorphic to polymorphic properties

Proposition 3.24 *Let us consider a property $P^{\tau[\tau']}$ of context transformers $\phi : \tau[\tau'] \in Tr_E$. We assume that $P^{\tau[\tau']}$ is true for constants ($P^{\tau[\tau']}(\lambda c.\emptyset)$ and $P^{\tau[\tau']}(\lambda c.Path(\tau'))$), identity ($P^{\tau[\tau]}(\lambda c.c)$), restriction ($P^{\tau[\tau']}(\lambda c.\delta^{-1}.c)$ where $\delta : \tau \to \tau'$), construction ($P^{\tau[\tau']}(\lambda c.\delta.c)$ where $\delta : \tau' \to \tau$), step ($P^{\tau[\tau']}(\lambda c.\text{if } \epsilon \in c \text{ then } Path(\tau') \text{ else } \emptyset)$). We also assume that $P^{\tau[\tau']}$ is preserved by composition ($P^{\tau[\tau']}(\phi)$ and $P^{\tau'[\tau'']}(\phi')$ implies $P^{\tau[\tau'']}(\phi' \circ \phi)$) and finite and infinite upper bound ($\forall j, P^{\tau[\tau']}(\phi_j)$ implies $P^{\tau[\tau']}(\sqcup_j \phi_j)$). These are all properties we prove when we want to prove property $P^{\tau[\tau']}$ in the monomorphic case.*

Then, in the case of a polymorphic language, $P^{\tau_M[\tau_x]}(E[\![M]\!]\rho x)$ where $x \in Var$ bound by a `fun` *expression (it is wrong in general if x is bound by a* `let` *or* `let rec` *expression), $P^{\tau_M[\tau_{M,i}]}(E[\![M]\!]\rho i)$ where $i \in Ind$, $P^{\tau_y^{\mathrm{g}}[\tau_{y,i}^{\mathrm{g}}]}(\rho[\![y]\!]i)$ for all $i \in Ind$.*

The key property of typing we use to prove this result is the following. In the instantiation $I_E^{\sigma,\tau_x^{\mathrm{g}}[\tau_{x,i}^{\mathrm{g}}]}$, the type variables substituted by σ have never been substituted during the computation of $\rho[\![x]\!]i$. Hence, the generalization $G_E^{\sigma',\tau_x^{\mathrm{g}}}$ has never been applied to these variables, and the property P is true in the cases mentioned above even if it is wrong for the generalization.

Proof: We consider the property P_2 defined by $P_2^{\tau[\tau']}(\phi) \Leftrightarrow \forall\sigma$ substitution on type variables, $\forall K \subseteq Path(\tau) - Path(\sigma\tau), P^{\sigma\tau[\sigma\tau']}(\lambda c.\phi(c \cup K) \cap Path(\sigma\tau'))$. Then P_2 satisfies the same properties as P.

- $P_2^{\tau[\tau']}(\lambda c.\emptyset)$: because $P^{\sigma\tau[\sigma\tau']}(\lambda c.\emptyset)$ and $\lambda c.\phi(c \cup K) \cap Path(\sigma\tau') = \lambda c.\emptyset$.
- $P_2^{\tau[\tau']}(\lambda c.Path(\tau'))$: because $P^{\sigma\tau[\sigma\tau']}(\lambda c.Path(\sigma\tau'))$.
- $P_2^{\tau[\tau]}(\lambda c.c)$: because $P^{\sigma\tau[\sigma\tau]}(\lambda c.c)$ and $\phi(c \cup K) \cap Path(\sigma\tau) = c \cup K \cap Path(\sigma\tau) = c$ if $c \in Ctx_E^{\sigma\tau}$.
- $P_2^{\tau[\tau']}(\lambda c.\delta^{-1}.c)$: since $P^{\sigma\tau[\sigma\tau']}(\lambda c.\delta^{-1}.c)$ and $\phi(c \cup K) \cap Path(\sigma\tau') = \delta^{-1}.(c \cup K) \cap Path(\sigma\tau') = \delta^{-1}.c$. Indeed, $c \subseteq Path(\sigma\tau) = \cup_{\delta_j} \delta_j.Path(\sigma\tau_j)$ where $\delta_j : \tau \to \tau_j$, therefore $\delta^{-1}.c \subseteq Path(\sigma\tau')$ and $\delta^{-1}.K \cap Path(\sigma\tau') = \emptyset$ because paths in K are not in $Path(\sigma\tau)$.
- $P_2^{\tau[\tau']}(\lambda c.\delta.c)$ similarly.
- $P_2^{\tau[\tau']}(\lambda c.\text{if } \epsilon \in c \text{ then } Path(\tau') \text{ else } \emptyset)$: since $P^{\sigma\tau[\sigma\tau']}(\lambda c.\text{if } \epsilon \in c \text{ then } Path(\sigma\tau') \text{ else } \emptyset)$ (notice that $\epsilon \notin K$, therefore $\epsilon \in c$ if and only if $\epsilon \in c \cup K$).
- $P_2^{\tau[\tau']}(\phi)$ and $P_2^{\tau'[\tau'']}(\phi')$ implies $P_2^{\tau[\tau'']}(\phi' \circ \phi)$: $\lambda c.(\phi' \circ \phi)(c \cup K) \cap Path(\sigma\tau'') = (\lambda c'.\phi'(c' \cup (\phi(c \cup K) - Path(\sigma\tau')) \cap Path(\sigma\tau'')) \circ (\lambda c.\phi(c \cup K) \cap Path(\sigma\tau'))$. Since P is preserved by composition, we have the result.
- $\forall j, P_2^{\tau[\tau']}(\phi_j)$ implies $P_2^{\tau[\tau']}(\sqcup_j \phi_j)$: $\lambda c.(\sqcup_j \phi_j)(c \cup K) \cap Path(\sigma\tau') = \sqcup_j(\lambda c.\phi_j(c \cup K) \cap Path(\sigma\tau'))$. Since P is preserved by upper bound, we have the result.

Then we prove that

$$\begin{cases} \text{If } y \in Var, & P_2^{\tau_M[\tau_x]}(\phi \circ (E[\![M]\!]\rho y)) \\ \text{If } i \in Ind, & P_2^{\tau_M[\tau_{M,i}]}(E[\![M]\!]\rho i) \end{cases}$$

where $\phi : \tau_y^{\mathrm{g}}[\tau_x]$, τ_x does not contain any type variable which is bound in the type scheme of τ_y^{g} (or in a type scheme of a variable of smaller scope than y), ϕ is additive and $P_2^{\tau_y^{\mathrm{g}}[\tau_x]}(\phi)$. ρ is such that $P_2^{\tau_z^{\mathrm{g}}[\tau_{z,i}^{\mathrm{g}}]}(\rho[\![z]\!]i)$. The proof is by induction on M.

For the variable case z, we have in fact three cases:

- First case: $y \in \mathit{Var}$ and $y \neq z$. We have $E[\![z]\!]\rho y = \lambda c.\emptyset$, hence $\phi \circ (E[\![M]\!]\rho y) = \lambda c.\emptyset$ hence the result.

- Second case: $y = z$ (Generalization). If y is bound by `fun`, $E[\![z]\!]\rho y = \lambda c.c$, and $P_2^{\tau_y^{\mathrm{g}}[\tau_x]}(\phi)$ hence the result.

  If y is bound by `let` or `let rec`, we have to prove $P_2^{\sigma\tau_y^{\mathrm{g}}[\tau_x]}(\phi)$. We have $P_2^{\tau_y^{\mathrm{g}}[\tau_x]}(\phi)$ i.e. $\forall\sigma'$ substitution on type variables, $\forall K \subseteq \mathit{Path}(\tau_y^{\mathrm{g}}) - \mathit{Path}(\sigma'\tau_x), P^{\sigma'\tau_y^{\mathrm{g}}[\sigma'\tau_x]}(\lambda c.\phi(c \cup K) \cap \mathit{Path}(\sigma'\tau_x))$. Put $\sigma' = \sigma'' \circ \sigma$. Then $\forall K \subseteq \mathit{Path}(\tau_y^{\mathrm{g}}) - \mathit{Path}(\sigma''\tau_x), P^{\sigma''\sigma\tau_y^{\mathrm{g}}[\sigma''\tau_x]}(\lambda c.\phi(c \cup K) \cap \mathit{Path}(\sigma''\tau_x))$ since τ_x is not changed by σ since it does not contain type variables bound by the `let` defining y or another variable of smaller scope than y. This yields $P_2^{\sigma\tau_y^{\mathrm{g}}[\tau_x]}(\phi)$.

- Third case: $i \in \mathit{Ind}$ (Instantiation). We have $P_2^{\tau_z^{\mathrm{g}}[\tau_{z,i}^{\mathrm{g}}]}(\rho[\![z]\!]i)$, and we prove $P_2^{\sigma\tau_z^{\mathrm{g}}[\sigma\tau_{z,i}^{\mathrm{g}}]}(\lambda c.\rho[\![z]\!]ic \cap \mathit{Path}(\sigma\tau_{z,i}^{\mathrm{g}}))$. $\forall\sigma'$ substitution on type variables, $\forall K \subseteq \mathit{Path}(\tau_z^{\mathrm{g}}) - \mathit{Path}(\sigma'\tau_{z,i}^{\mathrm{g}}), P^{\sigma'\tau_z^{\mathrm{g}}[\sigma'\tau_{z,i}^{\mathrm{g}}]}(\lambda c.\phi(c \cup K) \cap \mathit{Path}(\sigma'\tau_{z,i}^{\mathrm{g}}))$. Put $\sigma' = \sigma'' \circ \sigma$, and we have the result immediately.

Other cases are straightforward by induction hypothesis. Let us prove the `let` $z = M$ `in` N case to give an example.

- First ρ' satisfies the hypothesis on ρ, by induction hypothesis on M.

- Case $y \in \mathit{Var}$.

$$\phi \circ (E[\![N]\!]\rho' y \sqcup (E[\![M]\!]\rho y) \circ (E[\![N]\!]\rho' z)) = \phi \circ E[\![N]\!]\rho' y \sqcup \phi \circ (E[\![M]\!]\rho y) \circ (E[\![N]\!]\rho' z)$$

  by additivity of ϕ. The first term satisfies P_2 by induction hypothesis on N. $\psi = \phi \circ (E[\![M]\!]\rho y)$ satisfies P_2 by induction hypothesis on M. ψ is additive, and variables of τ_x are not bound in the type scheme of τ_z^{g} or of a variable of smaller scope than z (since z is of smaller scope than y, and by hypothesis the variables of τ_x are not bound in type schemes of variables of smaller scope than y). Therefore we can apply the induction hypothesis on N (with function ψ instead of ϕ), and prove that the second term satisfies P_2. Hence the result.

- Case $i \in \mathit{Ind}$. Obvious by induction hypothesis on N.

We also prove the `fun` $z \to M$ case.

- Notice that $\rho_\top$ satisfies the hypothesis on ρ, since $P_2^{\tau[\tau']}(\lambda c.\mathit{Path}(\tau'))$.

- For the first parameter, take $z = x$, $\phi = \mathrm{Id}$ (possible since the type scheme of z has no bound variables) to apply the induction hypothesis on M: $P_2^{\tau_M[\tau_z]}(E[\![M]\!]\rho[z \mapsto \rho_\top]z)$. Moreover, $P_2^{\tau[\tau_M]}(\lambda c.\mathrm{app}^{-1}.c)$. Since P_2 is preserved by composition, this yields the result.

- For the other parameters, the result is easy to prove: $P_2^{\tau[\tau_M]}(\lambda c.\mathrm{app}^{-1}.c)$, $E[\![M]\!]\rho[y \mapsto \rho_\top]i$ satisfies P_2 by induction hypothesis and P_2 is preserved by composition. For the variables, the technique is similar except that the induction hypothesis becomes: $\phi \circ E[\![M]\!]\rho[y \mapsto \rho_\top]x$ satisfies P_2.

The last step is to infer P from P_2. First, we can notice that if y is bound by `fun`, we can take $\phi = \mathrm{Id}$. Then, if $y \in \mathit{Var}$ is bound by `fun`, $P_2^{\tau_M[\tau_y]}(E[\![M]\!]\rho y)$. If we take $\sigma = \mathrm{Id}$, $K = \emptyset$ in the definition of P_2, we have $P_2^{\tau[\tau']}(\phi) \Rightarrow P^{\tau[\tau']}(\lambda c.\phi(c) \cap \mathit{Path}(\tau')) \Rightarrow P^{\tau[\tau']}(\phi)$ hence the expected result. $\square$

Appendix B

Proofs for analysis T

B.1 Definition of the graph of a type

Lemma 3.28 *If there is a morphism ϕ from G to G', and another graph morphism ϕ' from G' to G, then $\phi = {\phi'}^{-1}$ and ϕ is an isomorphism from G to G'.*

Proof: Let n_0 be the top node of G, n'_0 of G'. Then $\phi(n_0) = n'_0$ and $\phi'(n'_0) = n_0$.

If $n[\tau]$ is a node of G and $n'[\tau]$ a node of G' such that $\phi(n) = n'$ and $\phi(n') = n$, then for each accessor $\delta : \tau \to \tau'$, there is a single edge $n \xrightarrow{\delta} n_2$ in G and a single edge $n' \xrightarrow{\delta} n'_2$ in G'. By definition of a morphism, $\phi(n_2)$ is the end of an edge $\phi(n) \xrightarrow{\delta}$, therefore $\phi(n_2) = n'_2$. In the same way, $\phi'(n'_2) = n_2$.

Let n be any node in G. n is reachable from n_0 by a path $n_0 \xrightarrow{\delta} n_1 .... \xrightarrow{\delta'} n$. By the first step, $\phi'(\phi(n_0)) = n_0$. By the second step, $\phi'(\phi(n_1)) = n_1$, ..., $\phi'(\phi(n)) = n$. We can perform a similar proof for n' node of G'. Hence the result $\phi = {\phi'}^{-1}$. □

Theorem 3.31 (Existence and uniqueness) *For each type τ, there exists a unique (up to isomorphism) graph satisfying Definition 3.30.*

Proof: For the type τ, let us define a tree T, with exactly the same definition as the graphs of τ, but all nodes are distinct (that is there is one incoming edge for each node, except for the top which has no incoming edge). This tree is infinite for recursive types. Let n_0 be the top node of T.

Let R be the relation: $n_1 R n_2$ if and only if n_1 and n_2 are labelled by the same type, and there is a path in T from n_1 to n_2. Let R' be defined by: $n'_1 R' n'_2$ if and only if there are two nodes n_1 and n_2 such that $n_1 R n_2$ and there are two paths from n_1 to n'_1 and from n_2 to n'_2 labelled by the same sequence of accessors. Let R_{eq} be the symmetric and transitive closure of R'. R_{eq} is an equivalence relation.

Let G be the quotient of T by R_{eq}, that is the graph defined by

- its nodes are the equivalence classes of R_{eq}, labelled by the common type labelling all nodes of the same equivalence class of R_{eq};
- its edges are $n_1 \xrightarrow{\delta} n_2$ if there is $n'_1 \in n_1$, $n'_2 \in n_2$ such that $n'_1 \xrightarrow{\delta} n'_2$ is an edge of T.

Let us show that G satisfies Definition 3.30.

First, we show that if $n_1 R' n_2$, $n_1 \xrightarrow{\delta} n'_1$ and $n_2 \xrightarrow{\delta} n'_2$ then $n'_1 R' n'_2$.

Since $n_1 R' n_2$, there are N_1 and N_2 such that $N_1 R N_2$, there is a path p_1 from N_1 to n_1, p_2 from N_2 to n_2 and p_1 and p_2 are labelled by the same accessors. If we consider the paths $p_1 : n_1 \xrightarrow{\delta} n'_1$ and $p_2 : n_2 \xrightarrow{\delta} n'_2$, they are labelled by the same accessors, and thus $n'_1 R' n'_2$.

Now, applying several times this result, we can extend it to the transitive closure R_{eq} of R': if $n_1 R_{\text{eq}} n_2$, $n_1 \xrightarrow{\delta} n'_1$ and $n_2 \xrightarrow{\delta} n'_2$ then $n'_1 R_{\text{eq}} n'_2$.

The first consequence of this result is that there is at most one edge labelled by a given accessor δ outgoing from a given node n in G. Indeed, if $n_1 \in n$, $n_2 \in n$, $n_1 \xrightarrow{\delta} n'_1$ and $n_2 \xrightarrow{\delta} n'_2$ then $n_1 R_{\text{eq}} n_2$, hence by the above result $n'_1 R_{\text{eq}} n'_2$. Therefore, n'_1 and n'_2 are in the same node of G.

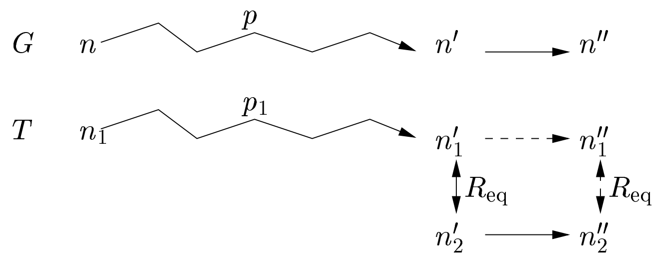


Figure B.1: For each path in G, there is a corresponding path in T.

The second consequence of this result is that if we have a path p in G from n to n', then there is a path in T from $n_1 \in n$ to $n'_1 \in n'$ labelled by the same accessors as p (see Figure B.1). This is obvious by definition of G is p has only one edge. Let us assume that this is true for p from n to n', and let us prove that it is also true for $p : n' \xrightarrow{\delta} n''$. By induction hypothesis, there is a path p_1 in T from $n_1 \in n$ to $n'_1 \in n'$. By definition of G, there an edge $n'_2 \xrightarrow{\delta} n''_2$ with $n'_2 \in n'$, $n''_2 \in n''$. Since n'_1 and n'_2 are both in n', $n'_1 R_{\text{eq}} n'_2$. There is an edge $n'_1 \xrightarrow{\delta} n''_1$ in T. By the above result, $n''_1 R_{\text{eq}} n''_2$, therefore $n''_1 \in n''$, and the path $p_1 : n'_1 \xrightarrow{\delta} n''_1$ gives the result.

If there is a path p in G from n to n' and n and n' are labelled by the same type, then there is a path p_1 in T from $n_1 \in n$ to $n'_1 \in n'$. This proves that $n_1 R n'_1$, hence $n_1 R_{\text{eq}} n'_1$. Then n and n' are in fact the same node, and the path p is cyclic. We have proved the first property of the graph of τ.

To prove the second property, let us consider another graph G' of τ which has the same property. We define a graph morphism ϕ from G to G'. We can easily define a graph morphism ϕ_T from T to G' (the definition of a morphism gives a unique solution). We only have to show that if $n_1 R_{\text{eq}} n_2$, then $\phi_T(n_1) = \phi_T(n_2)$. Then, the image by ϕ of the equivalence class of n_1 will be $\phi_T(n_1)$, which is independent of n_1.

If $n_1 R n_2$, there is a path p from n_1 to n_2 in T, hence by the graph morphism ϕ_T, there is a path p' from $\phi_T(n_1)$ to $\phi_T(n_2)$ in G' (just take the image of p by ϕ_T). If $\phi_T(n_1) \neq \phi_T(n_2)$, we remove all cycles of p' and get an acyclic path from $\phi_T(n_1)$ to $\phi_T(n_2)$ which contradicts the first property. Therefore, $\phi_T(n_1) = \phi_T(n_2)$.

If $n_1 R' n_2$, there are N_1 and N_2 such that $N_1 R N_2$, there is a path p_1 from N_1 to n_1 and p_2 from N_2 to n_2 in T, and p_1 and p_2 are labelled by the same accessors. $\phi_T(N_1) = \phi_T(N_2)$.

We want to show that $\phi_T(n_1) = \phi_T(n_2)$.

We use the following intermediate result. If we have $n = \phi_T(n_1) = \phi_T(n_2)$, $n_1 \xrightarrow{\delta} n'_1$, $n_2 \xrightarrow{\delta} n'_2$, then we have edges $n \xrightarrow{\delta} \phi_T(n'_1)$ and $n \xrightarrow{\delta} \phi_T(n'_2)$ in G'. Since there is only one outgoing edge of n for each accessor, $\phi_T(n'_1) = \phi_T(n'_2)$. Applying this result for each edge of path p_1, we get the required result.

If $n_1 R_{\mathrm{eq}} n_2$, we have the same result by transitivity and symmetry of equality.

This terminates the proof of existence.

Uniqueness (up to isomorphism): if we have two graphs of τ, G and G', satisfying Definition 3.30, $G \geq G'$ and $G' \geq G$. Therefore, by Lemma 3.28, G and G' are isomorphic. $\square$

B.2 Generalization

Lemma 3.35 *If two paths p and p' lead to the same node in $\mathrm{Gr}(\tau)$ (i.e. $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p')$), and they do not go inside a type variable substituted by σ (i.e. $\exists n, \tau_1$, such that $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) = \{n[\sigma\tau_1]\}$, and idem for p') then they also lead to the same node in $\mathrm{Gr}(\sigma\tau)$.*

Proof: Let T_τ be the tree corresponding to type τ as in Theorem 3.31. Let n such that $\{n\} = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p')$. Then, as we have seen in Theorem 3.31, n is an equivalence class of R_{eq} that contains both $\{n_1\} = \mathrm{NofP}_{T_\tau}(p)$ and $\{n'_1\} = \mathrm{NofP}_{T_\tau}(p')$ (we extend the definition of NofP to work on trees).

Now, the tree T_τ and the tree $T_{\sigma\tau}$ restricted to nodes labelled by types of the form $\sigma\tau'$ are clearly isomorphic. Let $\psi : T_\tau \to T_{\sigma\tau}$ be the isomorphism. Then $\mathrm{NofP}_{T_{\sigma\tau}}(p) = \{\psi(n_1)\}$, $\mathrm{NofP}_{T_{\sigma\tau}}(p') = \{\psi(n'_1)\}$. Noticing that $\tau_2 = \tau'_2 \Rightarrow \sigma\tau_2 = \sigma\tau'_2$ (this simple remark is in fact the key point of the proof), we have $n_2[\tau_2]\, R\, n'_2[\tau'_2] \Rightarrow \psi(n_2)[\sigma\tau_2]\, R\, \psi(n'_2)[\sigma\tau'_2]$, then $n_3\, R'\, n'_3 \Rightarrow \psi(n_3)\, R'\, \psi(n'_3)$ and $n_4\, R_{\mathrm{eq}}\, n'_4 \Rightarrow \psi(n_4)\, R_{\mathrm{eq}}\, \psi(n'_4)$. (The converse of all these implications is wrong, because $\sigma\tau_1 = \sigma\tau'_1$ does not imply $\tau_1 = \tau'_1$).

Therefore, $\psi(n_1)$ and $\psi(n'_1)$ are in the same equivalence class of R_{eq} hence they correspond to the same node in $\mathrm{Gr}(\sigma\tau)$. $\square$

B.3 Instantiation

Lemma 3.37 *Assume that $\phi_E : \tau[\tau']$. We say that $\phi_E \in Tr'_E$ if ϕ_E satisfies the property:*

> *If* $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon$, *let* $c = \phi_E(p) - \phi_E(\emptyset)$. *Then* $\forall p'' \in c, \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p'') = \epsilon$ *and* $\forall p' \in Path, \phi_E(p.p') = \mathrm{Ins}^{\tau'}_{p'}(c) \cup \phi_E(\emptyset)$.

In the monomorphic case, all context transformers ϕ_E are in Tr'_E. In the polymorphic case, we only have $E[\![M]\!]\rho x \in Tr'_E$ if $x \in Ind$ or $x \in Var$ bound by a `fun` expression (it is wrong in general if x is bound by a `let` or `let rec` expression) and $\rho[\![y]\!]i \in Tr'_E$ if $i \in Ind$.

Proof: We prove the properties required by Proposition 3.24, P being the property:

$$\begin{aligned} P^{\tau[\tau']}(\phi_E) \Leftrightarrow & (P1)\phi_E \in Tr'_E \wedge \\ & (P2)\text{if } c \in Ctx^{\tau}_E \text{ is closed, } \phi_E(c) \in Ctx^{\tau'}_E \text{ is closed.} \end{aligned}$$

Proposition 3.24 will then give the result immediately. For $(P2)$, the required properties have already been proved in Proposition 3.16, so we will not prove them again. However, property $(P2)$ is necessary because we need to know that all ϕ_E manipulated in this proof satisfy $(P2)$, and property $(P2)$ is not true for all ϕ_E (see Proposition 3.25).

- **Constants.** Obvious. $c = \emptyset$, $\phi_E(p.p') = \phi_E(\emptyset)$.

- **Identity.** Obvious. $c = \{p\}$, $\phi_E(p.p') = \{p.p'\} = \mathrm{Ins}_{p'}^{\tau'}(\{p\})$.

- **Construction.** Let $\delta : \tau' \to \tau$. We have $\mathrm{Constr}_E(\delta, \tau')(c') = \{\delta.p | p \in c'\}$. Then $c = \{\delta.p\}$.

$$\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(\delta.p) = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon$$
$$\mathrm{Ins}_{p'}^{\tau'}(c) \cup \mathrm{Constr}_E(\delta, \tau')(\emptyset) = \{\delta.p.p'\} = \mathrm{Constr}_E(\delta, \tau)(p.p')$$

- **Restriction.** Let $\delta : \tau \to \tau'$. The restriction is defined by $\mathrm{Restr}_E(\delta, \tau)(c') = \{p | \delta.p \in c'\}$.
  - If $p = \delta.p''$, $c = \{p''\}$ and $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p'') = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(\delta.p'') = \epsilon$.
  $$\mathrm{Ins}_{p'}^{\tau'}(c) \cup \mathrm{Restr}_E(\delta, \tau)(\emptyset) = \{p''.p'\} = \mathrm{Restr}_E(\delta, \tau)(p.p')$$
  - If $p = \delta'.p''$, with $\delta' \neq \delta$, then $c = \emptyset$.
  $$\mathrm{Ins}_{p'}^{\tau'}(c) \cup \mathrm{Restr}_E(\delta, \tau)(\emptyset) = \emptyset = \mathrm{Restr}_E(\delta, \tau)(p.p')$$
  - It is impossible to have $p = \epsilon$. Indeed, if we had $p = \epsilon$, $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(\epsilon) = \epsilon$ implies τ is a type variable and there is no accessor for type variables.

- **Step.** $\phi_E = \mathrm{test}_E^{\tau,\tau'} = \lambda c.\mathrm{if}\ \epsilon \in c\ \mathrm{then}\ \psi(Path)\ \mathrm{else}\ \emptyset$. If we had $p = \epsilon$, $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(\epsilon) = \epsilon$ implies τ is a type variable. Since the step $\mathrm{test}_E^{\tau,\tau'}$ is only used with τ function type, this is impossible. Therefore $p \neq \epsilon$, $\phi_E(p) = \emptyset$, $c = \emptyset$, $\phi_E(p.p') = \emptyset = \phi_E(\emptyset)$.

- **Composition.** Assume by induction hypothesis that ϕ_E and ϕ'_E satisfy this Lemma:

$$\begin{aligned}
&\text{If } \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon, c = \phi_E(p) - \phi_E(\emptyset),\\
&\quad \forall p' \in c, \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p') = \epsilon \text{ and } \phi_E(p.p_2) = \mathrm{Ins}_{p_2}^{\tau'}(c) \cup \phi_E(\emptyset)\\
&\text{If } \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p') = \epsilon, c' = \phi'_E(p') - \phi'_E(\emptyset),\\
&\quad \forall p'' \in c', \mathrm{RemPath}_{\mathrm{Gr}(\tau'')}(p'') = \epsilon \text{ and } \phi'_E(p'.p_2) = \mathrm{Ins}_{p_2}^{\tau''}(c') \cup \phi'_E(\emptyset)
\end{aligned}$$

We apply this hypothesis for all $p' \in c$ (since they are such that $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p') = \epsilon$) and take the upper bound, we obtain:

$$\phi'_E(\{p'.p_2 | p' \in c\}) = \mathrm{Ins}_{p_2}^{\tau''}(\phi'_E(c) - \phi'_E(\emptyset)) \cup \phi'_E(\emptyset)$$

Take the upper bound of each equation member with $(\phi'_E \circ \phi_E)(\emptyset)$:

$$\begin{aligned}
\phi'_E(\mathrm{Ins}_{p_2}^{\tau'}(c) \cup \phi_E(\emptyset)) &= \mathrm{Ins}_{p_2}^{\tau''}(\phi'_E(c) - \phi'_E(\emptyset)) \cup (\phi'_E \circ \phi_E)(\emptyset)\\
(\phi'_E \circ \phi_E)(p.p_2) &= \mathrm{Ins}_{p_2}^{\tau''}(\phi'_E(c) - \phi'_E(\emptyset)) \cup (\phi'_E \circ \phi_E)(\emptyset)\\
(\phi'_E \circ \phi_E)(p.p_2) &= \mathrm{Ins}_{p_2}^{\tau''}(\phi'_E(c) - (\phi'_E \circ \phi_E)(\emptyset)) \cup (\phi'_E \circ \phi_E)(\emptyset)
\end{aligned}$$

since all paths in $\mathrm{Ins}_{p_2}^{\tau''}((\phi'_E \circ \phi_E)(\emptyset)) = \{p'_1.p_2 | p'_1 \in (\phi'_E \circ \phi_E)(\emptyset), \mathrm{RemPath}_{\mathrm{Gr}(\tau'')}(p'_1) = \epsilon\}$ are anyway contained in $(\phi'_E \circ \phi_E)(\emptyset)$ (since $\phi'_E \circ \phi_E$ satisfies $(P2)$ since $(P2)$ is preserved by composition) and therefore they are always contained in the old and the new equality second members.

Let $c'' = (\phi'_E \circ \phi_E)(p) - (\phi'_E \circ \phi_E)(\emptyset)$.

$$c'' = \phi'_E(\phi_E(p) - \phi_E(\emptyset) \cup \phi_E(\emptyset)) - (\phi'_E \circ \phi_E)(\emptyset) = \phi'_E(c) - (\phi'_E \circ \phi_E)(\emptyset)$$

So, $(\phi'_E \circ \phi_E)(p.p_2) = \mathrm{Ins}_{p_2}^{\tau''}(c'') \cup (\phi'_E \circ \phi_E)(\emptyset)$. Moreover, $c'' \subseteq \phi'_E(c) - \phi'_E(\emptyset)$. For all $p' \in c, \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p') = \epsilon$, therefore by induction hypothesis on ϕ'_E, we have $\forall p'' \in c''$, $\mathrm{RemPath}_{\mathrm{Gr}(\tau'')}(p'') = \epsilon$ hence the result.

- **Upper bound.** Assume by induction hypothesis that $\phi_{Ej} \in Tr'_E$

$$c_j = \phi_{Ej}(p) - \phi_{Ej}(\emptyset), \forall p' \in c_j, \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p') = \epsilon, \phi_{Ej}(p.p_2) = \mathrm{Ins}_{p_2}^{\tau'}(c_j) \cup \phi_{Ej}(\emptyset)$$

Taking the upper bound of these equations, we obtain

$$\begin{aligned}
c'' &= (\sqcup_j \phi_{Ej})(p) - (\sqcup_j \phi_{Ej})(\emptyset) \subseteq \cup_j c_j \\
(\sqcup_j \phi_{Ej})(p.p_2) &= \mathrm{Ins}_{p_2}^{\tau'}(\cup_j c_j) \cup (\sqcup_j \phi_{Ej})(\emptyset) \\
&= \mathrm{Ins}_{p_2}^{\tau'}(c'') \cup \mathrm{Ins}_{p_2}^{\tau'}(\cup_j c_j - c'') \cup (\sqcup_j \phi_{Ej})(\emptyset)
\end{aligned}$$

and $\cup_j c_j - c'' = \cup_j(\phi_{Ej}(p) - \phi_{Ej}(\emptyset)) - (\cup_j \phi_{Ej}(p) - \cup_j \phi_{Ej}(\emptyset)) \subseteq \cup_j \phi_{Ej}(p) - (\cup_j \phi_{Ej}(p) - \cup_j \phi_{Ej}(\emptyset)) \subseteq \cup_j \phi_{Ej}(\emptyset)$. Therefore $\mathrm{Ins}_{p_2}^{\tau'}(\cup_j c_j - c'') \subseteq \mathrm{Ins}_{p_2}^{\tau'}(\cup_j \phi_{Ej}(\emptyset)) \subseteq \cup_j \phi_{Ej}(\emptyset)$ since this context is closed by $(P2)$. Then

$$(\sqcup_j \phi_{Ej})(p.p_2) = \mathrm{Ins}_{p_2}^{\tau'}(c'') \cup (\sqcup_j \phi_{Ej})(\emptyset)$$

and $\forall p'' \in c''$, $\exists j, p'' \in c_j$, therefore $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p'') = \epsilon$ which is the expected result. □

Definition 3.38

$$\gamma_T^{\tau[\tau']}(\phi_T) = \lambda c. \cup \{\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \gamma_T^{\tau'}(\phi_T(\tau))) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau))$$

The instantiation function for predicate transformers is then:

$$\begin{aligned}
I_T^{\tau[\tau']} &= \alpha_T^{\sigma\tau[\sigma\tau']} \circ \gamma_T^{\tau[\tau']} \\
I_T^{\tau[\tau']}(\phi_T)(c) &= \alpha_T^{\sigma\tau'}(\cup\{\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau'}(\phi_T \circ \alpha_T^{\tau}(p)) - \gamma_T^{\tau'}(\phi_T(\tau))) | p \in \gamma_T^{\sigma\tau}(c)\} \\
&\qquad \cup \gamma_T^{\tau'}(\phi_T(\tau)))
\end{aligned}$$

We also define the instantiation function for contexts by: $I_T^{\tau} = \alpha_T^{\sigma\tau} \circ \gamma_T^{\tau}$

Proposition 3.39 *The context transformers $\phi : Ctx_T^{\tau} \to Ctx_T^{\tau'}$ for analysis T are additive, that is: $\phi(\cup_i c_i) = \cup_i \phi(c_i)$. They can be put into the form:*

$$\lambda c. \cup \{i_s | s \in c\} \cup i_0 = \lambda c. \cup \{\phi(s) | s \in c\} \cup \phi(\emptyset)$$

where $i_s, i_0 \in Ctx_T^{\tau'}$.

Proof: We can easily check for each elementary operation of analysis T that it is additive. Additivity is preserved by instantiation, composition, upper bound and fixed point, therefore all context transformers of analysis T are additive. The above forms can represent all additive functions. □

Definition 3.40 The set of types labelling nodes of a context c is

$$\mathrm{Types}(c) = \{\tau | n[\tau] \in c\}.$$

These types are said to be the types marked in context c.

Lemma 3.41 $\mathrm{Types}(G_T^\tau(c)) = \{\tau' | \sigma\tau' \in \mathrm{Types}(c)\} \cup \{\beta$ *type variable* $| \exists \tau' \in \mathrm{Types}(c), \tau'$ *is the label of a node of* $\mathrm{Gr}(\sigma\beta)\}$.

Similarly for instantiation, $\mathrm{Types}(I_T^\tau(c)) = \{\sigma\tau' | \tau' \in \mathrm{Types}(c)\} \cup \{$ *types labelling* $\mathrm{Gr}(\sigma\beta) | \beta \in \mathrm{Types}(c), \beta$ *type variable* $\}$.

Proof: Let $p \in \mathit{Path}(\sigma\tau)$. We prove by induction on p that $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p)$ is labelled τ_1 and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p)$ is labelled τ_2 with $\tau_2 = \sigma\tau_1$ or τ_1 is a type variable and τ_2 is a type of the graph of $\sigma\tau_1$.

If $p = \epsilon$, $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p)$ is labelled τ, and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p)$ is labelled $\sigma\tau$.

If $p = p'.\delta$, by induction hypothesis, $\{n_1\} = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p')$ is labelled τ_1, and $\{n_2\} = \mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p')$ is labelled τ_2 with $\tau_2 = \sigma\tau_1$ or τ_1 is a type variable and τ_2 is a type of the graph of $\sigma\tau_1$. If τ_1 is not a type variable, then $\mathrm{Gr}(\tau)$ contains an edge $n_1 \xrightarrow{\delta} n_1'$ with n_1' labelled τ_1', and $\mathrm{Gr}(\sigma\tau)$ contains an edge $n_2 \xrightarrow{\delta} n_2'$ with n_2' labelled $\tau_2' = \sigma\tau_1'$. Then $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p) = \{n_1'\}$ and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) = \{n_2'\}$ hence we have the result. If τ_1 is a type variable, $\{n_1\} = \mathrm{NofP}_{\mathrm{Gr}(\tau)}(p)$ and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p)$ is a node since $p \in \mathit{Path}(\sigma\tau)$ and this node is reachable from n_2. Since n_2 is labelled by a type of $\mathrm{Gr}(\sigma\tau_1)$, this node $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p)$ is also labelled by a type of $\mathrm{Gr}(\sigma\tau_1)$.

We have $t \in \mathrm{Types}(G_T^\tau(c))$ if and only if there exists $p \in \mathit{Path}(\sigma\tau)$, $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p)$ is labelled t and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) \subseteq c$. If t is not a type variable, $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p)$ is labelled σt hence $\sigma t \in \mathrm{Types}(c)$. Conversely, if $\sigma t \in \mathrm{Types}(c)$, then there exists $p \in \mathit{Path}(\sigma\tau)$ such that $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p) \subseteq c$ and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p)$ is labelled σt. Then $t \in \mathrm{Types}(G_T^\tau(c))$.

We proceed similarly if t is a type variable, and in the case of instantiation. □

Definition B.9 A type τ is said to contain a type τ' if there exists a node in the graph of τ which is labelled τ'.

Proposition 3.42 *If the language is monomorphic,*

$$\text{if } c \text{ does not contain a node labelled by a function type, } \mathrm{Types}(\phi(c) - \phi(\tau)) \subseteq \mathrm{Types}(c) \tag{3.1}$$

In the case of a polymorphic language, (3.1) *is only true if* $\phi = T[\![M]\!]\rho x$ *with* $x \in \mathit{Ind}$ *or* x *has been bound in by a* `fun` *expression, or* $\phi = \rho[\![y]\!]i$ *with* $i \in \mathit{Ind}$. *In particular, this is true for all functions* ϕ *that may be the parameters of* $I_T^{\tau[\tau']}$. *It is not always true for* $\phi = T[\![M]\!]\rho x$ *if* x *has been bound by a* `let` *or* `let rec` *expression.*

Proof: To show (3.1) in the case of a monomorphic language, we only have to perform an induction on elementary operations used to compute T.

- Step. If c does not contain a node labelled by a function type, $\phi(c) = \phi(\tau) = \tau'$ since only functions are used as steps. Hence the result.

- Construction. $\mathrm{Constr}_T(\delta, \tau)(c) = \{\psi(n) | n \in c\}$ where ψ is a graph isomorphism. Therefore, the types marked in $\mathrm{Constr}_T(\delta, \tau)(c)$ are exactly the ones marked in c, therefore $\mathrm{Types}(\phi(c)) \subseteq \mathrm{Types}(c)$.

- Restriction. The types marked in $\mathrm{Restr}_T(\delta, \tau)(c)$ are marked in c (the only difference is that nodes of $\mathrm{Gr}(\tau)$ which are not reachable by a path beginning with δ in $\mathrm{Gr}(\tau)$ are not in $\mathrm{Restr}_T(\delta, \tau)(c)$). This gives $\mathrm{Types}(\phi(c)) \subseteq \mathrm{Types}(c)$.

- Constant. $\mathrm{Types}(\phi(c) - \phi(\tau)) = \emptyset \subseteq \mathrm{Types}(c)$.

- Identity. $\mathrm{Types}(\phi(c) - \phi(\tau)) = \mathrm{Types}(c)$.

Moreover (3.1) is preserved by composition, (finite and infinite) upper bound, and therefore by fixed point.

- Composition. Since $\mathrm{Types}(\phi_2(c) - \phi_2(\tau)) \subseteq \mathrm{Types}(c)$, $\phi_2(c) - \phi_2(\tau)$ does not mark functions. If c' does not mark functions, $\mathrm{Types}(\phi_1(c') - \phi_1(\tau')) \subseteq \mathrm{Types}(c')$, then $\mathrm{Types}(\phi_1(\phi_2(c) - \phi_2(\tau)) - \phi_1(\tau')) \subseteq \mathrm{Types}(\phi_2(c) - \phi_2(\tau)) \subseteq \mathrm{Types}(c)$. Moreover, $\phi_1(\tau') \subseteq \phi_1(\phi_2(\tau))$ therefore $\mathrm{Types}(\phi_1(\phi_2(c)) - \phi_1(\phi_2(\tau))) \subseteq \mathrm{Types}(c)$.

- Upper bound.

$$\begin{aligned}\mathrm{Types}((\sqcup_j \phi_j)(c) - (\sqcup_j \phi_j)(\tau)) &= \mathrm{Types}(\cup_j \phi_j(c) - \cup_j \phi_j(\tau)) \\ &\subseteq \mathrm{Types}(\cup_j (\phi_j(c) - \phi_j(\tau))) \subseteq \mathrm{Types}(c)\end{aligned}$$

The technique used to extend the proof to a polymorphic language is very similar to the proof of Proposition 3.24. For a polymorphic language, we prove that

$$\begin{cases}\text{If } y \in \mathit{Var}, & \phi_T \circ (T[\![M]\!]\rho y) \text{ satisfies (3.1)} \\ \text{If } y \in \mathit{Ind}, & T[\![M]\!]\rho y \text{ satisfies (3.1)}\end{cases}$$

where $\phi_T : \mathit{Ctx}_T^{\tau_y^{\mathrm{g}}} \to \mathit{Ctx}_T^{\tau_x}$, τ_x does not contain any type variable which is bound in the type scheme of τ_y^{g} (or in a type scheme of a variable of smaller scope than y), ϕ_T is additive and satisfies (3.1). ρ is such that $\rho[\![z]\!]i$ satisfies (3.1). The proof is by induction on M.

For the variable case z, we have in fact three cases:

- First case: $y \in \mathit{Var}$ and $y \neq z$. We have $T[\![z]\!]\rho y = \lambda c.\emptyset$, hence the result is obvious.

- Second case: $y = z$ (Generalization). If y is not bound by `let`, $T[\![z]\!]\rho y = \lambda c.c$, and ϕ_T satisfies (3.1), therefore the result is obvious. If y is bound by `let`, $T[\![z]\!]\rho y = G_T^{\sigma, \tau_y^{\mathrm{g}}}$.

  The key point is that, by hypothesis, no type of the graph of τ_x contains a type variable substituted by σ, since it contains no variable bound in the type scheme of y.

  Now, let $\tau = \tau_y^{\mathrm{g}}$. If $t \in \mathrm{Types}(\phi_T \circ G_T^{\sigma,\tau}(c) - \phi_T \circ G_T^{\sigma,\tau}(\sigma\tau))$, then $t \in \mathrm{Types}(G_T^{\sigma,\tau}(c))$ (applying induction hypothesis (3.1) to ϕ_T), and $\sigma t \in \mathrm{Types}(c)$ by Lemma 3.41. The key point above proves that $t = \sigma t$. Then $t \in \mathrm{Types}(c)$ and $\phi_T \circ G_T^{\sigma,\tau}$ satisfies (3.1).

- Third case: $y \in Ind$ (Instantiation). Let $p' = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$. Let $\phi'_T = \rho[\![z]\!]y$.

  $$I_T^{\tau[\tau']}(\phi'_T)(c) - I_T^{\tau[\tau']}(\phi'_T)(\sigma\tau) \subseteq \alpha_T^{\sigma\tau'}(\cup\{\mathrm{Ins}_{p'}^{\tau'}(\gamma_T^{\tau'}(\phi'_T \circ \alpha_T^{\tau}(p)) - \gamma_T^{\tau'}(\phi'_T(\tau)))|p \in \gamma_T^{\sigma\tau}(c)\})$$

  Let $\phi'_T(c) - \phi'_T(\tau) = \psi_T(c)$. By hypothesis on ρ, ϕ'_T satisfies (3.1), hence $\mathrm{Types}(\psi_T(c)) \subseteq \mathrm{Types}(c)$. We want to prove that $\mathrm{Types}(\psi'_T(c)) \subseteq \mathrm{Types}(c)$ where

  $$\psi'_T(c) = \alpha_T^{\sigma\tau'}(\cup\{\mathrm{Ins}_{p'}^{\tau'}((\gamma_T^{\tau'} \circ \psi_T \circ \alpha_T^{\tau})(p))|p \in \gamma_T^{\sigma\tau}(c)\})$$

  If a node $n[\tau'']$ is in $\psi'_T(c)$, then there exist paths p_1, p_2 such that $n \in \alpha_T^{\sigma\tau'}(p_1.\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_2))$, $p_1 \in \gamma_T^{\tau'}(c_1)$, $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p_1) = \epsilon$, $c_1 = \psi_T(c_2)$, $c_2 = \alpha_T^{\tau}(p_2)$, $p_2 \in \gamma_T^{\sigma\tau}(c)$.

  Let $p' = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_2)$. In a way similar to what has been done above to prove Lemma 3.41, let τ_1 be the type labelling the node $\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p_1)$. Then $\sigma\tau_1$ labels $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau')}(p_1)$. The type τ_1 is the type of a node in c_1, therefore also in c_2 by hypothesis on ψ_T. Therefore, τ_1 is the label of the node $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p_2)$. If we define p'_2 such that $p'_2.p' = p_2$, then τ_1 is the label of the node $\mathrm{NofP}_{\mathrm{Gr}(\tau)}(p'_2)$ and $\sigma\tau_1$ is the label of $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p'_2)$. Since $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p'_2)$ and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau')}(p_1)$ are labelled by the same type $\sigma\tau_1$, $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p'_2.p')$ and $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau')}(p_1.p')$ are labelled by the same type. By hypothesis, this type is τ'', therefore $\mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p'_2.p') = \mathrm{NofP}_{\mathrm{Gr}(\sigma\tau)}(p_2)$ is also labelled by τ'', and therefore c contains a node labelled by τ''.

  Then $I_T^{\tau[\tau']}(\phi'_T)$ satisfies (3.1).

Other cases are straightforward by induction hypothesis. Let us prove the `let` $z = M$ `in` N case to give an example.

- First ρ' satisfies the hypothesis on ρ, by induction hypothesis on M.

- Case $y \in Var$.

  $$\phi_T \circ (T[\![N]\!]\rho' y \sqcup (T[\![M]\!]\rho y) \circ (T[\![N]\!]\rho' z)) = \phi_T \circ T[\![N]\!]\rho' y \sqcup \phi_T \circ (T[\![M]\!]\rho y) \circ (T[\![N]\!]\rho' z)$$

  by additivity of ϕ_T. The first term satisfies (3.1) by induction hypothesis on N. $\psi_T = \phi_T \circ (T[\![M]\!]\rho y)$ satisfies (3.1) by induction hypothesis on M. ψ_T is additive, and variables of τ_x are not bound in the type scheme of τ_z^{g} or of a variable of smaller scope than z, then we can apply the induction hypothesis on N, and prove that the second term satisfies (3.1). Hence the result.

- Case $y \in Ind$. Obvious by induction hypothesis on N.

□

Definition 3.43 A context $c \in Ctx_T^{\tau}$ is closed if and only if

$$\forall n \in c, \forall n' \text{ reachable from } n \text{ in the graph of } \tau, n' \in c.$$

Proposition 3.44 (Relation between closure for T and closure for E) *If $c \in Ctx_T^{\tau}$ is closed, $\gamma_T^{\tau}(c) \in Ctx_E^{\tau}$ is closed. Conversely, if $c \in Ctx_E^{\tau}$ is closed, $\alpha_T^{\tau}(c) \in Ctx_T^{\tau}$ is also closed.*

Proof: Assume that $c \in Ctx_T^{\tau}$ is closed. Let $p \in \gamma_T^{\tau}(c)$, $G = \mathrm{Gr}(\tau)$. Then $\mathrm{NofP}_G(p) \subseteq c$. Let p' such that $p.p' \in Path(\tau)$. If $\mathrm{NofP}_G(p) = \{n\}$, $\mathrm{NofP}_G(p.p') = \mathrm{NofP}_G(n, p')$ is a node reachable from n in G. Hence, $\mathrm{NofP}_G(p.p') \subseteq c$ and $p.p' \in \gamma_T^{\tau}(c)$. $\gamma_T^{\tau}(c)$ is therefore closed.

Conversely, assume that $c \in Ctx_E^{\tau}$ is closed. Let $n \in \alpha_T^{\tau}(c)$ and n' reachable from n in $G = \mathrm{Gr}(\tau)$. Let p' be the access path built from the accessors labelling a path from n to n' in G. There exists $p \in c$ such that $\mathrm{NofP}_G(p) = \{n\}$. Then $\mathrm{NofP}_G(p.p') = \{n'\}$ and $p.p' \in c$ since c is closed. Then $n' \in \alpha_T^{\tau}(c)$, which proves that $\alpha_T^{\tau}(c)$ is closed. $\square$

Lemma B.13 *Let $c \in Ctx_E^{\sigma\tau}$ such that if a path corresponding to a node n labelled by a type variable is in c, all paths corresponding to the node n are also in c. More formally, $\forall p \in c$ such that $\alpha_T^{\tau}(\{p\}) = \{n[\beta]\}$, $\forall p' \in Path(\sigma\tau)$ such that $\alpha_T^{\tau}(\{p'\}) = \{n\}$, we have $p' \in c$. Then $\gamma_T^{\tau[\tau']}(\phi_T)(c) \supseteq (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(c) \cap Path(\sigma\tau')$.*

Proof: Let $p_1 \in (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(c) \cap Path(\sigma\tau')$.

If $p_1 \in \gamma_T^{\tau'}(\phi_T(\tau))$, we have the result. We assume from now on that $p_1 \notin \gamma_T^{\tau'}(\phi_T(\tau))$.

There exists $p_2 \in c$ such that $p_1 \in (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(\{p_2\})$. If $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_2) = NV$, $\gamma_T^{\tau[\tau']}(\phi_T)(\{p_2\}) = \mathrm{Ins}_{NV}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p_2) - \gamma_T^{\tau'}(\phi_T(\tau))) \cup \gamma_T^{\tau'}(\phi_T(\tau)) = (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p_2)$. Hence $p_1 \in \gamma_T^{\tau[\tau']}(\phi_T)(\{p_2\})$.

If $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_2) = p_3 \neq NV$, we reduce the problem to the case $p_3 = \epsilon$. Indeed, $p_2 = p_4.p_3$, where p_4 represents the same type variable β as p_2, then $p_4 \in c$, by hypothesis on c. And $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_4) = \epsilon$, $\alpha_T^{\tau}(\{p_4\}) = \alpha_T^{\tau}(\{p_2\})$, therefore $p_1 \in (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(\{p_4\})$. By (3.1), p_1 corresponds to a node n' labelled by the same type variable β as p_4. Let $p_5 = \mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p_1) \neq NV$. $p_1 = p_6.p_5$ with $p_6 \in (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(\{p_4\})$ (since p_6 and p_1 correspond to the same node n'). $p_6 \notin (\gamma_T^{\tau'} \circ \phi_T)(\tau)$ (since $p_1 \notin \gamma_T^{\tau'}(\phi_T(\tau))$, and p_6 and p_1 correspond to the same node n'), then $p_6 \in \mathrm{Ins}_{\epsilon}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(\{p_4\}) - \gamma_T^{\tau'}(\phi_T(\tau)))$ (since $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p_6) = \epsilon$). Then $p_1 = p_6.p_5 \in \mathrm{Ins}_{p_5}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(\{p_4\}) - \gamma_T^{\tau'}(\phi_T(\tau))) = \gamma_T^{\tau[\tau']}(\phi_T)(\{p_4.p_5\})$. Moreover, $p_1 = p_6.p_5 \in Path(\sigma\tau')$, and p_6 represents type variable β and $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p_6) = \epsilon$, then $p_5 \in Path(\sigma\beta)$, then $p_4.p_5 \in Path(\sigma\tau)$. $p_4.p_5$ represents the same type variable β as p_4, then $p_4.p_5 \in c$. This yields the expect result: $p_1 \in \gamma_T^{\tau[\tau']}(\phi_T)(c)$. $\square$

Proposition 3.45 (Closure) *If $c \in Ctx_T^{\tau_M}$ is closed, then $\forall x \in Var \cup Ind, T[\![M]\!]\rho x c$ is also closed.*

Proof: By cases on elementary operations.

- Construction. Let $\delta : \tau \to \tau'$ be an accessor. Assume that $n \in \mathrm{Constr}_T(\delta, \tau)(c) = \{\psi(n_0) | n_0 \in c\}$ where ψ is the isomorphism from $\mathrm{Gr}(\tau')$ to G'' subgraph of $\mathrm{Gr}(\tau)$ as we have seen in the definition of constructions. Then $\exists n_0 \in c, n = \psi(n_0)$. If n' is reachable from n in $\mathrm{Gr}(\tau)$, then $\psi^{-1}(n')$ is reachable from $\psi^{-1}(n) = n_0$ in $\mathrm{Gr}(\tau')$. Then $\psi^{-1}(n') \in c$ since c is closed, and $n' \in \mathrm{Constr}_T(\delta, \tau)(c)$.

- Restriction. Let $\delta : \tau \to \tau'$ be an accessor. Assume that $n \in \mathrm{Restr}_T(\delta, \tau)(c) = \{\psi^{-1}(n_0) | n_0 \in c \cap \mathrm{Nodes}(G'')\}$ where ψ is the isomorphism from $\mathrm{Gr}(\tau')$ to G'' subgraph of $\mathrm{Gr}(\tau)$ as we have seen in the definition of constructions. Then $\exists n_0 \in c \cap \mathrm{Nodes}(G''), n = \psi^{-1}(n_0)$. If n' is reachable from n in $\mathrm{Gr}(\tau')$, then $\psi(n')$ is reachable from $\psi(n) = n_0$ in $\mathrm{Gr}(\tau')$. Then $\psi(n') \in c \cap \mathrm{Nodes}(G'')$ since c is closed, and $n' \in \mathrm{Restr}_T(\delta, \tau)(c)$.

- Step. If the test is false, then the result is the empty context, which is closed. If the test is true, then the result is the full context $\underline{\tau_z^{\mathrm{g}}}$, which is also closed.

- Constant contexts τ and $\underline{\tau}$ are closed.

- Generalization. Let $c \in Ctx_E^{\sigma\tau}$ closed. We have to prove that $\alpha_T^{\tau}(c) \in Ctx_T^{\tau}$ is closed. Let $n \in \alpha_T^{\tau}(c)$ and n' reachable from n. Then there exists p_0 such that $n \in \alpha_T^{\tau}(\{p_0\})$, $p_0 \in c$, and there exists p such that $n' = \alpha_T^{\tau}(p_0.p)$. The key point is to choose p_0 and p such that p_0 and $p_0.p$ do not go inside a type variable of τ, i.e. $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0) \in \{NV, \epsilon\}$, $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0.p) \in \{NV, \epsilon\}$ (if p_0 goes inside a type variable, just cut the end of p_0, $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0)$ and similarly for $p_0.p$). Then $p_0.p \in Path(\sigma\tau)$ (as long as we do not consider paths that go inside type variables of τ, $Path(\sigma\tau)$ and $Path(\tau)$ contain the same paths). Then $p_0.p \in c$ (since c is closed), and $n' \in \alpha_T^{\tau}(c)$.

  By Proposition 3.44, if $c \in Ctx_T^{\sigma\tau}$ is closed, $\gamma_T^{\sigma\tau}(c) \in Ctx_E^{\sigma\tau}$ is closed, then using the above result $G_T^{\tau}(c)$ is closed.

- Instantiation. We assume that ϕ_T preserves closure. It is enough to show that if $c \in Ctx_E^{\sigma\tau}$ is closed, then $\gamma_T^{\tau[\tau']}(\phi_T)(c) \cap Path(\sigma\tau') \in Ctx_E^{\sigma\tau'}$ is closed. Then Proposition 3.44 gives immediately the expected result.

  Let $p \in \gamma_T^{\tau[\tau']}(\phi_T)(c) \cap Path(\sigma\tau')$ and $p.p' \in Path(\sigma\tau')$. We prove that in this case $p.p' \in \gamma_T^{\tau[\tau']}(\phi_T)(c)$.

  If $p \in \gamma_T^{\tau}(\phi_T(\tau))$, then $p.p' \in \gamma_T^{\tau}(\phi_T(\tau))$ since τ is closed, ϕ_T and γ_T^{τ} preserve closure. This proves the result when $p \in \gamma_T^{\tau}(\phi_T(\tau))$.

  Otherwise, $\exists p_0 \in c, p \in \mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0)}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p_0) - \gamma_T^{\tau'}(\phi_T(\tau)))$.

  – If $p_1 = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0) \neq NV$, $\alpha_T^{\tau}(p_0) = \{n[\beta]\}$. By (3.1), $(\phi_T \circ \alpha_T^{\tau})(p_0) - \phi_T(\tau)$ is labelled by the same type β. Moreover, $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p) = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0) = p_1$ by definition of $\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0)}$. Then $p.p' \in Path(\sigma\tau')$ implies $p_1.p' \in Path(\sigma\beta)$ implies $p_0.p' \in Path(\sigma\tau)$. Since $c \in Ctx_E^{\sigma\tau}$ is closed, $p_0.p' \in c$. And $p.p' \in \mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0.p')}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p_0.p') - \gamma_T^{\tau'}(\phi_T(\tau)))$ since $\alpha_T^{\tau}(p_0.p') = \alpha_T^{\tau}(p_0)$ and $\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0.p')}(c') = \{p_2.p' | p_2 \in \mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0)}(c')\}$. Therefore we have the result: $p.p' \in \gamma_T^{\tau[\tau']}(p_0.p')$.

  – If $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_0) = NV$, $p \in (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p_0) - \gamma_T^{\tau'}(\phi_T(\tau))$. Let $c' = \{p_0.p'' | p_0.p'' \in Path(\sigma\tau)\}$. $c' \subseteq c$ since c is closed.

    Notice that $c' \in Ctx_E^{\sigma\tau}$ is also closed. By the above result proved in the generalization case, $\alpha_T^{\tau}(c')$ is closed, so is $(\phi_T \circ \alpha_T^{\tau})(c')$ and $(\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(c') \in Ctx_E^{\tau'}$ is also closed.

    Then $p.p' \in (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(c')$. All paths in $Path(\sigma\tau)$ corresponding to a node n labelled by a type variable β are of the form $\{p_2.p_3 | p_3 \in Path(\sigma\beta)\}$ with $\alpha_T^{\tau}(\{p_2\}) = \{n\}$ and $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p_2) = \epsilon$. If one of these paths is in c', p_0 is a prefix of p_2, and the other paths are also in c'. Then we can apply Lemma B.13. This yields $p.p' \in \gamma_T^{\tau[\tau']}(\phi_T)(c') \subseteq \gamma_T^{\tau[\tau']}(\phi_T)(c)$ hence the result.

- This property is obviously true for identity. It is obviously preserved by composition, finite and infinite upper bound, and therefore by fixed point. □

We can now prove the main theorem, which gives the correctness of our instantiation. The role of closed contexts in this theorem is the following. It would be a more natural concretization to use

$$\gamma_2^{\tau[\tau']}(\phi_T) = \lambda c. \cup \{\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}((\gamma_T^{\tau'} \circ \phi_{TNC} \circ \alpha_T^{\tau})(p)) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau))$$

where ϕ_{TNC} is defined in the same way as ϕ_T except that all constant context transformers are replaced with $\lambda c.\emptyset$. This implies in particular $\phi_{TNC}(\tau) = \tau'$. We have $\phi_T(c) - \phi_T(\tau) \subseteq \phi_{TNC}(c) \subseteq \phi_T(c)$ but we do not have $\phi_T(c) - \phi_T(\tau) = \phi_{TNC}(c)$ in general. The above concretization could be proved correct without results on closed contexts. However, its drawback is that it cannot be defined from function ϕ_T only. We would have to define a second escape analysis ϕ_{TNC}. Closed contexts are useful to prove that

$$\begin{aligned} &\mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}((\gamma_T^{\tau'} \circ \phi_{TNC} \circ \alpha_T^{\tau})(p)) \cup \gamma_T^{\tau'}(\phi_T(\tau)) \\ &= \mathrm{Ins}^{\tau'}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}((\gamma_T^{\tau'}((\phi_T \circ \alpha_T^{\tau})(p) - \phi_T(\tau)) \cup \gamma_T^{\tau'}(\phi_T(\tau))\end{aligned}$$

and thus we can replace $\phi_{TNC}(c)$ by $\phi_T(c) - \phi(\tau)$ and prove the correctness of our concretization.

Theorem 3.46 *For context transformers corresponding to $E[\![M]\!]\rho x$, $x \in Var$ bound by* `fun` *or $x \in Ind$, and for context transformers $\rho[\![y]\!]i$, $i \in Ind$ (and in particular for context transformers that may be parameters of $I_T^{\tau[\tau']}$), $\gamma_T^{\tau[\tau']}$ is a correct concretization:*

$$\gamma_T^{\tau[\tau']}(\phi_T) = \sqcup\{\phi_E \in Tr'_E | \alpha_T^{\tau[\tau']}(\phi_E) \leq \phi_T\}.$$

Proof: Let $p' = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$. Let $\psi_E = \gamma_T^{\tau[\tau']}(\phi_T) = \lambda c. \cup \{\mathrm{Ins}^{\tau'}_{p'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \gamma_T^{\tau'}(\phi_T(\tau))) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau))$. Let $\phi_E \in Tr'_E$ such that $\alpha_T^{\tau[\tau']}(\phi_E) \leq \phi_T$. Then $\phi_E \leq \gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau}$.

- If p is a path such that $p' = NV$, let $c = \{p\}$, then $\cup\{\mathrm{Ins}^{\tau'}_{p'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \phi_T(\tau))) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau)) = \gamma_T^{\tau'}((\phi_T \circ \alpha_T^{\tau})(c) - \phi_T(\tau)) \cup \gamma_T^{\tau'}(\phi_T(\tau)) = \gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau}(c)$
  Therefore $\phi_E(p) \subseteq \cup\{\mathrm{Ins}^{\tau'}_{p'}(\gamma_T^{\tau'}(\phi_T \circ \alpha_T^{\tau}(p) - \phi_T(\tau))) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau))$
- If $p' \neq NV$, $p = p_0.p'$. By Lemma 3.37,

  $$\begin{aligned}&\phi_E(p) = \mathrm{Ins}^{\tau'}_{p'}(\phi_E(p_0) - \phi_E(\emptyset)) \cup \phi_E(\emptyset)\\ &\phi_E(p_0) \subseteq (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p_0) = (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p)\\ &\phi_E(\emptyset) \subseteq (\gamma_T^{\tau'} \circ \phi_T)(\tau)\end{aligned}$$

  Therefore, $\phi_E(p) \subseteq \mathrm{Ins}^{\tau'}_{p'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \phi_E(\emptyset)) \cup (\gamma_T^{\tau'} \circ \phi_T)(\tau)$

  We prove that $\mathrm{Ins}^{\tau'}_{p'}(A - \phi_E(\emptyset)) \cup B \subseteq \mathrm{Ins}^{\tau'}_{p'}(A - B) \cup B$ where $A = (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p)$, $B = (\gamma_T^{\tau'} \circ \phi_T)(\tau) \supseteq \phi_E(\emptyset)$.

  $\mathrm{Ins}^{\tau'}_{p'}(A - \phi_E(\emptyset)) = \mathrm{Ins}^{\tau'}_{p'}(A - B) \cup \mathrm{Ins}^{\tau'}_{p'}(B - \phi_E(\emptyset))$ since $A \supseteq B \supseteq \phi_E(\emptyset)$

  $\mathrm{Ins}^{\tau'}_{p'}(B - \phi_E(\emptyset)) \subseteq \mathrm{Ins}^{\tau'}_{p'}(B) \subseteq B$ since B is closed by Propositions 3.45 and 3.44.

  So $\mathrm{Ins}^{\tau'}_{p'}(A - \phi_E(\emptyset)) \subseteq \mathrm{Ins}^{\tau'}_{p'}(A - B) \cup B$. So we have proved the intermediate result.

  We obtain: $\phi_E(p) \subseteq \mathrm{Ins}^{\tau'}_{p'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - (\gamma_T^{\tau'} \circ \phi_T)(\tau)) \cup (\gamma_T^{\tau'} \circ \phi_T)(\tau)$.

Taking the upper bound of all these results, we have:

$$\phi_E(c) \subseteq \cup\{\mathrm{Ins}_{p'}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - (\gamma_T^{\tau'} \circ \phi_T)(\tau)) | p \in c\} \cup (\gamma_T^{\tau'} \circ \phi_T)(\tau) = \psi_E(c)$$

which gives $\gamma_T^{\tau[\tau']}(\phi_T) \leq \psi_E$.

To prove the other inequality, we have to show that $\psi_E \in \mathit{Tr}'_E$ and $\alpha_T^{\tau[\tau']}(\psi_E) \leq \phi_T$. Clearly $\psi_E \in \mathit{Tr}_E$. We only have to prove that ψ_E satisfies Lemma 3.37. $\psi_E(\emptyset) = \gamma_T^{\tau'}(\phi_T(\tau))$. Let p such that $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) = \epsilon$. Then $c = \psi_E(p) - \psi_E(\emptyset) = \mathrm{Ins}_{\epsilon}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \gamma_T^{\tau'}(\phi_T(\tau)))$.

$$\begin{aligned}
\psi_E(p.p'') &= \mathrm{Ins}_{p''}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \gamma_T^{\tau'}(\phi_T(\tau))) \cup \gamma_T^{\tau'}(\phi_T(\tau)) \\
&= \mathrm{Ins}_{p''}^{\tau'}(\mathrm{Ins}_{\epsilon}^{\tau'}((\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \gamma_T^{\tau'}(\phi_T(\tau)))) \cup \gamma_T^{\tau'}(\phi_T(\tau)) \\
&= \mathrm{Ins}_{p''}^{\tau'}(c) \cup \psi_E(\emptyset)
\end{aligned}$$

This proves that $\psi_E \in \mathit{Tr}'_E$. To prove $\alpha_T^{\tau[\tau']}(\psi_E) \leq \phi_T$, we can prove $\psi_E \leq \gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau}$. If $p' = \mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p) \neq NV$, let $p'' \in c'' = \gamma_T^{\tau'}(\phi_T(\alpha_T^{\tau}(p)) - \phi_T(\tau))$. If $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p'') = \epsilon$, then $\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p'') = n[\beta]$ is labelled by a type variable, and $\mathrm{NofP}_{\mathrm{Gr}(\tau')}(p''.p') = n[\beta]$ marks the same node, therefore $p''.p' \in \gamma_T^{\tau'}(n) \subseteq c''$. Hence, if $p'' \in c''$ and $\mathrm{RemPath}_{\mathrm{Gr}(\tau')}(p'') = \epsilon$, then $p''.p' \in c''$.

This gives $\mathrm{Ins}_{p'}^{\tau'}(\gamma_T^{\tau'}(\phi_T(\alpha_T^{\tau}(p)) - \phi_T(\tau))) \subseteq \gamma_T^{\tau'}(\phi_T(\alpha_T^{\tau}(p)) - \phi_T(\tau))$. This is of course also true if $p' = NV$. Then

$$\begin{aligned}
\psi_E(c) &\subseteq \cup\{(\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(p) - \gamma_T^{\tau'}(\phi_T(\tau)) | p \in c\} \cup \gamma_T^{\tau'}(\phi_T(\tau)) \\
&\subseteq (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(c) - \gamma_T^{\tau'}(\phi_T(\tau)) \cup \gamma_T^{\tau'}(\phi_T(\tau)) \\
&\subseteq (\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau})(c)
\end{aligned}$$

which is the required result. □

Proposition 3.47

$$I_T^{\tau[\tau']}(\phi_T)(\sigma\tau) = I_T^{\tau'}(\phi_T(\tau)), \quad I_T^{\tau[\tau']}(\phi_T)(\underline{\sigma\tau}) = I_T^{\tau'}(\phi_T(\underline{\tau})), \quad I_T^{\tau[\tau']}(\phi_T) \leq I_T^{\tau'} \circ \phi_T \circ G_T^{\tau}.$$

Proof: With the formula of instantiation, $I_T^{\tau[\tau']}(\phi_T)(\sigma\tau) = \alpha_T^{\sigma\tau'}(\gamma_T^{\tau'}(\phi_T(\tau)) = I_T^{\tau'}(\phi_T(\tau))$.

By Theorem 3.46, $I_T^{\tau[\tau']}(\phi_T) \leq \alpha_T^{\sigma\tau'} \circ \gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau} \circ \gamma_T^{\sigma\tau} = I_T^{\tau'} \circ \phi_T \circ G_T^{\tau}$.

To prove $I_T^{\tau[\tau']}(\phi_T)(\underline{\sigma\tau}) = I_T^{\tau'}(\phi_T(\underline{\tau}))$, notice that $\gamma_T^{\sigma\tau}(\underline{\sigma\tau})$ clearly satisfies the hypothesis of Lemma B.13. Then $(\gamma_T^{\tau'} \circ \phi_T \circ \alpha_T^{\tau} \circ \gamma_T^{\sigma\tau})(\underline{\sigma\tau}) \cap \mathit{Path}(\sigma\tau') \subseteq \gamma_T^{\tau[\tau']}(\phi_T)(\gamma_T^{\sigma\tau}(\underline{\sigma\tau}))$. This yields $(I_T^{\tau'} \circ \phi_T \circ G_T^{\tau})(\underline{\sigma\tau}) \subseteq I_T^{\tau[\tau']}(\phi_T)(\underline{\sigma\tau})$. The converse inclusion has been proved above, then $(I_T^{\tau'} \circ \phi_T)(\underline{\tau}) = I_T^{\tau[\tau']}(\phi_T)(\underline{\sigma\tau})$. □

Appendix C

Other integer escape analyses

C.1 Analysis L' (finite contexts only)

In analyses L and F (Section 3.6), the height of closures if ∞. This leads to a relatively imprecise analysis for stack allocation of closures, and for all data structures which contain functions. Our goal here is to design a more precise analysis still using integers. We will see in Section C.3 that this goal is indeed achieved in most cases.

The definition of L' is the same as the one of L, except that we consider the construction of a closure in the same way as the construction of an ordinary constructed type. Equation (3.3) is therefore removed from the definition of the height of types. The new height is denoted $\top_{L'}$. Except in the case of recursive types, we have then $\top_{L'}[\tau_1 \to \tau_2] = 1 + \top_{L'}[\tau_2]$. Then $Ctx_{L'} = \mathbb{N}$. The definition of context transformers is very similar to that of analysis L. It is summarized in Figure C.1.

The difficulty with this new definition is that the height of a closure may be smaller than the height of the free variables it contains. This prevents us from using the fast solving algorithm due to Knuth (see Section 4.3). This algorithm only applies to functions which give a result smaller than their parameters (such functions are said to be inferior) and steps are not inferior in general when the step value is finite, and may be smaller than the result of the step. We will therefore have to use an iterative fixed point algorithm to solve the equations.

Remark. As we have seen in the two definitions of the height of a type, computing the height of a closure is a difficult problem. This comes from the fact that we do not know the types of the free variables it contains.

It is interesting to notice that this problem is also the source of seemingly unrelated problems in the typing of polymorphic programs with imperative operations, as noticed in [Ler92]. A solution suggested in [Ler92] is to add "closure types" which give more information on the contents of closures.

This problem appears in a very different manner in object-oriented languages such as Java. The type of objects indicates explicitly the fields it contains (which correspond to free variables of closures). However, type conversions become more complicated since subtyping enables us to forget fields, in a similar way as we can forget free variables of closures.

The general form of context transformers is different from the one of analysis L:

Contexts:	$Ctx^{\tau}_{L'} = [\![0, \top_{L'}[\tau]]\!]$
Context transformers:	$Tr^{\tau[\tau']}_{L'} = Ctx^{\tau}_{L'} \xrightarrow{\text{m}} Ctx^{\tau'}_{L'}$
Identity:	$\mathrm{Id}^{\tau}_{L'} = \lambda c.c$
Greatest element:	$\top^{\tau}_{Tr_{L'}} = \lambda c.\top_{L'}[\tau]$
Least element:	$\bot^{\tau}_{Tr_{L'}} = \lambda c.0$
Construction:	$\mathrm{Constr}_{L'}(\delta, \tau) = \lambda c.c$
Restriction:	$\mathrm{Restr}_{L'}(\delta, \tau) = \lambda c.c \sqcap \top_{L'}[\tau']$ where $\delta : \tau \to \tau'$.
Step:	$\mathrm{test}^{\tau_y \to \tau, \tau'}_{L'} = \lambda c.\text{if } c = \top_{L'}[\tau_y \to \tau] \text{ then } \top_{L'}[\tau'] \text{ else } 0$
Upper bound:	$\sqcup_{L'}(\phi_i)_i = \lambda c. \sqcup_i \phi_i(c)$
Composition:	$\phi \circ_{L'} \psi = \lambda c.\phi(\psi(c))$
Generalization:	

$$G^{\sigma,\tau}_{L'} = \lambda c. \begin{cases} \sqcup\{\top_{L'}[\tau'] \mid \tau' \text{ label of a node of } \mathrm{Gr}(\tau), \top_{L'}[\sigma\tau'] \leq c\} \sqcup 1 & \text{if } c > 0 \\ 0 & \text{if } c = 0 \end{cases}$$

Instantiation: see Theorem C.2.

Figure C.1: Lattice of context transformers for analysis L'

Lemma C.1 *The general form of context transformers $Ctx^{\tau}_{L'} \to Ctx^{\tau'}_{L'}$ of analysis L' is*

$$\lambda c. \sqcup_{j \in J} (\text{if } c \geq s_j \text{ then } i^+_j \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$$

where $J \subseteq Ctx^{\tau}_{L'}$ is a finite set.

Proof: The simplest proof is probably to notice that this form can represent all monotone functions of finite domain. Indeed, if ϕ is such a function,

$$\phi(x) = \sqcup_{j \mid \phi(j-1) \neq \phi(j)} (\text{if } c \geq j \text{ then } \phi(j) \text{ else } 0)$$

and we can prove that all context transformers are monotone.

We have used this form and not only steps, because in practice the number of steps can be reduced by using a term of the form $c \sqcap f$. □

The instantiation is then slightly more complex than in analysis L.

Theorem C.2 (Instantiation) *Let us define $I^{\tau}_{min}(s) = \sqcap\{\top_{L'}[\sigma\tau'] \mid \tau'$ label of a type of $\mathrm{Gr}(\tau), \top_{L'}[\tau'] \geq s\}$. $I^{\tau}_{L'}$ is defined exactly like I^{τ}_{L} but using $\top_{L'}$ instead of $\top_L$. If*

$$\phi = \lambda c. \sqcup_{j \in J} (\text{if } c \geq s_j \text{ then } i^+_j \text{ else } 0) \sqcup (c \sqcap f) \sqcup i,$$

$$I^{\tau[\tau']}_{L'}(\phi) = (\lambda c. \sqcup_{j \in J} \text{if } c \geq I^{\tau}_{min}(s_j) \text{ then } I^{\tau'}_{L'}(i^+_j) \text{ else } 0) \sqcup (c \sqcap I^{\tau'}_{L'}(f)) \sqcup I^{\tau'}_{L'}(i).$$

The analysis is correct with this instantiation and we can use the usual criterion (same as in Theorem 3.56) to decide stack allocation.

Proof: We can split analysis T into two parts: $\phi_T(c) = a_T(c) \sqcup b_T(c)$ where $b_T(c)$ is computed by constructions, restrictions, upper bound, composition, generalization, fixed point and instantiation, and $a_T(c)$ contains the other terms (terms that contain at least a constant or a step). More precisely, the elementary operations on the pair (a_T, b_T) are as follows:

Constants k	$(\lambda c.k, \lambda c.\emptyset)$
Identity	$(\lambda c.\emptyset, \lambda c.c)$
Constructions	$(\lambda c.\emptyset, \lambda c.\mathrm{Constr}_T(\delta,\tau)(c))$
Restrictions	$(\lambda c.\emptyset, \lambda c.\mathrm{Restr}_T(\delta,\tau)(c))$
Upper bound	$\sqcup_i(a_i, b_i) = (\sqcup_i a_i, \sqcup_i b_i)$
Step	$(\lambda c.\text{if } c \geq \underline{\tau_y \to} \tau_M \text{ then } \underline{\tau_x^{\mathrm{g}}} \text{ else } \emptyset, \lambda c.\emptyset)$
Composition	$(a_1, b_1) \circ (a_2, b_2) = ((a_1 \circ (a_2 \sqcup b_2)) \sqcup (b_1 \circ a_2), b_1 \circ b_2)$
Generalization	$(\lambda c.\emptyset, G_T^\tau)$
Instantiation	$I_T^{\tau[\tau']}(a, b) = (I_T^{\tau[\tau']}(a), I_T^{\tau[\tau']}(b))$

In the same way, analysis L' can be split into two parts: $\phi_{L'}(c) = a_{L'}(c) \sqcup b_{L'}(c)$ where $b_{L'}$ is of the form $\lambda c.c \sqcap f$ or $b_{L'} = G_{L'}^\tau$, and $a_{L'}$ contains steps and constants. The elementary operations on the pair $(a_{L'}, b_{L'})$ are

Constants k	$(\lambda c.k, \lambda c.0)$
Identity	$(\lambda c.0, \lambda c.c)$
Constructions	$(\lambda c.0, \lambda c.c)$
Restrictions	$(\lambda c.0, \lambda c.c \sqcap \top_{L'}[\tau])$
Upper bound	$\sqcup_i(a_i, \lambda c.c \sqcap f_i) = (\sqcup_i a_i, \lambda c.c \sqcap (\sqcup_i f_i))$
Step	$(\lambda c.\text{if } c \geq \top_{L'}[\tau_y \to \tau_M] \text{ then } \top_{L'}[\tau_x^{\mathrm{g}}] \text{ else } 0, \lambda c.0)$
Composition	$(a_1, \lambda c.c \sqcap f_1) \circ (a_2, \lambda c.c \sqcap f_2) = ((a_1 \circ (a_2 \sqcup (\lambda c.c \sqcap f_2))) \sqcup (a_2 \sqcap f_1), \lambda c.c \sqcap (f_1 \sqcap f_2))$
	$(a_1, G_{L'}^\tau) \circ (a_2, \lambda c.c \sqcap f_2) = ((a_1 \circ (a_2 \sqcup (\lambda c.c \sqcap f_2))) \sqcup (G_{L'}^\tau \circ a_2), \lambda c.c \sqcap (G_{L'}^\tau(f_1)))$
Generalization	$(\lambda c.0, G_{L'}^\tau)$
Instantiation	$I_{L'}^{\tau[\tau']}(a, \lambda c.c \sqcap f) = (I_{L'}^{\tau'} \circ a \circ G_{L'}^\tau, \lambda c.c \sqcap I_{L'}^{\tau'}(f))$

For the generalization, we use the remark page 73 to convert it into the form $c \sqcap f$ or to compose it with a function of this form when necessary.

Then, we can show that $\phi_{L'} = a_{L'} \sqcup b_{L'}$, with $a_{L'} \geq \alpha_{L'} \circ a_T \circ \gamma_{L'}^\tau$ and $b_{L'} \geq \alpha_{L'} \circ b_T \circ \gamma_{L'}^\tau$. This is a more precise version of the correctness theorem, the proof is the same, by cases on the elementary operations. We have two more cases: generalization and instantiation. The generalization case is obvious by definition of $G_{L'}^\tau$. Let us prove the instantiation case.

We have to prove that $I_{L'}^{\tau'} \circ a_{L'} \circ G_{L'}^\tau \geq \alpha_{L'} \circ I_T^{\tau[\tau']}(a_T) \circ \gamma_{L'}^{\sigma\tau}$ and $\lambda c.c \sqcap I_{L'}^{\tau'}(f) \geq \alpha_{L'} \circ I_T^{\tau[\tau']}(b_T) \circ \gamma_{L'}^{\sigma\tau}$. (We assume that $b_{L'}$ is $\lambda c.c \sqcap f$. If $b_{L'}$ is $G_{L'}^\tau$, we take $\lambda c.c \sqcap \top_{L'}[\tau]$ as a correct approximation.)

Indeed,

$$\begin{aligned}
I_{L'}^{\tau'} \circ a_{L'} \circ G_{L'}^\tau &= (\alpha_{L'} \circ I_T^{\tau'} \circ \gamma_{L'}^{\tau'}) \circ a_{L'} \circ (\alpha_{L'} \circ G_T^\tau \circ \gamma_{L'}^{\sigma\tau}) \\
&\geq \alpha_{L'} \circ I_T^{\tau'} \circ \gamma_{L'}^{\tau'} \circ \alpha_{L'} \circ a_T \circ \gamma_{L'}^\tau \circ \alpha_{L'} \circ G_T^\tau \circ \gamma_{L'}^{\sigma\tau} \\
&\geq \alpha_{L'} \circ I_T^{\tau'} \circ a_T \circ G_T^\tau \circ \gamma_{L'}^{\sigma\tau} \qquad \text{since } \gamma_{L'}^{\tau'} \circ \alpha_{L'} \geq \mathrm{Id} \text{ and } \gamma_{L'}^\tau \circ \alpha_{L'} \geq \mathrm{Id} \\
&\geq \alpha_{L'} \circ I_T^{\tau[\tau']}(a_T) \circ \gamma_{L'}^{\sigma\tau}
\end{aligned}$$

since $I_T^{\tau'} \circ a_T \circ G_T^\tau \geq I_T^{\tau[\tau']}(a_T)$ (Proposition 3.47).

On the other hand, $c \sqcap I_{L'}^{\tau'}(f) \geq (\alpha_{L'} \circ I_T^{\tau[\tau']}(b_T) \circ \gamma_{L'}^{\sigma\tau})(c)$. First, $c \geq (\alpha_{L'} \circ I_T^{\tau[\tau']}(b_T) \circ \gamma_{L'}^{\sigma\tau})(c)$. This result comes from the fact that $\mathrm{Types}(I_T^{\tau[\tau']}(b_T)(c')) \subseteq \mathrm{Types}(c')$. Indeed, we can now prove that $I_T^{\tau[\tau']}(b_T)$ satisfies (3.1) without the restriction that c does not contain function types since b_T does not contain steps. Moreover, $I_T^{\tau[\tau']}(b_T)(\emptyset) = \emptyset$, therefore

$\mathrm{Types}(I_T^{\tau[\tau']}(b_T)(c')) \subseteq \mathrm{Types}(c')$ and $(\alpha_{L'} \circ I_T^{\tau[\tau']}(b_T))(c') \leq \alpha_{L'}(c')$. Applying this with $c' = \gamma_{L'}^{\sigma\tau}(c)$ gives the expected result.

Since $\forall c, f \geq (\alpha_{L'} \circ b_T \circ \gamma_{L'}^{\sigma\tau})(c)$, we have $f \geq \alpha_{L'}(b_T(\underline{\tau}))$. Therefore

$$\begin{aligned}
I_{L'}^{\tau'}(f) &= (\alpha_{L'} \circ I_T^{\tau'} \circ \gamma_{L'}^{\tau'})(f) \\
&\geq (\alpha_{L'} \circ I_T^{\tau'} \circ \gamma_{L'}^{\tau'} \circ \alpha_{L'} \circ b_T)(\underline{\tau}) \\
&\geq \alpha_{L'}(I_T^{\tau'}(b_T(\underline{\tau}))) && \text{since } \gamma_{L'}^{\tau'} \circ \alpha_{L'} \geq \mathrm{Id} \\
&\geq \alpha_{L'}(I_T^{\tau[\tau']}(b_T)(\underline{\sigma\tau})) && \text{by Proposition 3.47} \\
&\geq \alpha_{L'}(I_T^{\tau[\tau']}(b_T)(\gamma_{L'}^{\sigma\tau}(c)))
\end{aligned}$$

We have therefore $c \sqcap I_{L'}^{\tau'}(f) \geq (\alpha_{L'} \circ I_T^{\tau[\tau']}(b_T) \circ \gamma_{L'}^{\sigma\tau})(c)$ which terminates the proof of the instantiation case.

The two correctness formulas imply $\phi_{L'} \geq \alpha_{L'} \circ \phi_T \circ \gamma_{L'}^{\sigma\tau}$ by taking the upper bound of the two formulas. This is the usual correctness criterion, and therefore we can terminate the proof of the correctness theorem in the same way as for the other analyses (Proposition 3.22).

We have to show that the expected formula of instantiation corresponds to the formula used in this proof, that is: $I_{L'}^{\tau[\tau']}(\phi_{L'}) = I_{L'}^{\tau'} \circ a_{L'} \circ G_{L'}^{\tau} \sqcup (\lambda c.c \sqcap I_{L'}^{\tau'}(f))$ where $\phi_{L'}(c) = a_{L'}(c) \sqcup (c \sqcap f)$ and $a_{L'}(c) = \sqcup_j(\text{if } c \geq s_j \text{ then } i_j^+ \text{ else } 0) \sqcup i_0$. Indeed

$$(I_{L'}^{\tau'} \circ a_{L'} \circ G_{L'}^{\tau})(c) = \sqcup_j(\text{if } G_{L'}^{\tau}(c) \geq s_j \text{ then } I_{L'}^{\tau'}(i_j^+) \text{ else } 0) \sqcup I_{L'}^{\tau'}(i_0)$$

and $G_{L'}^{\tau}(c) \geq s_j \Leftrightarrow \exists \tau', \top_{L'}[\tau'] \geq s_j \wedge \tau'$ label of a node of $\mathrm{Gr}(\tau), \top_{L'}[\sigma\tau'] \leq c \Leftrightarrow c \geq I_{\min}^{\tau}(s_j)$.
□

Remark. The above result becomes wrong if simplifications are possible during the computation of L', like transforming $(c \sqcap 1) \sqcup (\text{if } c \geq 2 \text{ then } 2 \text{ else } 0)$ into $c \sqcap 2$. Let us consider for example:

```
fun x y ->
  let z = (x, y) in
  fun a -> fst(z)
```

Let $M =$ `fun a -> fst(z)`. The type of `x` is α, `y` is of type β. We have $L'[\![M]\!]\rho\mathtt{z}c = (\text{if } c \geq 2 \text{ then } 2 \text{ else } 0) \sqcup (c \sqcap 1) = c \sqcap 2$. If we instantiate the simplified form for types $\alpha =$ `int`, $\beta =$ `int list`, we get $c \sqcap 3$. If $c = 2$, the closure M escapes, hence `z` escapes, which is of height 3. Then this escape information is incorrect. On the contrary, the instantiation of the non-simplified form yields $(\text{if } c \geq 2 \text{ then } 3 \text{ else } 0) \sqcup (c \sqcap 1)$, which is correct.

This is why, in the above proof, we had to split the function into two distinct parts, one for $c \sqcap f$ and the other one for steps and constants. This expresses that simplifications must not transfer values from the steps part to the $c \sqcap f$ part.

Therefore, we have to take into account not only the value of the function, but also the form in which it is written.

C.2 Analysis L_1 (analysis L' limited to one step)

The general form of context transformers of analysis L' is still relatively complex. We will simplify it again to have a more efficient implementation. The idea is to restrict oneself to one step. That is, we consider context transformers of the form:

$$\lambda c.(\text{if } c \geq s \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i.$$

To do this, approximated upper bound and composition have to be used. The new analysis will be denoted L_1. If $\phi_k(c) = (\text{if } c \geq s_k \text{ then } i_k^+ \text{ else } 0) \sqcup (c \sqcap f_k) \sqcup i_k$, we define:

$$(\sqcup_{L_1}(\phi_k)_k)(c) = (\text{if } c \geq \sqcap_k s_k \text{ then } \sqcup_k i_k^+ \text{ else } 0) \sqcup (c \sqcap (\sqcup_k f_k)) \sqcup (\sqcup_k i_k).$$

For the composition, the exact value of $\phi_1 \circ \phi_2$ is:

$$\begin{aligned}(\phi_1 \circ \phi_2)(c) = \text{if } c \geq s_2 \text{ then } & [\text{if } c \geq s_1 \text{ then } (\text{if } f_2 \sqcup i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0) \\ & \qquad \qquad \text{else } (\text{if } i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0)] \\ & \sqcup (i_2^+ \sqcap f_1) \text{ else } [\text{if } c \geq s_1 \text{ then } (\text{if } f_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0) \text{ else } 0] \\ & \sqcup (c \sqcap (f_2 \sqcap f_1)) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0)).\end{aligned}$$

The approximate value[1] is:

$$\begin{aligned}(\phi_1 \circ_{L_1} \phi_2)(c) = \text{if } c \geq & (\text{if } f_2 \geq s_1 \text{ then } s_1 \sqcap s_2 \text{ else } s_2) \\ & \text{then } [(\text{if } f_2 \sqcup i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0) \sqcup (i_2^+ \sqcap f_1)] \text{ else } 0 \\ & \sqcup (c \sqcap (f_2 \sqcap f_1)) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0)).\end{aligned}$$

Proof: We can prove by cases on the value of c that $(\phi_1 \circ_{L_1} \phi_2)(c) \geq (\phi_1 \circ \phi_2)(c)$ (distinguish $c < s_1$ and $c < s_2$, $s_1 \leq c < s_2$, $s_2 \leq c < s_1$, $c \leq s_1$ and $c \leq s_2$). □

Notice that this composition is *not* associative. But we can easily see that all ways of composing context transformers give (possibly different) correct approximations of analysis L', since the L_1-composition is greater than the corresponding L'-composition (and the L'-composition is associative). Therefore, we can compose context transformers in any order.

The other elementary operations are the same as for L' (notice that the instantiation of a one-step function still has only one step). However, notice that the definition of operations for analysis L_1 depends on the precise values of (f, i, s, i^+), not only on the value of the function. Then context transformers can be better represented by the quadruple of integers (f, i, s, i^+) ordered by

$$(f_1, i_1, s_1, i_1^+) \leq (f_2, i_2, s_2, i_2^+) \Leftrightarrow f_1 \leq f_2 \wedge i_1 \leq i_2 \wedge s_1 \geq s_2 \wedge i_1^+ \leq i_2^+$$

and we can define a concretization $\gamma_{L_1} : \mathbb{N}^2 \times (\mathbb{N} \cup \{\infty\}) \times \mathbb{N} \rightarrow Tr_{L'}$:

$$\gamma_{L_1}(f, i, s, i^+) = \lambda c.(\text{if } c \geq s \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i.$$

We can easily check that γ_{L_1} is monotone. The best precision is achieved by choosing the least quadruple of integers representing a given function. This yields the following definition of elementary operations:

[1] We could slightly improve the precision of the analysis by noticing that the step s is always the height of a function type. Therefore, if $\phi : \tau[\tau']$ is defined by $\phi(c) = (\text{if } c \geq s \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$, $s \geq \top_0[\tau]$ where $\top_0[\tau] = \sqcap\{\top_{L'}[\tau_f], \tau_f$ function type labelling a node of the graph of $\tau\}$. We do not detail this point, since it has not been implemented yet.

Identity and constructions	$\mathrm{Id}^{\tau}_{L_1} = \mathrm{Constr}_{L_1}(\delta,\tau) = (\top_{L'}[\tau], 0, \infty, 0)$
Constants	$\perp^{\tau}_{Tr_{L_1}} = (0,0,\infty,0), \top^{\tau}_{Tr_{L_1}} = (0, \top_{L'}[\tau], \infty, 0)$
Restriction	$\mathrm{Restr}_{L_1}(\delta,\tau) = (\top_{L'}[\tau'], 0, \infty, 0)$ where $\delta : \tau \to \tau'$
Step	$\mathrm{test}^{\tau_y \to \tau, \tau'}_{L_1} = (0, 0, \top_{L'}[\tau_y \to \tau], \top_{L'}[\tau'])$
Composition	$(f_1, i_1, s_1, i_1^+) \circ_{L_1} (f_2, i_2, s_2, i_2^+) = (f_2 \sqcap f_1,$ $(i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0),$ $\text{if } f_2 \geq s_1 \text{ then } s_1 \sqcap s_2 \text{ else } s_2,$ $(\text{if } f_2 \sqcup i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0) \sqcup (i_2^+ \sqcap f_1))$
Upper bound	$\sqcup_{L_1}(f_k, i_k, s_k, i_k^+)_k = (\sqcup_k f_k, \sqcup_k i_k, \sqcap_k s_k, \sqcup_k i_k^+)$

Remark. It would seem natural to define an upper closure operator [CC79, Theorem 5.2.0.2] from monotone functions to one-step functions. However, the set of one-step functions approximating a given monotone function has no least element. Thus, there is no best upper closure operator.

The general form of context transformers of analysis L_1 gives an implementation very similar to that of L (see Chapter 4). For solving the equations of analysis L_1, we cannot use the fast solving algorithm by Knuth, because we have to represent functions whose result is greater than their parameters (in the step case). We use a more classical algorithm [HDT87], which proceeds in two steps. First, the equations system is split into strongly connected components, to reduce the number of equations in each system. Second, the strongly connected components are visited in dependency order, and each component is solved using a classical iterative fixed point computation. Inside a strongly connected component, the nodes are ordered depth first. This algorithm gives the solution of the equations. The speed of the resulting algorithm is satisfying, since on average, each strongly connected component of the equation system is small.

The complexity of this analysis can be determined as follows. The number of calls to link and to eval is at most $\mathcal{O}(n^2)$, the length of path compressions is bounded by $\mathcal{O}(n^2 \log n)$, the number of generated equations and unknowns is in the same bound. Let H be the maximum height of a type. The number of iterations for solving the system is at most the height of the lattice of u-tuples of escapes informations, i.e. $\mathcal{O}(uH) = \mathcal{O}(n^2 \log nH)$. Computing one iteration takes $\mathcal{O}(n^2 \log n)$, hence the total complexity of solving: $\mathcal{O}(n^4 \log^2 nH)$. Thanks to the splitting into strongly connected components, the number of equations we have to solve in each component is small, and the number of iterations is therefore much reduced. Finally, the complexity is $\mathcal{O}(n^4 \log^2 nH + t)$ where t is the compute time of type heights. In practice, the complexity is $\mathcal{O}(n^2 \log^2 nH + t)$ in most cases since one square comes only from the enumeration of free variables of closures, which is quadratic in the worst case, but linear in most cases.

C.3 Comparison between different integer escape analyses

By definition of analysis L_1, L_1 is less precise than L'. Let us consider for example the following expression M:

```
fun a -> print y;
        let x = hd y in fun z -> print x
```

where `print` is a function which does not make its parameter escape. Let us assume that `y` is of type α `list`. We have:

$$L'[\![M]\!]\rho_L \mathtt{y} c = \text{if } c \geq 3 \text{ then } 2 \text{ else } 0 \sqcup \text{if } c \geq 2 \text{ then } 1 \text{ else } 0$$

whereas L_1 would yield:

$$L_1[\![M]\!]\rho_L \mathtt{y} c = \text{if } c \geq 2 \text{ then } 2 \text{ else } 0$$

because of approximations done during the computation of the upper bound.

Analysis F is also clearly less precise than L. Consider for example `x; fun y -> y`. By Lemma 3.59, `x` escapes from this expression (the input context c is ∞ since the result is a closure) for analysis F, but `x` does not escape for analysis L.

In the general case, L_1 and L are not comparable, L and L' are not comparable. Indeed, let us consider for example:

```
let f = fun x -> (print x; fun y -> print y)
in f [2]
```

Analysis L yields: $\rho_L[\![\mathtt{f}]\!]1\infty = \top_L[\tau_x]$ therefore the list `[2]` cannot be stack allocated. On the contrary, L_1 yields $\rho_{L_1}[\![\mathtt{f}]\!]1\ \top_{L'}[\tau_y \to \mathtt{unit}] = 0$ which proves that the list `[2]` can be stack allocated. L' gives the same result as L_1 here, since only one step intervenes. L can therefore be less precise than L' and L_1.

Conversely, let us consider:

```
g = fun f -> a := fst ([2], f)
```

where `f` is a function of type $\alpha \to \alpha$. Analysis L proves that the first parameter of `g` does not escape: $\rho_L[\![\mathtt{g}]\!]1\top_L[\mathtt{unit}] = \top_L[\mathtt{int\ list}] < \infty = \top_L[\alpha \to \alpha]$. On the contrary, L' cannot find this information: $\rho_{L'}[\![\mathtt{g}]\!]1\top_{L'}[\mathtt{unit}] = \top_{L'}[\mathtt{int\ list}] = 2 = \top_{L'}[\alpha \to \alpha]$. A fortiori, L_1 cannot prove either that the closure `f` does not escape. In this case, L is therefore more precise than L_1 and L'.

We define ***first-order*** programs as programs in which `fun` expressions only appear in `let` bindings: `let` $x =$ `fun` $y_1 \to ...$ `fun` $y_n \to M$ `in` N or `let rec` $x =$ `fun` $y_1 \to ...$ `fun` $y_n \to M$ `in` N and variables of function types only appear in applications in which they are completely applied. Expressions such as (`fun` $y_1 \to ...$ `fun` $y_n \to M)M_1...M_n$ may also be allowed, but are not really interesting (they can be β-reduced). For such programs, analyses L, L', L_1 and F are equivalent (the height of types differs only for functions, and steps have no contribution in the first-order case since no closure may escape). Only the precision of the higher-order extension differs.

If we restrict the kind of data structures that may contain functions, we can have more precise comparison results in the higher-order case. Let us assume that all types which contain functions have the form of Figure C.2. There is a node n_1, alone in its strongly connected component, labelled by a function type such that the result of this function contains no functions. The part represented above n_1 may contain recursive types, but each strongly connected component must have one single outgoing edge, such that n_1 is reachable from all nodes represented above it. If τ is a type labelling a node above n_1, τ_f the type labelling n_1 and τ' a type labelling a node below n_1, by (3.5), $\top_{L'}[\tau] > \top_{L'}[\tau_f] > \top_{L'}[\tau']$. Then τ_f is the function type of minimal height in this graph.

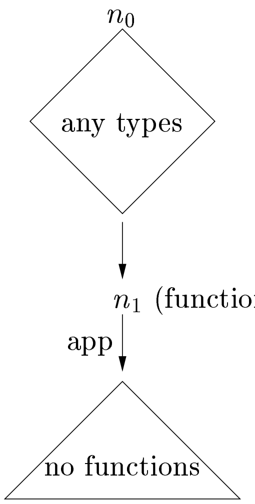


Figure C.2: Graph of functional types considered in this section.

For example, functions can be contained in lists, but not in pairs (except if these pairs are included in a strongly component above n_1). The type $\texttt{int} * (\texttt{bool} \to \texttt{int})$ does not fit in this restricted case. A type such as $(\texttt{bool} \to \texttt{int})\ \texttt{list}$ and all function types which are not in data structures are included in this restricted case.

We also assume that the language does not contain polymorphism. Indeed, with polymorphism, analysis L_1 may also be less precise than L. For example, consider

```
M = let y = fun a b c -> print x; a b c; a b
    in
    y a' b' c'
```

The type of `y` is $\tau' = (\beta_1 \to \beta_2 \to \beta_3) \to \beta_1 \to \beta_2 \to (\beta_2 \to \beta_3)$. We assume that when using `y`, its type is instantiated by substitution σ, i.e., the type of `a'` is $\sigma\beta_1 \to \sigma\beta_2 \to \sigma\beta_3$, of `b'`, $\sigma\beta_1$, of `c'`, $\sigma\beta_2$. We assume that $\sigma\beta_3$ contains no function. And we have:

$$\begin{aligned}
L_1[\![\texttt{M}]\!]\rho\texttt{x} &= L_1[\![\texttt{fun a b c -> ...}]\!]\rho\texttt{x} \circ_{L_1} G_{L_1}^{\tau'} \circ_{L_1} L_1[\![\texttt{y a' b' c'}]\!]\rho\texttt{y} \\
&= (\lambda c.\text{if } c \geq \top_{L'}[\beta_2 \to \beta_3] \text{ then } \top_{L'}[\tau_{\texttt{x}}] \text{ else } 0) \circ_{L_1} G_{L_1}^{\tau'} \circ_{L_1} (\lambda c.c \sqcap \top_{L'}[\sigma\beta_2 \to \sigma\beta_3]) \\
&= (\lambda c.\text{if } c \geq \top_{L'}[\beta_2 \to \beta_3] \text{ then } \top_{L'}[\tau_{\texttt{x}}] \text{ else } 0) \circ_{L_1} (\lambda c.c \sqcap \top_{L'}[\beta_2 \to \beta_3]) \\
&= (\lambda c.\text{if } c \geq \top_{L'}[\beta_2 \to \beta_3] \text{ then } \top_{L'}[\tau_{\texttt{x}}] \text{ else } 0)
\end{aligned}$$

The value of the step $\top_{L'}[\beta_2 \to \beta_3]$ is smaller than (or equal to) $\top_f[\tau_M] = \top_{L'}[\sigma\beta_3]$, if σ is well chosen.

This is the real cause of imprecision of analysis L_1. Assume that the step of a context transformer $\phi : \tau[\tau']$ of analysis L_1 is always greater than $\top_f[\tau]$. Also assume that the input context of ϕ contains no escaping function. Then the contribution of the step part of L_1 will be zero, and analyses L_1 and L will give the same result. If the input context of ϕ contains an escaping function, the corresponding input context for analysis L will be ∞, and L_1 is clearly more precise than L. On the contrary, if the step of ϕ is smaller than or equal to $\top_f[\tau]$, even if the input context of ϕ contains no escaping function, the contribution of the step may be greater than 0, and analysis L_1 claims that free variables of some closures may escape when this is not the case. In this situation, analysis L is more precise than L_1.

If we consider the above example, $L_1[\![\texttt{M c'}]\!]\rho\texttt{x}\top_{L'}[\tau_{\texttt{c}'}] = \top_{L'}[\tau_{\texttt{x}}]$: `x` may escape. The analysis L is more precise: $L[\![\texttt{M}]\!]\rho\texttt{x} = \lambda c.\text{if } c = \infty \text{ then } \infty \text{ else } 0$, and $L[\![\texttt{M c'}]\!]\rho\texttt{x}\top_L[\tau_{\texttt{c}'}] = 0$: analysis L proves that `x` does not escape.

We are going to prove that in this case without polymorphism and with restrictions on types containing functions, L is less precise than L_1, and therefore also than L'.

In the following, $\top_f[\tau]$ is defined with respect to analysis L, and not with respect to L' (anyway, for L', $\top_f[\tau] = \top_{L'}[\tau]$, hence the notation $\top_{L'}[\tau]$ is sufficient).

A semi-dual Galois connection can then be defined between L' and L:

$$\alpha^\tau(c) = \begin{cases} c & \text{if } c \leq \top_f[\tau], \\ \infty & \text{otherwise.} \end{cases}$$

$$\gamma^\tau(c) = c \sqcap \top_{L'}[\tau]$$

$$(Ctx^\tau_{L'}, \leq) \underset{\alpha^\tau}{\overset{\gamma^\tau}{\leftrightarrows}} (Ctx^\tau_L, \leq)$$

We can easily check that $\forall c \in Ctx^\tau_L, L[\![M]\!]\rho x c \geq \alpha^{\tau^{\mathrm{g}}_x}(L'[\![M]\!]\rho' x(\gamma^{\tau_M}(c)))$, which already proves that L' is more precise than L.

Let us consider now $\phi = L_1[\![M : \tau]\!]\rho(x : \tau')$.

$$\phi(c) = (\text{if } c \geq s \text{ then } i^+ \text{ else } 0) \sqcup (c \sqcap f) \sqcup i$$

Let

$$\phi_k(c) = (\text{if } c \geq s_k \text{ then } i^+_k \text{ else } 0) \sqcup (c \sqcap f_k) \sqcup i_k$$

Lemma C.3 *If $f > \top_f[\tau']$, then there exists τ_f, function type, label of a node of* $\mathrm{Gr}(\tau)$ *and of a node of* $\mathrm{Gr}(\tau')$*, such that $\top_{L'}[\tau_f] = \top_f[\tau'] + 1$.*

By the restriction on functional types, all non-functional elements of τ or τ' have a height strictly smaller than $\top_{L'}[\tau_f]$, and τ and τ' contain a function type of height $\top_{L'}[\tau_f]$. Then in particular, if $f > \top_f[\tau']$, $\top_f[\tau] = \top_{L'}[\tau_f] - 1 = \top_f[\tau']$.

Intuitively, if $\phi = L_1[\![M]\!]\phi x$ is such that $f > \top_f[\tau']$, a function of type τ_f is read in variable x by the analyzed expression M and returned in the result of M.

Proof: By induction on used elementary operations.

- This property is preserved by composition. If $\phi_1 : \tau'[\tau'']$, $\phi_2 : \tau[\tau']$, $\phi = \phi_1 \circ_{L_1} \phi_2$, $f = f_1 \sqcap f_2$, therefore if $f > \top_f[\tau'']$, $f_1 > \top_f[\tau'']$, hence by induction hypothesis, there exists τ'_f function type, common label of nodes of $\mathrm{Gr}(\tau')$ and $\mathrm{Gr}(\tau'')$, such that $\top_{L'}[\tau'_f] = \top_f[\tau''] + 1$. Then $f_2 > \top_f[\tau''] = \top_f[\tau']$. Therefore there exists τ_f common label of nodes of $\mathrm{Gr}(\tau)$ and $\mathrm{Gr}(\tau')$. By the restriction on functional types, τ_f is the **only** function type of minimal height labelling a node of the graph of τ', hence $\tau_f = \tau'_f$ is common to τ and τ'', q.e.d.

- This property is preserved by upper bound. If $\phi = \sqcup_{L_1}(\phi_k)_k$, $f = \sqcup_k f_k$, therefore if $f > \top_f[\tau']$, $\exists k, f_k > \top_f[\tau']$, hence the existence of τ_f.

- This property is true in base cases.

□

Lemma C.4 $s > \top_f[\tau]$.

This lemma corresponds to the fact that steps only test on functions (in analysis T, the corresponding operation is if $c \geq c_0$ then x else 0 in which c_0 must be labelled by a function type), therefore only on data of height strictly greater than $\top_f[\tau]$.

Proof: By induction on used elementary operations.

- This property is preserved by composition. If $\phi_1 : \tau'[\tau'']$, $\phi_2 : \tau[\tau']$, $\phi = \phi_1 \circ_{L_1} \phi_2$, $s =$ if $f_2 \geq s_1$ then $s_1 \sqcap s_2$ else s_2. If $f_2 \geq s_1 > \top_f[\tau']$, $\top_f[\tau'] = \top_f[\tau]$ (by Lemma C.3) therefore $s_1 > \top_f[\tau]$. Moreover, $s_2 > \top_f[\tau]$ therefore $s = s_1 \sqcap s_2 > \top_f[\tau]$. Otherwise, $s = s_2 > \top_f[\tau]$.
- This property is preserved by upper bound. If $\phi = \sqcup_{L_1}(\phi_k)_k$, $s = \sqcap_k s_k > \top_f[\tau]$ since for all k, s_k satisfies the same property.
- This property is true in base cases. For closures, $s > \top_f[\tau]$. In other base cases, $s = \infty$ (there is no step).

□

Theorem C.5 $\forall c \in Ctx^{\tau}_{L}, L[\![M]\!]\rho x c \geq \alpha^{\tau^{g}_{x}}(L_1[\![M]\!]\rho_1 x(\gamma^{\tau_M}(c)))$. *This means that L_1 is more precise than L.*

Proof: By induction on used elementary operations. We use ϕ, ϕ_1, ϕ_2 for operators of analysis L_1, ϕ', ϕ'_1, ϕ'_2 for the ones of analysis L. We prove that if $\phi : \tau[\tau']$,

$$\forall c \in Ctx^{\tau}_{L}, \phi'(c) \geq (\alpha^{\tau'} \circ \phi \circ \gamma^{\tau})(c).$$

- **This property is preserved by composition.** If $\phi_1 : \tau'[\tau'']$, $\phi_2 : \tau[\tau']$, $\phi = \phi_1 \circ_{L_1} \phi_2$ and $\phi' = \phi'_1 \circ \phi'_2$, by induction hypothesis

$$\begin{aligned}\phi'(c) &\geq (\alpha^{\tau''} \circ \phi_1 \circ \gamma^{\tau'} \circ \alpha^{\tau'} \circ \phi_2 \circ \gamma^{\tau})(c)\\ &\geq (\alpha^{\tau''} \circ \phi_1 \circ \phi_2 \circ \gamma^{\tau})(c).\end{aligned}$$

First case: $f_2 < s_1$. In this case, $\phi_1 \circ \phi_2 = \phi_1 \circ_{L_1} \phi_2$, therefore $\phi'(c) \geq (\alpha^{\tau''} \circ (\phi_1 \circ_{L_1} \phi_2) \circ \gamma^{\tau})(c) = (\alpha^{\tau''} \circ \phi \circ \gamma_{\tau})(c)$.
Second case: $f_2 \geq s_1$.

$$\begin{aligned}(\phi_1 \circ \phi_2)(c) = &\text{ if } c \geq s_2 \text{ then } [\text{if } c \geq s_1 \text{ then } i_1^+ \text{ else (if } i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0)] \sqcup (i_2^+ \sqcap f_1)\\ &\text{ else } [\text{if } c \geq s_1 \text{ then } i_1^+ \text{ else } 0]\\ &\sqcup (c \sqcap f_2 \sqcap f_1) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0))\end{aligned}$$

$$\begin{aligned}(\phi_1 \circ_{L_1} \phi_2)(c) = {} & \text{if } c \geq s_1 \sqcap s_2 \text{ then } i_1^+ \sqcup (i_2^+ \sqcap f_1) \text{ else } 0 \\ & \sqcup (c \sqcap f_2 \sqcap f_1) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0))\end{aligned}$$

By Lemma C.4, $s_1 > \top_f[\tau']$, therefore $f_2 > \top_f[\tau']$ and by Lemma C.3, $\top_f[\tau'] = \top_f[\tau]$, therefore $s_1 > \top_f[\tau]$. Using Lemma C.4 again, $s_2 > \top_f[\tau]$. Then $\gamma^\tau(c) \geq s_1$ if and only if $c = \infty$ and the same for s_2. Then

$$\begin{aligned}(\phi_1 \circ \phi_2 \circ \gamma^\tau)(c) = {} & \text{if } c = \infty \text{ then [if } c = \infty \text{ then } i_1^+ \text{ else (if } i_2^+ \geq s_1 \text{ then } i_1^+ \text{ else } 0)] \\ & \qquad \sqcup (i_2^+ \sqcap f_1) \\ & \quad \text{else [if } c = \infty \text{ then } i_1^+ \text{ else } 0] \\ & \sqcup (\gamma^\tau(c) \sqcap f_2 \sqcap f_1) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0)) \\ = {} & \text{if } c = \infty \text{ then } i_1^+ \sqcup (i_2^+ \sqcap f_1) \text{ else } 0 \\ & \sqcup (\gamma^\tau(c) \sqcap f_2 \sqcap f_1) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0))\end{aligned}$$

$$\begin{aligned}(\phi_1 \circ_{L_1} \phi_2 \circ \gamma^\tau)(c) = {} & \text{if } c = \infty \text{ then } i_1^+ \sqcup (i_2^+ \sqcap f_1) \text{ else } 0 \\ & \sqcup (\gamma^\tau(c) \sqcap f_2 \sqcap f_1) \sqcup ((i_2 \sqcap f_1) \sqcup i_1 \sqcup (\text{if } i_2 \geq s_1 \text{ then } i_1^+ \text{ else } 0))\end{aligned}$$

Therefore $(\phi_1 \circ \phi_2 \circ \gamma^\tau)(c) = (\phi_1 \circ_{L_1} \phi_2 \circ \gamma^\tau)(c) = (\phi \circ \gamma^\tau)(c)$, hence $\phi' \geq \alpha^{\tau'} \circ \phi \circ \gamma^\tau$.

• **This property is preserved by upper bound.** If $\phi = \sqcup_{L_1}(\phi_k)_k$ and $\phi' = \sqcup_k \phi'_k$, by induction hypothesis,

$$\begin{aligned}\phi'(c) & \geq \sqcup_k(\alpha^{\tau'} \circ \phi_k \circ \gamma^\tau)(c) \\ & \geq (\alpha^{\tau'} \circ (\sqcup_k \phi_k) \circ \gamma^\tau)(c)\end{aligned}$$

$$(\sqcup_k \phi_k)(c) = \sqcup_k(\text{if } c \geq s_k \text{ then } i_k^+ \text{ else } 0) \sqcup (c \sqcap (\sqcup_k f_k)) \sqcup (\sqcup_k i_k)$$

and

$$(\sqcup_{L_1}(\phi_k)_k)(c) = (\text{if } c \geq \sqcap_k s_k \text{ then } \sqcup_k i_k^+ \text{ else } 0) \sqcup (c \sqcap (\sqcup_k f_k)) \sqcup (\sqcup_k i_k).$$

By Lemma C.4, $s_k > \top_f[\tau]$, therefore $\gamma^\tau(c) \geq s_k$ if and only if $c = \infty$. Then:

$$(\sqcup_k \phi_k)(\gamma^\tau(c)) = \sqcup_k(\text{if } c = \infty \text{ then } i_k^+ \text{ else } 0) \sqcup (\gamma^\tau(c) \sqcap (\sqcup_k f_k)) \sqcup (\sqcup_k i_k)$$

and

$$(\sqcup_{L_1}(\phi_k)_k)(\gamma^\tau(c)) = (\text{if } c = \infty \text{ then } \sqcup_k i_k^+ \text{ else } 0) \sqcup (\gamma^\tau(c) \sqcap (\sqcup_k f_k)) \sqcup (\sqcup_k i_k)$$

therefore $((\sqcup_k \phi_k) \circ \gamma^\tau)(c) = ((\sqcup_{L_1}(\phi_k)_k) \circ \gamma^\tau)(c) = (\phi \circ \gamma^\tau)(c)$ hence $\phi'(c) \geq (\alpha^{\tau'} \circ \phi \circ \gamma^\tau)(c)$.

• **This property is true in base cases,** since in these cases L_1 is equal to L'.

□

We have therefore a collection of integer escape analyses: L' more precise than L_1, often more precise than L which is more precise than F.

In practice, the results obtained by analysis L_1 are very similar to the ones of analysis F. The percentage of stack allocated data is the same for all benchmarks except coq, in which analysis L_1 allocate 1% to 2% more data on the stack than analysis F. Analysis L_1 is more costly than analysis F. The compile time increases by up to 43% for coq with inlining with analysis L_1, instead of 35% with analysis F. For small benchmarks, analysis L_1 costs up to 6% of the compile time more than analysis F.

Appendix D

Correctness of the implementation of analysis T

D.1 Graph building

We prove the correctness of the graph building algorithm of Section 5.1.2.

We first define the correspondence between the graphs of types $\mathrm{Gr}(\tau)$ and the exploded super-graph. Since the domain $D = \mathrm{Type}(\tau)$ is simply a copy of $\mathrm{Gr}(\tau)$, we can define an isomorphism $\phi_D : \mathrm{Gr}(\tau) \to D$ which maps nodes of the graph of a type to nodes of the exploded super-graph.

Let $\mathrm{Nodes} : \mathit{AnnotatedType} \times \mathit{Ctx}_T \to \mathcal{P}(\mathit{Node})$ be such that $\mathrm{Nodes}(D, c) = \{\phi_D(n') | n' \in c\}$ is the set of nodes of D whose corresponding nodes in $\mathrm{Gr}(\tau)$ are in c.

Conversely, let $\mathrm{OccType} : \mathit{AnnotatedType} \times \mathit{Node} \to \mathit{Ctx}_T$ be such that $\mathrm{OccType}(D, n) = \{\phi_D^{-1}(n)\}$ is the context $c \in \mathit{Ctx}_T$ containing the node of $\mathrm{Gr}(\tau)$ corresponding to the node n in domain D.

$$\cup\{\mathrm{OccType}(D, n) | n \in \mathrm{Nodes}(D, c)\} = c \text{ and } \mathrm{Nodes}(D, \mathrm{OccType}(D, n)) = \{n\}$$

If $pD \in p\mathit{AnnotatedType}$, the ith parameter of pD is denoted by p_iD and the result of pD is denoted by D, i.e. $pD = ([p_1D, ..., p_kD], D)$.

If M is an expression, $pD_M = ([p_1D_M, ..., p_kD_M], D_M)$ is the associated domain, the one given as parameter to function Graph during the analysis of M. If x is a variable, $\rho(x) = ([p_1\rho[\![x]\!], ..., p_k\rho[\![x]\!]], \rho[\![x]\!])$ is the domain corresponding to x (the one in the environment ρ during the runtime of Graph). Notice that, even if in the definition of D_M the expression M may in particular be a variable x, $\rho[\![x]\!] \neq D_x$!

We now define the context transformer represented by a certain part of the graph. An intuitive definition would be:

$$\begin{aligned} \mathrm{Path}_0(D_n, D_m) = &\lambda c. \cup \{\mathrm{OccType}(D_n, n) | (\exists m \in \mathrm{Nodes}(D_m, c) \text{ such that } \mathit{SLRP}(n, m) \neq \emptyset) \\ &\text{or } (\exists m' \text{ such that } \mathrm{Mark}(m') \text{ and } \mathit{RP}(n, m') \neq \emptyset) \end{aligned}$$

However, to perform the proof, we have to be able to distinguish which escaping part comes from which expression P. This is achieved by adding the condition: the first edge of the path $q \in \mathit{SLRP}(n, m)$ or $q \in \mathit{RP}(n, m')$ must have been added during the analysis of P.

The first edge of q may be created by calling $\mathrm{Link}(n, n')$ in two cases: in Generalize when we analyze a variable, and in the case of a closure, to link each free variable occurrence to the closure. After the call to $\text{Link}(n, n')$, the edge $n \xrightarrow{\iota} n'$ is in *Edge*. It may also be created by calling Call. The phrase "the first edge of q has been added during the analysis of P" means either this edge is $n \xrightarrow{\iota} n'$ and $\text{Link}(n, n')$ has been called during the analysis of P (even if this edge already existed before), or this edge has been created by a call to Call. That is why we use "added" instead of "created".

Moreover, the path q should not go outside of the part of the graph between D_n and D_m, and come back to D_m later. More formally, $q \in \mathit{SLRP}(n, m)$ is not of the form $q = r : s, r \in \mathit{SLRP}(n, m), m \in \text{Nodes}(D_m), s \neq \epsilon$. We have a similar property for $q \in \mathit{RP}(n, m')$.

The formal definition is then:

$$
\begin{aligned}
&\text{PathRest} : \mathit{AnnotatedType} \times \mathit{Expression} \times \mathit{AnnotatedType} \to \mathit{Tr}_T \\
&\text{PathRest}(D_n, P, D_m) = \\
&\quad \lambda c. \cup \{\text{OccType}(D_n, n) | n \text{ has been marked during the analysis of } P \text{ or} \\
&\quad (\exists m \in \text{Nodes}(D_m, c) \text{ and } q \in \mathit{SLRP}(n, m) \text{ such that} \\
&\qquad q \text{ is not of the form } q = r : s, r \in \mathit{SLRP}(n, m'), m' \in \text{Nodes}(D_m), s \neq \epsilon \\
&\quad\ \text{or } \exists m' \text{ such that } \text{Mark}(m') \text{ and } q \in \mathit{RP}(n, m') \text{ such that} \\
&\qquad q \text{ is not of the form } q = r : s, r \in \mathit{SLRP}(n, m), m \in \text{Nodes}(D_m)) \\
&\quad\ \text{and the first edge of } q \text{ has been added during the analysis of } P\}
\end{aligned}
$$

The following particular cases of function PathRest will be useful in the proof:

$$
\begin{aligned}
&\text{Path} : \mathit{AnnotatedType} \times \mathit{Expression} \to \mathit{Tr}_T \\
&\text{Path}(D, M) = \text{PathRest}(D, M, D_M) \\
&\text{Path}_\text{v} : \mathit{Var} \times \mathit{Expression} \to \mathit{Tr}_T \\
&\text{Path}_\text{v}(z, M) = \text{PathRest}(\rho[\![z]\!], M, D_M) \\
&\text{Path}_\text{a} : \mathit{AnnotatedType} \times \mathit{AnnotatedType} \to \mathit{Tr}_T \\
&\text{Path}_\text{a}(p_i\rho[\![z]\!], \rho[\![z]\!]) = \text{PathRest}(p_i\rho[\![z]\!], M, \rho[\![z]\!]) \text{ where } M \text{ is the whole body of the program} \\
&\text{PathClo} : \mathit{AnnotatedType} \times \mathit{Expression} \to \mathit{Tr}_T \\
&\text{PathClo}(D_n, M) = \lambda c. \cup \{\text{OccType}(D_n, n) | \exists m \text{ node corresponding to the top of the} \\
&\qquad\qquad \text{closure } M \text{ such that } n \xrightarrow{\iota} m \in \mathit{Edge} \text{ and } m \in \text{Nodes}(D_M, c)\}
\end{aligned}
$$

The function Path_a defines a concretization from the graph to the context transformers Tr_T of analysis T.

Lemma D.1 *The same node n cannot be the target of both an edge labelled $(_i$ and an edge labelled ι or $)_i$.*

Proof: If n is the target of an edge labelled ι or $)_i$, n is in a domain D_x or $D_{\texttt{fun}\ x \to N}$ (such edges are created by Generalize or CallD in the x case, by Link in the `fun` case). If n is the target of a $(_i$ edge, then there exists x such that n is in $p_k\rho[\![x]\!]$ (such an edge is created by CallD in the x case). We prove that no node n can be in both domains, by cases on the creation site of n.

If n is in the domain D_N created in the `let` or `let rec` cases, n appears in domains D_M for an expression M and $\rho[\![x]\!]$ for a variable x. We consider 4 different kinds of domains:

domains of the form D_M, p_iD_M, $\rho[\![x]\!]$ and $p_i\rho[\![x]\!]$. During a call to Graph(M, pD_M, ρ), nodes in D_M remain in domains of the form $D_{M'}$, nodes in $\rho[\![x]\!]$ remain in $\rho[\![x']\!]$, nodes in $p_i\rho[\![x]\!]$ remain in $p_{i'}\rho[\![x']\!]$, only nodes in p_iD_M may also appear in $\rho[\![x]\!]$ (function case). Then such a node n cannot be in a $p_k\rho[\![x]\!]$.

If n is in p_kD_N created in the `let` or `let rec` cases, n may appear in domains p_iD_M, $p_i\rho[\![x]\!]$ or $\rho[\![x]\!]$. It cannot be in a D_M.

If n is in D_P created in the $N\ P$ case, n may appear in domains p_iD_M, D_M and $\rho[\![x]\!]$. It cannot be in a $p_i\rho[\![x]\!]$.

If n is in p_kD_P created in the $N\ P$ case, n may appear in $p_k\rho[\![x]\!]$. It cannot be in a D_M.

If n is created by MarkParams in the function case, n is in $p_i\rho[\![x]\!]$. n cannot be in D_M.

If n is in a domain D_N created in the $\overline{C_i}(N) \leftarrow P$, $N; P$, `if` or main program cases, n may appear in D_M for expressions M only, and not in $p_i\rho[\![x]\!]$.

If n is in a domain p_kD_N created in the $\overline{C_i}(N) \leftarrow P$, $N; P$, `if` or main program cases, n may appear in p_iD_M and $\rho[\![x]\!]$. n cannot appear in D_M nor in $p_i\rho[\![x]\!]$. □

Lemma D.2 (Suffixes of *SLRP* paths) *If $q = q_1 : q_2 \in SLRP$ and $q_1 \in SLRP$ then $q_2 \in SLRP$.*

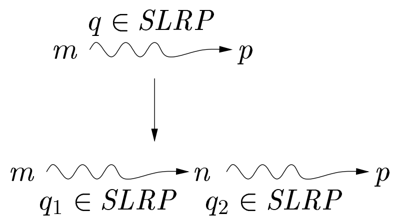
Proof: This is obvious if q_1 or q_2 is empty. Otherwise, the path q has been built using Rule (5.4) (this is the only rule which builds a path which has a strict prefix in *SLRP*). Therefore $q = q_1 : q_3 : ... : q_n$, with $\forall i \geq 3, q_i \in SLRPEL$. We have to prove that $q_2 = q_3 : ... : q_n \in SLRP$. Therefore, applying Rule (5.4), we only have to prove that there is no $(_i$ edge whose target is the source node n of q_3. n is the target of an *SLRP* path, therefore n is the target of an edge labelled ι or $)_i$, hence by Lemma D.1, n is not the target of an edge $(_i$. □

Lemma D.3 (Suffixes of *RP* paths) *If $q = q_1 : q_2 \in RP$ and $q_1 \in SLRP$ then $q_2 \in RP$.*

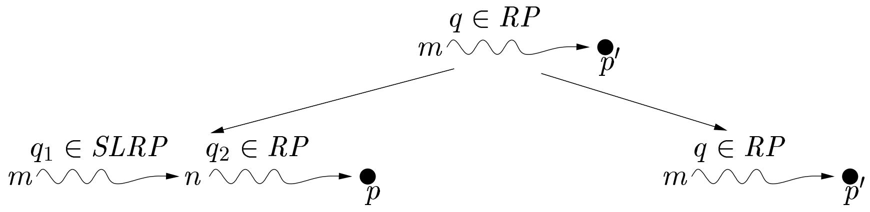
Proof: This is obvious if q_1 is empty. Otherwise, assume that q_2 is not empty. The path q has been built by Rule (5.6) (this is the only rule which builds a path which has a strict prefix in *SLRP*). Therefore $q = q_1' : m \xrightarrow{(_i} n : q_2'$ with $q_1' \in SLRP$ and $q_2' \in RP$. Since q_1 is *SLRP*, it does not contain the edge $m \xrightarrow{(_i} n$ which has no corresponding $)_i$ edge in q, and it is shorter than q_1'. First case: q_1 is strictly shorter than q_1', $q_1' = q_1 : q_2''$, therefore $q_2'' \in SLRP$ by Lemma D.2. $q_2 = q_2'' : m \xrightarrow{(_i} n : q_2'$ and we use Rule (5.6) taking into account Lemma D.1 which proves that there exists no edge $n' \xrightarrow{(_j} m'$ whose target is the source node of q_2''. Second case: q_1' and q_1 have the same length, and $q_2 = m \xrightarrow{(_i} n : q_2' \in RP$ by applying (5.6) (using again Lemma D.1 to prove that there exists no edge $n' \xrightarrow{(_j} m$ whose target is the source node of q_2).

If $q_2 = m : \epsilon : m$ is empty, by Lemma D.1, there is no $(_i$ edge of target m, then $q_2 \in RP(m, m)$ by Rule (5.5). □

Lemma D.4 *Let $q = q_1 : ... : q_n \in SLRP$ with $q_i \in SLRPEL$. Let us assume that the first edge of q has been added during the analysis of expression M, and that no source or target of q_i is in the corresponding domain D_M. Then all targets and sources of the paths q_i have been created during the analysis M (except perhaps the source node of q).*



Case of unmarked nodes.



Case of marked nodes. The case on the left is the one where q can be written in the form $q = t : u, t \in SLRP(m', n), n \in \mathrm{Nodes}(D_n)$.

Figure D.1: Splitting of paths in Lemma D.5

Proof: First notice that all edges labelled $)_i$ or ι added during the analysis of M have a target which is either in D_M, or has been created during the analysis of M. Then we proceed by induction on the number n of subpaths q_i.

First case: the first edge of q is labelled ι (and in this case, it is the only edge of q_1). Then its target is either a node of D_M which is impossible since no target of q_i is a node of D_M, or a node created during the analysis of M. Second case: the first edge of q is labelled $(_i$. Since it has been added during the analysis of M, all edges labelled $(_i$ or $)_i$ have been created during the analysis of M. Then the last edge of q_1 labelled $)_i$ has been created during the analysis of M, hence its target is either in D_M (impossible since no target of q_i is in D_M) or it is a node created during the analysis of M.

If there is only one subpath $q = q_1 \in SLRPEL$, we have proved the result. Otherwise, the source of the first edge of q_2 is only known during the analysis of M, hence its first edge has been created during the analysis of M. Therefore we can apply the induction hypothesis to the path $q_2 : ... : q_n$ and get the expected result. $\square$

Lemma D.5 *We assume that N is an immediate subexpression of P, pD_N is created during the analysis of P (by a call to* pType*), and z is variable which is bound outside of P. In this case,*

$$\mathrm{Path}(\rho[\![z]\!], N) \circ \mathrm{Path}(D_N, P) = \mathrm{PathRest}(\rho[\![z]\!], N, D_P).$$

Proof: Notice that

- **Remark 1.** By hypothesis, D_N is created during the analysis of P, therefore all edges whose source is in $\mathrm{Nodes}(D_N)$ have been created during the analysis of P. Indeed, the graph building algorithm does not have access to nodes of D_N outside the analysis of P.

- **Remark 2.** $\text{Nodes}(D_N) \cap \text{Nodes}(D_P) = \emptyset$ since D_N has been created during the analysis of P and D_P already existed before.

- **Remark 3.** D_P has not been created during the analysis of N, since D_P existed before the beginning of the analysis of P, hence before the beginning of the analysis of N since N is a subexpression of P.

Let us first prove: $\text{PathRest}(\rho[\![z]\!], N, D_P) \leq \text{Path}(\rho[\![z]\!], N) \circ \text{Path}(D_N, P)$. Let $c' \in \text{PathRest}(\rho[\![z]\!], N, D_P)(c)$. Then there exists m such that $c' \in \text{OccType}(\rho[\![z]\!], m)$. Let us consider the case of unmarked nodes: $\exists p \in \text{Nodes}(D_P, c), q \in \mathit{SLRP}(m,p)$ such that the first edge of q has been added during the analysis of N. $q = q'_1 : ... : q'_k$ with $q'_i \in \mathit{SLRPEL}$. Let us assume that no source or target of q'_i is in D_N. Then, by Lemma D.4, all targets of q'_i have been created during the analysis of N. This is impossible since the target of q is in D_P which has not been created during the analysis of N (Remark 3). Then, there exists a target of q'_i in D_N. Let us suppose that the target of q'_i is the first such target. Let $q_1 = q'_1 : ... : q'_i$ and $q_2 = q'_{i+1} : ... : q'_k$. q is then of the form $q_1 : q_2$, with $q_1 \in \mathit{SLRP}(m,n), n \in \text{Nodes}(D_N)$ (see Figure D.1). q_1 is not of the form $q_1 = r : s, r \in \mathit{SLRP}(m,n'), n' \in \text{Nodes}(D_N), s \neq \epsilon$ since the target of q_1 is the *first* target of a q'_j being in $\text{Nodes}(D_N)$. By Lemma D.2, $q_2 \in \mathit{SLRP}(n,p)$. q_2 is not of the form $q_2 = r : s, r \in \mathit{SLRP}(n,p'), p' \in \text{Nodes}(D_P), s \neq \epsilon$ since q is not of this form. By Remark 1, all edges whose source is in $\text{Nodes}(D_N)$ have been created during the analysis of P. In particular, the first edge of q_2 has been created during the analysis of P (q_2 is not empty since its source is in $\text{Nodes}(D_N)$ and its target in $\text{Nodes}(D_P)$, and these sets are disjoint by Remark 2). Then $\text{OccType}(D_N, n) \subseteq \text{Path}(D_N, P)(c)$ and $c' \in \text{OccType}(\rho[\![z]\!], m) \subseteq \text{Path}(\rho[\![z]\!], N)(\text{OccType}(D_N, n))$. Therefore $c' \in \text{Path}(\rho[\![z]\!], N)(\text{Path}(D_N, P)(c))$.

Case of marked nodes. If m has been marked during the analysis of N, $c' \in \text{Path}(\rho[\![z]\!], N)(c'')$ for all c'' hence the result. Otherwise, $\exists p'$ such that $\text{Mark}(p')$ and $q \in \mathit{RP}(m,p')$, q is not of the form $q = r : s, r \in \mathit{SLRP}(m,p), p \in \text{Nodes}(D_P)$ and the first edge of q has been added during the analysis of N.

- First assume that q can be put into the form $q = t : u, t \in \mathit{SLRP}(m,n), n \in \text{Nodes}(D_N)$. Since t is SLRP, u cannot start with an edge labelled $)_i$, hence u starts with an edge labelled ι or $(_i$, which is created during the analysis of P by Remark 1 (or u is the empty path). By Lemma D.3, $u \in \mathit{RP}(n,p')$. Therefore $\text{OccType}(D_N, n) \subseteq \text{Path}(D_N, P)(c)$, and $c' \in \text{OccType}(\rho[\![z]\!], m) \subseteq \text{Path}(\rho[\![z]\!], N)(\text{OccType}(D_N, n))$, therefore $c' \in \text{Path}(\rho[\![z]\!], N)(\text{Path}(D_N, P)(c))$.

- Otherwise, $q \in \mathit{RP}(m,p')$ and q is not of the form $q = t : u, t \in \mathit{SLRP}(m,n), n \in \text{Nodes}(D_N)$, therefore $c' \in \text{OccType}(\rho[\![z]\!], m) \subseteq \text{Path}(\rho[\![z]\!], N)(c_0)$ for all context c_0. In particular, $c' \in \text{Path}(\rho[\![z]\!], N)(\text{Path}(D_N, P)(c))$.

Let us prove the converse: $\text{Path}(\rho[\![z]\!], N)(\text{Path}(D_N, P)(c)) \subseteq \text{PathRest}(\rho[\![z]\!], N, D_P)(c)$. Preliminary remark: let $q \in \mathit{SLRP}(m,n)$, $m \in \text{Nodes}(\rho[\![z]\!])$, $n \in \text{Nodes}(D_N)$. Let $q' \in \mathit{SLRP}(n,p')$ such that q' is not of the form $q' = r' : s', r' \in \mathit{SLRP}(n,p), p \in \text{Nodes}(D_P)$. Assume that there exists a path $c \xrightarrow{(_j} s : q : q' : e \xrightarrow{)_j} r$. Then $s = m \in \text{Nodes}(\rho[\![z]\!])$, and when edges $c \xrightarrow{(_j} s$ and $e \xrightarrow{)_j} r$ are created, $e = p'$ is a node of a domain $\rho[\![x]\!]$, and s is a node of a parameter $p_k\rho[\![x]\!]$ of $\rho[\![x]\!]$. By Lemma D.4, $q' = q'_1 : ... : q'_l$, with $q'_i \in \mathit{SLRPEL}$ and all targets of q'_i have been created during the analysis of P. In particular, the target p' of q'_l has been created during the analysis of P. The domain $\rho[\![x]\!]$, and therefore the parameter domain

$p_k\rho[\![x]\!]$ which is created at the same time as $\rho[\![x]\!]$ or afterwards, have been created during the analysis of P therefore after nodes of $\rho[\![z]\!]$ have been inserted in the environment (nodes are only inserted in ρ in the `let` and `let rec` cases and the parameters and result are created by pType at the same time, or in the `fun` case, and the parameters are created by MarkParams whereas the result already existed). Then $s \in p_k\rho[\![x]\!]$ has been created after nodes $\rho[\![z]\!]$ have been inserted in the environment. Thus $s \notin \mathrm{Nodes}(\rho[\![z]\!])$. Contradiction. Therefore there exists no path $c \xrightarrow{(_j} s : q : q' : e \xrightarrow{)_j} r$. Then rules (5.4) and (5.6) can be applied.

Let us consider $c' \in \mathrm{Path}(\rho[\![z]\!], N)(\mathrm{Path}(D_N, P)(c))$. Then there exists m, such that $c' \in \mathrm{OccType}(\rho[\![z]\!], m)$. Let us consider the case of unmarked nodes. $\exists n \in \mathrm{Nodes}(D_N, \mathrm{Path}(D_N, P)(c)), q \in \mathit{SLRP}(m, n)$ such that the first edge of q has been added during the analysis of N. Since $n \in \mathrm{Nodes}(D_N, \mathrm{Path}(D_N, P)(c))$, $\exists q' \in \mathit{SLRP}(n, p), p \in \mathrm{Nodes}(D_P, c)$. By concatenating these paths: $q : q' \in \mathit{SLRP}(m, p)$ (Rule (5.4)) is a path which proves that $c' \in \mathrm{PathRest}(\rho[\![z]\!], N, D_P)(c)$ ($q : q'$ is not of the form $q : q' = r : s, r \in \mathit{SLRP}(m, p'), p' \in \mathrm{Nodes}(D_P), s \neq \epsilon$, since if p' were in q' this would contradict the form of q' and if p' were in q, this would contradict Lemma D.4)

Case of marked nodes. If m is marked during the analysis of N, then $c' \in \mathrm{PathRest}(\rho[\![z]\!], N, D_P)(c)$.

Let us assume that there exists $q \in \mathit{RP}(m, n')$ with n' marked and q is not of the form $q = r : s$, $r \in \mathit{SLRP}(m, n)$, $n \in \mathrm{Nodes}(D_N)$. Moreover, the first edge of q has been added during the analysis of N. Then q is not of the form $q = r' : s'$, $r' \in \mathit{SLRP}(m, p), p \in \mathrm{Nodes}(D_P)$ either: by contradiction, if it were, we could split the path $r' \in \mathit{SLRP}(m, p)$ into $r' = r : s''$ with $r \in \mathit{SLRP}(m, n), n \in \mathrm{Nodes}(N)$ (this is what we have proved in the case of unmarked nodes of the direct implication). This would contradict the hypothesis on q. Therefore q is not of the form $q = r' : s'$, $r' \in \mathit{SLRP}(m, p), p \in \mathrm{Nodes}(D_P)$, therefore $c' \in \mathrm{OccType}(\rho[\![z]\!], m) \subseteq \mathrm{PathRest}(\rho[\![z]\!], N, D_P)(c)$.

Let us now assume that there exists $n \in \mathrm{Nodes}(D_N, \mathrm{Path}(D_N, P)(c)), q \in \mathit{SLRP}(m, n)$ such that the first edge of q has been added during the analysis of N. Since $n \in \mathrm{Nodes}(D_N, \mathrm{Path}(D_N, P)(c))$, either n is marked (in which case we take $p' = n$ and $q' = \epsilon$) or $\exists q' \in \mathit{RP}(n, p'), p'$ marked and q' is not of the form $q' = r : s$, with $r \in \mathit{SLRP}(n, p), p \in \mathrm{Nodes}(D_P)$. Then the path $q : q'$ is in $\mathit{RP}(m, p')$ (Rule (5.6)) and is not of the form $r' : s'$, $r' \in \mathit{SLRP}(m, p), p \in \mathrm{Nodes}(D_P)$ (same reasoning as above). Hence the result: $c' \in \mathrm{OccType}(\rho[\![z]\!], m) \subseteq \mathrm{PathRest}(\rho[\![z]\!], N, D_P)(c)$. □

Lemma D.6 *We assume that N is an immediate subexpression of P, pD_N is created during the analysis of P (by a call to* pType*), and z is variable which is bound outside of P. We also assume that no edge has its source in* $\mathrm{Nodes}(D_N)$ *and no node of* $\mathrm{Nodes}(D_N)$ *is marked. In this case,*

$$\mathrm{Path}(\rho[\![z]\!], N)(\tau_N) = \mathrm{PathRest}(\rho[\![z]\!], N, D_P)(c).$$

Proof: $\mathrm{Path}(D_N, P) = \mathrm{PathRest}(D_N, P, D_P) = \lambda c.\emptyset$ since no edge has its source in $\mathrm{Nodes}(D_N)$ and no node of $\mathrm{Nodes}(D_N)$ is marked. Then by Lemma D.5, we have the result. □

Lemma D.7 *All edges added during a call to* $\mathrm{Graph}(M, pD_M, \rho)$ *have their source either in p_iD_M or in $\rho[\![x]\!]$ or in nodes created during the analysis of M.*

All nodes marked during a call to $\mathrm{Graph}(M, pD_M, \rho)$ *are in p_iD_M or in nodes created during the analysis of M.*

Proof: Easy by induction on expression M. In the x case, Generalize adds edges starting from $\rho[\![x]\!]$. CallD adds $(_i$ edges starting from $p_i D_M$ and $)_i$ edges starting from $\rho[\![x]\!]$. In the `fun` case, Link adds edges starting from $\rho[\![x]\!]$. The recursive call $\mathrm{Graph}(N, pD_N, \rho[x \mapsto \mathrm{MarkParams}(D_x)])$ adds edges starting from $p_i D_N$ or from $\rho[\![x]\!]$ or from D_x or from new nodes, by induction hypothesis. $p_i D_N = p_{i+1} D_M$ and $D_x = p_1 D_M$ are $p_j D_M$ hence the result. The other cases are obvious by induction hypothesis. □

Lemma D.8 *If* $\mathrm{Graph}(M, pD_M, \rho)$ *is called,* $\mathrm{Nodes}(p_i D_M) \cap \cup_x \mathrm{Nodes}(\rho[\![x]\!]) = \emptyset$.

Proof: Easy by induction on expression M. □

Lemma D.9 *If* pD_N *is created during the analysis of* M, $\mathrm{PathRest}(p_i D_M, N, D_M)(c) = \emptyset$ *(*M *is either* `let` $x = N$ `in` P, `let rec` $x = N$ `in` P, $P\ N$, $N; P$, `if` N `then` P `else` Q*).*

Proof: If there existed a path $q \in SLRP(n, m)$ or $q \in RP(n, m')$, $n \in \mathrm{Nodes}(p_i D_M)$, with a first edge added during the analysis of N, $\mathrm{Graph}(N, pD_N, \rho')$, then by Lemma D.7, the source n of this edge would be in $p_i D_N$ or in $\rho'[\![x]\!]$ or in nodes created during the analysis of N.

- First case: n is in $p_i D_N$. This case is impossible since $p_i D_N$ is created during the analysis of M, and therefore $\mathrm{Nodes}(p_i D_N) \cap \mathrm{Nodes}(p_i D_M) = \emptyset$.
- Second case: n is in $\rho'[\![x]\!]$. Assume that the analysis of M is $\mathrm{Graph}(M, pD_M, \rho)$. By Lemma D.8, $\mathrm{Nodes}(p_i D_M) \cap \cup_x \mathrm{Nodes}(\rho[\![x]\!]) = \emptyset$. In the $N; P$, `if`, `let` cases, $\rho' = \rho$. In the `let rec` case, $\rho' = \rho[x \to pD_N]$. Since D_N is new, $\mathrm{Nodes}(p_i D_M) \cap \mathrm{Nodes}(D_N) = \emptyset$. In all cases, $\mathrm{Nodes}(p_i D_M) \cap \cup_x \mathrm{Nodes}(\rho'[\![x]\!]) = \emptyset$. Therefore, the second case is impossible.
- Third case: n is created during the analysis of N. This case is impossible since $n \in \mathrm{Nodes}(p_i D_M)$ has not been created during the analysis of N (N is a subexpression of M).

Therefore, there exists no such path q.

On the other hand, if $n \in \mathrm{Nodes}(p_i D_M)$ was marked during the analysis of N, by Lemma D.7, n would be in $p_i D_N$ or in nodes created during the analysis of N, and we have just seen that this is impossible. Then we have the required result. □

Lemma D.10 $G_T^{\tau_x^{\mathrm{g}}} = \mathrm{Path}(\rho[\![x]\!], x)$.

Proof: All edges added and nodes marked during the analysis of x are added or marked in Generalize or CallD. Sources of edges added and nodes marked in CallD are in $\mathrm{Nodes}(p_i D_M)$. By Lemma D.8, they are not in $\mathrm{Nodes}(\rho[\![x]\!])$. Therefore, only edges added by Generalize have to be considered.

We can notice that the algorithm of Generalize calls $\mathrm{Link}(n, m)$ if and only if the generalization algorithm mentioned in the definition of analysis T (page 59) asserts that $\mathrm{OccType}(\rho[\![x]\!], n) \in c$ implies $\mathrm{OccType}(D_x, m) \in G_T^{\tau_x^{\mathrm{g}}}(c)$. The correctness of this algorithm yields the correctness of Generalize. □

Lemma D.11 *Let* $\tau = \tau_x^{\mathrm{g}}$ *and* $\tau_i = \tau_{x,i}^{\mathrm{g}}$. $I_T^{\tau[\tau_i]}(\mathrm{Path_a}(p_i \rho[\![x]\!], \rho[\![x]\!])) = \mathrm{Path}(p_i D_x, x)$ *if* $\mathrm{Path_a}(p_i \rho[\![x]\!], \rho[\![x]\!])$ *satisfies* (3.1)*: if* c *does not contain a node labelled by a function type,* $\mathrm{Types}(\mathrm{Path_a}(p_i \rho[\![x]\!], \rho[\![x]\!])(c) - \mathrm{Path_a}(p_i \rho[\![x]\!], \rho[\![x]\!])(\emptyset)) \subseteq \mathrm{Types}(c)$.

Proof: Let PathNotMarked be the part of Path corresponding to paths whose target is an unmarked node. We prove the following properties:

1. $\alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(\mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])\circ\alpha_T^{\tau}(p)-\mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\emptyset)))|p\in\gamma_T^{\sigma\tau}(c_r)\})\subseteq \mathrm{PathNotMarked}(p_iD_x,x)(c_r)$ where c_r is a context which contains a single node.

2. $\alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(\mathrm{PathNotMarked_a}(p_i\rho[\![x]\!],\rho[\![x]\!])\circ\alpha_T^{\tau}(p)))|p\in\gamma_T^{\sigma\tau}(c_r)\})\supseteq \mathrm{PathNotMarked}(p_iD_x,x)(c_r)$ where c_r is a context which contains a single node.

3. $I_T^{\tau_i}(\mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\tau))\subseteq \mathrm{Path}(p_iD_x,x)(\sigma\tau)$.

4. $I_T^{\tau_i}(\mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\tau))\supseteq \mathrm{Path}(p_iD_x,x)(\sigma\tau)$.

• **Proof of property 1.** Let $c_e = G_T^{\tau}(c_r)$, $c_s = \mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(c_e) - \mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\emptyset)$ and $c_c = \alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(c_s))|p\in\gamma_T^{\sigma\tau}(c_r)\})$. Let $c\in \mathrm{Nodes}(p_iD_x,c_c)$.

There exists $s\in \mathrm{Nodes}(p_i\rho[\![x]\!],c_s),(c,s)\in\cup_{k\in[\![1,m]\!]}\mathrm{Collect}(p_kD_x,p_k\rho[\![x]\!],\mathrm{true})$ (since $c_c = \alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(c_s))|p\in\gamma_T^{\sigma\tau}(c_r)\})$ therefore c_c and c_s correspond to the same occurrences respectively in the instantiated and general type or to a type and the corresponding type variable in the general type, and Collect groups pairwise the nodes corresponding to these occurrences in p_iD_x and $p_i\rho[\![x]\!]$). Then an edge $c\overset{(_j}{\rightarrow}s$ is created by Call. Since $c_s = \mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(c_e) - \mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\emptyset)$, there exists $q\in SLRP(s,e)$, $e\in\mathrm{Nodes}(\rho[\![x]\!],c_e)$ (if the target of q was a marked node, $s\in\mathrm{Nodes}(p_i\rho[\![x]\!],\mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\emptyset))$ which is not the case). Let us consider two cases:

- If $\mathrm{Types}(c_e)$ is not a type variable, $\exists r\in\mathrm{Nodes}(D_x,c_r),(r,e)\in\mathrm{Collect}(D_x,\rho[\![x]\!],\mathrm{false})$ (same reason as for (c,s) above). Then an edge $e\overset{)_j}{\rightarrow}r$ is created by Call. $q' = c\overset{(_j}{\rightarrow}s : q : e\overset{)_j}{\rightarrow}r\in SLRP(c,r)$ (Rules (5.2) and (5.4)). q' is not of the form $q' = r' : s', r'\in SLRP(c,m'), s'\neq\epsilon$. Then $c_c\subseteq\mathrm{PathNotMarked}(p_iD_x,x)(c_r)$.

- If $\mathrm{Types}(c_e)$ is a type variable β, $\mathrm{Types}(c_s)$ is the same type variable β (By (3.1), since $\mathrm{Types}(c_e)$ does not contain a function, $\mathrm{Types}(c_s) = \mathrm{Types}(\mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(c_e) - \mathrm{Path_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(\emptyset))\subseteq\mathrm{Types}(c_e)$). Let o be an occurrence of c_c in the graph of $\sigma\beta$. For this value of o, during the runtime of CallD, $\exists s\in\mathrm{Nodes}(\rho[\![x]\!],c_s),(c,s)\in par$. Then an edge $c\overset{(_{j'}}{\rightarrow}s$ is created. Since $c_s = \mathrm{PathNotMarked_a}(p_i\rho[\![x]\!],\rho[\![x]\!])(c_e)$, there exists $q\in SLRP(s,e)$, $e\in\mathrm{Nodes}(\rho[\![x]\!],c_e)$. c_e is labelled by the type variable β, c_r corresponds to the part of $\mathrm{Gr}(\sigma\beta)$ defined by the occurrence $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$. Since $c_c = \alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(c_s))|p\in\gamma_T^{\sigma\tau}(c_r)\})$, and o is an occurrence of c_c in $\mathrm{Gr}(\sigma\beta)$, o is the occurrence corresponding to $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$ for a certain $p\in\gamma_T^{\sigma\tau}(c_r)$. Consider now the pair $(r,e)\in res$ where r corresponds to the occurrence o in $\mathrm{Gr}(\sigma\beta)$. We have then $r\in\mathrm{Nodes}(D_x,c_r)$. Since $(r,e)\in res$, there exists an edge $e\overset{)_{j'}}{\rightarrow}r$. The end of the proof is then the same as in the first case.

• **Proof of property 2.** Let $c_c = \mathrm{PathNotMarked}(p_i D_x, x)(c_r)$ and $c \in \mathrm{Nodes}(D_x, c_c)$. There exists $q \in \mathit{SLRP}(c, r)$, $r \in \mathrm{Nodes}(D_x, c_e)$. The first edge of q has been added during the analysis of x, therefore q starts with an edge $c \xrightarrow{(_j} s$. Since $q \in \mathit{SLRP}(c, r)$, q contains an edge $e \xrightarrow{)_j} r'$. All edges labelled $)_j$ have their source in $\mathrm{Nodes}(\rho[\![x]\!])$ and their target in $\mathrm{Nodes}(D_x)$. Since q does not have a strict *SLRP* prefix with a target in $\mathrm{Nodes}(D_x)$, r' is in fact the target of q and $r' = r$. Then $q = c \xrightarrow{(_j} s : q' : e \xrightarrow{)_j} r$, $q' \in \mathit{SLRP}(s, e)$. q' has no strict *SLRP* prefix whose target is in $\mathrm{Nodes}(\rho[\![x]\!])$. Indeed, assume that there exists such a prefix $q'' \in \mathit{SLRP}(s, e')$, $e' \in \mathrm{Nodes}(\rho[\![x]\!])$. There are two cases.

First case: e' does not correspond to a type variable. In this case, CallD creates an edge $e' \xrightarrow{)_j} r'$ since $\mathrm{Collect}(D_{ir}, D_{gr}, \mathrm{false})$ contains a pair of the form (e', r') for a certain r'.

Second case: e' corresponds to a type variable. Then s also corresponds to a type variable. (We prove by induction that if an *SLRP* path has a target which corresponds to a type variable, then its source also corresponds to a type variable. An edge labelled ι with a target corresponding to a type variable can only be created by $\mathrm{Generalize}(D_i, D_g)$ in the case $D_i = n[\beta]$, which implies $D_g = n'[\beta']$ since D_g is more general than D_i. Then the edge links a type variable to a type variable. Similarly, when creating edges $c \xrightarrow{(_j} s$ and $e \xrightarrow{)_j} r$, if r corresponds to a type variable, so does e, and therefore also s by induction hypothesis. Then we are in the case $(n[\beta], m[\beta])$ of Collect, and therefore c also corresponds to a type variable.) We can choose for occurrence o the top of the type, for example. Then CallD creates edges $c' \xrightarrow{(_{j'}} s$ and $e' \xrightarrow{)_{j'}} r'$.

In both cases, there exists a path $c' \xrightarrow{(_{j'}} s : q'' : e' \xrightarrow{)_{j'}} r'$ and Rule (5.4) could not be used to prove that $q' \in \mathit{SLRP}(s, e)$. This proves that q' has no strict *SLRP* prefix whose target is in $\mathrm{Nodes}(\rho[\![x]\!])$. Therefore $c_s = \mathrm{OccType}(\rho[\![x]\!], s) \subseteq \mathrm{Path}_{\mathrm{a}}(p_i\rho[\![x]\!], \rho[\![x]\!])(c_e)$ with $c_e = \mathrm{OccType}(\rho[\![x]\!], e)$. The existence of the edge $c \xrightarrow{(_j} s$ proves that $c_c \subseteq \alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(c_s)) | p \in \gamma_T^{\sigma\tau}(c_r)\})$ and the existence of $e \xrightarrow{)_j} r$ proves that $c_e \subseteq G_T^{\tau}(c_r)$. (Two cases: if c_r goes inside a type variable β, then c_s and c_e are labelled β, c_c and c_r correspond to the same occurrence determined by $\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)$ inside β since the corresponding nodes are linked by edges labelled by the same number j. Otherwise, c_r and c_e correspond to the same occurrence respectively in the general and instantiated types; c_c, c_s correspond to the same occurrence respectively in the instantiated and general types.) Finally, we get the expected result:

$$c_c \subseteq \alpha_T^{\sigma\tau_i}(\cup\{\mathrm{Ins}^{\tau_i}_{\mathrm{RemPath}_{\mathrm{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(\mathrm{PathNotMarked}_{\mathrm{a}}(p_i\rho[\![x]\!], \rho[\![x]\!]) \circ \alpha_T^{\tau}(p))) | p \in \gamma_T^{\sigma\tau}(c_r)\}).$$

• **Proof of property 3.** Let $c_s = \mathrm{Path}_{\mathrm{a}}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau)$ and $c_c = I_T^{\tau_i}(c_s)$. Let $c \in \mathrm{Nodes}(p_i D_x, c_c)$.

There exists $s \in \mathrm{Nodes}(\rho[\![x]\!], c_s), (c, s) \in \cup_{k \in [\![1,m]\!]}\mathrm{Collect}(p_k D_x, p_k\rho[\![x]\!], \mathrm{true})$ (since $c_c = I_T^{\tau_i}(c_s)$ therefore c_c and c_s correspond to the same occurrences respectively in the instantiated and general type or to a type and the corresponding type variable in the general type, and Collect groups pairwise the nodes corresponding to these occurrences in $p_i D_x$ and $p_i\rho[\![x]\!]$). Then an edge $c \xrightarrow{(_j} s$ is created. Since $c_s = \mathrm{Path}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau)$, there exists $q \in RP(s, e')$, e' marked. $q' = c \xrightarrow{(_j} s : q \in RP(c, e')$ (Rule (5.6)). q' is not of the form $q' = r' : s', r' \in$

$SLRP(c, m')$. Then $c_c \subseteq \text{Path}(p_i D_x, x)(\sigma\tau)$.

• **Proof of property 4.** This is the converse of property 3. Let $c_c = \text{Path}(p_i D_x, x)(\sigma\tau)$. $c \in \text{Nodes}(D_x, c_c)$. There exists $q \in RP(c, e')$ such that e' is marked and the first edge of q has been added during the analysis of x. Then q starts with an edge $c \xrightarrow{(_j} s$. Let q' be q without its first edge. $q' \in RP(s, e')$. We prove that q' has no $SLRP$ prefix going to a node of $\text{Nodes}(D_x)$. By contradiction, let q_1 be such a prefix. There exists a $(_j$ edge whose target is the source of q_1 and every node of $\text{Nodes}(D_x)$ which may be the target of q_1 is the source of a $)_j$ edge. If $q_1 = q'$, Rule (5.5) would be used to prove that $q' \in RP(s, e')$, but this rule could not be applied because there is a path $c \xrightarrow{(_j} s : q_1 : m \xrightarrow{)_j} n$. Contradiction. If $q_1 \neq q'$, q_1 is a strict prefix of q', but Rules (5.2) and (5.6) cannot be applied to build an $SLRP$ or RP path whose q_1 is a strict prefix (again because of the path $c \xrightarrow{(_j} s : q_1 : m \xrightarrow{)_j} n$). Contradiction. Therefore q' has no $SLRP$ prefix going to a node of $\text{Nodes}(D_x)$. Therefore $c_s \in \text{OccType}(\rho[\![x]\!], s) \subseteq \text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau)$. The existence of the edge $c \xrightarrow{(_j} s$ proves that $c_c \subseteq I_T^{\tau_i}(c_s)$ hence the result.

• **Proof of the lemma from the 4 properties.**

$$
\begin{aligned}
I_T^{\tau[\tau_i]}(\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!]))(c) &\subseteq \text{PathNotMarked}(p_i D_x, x)(c) \cup \text{Path}(p_i D_x, x)(\sigma\tau) \\
&\qquad\qquad\text{by properties 1 and 3} \\
&\subseteq \text{Path}(p_i D_x, x)(c)
\end{aligned}
$$

$$
\begin{aligned}
X &= \text{Path}(p_i D_x, x)(c) \\
&= \text{PathNotMarked}(p_i D_x, x)(c) \cup \text{Path}(p_i D_x, x)(\sigma\tau) \\
&\subseteq \alpha_T^{\sigma\tau_i}(\cup\{\text{Ins}^{\tau_i}_{\text{RemPath}_{\text{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(\text{PathNotMarked}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!]) \circ \alpha_T^{\tau}(p)))| \\
&\quad p \in \gamma_T^{\sigma\tau}(c_r)\} \cup I_T^{\tau_i}(\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau)) \qquad \text{by properties 2 and 4} \\
&\subseteq \alpha_T^{\sigma\tau_i}(\cup\{\text{Ins}^{\tau_i}_{\text{RemPath}_{\text{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(\text{PathNotMarked}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!]) \circ \alpha_T^{\tau}(p) - \\
&\quad \text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\emptyset)))|p \in \gamma_T^{\sigma\tau}(c_r)\}) \cup I_T^{\tau_i}(\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau))
\end{aligned}
$$

since $\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau)$ is taken into account in the second term

$$
\begin{aligned}
X &\subseteq \alpha_T^{\sigma\tau_i}(\cup\{\text{Ins}^{\tau_i}_{\text{RemPath}_{\text{Gr}(\tau)}(p)}(\gamma_T^{\tau_i}(\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!]) \circ \alpha_T^{\tau}(p) - \\
&\quad \text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\emptyset)))|p \in \gamma_T^{\sigma\tau}(c_r)\}) \cup I_T^{\tau_i}(\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!])(\tau))
\end{aligned}
$$

since $\text{PathNotMarked}_\text{a}(D, D')(c) - \text{Path}_\text{a}(D, D')(\emptyset) = \text{Path}_\text{a}(D, D')(c) - \text{Path}_\text{a}(D, D')(\emptyset)$ since all paths which have a marked target are taken into account by $\text{Path}_\text{a}(D, D')(\emptyset)$

$$
X \subseteq I_T^{\tau[\tau_i]}(\text{Path}_\text{a}(p_i\rho[\![x]\!], \rho[\![x]\!]))(c)
$$

□

If $\mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!])$ does not satisfy (3.1), we can change the definition of $I_T^{\tau[\tau_i]}$ to have $I_T^{\tau[\tau_i]}(\mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!])) = \mathrm{Path}(p_i D_x, x)$. Indeed, we have proved that in the definition of analysis T, $I_T^{\tau[\tau_i]}(\phi)$ is only used with functions ϕ which satisfy (3.1), therefore we have not changed the definition of analysis T. However, this change will make the proof simpler because we do not know *a priori* that $\mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!])$ also satisfies (3.1). We will know that when we have finished the correctness proof, and proved that $\mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!]) = \rho_T[\![x]\!]i$. The new definition of $I_T^{\tau[\tau_i]}$ is

$$\begin{aligned}
I_T^{\tau[\tau_i]}(\phi)(c_r) = \cup\{&\mathrm{OccType}(p_i D_x, c) | \exists s \in \mathrm{Nodes}(p_i\rho[\![x]\!], \phi(\mathrm{OccType}(\rho[\![x]\!], e))),\\
&\exists r \in \mathrm{Nodes}(D_x, c_r) \text{ such that CallD creates edges } c \overset{(_j}{\to} s \text{ and } e \overset{)_j}{\to} r\\
&\text{or } \exists s \in \mathrm{Nodes}(p_i\rho[\![x]\!], \phi(\emptyset)) \text{ such that CallD creates edge } c \overset{(_j}{\to} s\}
\end{aligned}$$

This definition only depends on ϕ, τ, and τ_i since the edges created by CallD do not depend on the specific nodes but only on the types labelling them. The condition "$\mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!])$ satisfies (3.1)" can then be removed.

Theorem D.12 *If ρ_T is defined by $\forall z \in FV(M), \forall i \in Ind, \rho_T[\![z]\!]i = \mathrm{Path_a}(p_i\rho[\![z]\!], \rho[\![z]\!])$ (ρ_T does not necessarily have the same value as when we compute analysis T), then*

$$\begin{aligned}
&\forall y \in Var, T[\![M]\!]\rho_T y = \mathrm{Path_v}(y, M)\\
&\forall i \in Ind, T[\![M]\!]\rho_T i = \mathrm{Path}(p_i D_M, M)
\end{aligned}$$

Proof: By induction on expression M.

- **Case** x. When there is no instantiation, the result is obvious by hypothesis on ρ_T. The general case is slightly more complex.

Let $y \in Var - \{x\}$. $T[\![x]\!]\rho_T y = \emptyset = \mathrm{Path}(\rho[\![y]\!], x)$ since no edge starting from $\rho[\![y]\!]$ is created during the analysis of x.

$$T[\![x]\!]\rho_T x = G_T^{\tau_x^{\mathrm{g}}} = \mathrm{Path}(\rho[\![x]\!], x) \qquad \text{by Lemma D.10}$$

Let $i \in Ind$.

$$\begin{aligned}
T[\![x]\!]\rho_T i &= I_T^{\tau_x^{\mathrm{g}}[\tau_{x,i}^{\mathrm{g}}]}(\rho_T[\![x]\!]i)\\
&= I_T^{\tau_x^{\mathrm{g}}[\tau_{x,i}^{\mathrm{g}}]}(\mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!]))\\
&= \mathrm{Path}(p_i D_x, x) \qquad \text{by Lemma D.11}
\end{aligned}$$

- **Case** $M = \texttt{let}\ x = N\ \texttt{in}\ P$. Let $\rho'_T = \rho_T[x \mapsto T[\![N]\!]\rho_{T|Ind}]$. First, $\rho'_T[\![x]\!]$ satisfies the expected hypothesis, since by induction on N, $T[\![N]\!]\rho_T i = \mathrm{Path}(p_i D_N, N) = \mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!])$, since every edge starting from $p_i D_N$ has been added during the analysis of N (during the analysis of P, D_N is in the environment, therefore no edge starting from $p_i D_N$ is added by Lemma D.7) therefore the induction hypothesis on P can be applied.

Let $y \in Var - \{x\}$.

$$\begin{aligned} T[\![M]\!]\rho_T y &= T[\![P]\!]\rho'_T y \sqcup (T[\![N]\!]\rho_T y) \circ (T[\![P]\!]\rho'_T x) \\ &= \mathrm{Path_v}(y,P) \sqcup \mathrm{Path_v}(y,N) \circ \mathrm{Path_v}(x,P) \end{aligned}$$

by induction hypothesis on P. Moreover, since P and $M = \texttt{let}\ x = N\ \texttt{in}\ P$ share the same domain D_P, $\mathrm{Path_v}(y,P) = \mathrm{PathRest}(\rho[\![y]\!], P, D_M)$.

On the other hand, let us prove that $\mathrm{Path_v}(y,N) \circ \mathrm{Path_v}(x,P) = \mathrm{PathRest}(\rho[\![y]\!], N, D_M)$. Since the domain D_N is equal to $\rho[\![x]\!]$, $\mathrm{Path_v}(y,N) \circ \mathrm{Path_v}(x,P) = \mathrm{Path}(\rho[\![y]\!], N) \circ \mathrm{Path}(D_N, P)$. On the other hand, $D_P = D_M$. All edges starting from D_N have been created during the analysis of P (they cannot be created during the analysis of N by Lemma D.7). Therefore, $\mathrm{Path}(D_N,P) = \mathrm{Path}(D_N,M)$ and $\mathrm{Path_v}(y,N) \circ \mathrm{Path_v}(x,P) = \mathrm{Path}(\rho[\![y]\!], N) \circ \mathrm{Path}(D_N, M) = \mathrm{PathRest}(\rho[\![y]\!], N, D_M)$ by Lemma D.5.

An edge created during the analysis of M has either been created during the analysis of N or of P. Then $T[\![M]\!]\rho_T y = \mathrm{Path_v}(y,M)$.

Let $i \in Ind$.

$$\begin{aligned} T[\![M]\!]\rho_T i &= T[\![P]\!]\rho'_T i \\ &= \mathrm{Path}(p_i D_P, P) && \text{by induction hypothesis on } P \\ &= \mathrm{PathRest}(p_i D_M, P, D_M) && \text{since } pD_P = pD_M \\ &= \mathrm{Path}(p_i D_M, M) \end{aligned}$$

since $\mathrm{PathRest}(p_i D_M, N, D_M) = \lambda c.\emptyset$ by Lemma D.9, therefore $\mathrm{PathRest}(p_i D_M, M, D_M) = \mathrm{PathRest}(p_i D_M, N, D_M) \sqcup \mathrm{PathRest}(p_i D_M, P, D_M) = \mathrm{PathRest}(p_i D_M, P, D_M)$.

• **Case** $M = \texttt{let rec}\ f = \texttt{fun}\ x \to N\ \texttt{in}\ P$. Let $Q = \texttt{fun}\ x \to N$. Let $\rho'_T = \mathrm{lfp}(\lambda\rho_1.\rho_T[f \mapsto T[\![Q]\!]\rho_{1|Ind}]$. Let us first prove the hypothesis on ρ'_T.

We have $\mathrm{Path_a}(p_i\rho[\![f]\!], \rho[\![f]\!]) = \mathrm{Path}(p_i D_Q, Q)$ since all edges starting from $p_i D_Q$ are created during the analysis of Q. They cannot be created during the analysis of P by Lemma D.7. By induction hypothesis on Q, if X satisfies $Xi = \mathrm{Path_a}(p_i\rho[\![f]\!], \rho[\![f]\!]) = \mathrm{Path}(p_i D_Q, Q)$, then $T[\![Q]\!]\rho_T[f \mapsto X]i = \mathrm{Path}(p_i D_Q, Q) = Xi$. Therefore $\rho_T[f \mapsto X]$ is a fixed point of $\lambda\rho_1.\rho_T[f \mapsto T[\![Q]\!]\rho_{1|Ind}]$, and $X \geq \rho'_T[\![f]\!]$.

To prove the other inequality, we perform an induction over the number of imbricated function calls ν. We define the set $SLRP^\nu$ of $SLRP$ paths with at most ν imbricated function calls:

$$\forall \nu \in \mathbb{N}, m \xrightarrow{\iota} n \in SLRPEL^\nu(m,n) \tag{D.1}$$

$$q \in SLRP^\nu(n,p) \Rightarrow m \xrightarrow{(_i} n : q : p \xrightarrow{)_i} r \in SLRPEL^{\nu+1}(m,r) \tag{D.2}$$

$$\forall \nu \in \mathbb{N}, m : \epsilon : m \in SLRP^\nu(m,m) \tag{D.3}$$

$$\begin{aligned} &q_1 \in SLRP^\nu(m,n), q_2 \in SLRPEL^\nu(n,p), \text{there exists no path } m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n' \\ &\qquad \Rightarrow q_1 : q_2 \in SLRP^\nu(m,p) \end{aligned} \tag{D.4}$$

We define RP^ν similarly being the RP paths with at most ν imbricated calls, $\mathrm{PathRest}^\nu$, $\mathrm{Path}^\nu_{\mathrm{a}}$ and Path^ν being the corresponding value of PathRest, $\mathrm{Path_a}$ and Path respectively. The previous lemmas can easily be extended to these paths.

Since the concatenation of two *SLRP* or *RP* paths with at most ν imbricated function calls also has at most ν imbricated function calls, Lemma D.5 becomes (with the same hypotheses)

$$\mathrm{Path}^{\nu}(\rho[\![z]\!], N) \circ \mathrm{Path}^{\nu}(D_N, P) = \mathrm{PathRest}^{\nu}(\rho[\![z]\!], N, D_P).$$

Since ι edges are in all $SLRP^{\nu}$, Lemma D.10 becomes

$$\forall \nu \in \mathbb{N}, G_T^{\tau_x^{\mathrm{g}}} = \mathrm{Path}^{\nu}(\rho[\![x]\!], x).$$

Lemma D.11 becomes

$$I_T^{\tau_x^{\mathrm{g}}[\tau_{x,i}^{\mathrm{g}}]}(\mathrm{Path}_{\mathrm{a}}^{\nu}(p_i\rho[\![x]\!], \rho[\![x]\!])) = \mathrm{Path}^{\nu+1}(p_i D_x, x).$$

Moreover,

$$\mathrm{Path}^0(p_i D_x, x) = \emptyset.$$

The current theorem can then be generalized as follows. If $\forall z \in FV(M), \forall i \in Ind, \rho_T[\![z]\!]i \geq \mathrm{Path}_{\mathrm{a}}^{\nu}(p_i\rho[\![z]\!], \rho[\![z]\!])$, then

$$\begin{aligned} &\forall y \in Var, T[\![M]\!]\rho_T y \geq \mathrm{Path}_{\mathrm{v}}^{\nu+1}(y, M) \\ &\forall i \in Ind, T[\![M]\!]\rho_T i \geq \mathrm{Path}^{\nu+1}(p_i D_M, M) \end{aligned}$$

Without any hypothesis on ρ_T,

$$\begin{aligned} &\forall y \in Var, T[\![M]\!]\rho_T y \geq \mathrm{Path}_{\mathrm{v}}^{0}(y, M) \\ &\forall i \in Ind, T[\![M]\!]\rho_T i \geq \mathrm{Path}^{0}(p_i D_M, M) \end{aligned}$$

The proof proceeds by induction on M, similarly to the present proof.

We have therefore $\mathrm{Path}^0(p_i D_Q, Q) \leq T[\![Q]\!]\rho'_T i = \rho'_T[\![f]\!]i$. Moreover, $\mathrm{Path}^{\nu}(p_i D_Q, Q) \leq \rho'_T[\![f]\!]i$ and implies $\mathrm{Path}^{\nu+1}(p_i D_Q, Q) \leq T[\![Q]\!]\rho'_T i = \rho'_T[\![f]\!]i$. Therefore, $\forall \nu \in \mathbb{N}, \mathrm{Path}^{\nu}(p_i D_Q, Q) \leq \rho'_T[\![f]\!]i$. Hence $\mathrm{Path}(p_i D_Q, Q) \leq \rho'_T[\![f]\!]i$.

We have therefore the equality $\mathrm{Path}(p_i D_Q, Q) = \rho'_T[\![f]\!]i$. On the other hand, if $x \neq f$, $\rho'_T[\![x]\!]i = \rho_T[\![x]\!]i$ therefore ρ'_T satisfies the expected property. We have

$$R(f) = \sqcup_{n\in\mathbb{N}}(T[\![Q]\!]\rho'_T f)^n \circ T[\![P]\!]\rho'_T f$$

(see the computation of R for the proof of analysis E, page 229). By induction hypothesis applied to P and Q,

$$T[\![P]\!]\rho'_T f = \mathrm{Path}_{\mathrm{v}}(f, P) = \mathrm{Path}(D_Q, P), \quad T[\![Q]\!]\rho'_T f = \mathrm{Path}_{\mathrm{v}}(f, Q) = \mathrm{Path}(D_Q, Q)$$

since $D_Q = \rho[\![f]\!]$. Then

$$R(f) = \sqcup_{n\in\mathbb{N}}(\mathrm{Path}(D_Q, Q))^n \circ \mathrm{Path}(D_Q, P).$$

By the same reasoning as in Lemma D.5, but splitting the path in $n+1$ parts: n parts from D_Q to D_Q and a last part from D_Q to $D_P = D_M$,

$$R(f) = \sqcup_{n\in\mathbb{N}}(\mathrm{Path}(D_Q, Q))^n \circ \mathrm{Path}(D_Q, P) = \mathrm{PathRest}(\rho[\![f]\!], M, D_M) = \mathrm{Path}_{\mathrm{v}}(f, M).$$

Let $y \in Var - \{f\}$.

$$T[\![M]\!]\rho_T y = R(y) = T[\![P]\!]\rho'_T y \sqcup (T[\![Q]\!]\rho'_T y) \circ R(f)$$

(see again the computation of R for the proof of analysis E, page 229). By induction hypothesis applied to P and Q, $T[\![P]\!]\rho'_T y = \mathrm{Path_v}(y, P)$ and $T[\![Q]\!]\rho'_T y = \mathrm{Path_v}(y, Q)$. Then

$$T[\![M]\!]\rho_T y = \mathrm{Path_v}(y, P) \sqcup \mathrm{Path_v}(y, Q) \circ \mathrm{Path_v}(f, M).$$

By Lemma D.5, and noticing that $\rho[\![f]\!] = D_Q$ and $D_M = D_P$,

$$\begin{aligned} T[\![M]\!]\rho_T y &= \mathrm{PathRest}(\rho[\![y]\!], P, D_M) \sqcup \mathrm{PathRest}(\rho[\![y]\!], Q, D_M) \\ &= \mathrm{Path_v}(y, M) \end{aligned}$$

Let $i \in Ind$. The induction hypothesis applied to P yields:

$$\begin{aligned} T[\![M]\!]\rho_T i &= T[\![P]\!]\rho'_T i \\ &= \mathrm{Path}(p_i D_P, P) \\ &= \mathrm{PathRest}(p_i D_M, P, D_M) && \text{since } pD_M = pD_P \\ &= \mathrm{Path}(p_i D_M, M) \end{aligned}$$

since $\mathrm{PathRest}(p_i D_M, N, D_M) = \lambda c.\emptyset$ by Lemma D.9.

- **Case** $N\ P$. Let $y \in Var$.

$$\begin{aligned} & T[\![N\ P]\!]\rho_T y \\ &= (T[\![N]\!]\rho_T y) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \sqcup (T[\![P]\!]\rho_T y) \circ (T[\![N]\!]\rho_T 1) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \\ &= \mathrm{Path_v}(y, N) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \sqcup \mathrm{Path_v}(y, P) \circ \mathrm{Path}(p_1 D_N, N) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \\ &= \mathrm{Path_v}(y, N) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \sqcup \mathrm{Path_v}(y, P) \circ \mathrm{Path}(D_P, N) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \\ & \qquad \text{since } p_1 D_N = D_P \\ &= \mathrm{PathRest}(\rho[\![y]\!], N, D_{N\ P}) \sqcup \mathrm{Path_v}(y, P) \circ \mathrm{PathRest}(D_P, N, D_{N\ P}) \end{aligned}$$

since the nodes corresponding to context $\mathrm{Constr}_T(\mathrm{app}, \tau_N)(c)$ in N are the nodes corresponding to context c in $N\ P$.

$$= \mathrm{PathRest}(\rho[\![y]\!], N, D_{N\ P}) \sqcup \mathrm{Path_v}(y, P) \circ \mathrm{Path}(D_P, N\ P)$$

since all edges starting from D_P have been created during the analysis of N (they cannot be created during the analysis of P by Lemma D.7)

$$\begin{aligned} &= \mathrm{PathRest}(\rho[\![y]\!], N, D_{N\ P}) \sqcup \mathrm{PathRest}(\rho[\![y]\!], P, D_{N\ P}) && \text{by Lemma D.5.} \\ &= \mathrm{Path_v}(y, N\ P) \end{aligned}$$

Let $i \in Ind$.

$$\begin{aligned} T[\![N\ P]\!]\rho_T i &= T[\![N]\!]\rho_T(i+1) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \\ &= \mathrm{Path}(p_{i+1} D_N, N) \circ \mathrm{Constr}_T(\mathrm{app}, \tau_N) \\ &= \mathrm{PathRest}(p_i D_{(N\ P)}, N, D_{N\ P}) \end{aligned}$$

since $p_{i+1}D_N = p_iD_{(N\ P)}$, and nodes associated with context $\mathrm{Constr}_T(\mathrm{app}, \tau_N)(c)$ in N are nodes associated with context c in $N\ P$: $\mathrm{Nodes}(D_N, \mathrm{Constr}_T(\mathrm{app}, \tau_N)(c)) = \mathrm{Nodes}(D_{N\ P}, c)$.

$$T[\![N\ P]\!]\rho_T i = \mathrm{PathRest}(p_iD_{(N\ P)}, N\ P, D_{N\ P}) = \mathrm{Path}(p_iD_{(N\ P)}, N\ P)$$

since $\mathrm{PathRest}(p_iD_{N\ P}, P, D_{N\ P}) = \lambda c.\emptyset$ by Lemma D.9.

- **Case** $M = \mathtt{fun}\ x \to N$**.** Let $y \in \mathit{Var}$.

$$\begin{aligned} T[\![M]\!]\rho_T y = {} & (\lambda c.\text{if } c \geq \tau_x \underline{\to} \tau_N \text{ and } y \in FV(M) \text{ then } \underline{\tau_y} \text{ else } \tau_y) \\ & \sqcup (T[\![N]\!]\rho_T[x \mapsto \rho_\top]y) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) \end{aligned}$$

We have $\rho_\top ic = \underline{\tau_{x,i}} = \mathrm{Path_a}(p_i\rho[\![x]\!], \rho[\![x]\!])(c)$ since the parameters of x are marked. Therefore $\rho_T[x \mapsto \rho_\top]$ satisfies the hypothesis and the induction hypothesis can be applied to N. Let $\rho'_T = \rho_T[x \mapsto \rho_\top]$.

$$\begin{aligned} T[\![M]\!]\rho_T y &= (\lambda c.\text{if } c \geq \tau_x \underline{\to} \tau_N \text{ then } \mathrm{PathClo}(\rho[\![y]\!], M)(\underline{\tau_M}) \text{ else } \tau_y) \\ &\quad \sqcup \mathrm{Path_v}(y, N) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) \\ &= \mathrm{PathClo}(\rho[\![y]\!], M) \sqcup \mathrm{PathRest}(\rho[\![y]\!], N, D_M) \\ &= \mathrm{Path_v}(y, M) \end{aligned}$$

Indeed, an edge added during the analysis M is either added during the analysis of N (in which case, the paths beginning with this edge are taken into account by $\mathrm{PathRest}(\rho[\![y]\!], N, D_M)$), either links the top of the closure M (in this case, since the path must not have a strict *SLRP* prefix whose target is in D_M, this path contains one single edge whose target is the top of D_M, and it is taken into account by $\mathrm{PathClo}(\rho[\![y]\!], M)$).

By induction hypothesis applied to N:

$$\begin{aligned} T[\![\mathtt{fun}\ x \to N]\!]\rho_T 1 &= T[\![N]\!]\rho_T[x \mapsto \rho_\top]x \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) \\ &= \mathrm{Path_v}(x, N) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) \end{aligned}$$

Finally, $\rho[\![x]\!] = p_1D_M$ and D_N is also the domain of the result of M, therefore $\mathrm{Path_v}(x, N) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) = \mathrm{Path}(\rho[\![x]\!], N) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) = \mathrm{Path}(p_1D_M, M)$ which is the expected result.

Let $i > 1$. Again by induction hypothesis applied to N:

$$\begin{aligned} T[\![\mathtt{fun}\ x \to N]\!]\rho_T i &= T[\![N]\!]\rho_T[x \mapsto \rho_\top](i-1) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) \\ &= \mathrm{Path}(p_{i-1}D_N, N) \circ \mathrm{Restr}_T(\mathrm{app}, \tau_M) \\ &= \mathrm{Path}(p_iD_M, M) \end{aligned}$$

since the domain of the $(i-1)$th parameter of N is the one of the ith parameter of $M = \mathtt{fun}\ x \to N$, and the domain N is also the domain of the result of M.

- **Case** $M = C(N_1, ..., N_k)$**.** Let $y \in \mathit{Var}$.

$$\begin{aligned} T[\![C(N_1, ..., N_k)]\!]\rho_T y &= \sqcup_j T[\![N_j]\!]\rho_T y \circ \mathrm{Restr}_T(\overline{C_j}, \tau_M) \\ &= \sqcup_j \mathrm{Path_v}(y, N_j) \circ \mathrm{Restr}_T(\overline{C_j}, \tau_M) \\ &= \sqcup_j \mathrm{PathRest}(\rho[\![y]\!], N_j, D_M) \\ &= \mathrm{Path_v}(y, M) \end{aligned}$$

since D_{N_j} is the restriction by accessor $\overline{C_j}$ of D_M therefore $\mathrm{Path}_{\mathrm{v}}(y, N_j) \circ \mathrm{Restr}_T(\overline{C_j}, \tau_M) = \mathrm{PathRest}(\rho[\![y]\!], N_j, D_{N_j}) \circ \mathrm{Restr}_T(\overline{C_j}, \tau_M) = \mathrm{PathRest}(\rho[\![y]\!], N_j, D_M)$.

In this case, M has no parameters, hence the second part of the theorem is void.

- **Case** $N; P$**.** Let $y \in \mathit{Var}$.

$$\begin{aligned} T[\![N;P]\!]\rho_T y &= T[\![N]\!]\rho_T y \circ (\lambda c.\tau_N) \sqcup T[\![P]\!]\rho_T y \\ &= \mathrm{Path}_{\mathrm{v}}(y, N) \circ (\lambda c.\tau_N) \sqcup \mathrm{Path}_{\mathrm{v}}(y, P) \end{aligned}$$

By Lemma D.6, $\mathrm{Path}_{\mathrm{v}}(y, N)(\tau_N) = \mathrm{PathRest}(\rho[\![y]\!], N, D_{N;P})(c)$. Indeed, by Lemma D.7, $\mathrm{Graph}(N, pD_N, \rho)$ only creates edges whose source is in $p_i D_N$ or in $\rho[\![x]\!]$ or in nodes created during the analysis of N. It also marks only nodes of $p_i D_N$ or nodes created during the analysis of N. Hence, no edge has its source in nodes of D_N, and they are not marked.

Since $D_P = D_{N;P}$, $\mathrm{Path}_{\mathrm{v}}(y, P) = \mathrm{PathRest}(\rho[\![y]\!], P, D_{N;P})$.

Finally, $T[\![N;P]\!]\rho_T y = \mathrm{Path}_{\mathrm{v}}(y, N; P)$

Let $i \in \mathit{Ind}$.

$$\begin{aligned} T[\![N;P]\!]\rho_T i &= T[\![P]\!]\rho_T i \\ &= \mathrm{Path}(p_i D_P, P) \\ &= \mathrm{PathRest}(p_i D_{(N;P)}, P, D_{N;P}) && \text{since } pD_{N;P} = pD_P \\ &= \mathrm{Path}(p_i D_{(N;P)}, N; P) \end{aligned}$$

since $\mathrm{PathRest}(p_i D_{(N;P)}, N, D_{N;P}) = \lambda c.\emptyset$ by Lemma D.9.

- **Case** $M =$ `if` N `then` P `else` Q**.** Let $y \in \mathit{Var}$.

$$\begin{aligned} T[\![M]\!]\rho_T y &= (T[\![N]\!]\rho_T y) \circ (\lambda c.\tau_N) \sqcup T[\![P]\!]\rho_T y \sqcup T[\![Q]\!]\rho_T y \\ &= \mathrm{Path}_{\mathrm{v}}(y, N) \circ (\lambda c.\tau_N) \sqcup \mathrm{Path}_{\mathrm{v}}(y, P) \sqcup \mathrm{Path}_{\mathrm{v}}(y, Q) \\ &= \mathrm{PathRest}(\rho[\![y]\!], N, D_M) \sqcup \mathrm{PathRest}(\rho[\![y]\!], P, D_M) \sqcup \mathrm{PathRest}(\rho[\![y]\!], Q, D_M) \\ &= \mathrm{Path}_{\mathrm{v}}(y, M, c) \end{aligned}$$

For the first term, we apply Lemma D.6. For the last two terms, the equality comes from the fact that P, Q and $M =$ `if` N `then` P `else` Q have the same domain.

Let $i \in \mathit{Ind}$.

$$\begin{aligned} T[\![M]\!]\rho_T i &= T[\![P]\!]\rho_T i \sqcup T[\![Q]\!]\rho_T i \\ &= \mathrm{Path}(p_i D_P, P) \sqcup \mathrm{Path}(p_i D_Q, Q) \\ &= \mathrm{PathRest}(p_i D_M, P, D_M) \sqcup \mathrm{PathRest}(p_i D_M, Q, D_M) \\ &\qquad\qquad \text{since } pD_P = pD_Q = pD_M \\ &= \mathrm{Path}(p_i D_M, M) \end{aligned}$$

since $\mathrm{PathRest}(p_i D_M, N, D_M) = \lambda c.\emptyset$ by Lemma D.9.

• **Case** $M = \overline{C_i}(N) \leftarrow P$. Let $y \in Var$.

$$\begin{aligned} T[\![M]\!]\rho_T y &= (T[\![N]\!]\rho_T y) \circ (\lambda c.\tau_N) \sqcup (T[\![P]\!]\rho_T y) \circ (\lambda c.\underline{\tau_P}) \\ &= \mathrm{Path_v}(y, N) \circ (\lambda c.\tau_N) \sqcup \mathrm{Path_v}(y, P) \circ (\lambda c.\underline{\tau_P}) \\ &= \mathrm{PathRest}(\rho[\![y]\!], N, D_M) \sqcup \mathrm{PathRest}(\rho[\![y]\!], P, D_M) \\ &= \mathrm{Path_v}(y, M) \end{aligned}$$

Expression M has no parameter.

• **Case** $\overline{C_i}$. We have to prove that $\rho_T[\![\overline{C_i}]\!]k = \mathrm{Path_a}(p_k\rho[\![\overline{C_i}]\!], \rho[\![\overline{C_i}]\!])$ where $k \in Ind$.

For the first parameter, we have $k = 1$. $\mathrm{Constr}_T(\overline{C_i}, \tau') \circ \mathrm{Restr}_T(\mathrm{app}, \tau' \to \tau_i') = \mathrm{Path_a}(p_1\rho[\![\overline{C_i}]\!], \rho[\![\overline{C_i}]\!])$. The only paths whose sources are in $\mathrm{Nodes}(p_1\rho[\![\overline{C_i}]\!], c)$ are $n_1 \xrightarrow{\iota} n_2$ with $\mathrm{OccType}(p_1\rho[\![\overline{C_i}]\!], n_1) = (\mathrm{Constr}_T(\overline{C_i}, \tau') \circ \mathrm{Restr}_T(\mathrm{app}, \tau' \to \tau_i'))(\mathrm{OccType}(\rho[\![\overline{C_i}]\!], n_2))$, hence the expected equality.

For the other parameters: the nodes of $\mathrm{Nodes}(p_k\rho[\![\overline{C_i}]\!])$ are marked (for $k > 1$), hence $\mathrm{Path_a}(p_k\rho[\![\overline{C_i}]\!], \rho[\![\overline{C_i}]\!])(c) = \underline{\tau_{\overline{C_i},k}}$.

• **The other cases** of functions predefined in the environment can be handled in the same way. □

D.2 Computing summary edges

We prove here the correctness of the Propagate function of Section 5.2. We cannot reuse directly the results of [RSH94] because we may have in parallel ordinary and return edges, which is not the case in [RSH94]. This leads to a slightly different definition of the *SLRP* and *RP* paths. The values of *RP* and *SLRP* used in the following proofs are computed in the graph before any call to Propagate.

Definition D.13 Let $SOP(m, n)$ be the set of paths from m to n containing only ordinary and summary edges, generated by the following rules:

$$m : \epsilon : m \in SOP(m, m) \tag{D.5}$$

$$\begin{aligned} &q_1 \in SOP(m, n), \text{there exists no path } m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n' \\ &\quad \Rightarrow q_1 : n \xrightarrow{\iota} p \in SOP(m, p) \end{aligned} \tag{D.6}$$

Theorem D.14 *Let* Propagate′ *be*

Propagate′(n) =
  if Visited(n) **then return** ∞
  Visited(n) = true
  if Mark(n) **then return** 0
  foreach $s \in$ Callee(n),
    Propagate(s, s)
    if Mark(s) **then return** 0
  return $\min\{\text{Propagate}'(m) | m \in \text{Succ}(n)\}$.

(By convention $\min \emptyset = \infty$*. At the beginning,* $\mathrm{Visited}(n) = \mathrm{false}$*. At the end, we reset* $\mathrm{Visited}(n) = \mathrm{false}$*. The goal of the* Visited *flag is simply to avoid visiting several times the same node.) Let* $v = \mathrm{Propagate}'(m)$*. If there exists no edge labelled* $(_i$ *of target* m*,*

- $v = 0$ *if and only if there exists* p *initially marked such that* $RP(m, p) \neq \emptyset$*;*
- *If* $v > 0$*,* p *is reachable from* m *(by ordinary and summary edges) if and only if* $SLRP(m, p) \neq \emptyset$*.*

Proof: For the moment, we ignore the hypothesis on m, m can be any node. We first prove properties of the function Propagate (Properties 1 to 4), and then prove the correctness of Propagate$'$.

• **Property 1.** If $s \in \mathrm{PathEdge}(n)$, $SLRP(s, n) \neq \emptyset$ and if $SOP(m, p) \neq \emptyset$, $SLRP(m, p) \neq \emptyset$. If $p \in \mathrm{Succ}(m)$, $SLRPEL(m, p) \neq \emptyset$.

By induction on the total number of nodes inserted into PathEdge and Succ. The case 0 is obvious. When a new node s is inserted into $\mathrm{PathEdge}(n)$, $\mathrm{Propagate}(n, s)$ has been called, which may correspond to three cases:

- The calls to $\mathrm{Propagate}(s', s')$ (line 12). In this case, we have $SLRP(s', s') \neq \emptyset$ (it contains the empty path from s' to s').
- The call to $\mathrm{Propagate}(m, s)$ (line 10). We have $s \in \mathrm{PathEdge}(n)$ therefore $\exists q_1 \in SLRP(s, n)$ (induction hypothesis), and $m \in \mathrm{Succ}(n)$ therefore $\exists q_2 \in SLRPEL(n, m)$ (again by induction hypothesis) therefore $q_1 : q_2 \in SLRP(s, m) \neq \emptyset$ by concatenating the paths (Rule (5.4) since there exists no path $c \xrightarrow{(_i} s : q_1 : n \xrightarrow{)_i} n'$, otherwise this part of the code would not be executed; we would have found a corresponding c at line 8).
- The call to $\mathrm{Propagate}(r, s')$ (line 7). In this case, we have $s' \in \mathrm{PathEdge}(c)$, therefore by induction hypothesis, $\exists q_1 \in SLRP(s', c)$. Moreover $c \xrightarrow{(_i} s \in Edge$ since $c \in \mathrm{Callers}(i, s)$. $s \in \mathrm{PathEdge}(n)$, therefore $\exists q_2 \in SLRP(s, n)$. Finally, $n \xrightarrow{)_i} r \in Edge$ since $n \xrightarrow{)_i} r \in \mathrm{Ret}(n)$. Then $q_1 : c \xrightarrow{(_i} s : q_2 : n \xrightarrow{)_i} r \in SLRP(r, s') \neq \emptyset$ by Rule (5.2) and Rule (5.4) since there exists no path $c' \xrightarrow{(_j} s' : q_1 : c \xrightarrow{)_j} n'$. Indeed, assume that such a path exists. Since there exists an edge $c \xrightarrow{(_i} s$, c is in a $p_k D_x$ domain for a certain variable x and a parameter index k. Then c is in a $p_l D_M$ domain when it is created (notice that the only transfer of nodes between different kinds of domains $p_l D_M$, D_M, $p_l \rho[\![x]\!]$ and $\rho[\![x]\!]$ is from $p_l D_M$ to $\rho[\![y]\!]$ in the `fun` case). Since there exists an edge $c \xrightarrow{)_j} n'$, c is also in a domain $\rho[\![x]\!]$. Then it has been put in this domain during this analysis of a function, and $p_l \rho[\![x]\!]$ is created by MarkParams. Since there exists an edge $c' \xrightarrow{(_j} s'$, $\exists l, s' \in p_l \rho[\![x]\!]$. Then s' is always in a $p_l \rho[\![x]\!]$ and no edge has source s'. This contradicts the existence of the path q_1.

When a node r is inserted into $\mathrm{Succ}(c)$ (line 6), we have $c \xrightarrow{(_i} s \in Edge$, $s \in \mathrm{PathEdge}(n)$, therefore there exists a path $q_2 \in SLRP(s, n) \neq \emptyset$ and $n \xrightarrow{)_i} r \in Edge$ therefore $c \xrightarrow{(_i} s : q_2 : n \xrightarrow{)_i} r \in SLRPEL(c, r)$ (Rule (5.2)).

Assume that $q \in SOP(m,p)$ after adding r to Succ(c). If q does not contain edge $c \xrightarrow{\iota} r$, by induction hypothesis, we have obviously $SLRP(m,p) \neq \emptyset$. If q contains $c \xrightarrow{\iota} r$, q can be split into $q = q_1' : c \xrightarrow{\iota} r : q_2' : c \xrightarrow{\iota} r : ... : c \xrightarrow{\iota} r : q_k'$. Since q_1' exists, $\exists q_1'' \in SLRP(m,c) \neq \emptyset$; since q_k' exists, $\exists q_k'' \in SLRP(r,p) \neq \emptyset$. Then $q_1'' : c \xrightarrow{(_i} s : q_2 : n \xrightarrow{)_i} r : q_k'' \in SLRP(m,p)$ (Rule (5.4)).

- **Property 2.** If m is marked, then there exists p initially marked such that $RP(m,p) \neq \emptyset$.

By induction on the number of marked nodes added by Propagate. The case 0 is obvious. A node s may be marked in two cases:

- The node s is marked because n is marked and $s \in$ PathEdge(n) (line 9). By the first property, $\exists q_1 \in SLRP(s,n)$. Then by Lemma D.1, there exists no edge $c \xrightarrow{(_i} n$. Then no call of the form Propagate(x,n) is done, therefore n is never marked during the execution of Propagate. Therefore n was initially marked. Moreover, $q_1 \in RP(s,p) \neq \emptyset$ ((5.5) since there exists no path $c \xrightarrow{(_j} s : q_1 : n \xrightarrow{)_j} n'$, otherwise this part of the algorithm could not be executed).

- The node s is marked because s' is marked and $s' \in$ Call(n), $s \in$ PathEdge(n) (line 13). By induction hypothesis, there exists p initially marked such that $\exists q_1 \in RP(s',p) \neq \emptyset$. By the first property, $\exists q_2 \in SLRP(s,n) \neq \emptyset$. Then $q_2 : n \xrightarrow{(_i} s' : q_1 \in RP(s,p)$ (Rule (5.6) since there exists no path $c \xrightarrow{(_j} s : q_2 : n \xrightarrow{)_j} n'$, otherwise this part of the algorithm could not be executed).

- **Property 3.** If Propagate(m,m) has been called, and $SLRP(m,p) \neq \emptyset$, then at least one of the two following properties holds: 1) m is marked, or 2) $SOP(m,p) \neq \emptyset$ and $m \in$ PathEdge(p). By induction on the length of the path $q \in SLRP(m,p)$.

- If q is the empty path (built by Rule (5.3)), $SOP(m,p) = SOP(m,m) \neq \emptyset$ and $m \in$ PathEdge(m) since Propagate(m,m) has been called.

- If q has been built by Rule (5.4), $q = q_1 : q_2$, $q_1 \in SLRP(m,n)$, $q_2 \in SLRPEL(n,p)$, and there exists no path $m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n'$. We apply the induction hypothesis to q_1. Then either m is marked and we have the result, or $SOP(m,n) \neq \emptyset$ and $m \in$ PathEdge(n). Then Propagate(n,m) has been called. During this call, no corresponding c node is found (since there exists no path $m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n'$). If n was marked, Propagate marks m, therefore Property 3 is satisfied. Otherwise, we have again two cases.

  If $q_2 = n \xrightarrow{\iota} p$, Propagate($p,m$) is called (line 10) therefore $m \in$ PathEdge(p). Moreover $SOP(m,p) \neq \emptyset$ (Rule D.6).

  If $q_2 = n \xrightarrow{(_i} p : q' : r \xrightarrow{)_i} s$, Propagate($p,p$) is called (line 12, since n is not marked). By induction hypothesis applied to q', $SOP(p,r) \neq \emptyset$ and $p \in$ PathEdge(r) or p is marked. If p is marked, Propagate marks m. Otherwise, since $p \in$ PathEdge(r), Propagate(r,p) has been called. During this call, the existence of edges $n \xrightarrow{(_i} p$ and $r \xrightarrow{)_i} s$ has been noticed, the summary edge $n \xrightarrow{\iota} s$ has been added, and Propagate(m, s) has been called (line 7, since $m \in$ PathEdge(n)) therefore $m \in$ PathEdge(s).

• **Property 4.** If Propagate(m, m) has been called and there exists p initially marked such that $RP(m,p) \neq \emptyset$, then m is marked. By induction on the building of $q \in RP(m,p)$.

- If q has been built by Rule (5.5), $q \in SLRP(m,p)$. By Property 3, m is marked (and we have the result) or $SOP(m,p) \neq \emptyset$ and $m \in$ PathEdge(p). Then Propagate(p, m) has been called. During this call, no corresponding c node has been found, since there exists no path $m' \xrightarrow{(_i} m : q : p \xrightarrow{)_i} p'$, therefore since p is marked, Propagate marks m (line 9).

- If q has been build by Rule (5.6), $q = q_1 : n \xrightarrow{(_i} r : q_2$ with $q_1 \in SLRP(m,n)$, $q_2 \in RP(r,p)$. By Property 3, m is marked (and we have the result) or $SOP(m,n) \neq \emptyset$ and $m \in$ PathEdge(n). Then Propagate(n, m) has been called. During this call, no corresponding c node has been found, since there exists no path $m' \xrightarrow{(_i} m : q_1 : n \xrightarrow{)_i} n'$, then either n is marked and Propagate marks m (line 9), or Propagate(r, r) is called (line 12). By induction on q_2, r is marked. Then Propagate marks m when returning from Propagate(r, r) (line 13).

• **Property 5.** We now assume that node m satisfies the hypotheses of the theorem. Let us prove that, if Propagate$'(m) \neq 0$, and $SLRP(m,p) \neq \emptyset$ then $SOP(m,p) \neq \emptyset$ and Propagate$'(p)$ has been called. By induction on the building of the path $q \in SLRP(m,p)$.

- If q is an empty path (built by Rule (5.3)), $SOP(m,p) \neq \emptyset$.

- If q has been built by Rule (5.4), $q = q_1 : q_2$, $q_1 \in SLRP(m,n)$, $q_2 \in SLRPEL(n,p)$. By induction hypothesis on q_1, $\exists q_1' \in SOP(m,n)$, and Propagate$'(n)$ has been called. Two cases have to be distinguished according to the building of q_2.

  If $q_2 = n \xrightarrow{\iota} p$, then $q_1' : n \xrightarrow{\iota} p \in SOP(m,p)$ and Propagate$'(p)$ is called.

  If $q_2 = n \xrightarrow{(_i} n' : q_3 : p' \xrightarrow{)_i} p$, Propagate$(n', n')$ is called, therefore by Property 3, n' is marked or $\exists q_3' \in SOP(n', p')$ and $n' \in$ PathEdge(p') Then Propagate(p', n') has been called. Using the same reasoning as in the proof of Property 3, during this call, the summary edge $n \xrightarrow{\iota} p$ has been added. Therefore $q_1' : n \xrightarrow{\iota} p \in SOP(m,p)$, and Propagate$'(p)$ is called.

• **Proof of the theorem.** Let us assume that $v = 0$. Then two cases have to be distinguished. First case: the value 0 is returned at line 1 of Propagate$'$. In this case, there exists a marked node p reachable from m by ordinary and summary edges. Let q be a path from m to p. Since there exists no $(_i$ edge of target m, $q \in SOP(m,p)$. Then by Property 1, $\exists q' \in SLRP(m,p)$. Since p is marked but there exists no $(_i$ edge of target p (since there exists an edge labelled ι or $)_i$ with target p, Lemma D.1), no call of the form Propagate(x, p) is executed, and Propagate does not mark p. Therefore, p was in fact initially marked. Since no $(_i$ edge has target m, by Rule (5.5), $q' \in RP(m,p)$. Second case: the value 0 is returned at line 4 of Propagate$'$. In this case, there exists a node n reachable from m by ordinary and summary edges by a path q, an edge $n \xrightarrow{(_i} p$ and a marked node p. By property 1, there exists $q' \in SLRP(m, n)$. By property 2, $\exists p'$ initially marked such that $\exists q'_2 \in RP(p, p')$. Since there exists no $(_i$ edge of target m, by Rule (5.6), $q' : n \xrightarrow{(_i} p : q'_2 \in RP(m, p')$.

Conversely, let us assume that there exists p initially marked such that $q \in RP(m, p)$. Let us prove by contradiction that $v = 0$. We assume that $v > 0$. If $q \in SLRP(m, p)$, by Property 5, Propagate$'(p)$ has been called. Since p is marked, Propagate$'(p) = 0$ and this 0 value is returned to all callers, therefore $v = 0$. Contradiction. If $q = q_1 : n \xrightarrow{(_i} r : q_2$, $q_1 \in SLRP(m, n)$, $q_2 \in RP(r, p)$, by Property 5, Propagate$'(n)$ has been called. Therefore Propagate(r, r) is called. By property 4, r is marked, therefore Propagate$'(n) = 0$ therefore $v = 0$. Contradiction.

We now assume that $v > 0$. Let q be a path from m to p by ordinary and summary edges. Since there exists no $(_i$ edge of target m, $q \in SOP(m, p)$. By Property 1, $SLRP(m, p) \neq \emptyset$.

Conversely, if $q \in SLRP(m, p)$, since Propagate$'(m)$ has been called, by Property 5, $\exists q' \in SOP(m, p)$, therefore p is reachable from m by ordinary and summary edges. □

D.3 Correctness of the whole algorithm

Theorem D.14 is still true for the Propagate$'$ function of Section 5.3.1, since Propagate$'$ returns 0 in the same cases.

Theorem 5.14 (The implementation of T is sound and complete) *Let us consider two kinds of expressions:*

- `let` x `=` M `in` N. *In this case, let ℓ be the location at the top of x. Let us assume that the creation of this location is the last operation of M. Let n_0 be the node at the top of the graph* $\mathrm{Gr}(\tau_x^{\mathrm{g}})$. *Let* $\phi = T[\![N]\!]\rho\, x$.
- `let rec` $x =$ `fun` $y \rightarrow M$ `in` N. *In this case, let ℓ be the location of the closure x. Let n_0 be the node at the top of the graph* $\mathrm{Gr}(\tau_x^{\mathrm{g}})$. *Let $\phi = R(x)$ where R is defined in the* `let rec` *case of Figure 3.6.*

In these two cases, $n_0 \notin \phi(\underline{\tau_N})$ *if and only if* Propagate$'(\mathrm{TopNode}(\rho[\![x]\!])) >$ Label$(\mathrm{TopNode}(D_N))$. *If this condition is satisfied, ℓ can be stack allocated, and deallocated at the end of N.*

Proof: We have to distinguish the `let` and `let rec` cases.

- `let` case. By Theorem D.12, $\phi(\underline{\tau_N}) = T[\![N]\!]\rho_T x \underline{\tau_N} = \mathrm{Path}_{\mathrm{v}}(x, N)(\underline{\tau_N})$.

  Furthermore, all edges which have their source in $\rho[\![x]\!]$ have been created during the analysis of N (since $\rho[\![x]\!]$ is created during the analysis of the `let` expression, and during the analysis of M, $D_M = \rho[\![x]\!]$ is the domain corresponding to the result of the expression, and no edge is created with a source in the domain corresponding to the result of the expression). Then

$$
\begin{aligned}
\phi(\underline{\tau_N}) &= \mathrm{Path}_{\mathrm{v}}(x, N)(\underline{\tau_N}) \\
&= \cup\{\mathrm{OccType}(D_M, m) | (\exists n \in \mathrm{Nodes}(D_N), \exists q \in SLRP(m, n) \\
&\quad \text{or } \exists n' \text{ such that } \mathrm{Mark}(n') \text{ and } q \in RP(m, n'), q \text{ is not of the form} \\
&\quad q = r : s, r \in SLRP(m, n), n \in \mathrm{Nodes}(D_N))\}
\end{aligned}
$$

- `let rec` case. We have seen in the proof of Theorem D.12 that $\phi(\underline{\tau_N}) = R(x)(\underline{\tau_N}) = \text{Path}_\text{v}(x, P)(\underline{\tau_N})$ where $P = \texttt{let rec}\ x = \texttt{fun}\ y \to M\ \texttt{in}\ N$.

  Furthermore, all edges which have their source in $\rho[\![x]\!]$ have been created during the analysis of P, and $D_P = D_N$, then

$$\begin{aligned}\phi(\underline{\tau_N}) &= \text{Path}_\text{v}(x, P)(\underline{\tau_N}) \\ &= \cup\{\text{OccType}(D_M, m)|(\exists n \in \text{Nodes}(D_N), \exists q \in SLRP(m, n) \\ &\quad \text{or } \exists n' \text{ such that Mark}(n') \text{ and } q \in RP(m, n'), q \text{ is not of the form} \\ &\quad q = r : s, r \in SLRP(m, n), n \in \text{Nodes}(D_N))\}\end{aligned}$$

Assume that there exists $q \in RP(m, n')$ and that q is of the form $q = r : s$, $r \in SLRP(m, n)$, $n \in \text{Nodes}(D_N)$. Then the path r proves that $\text{OccType}(D_M, m) \subseteq \phi(\underline{\tau_N})$. The condition on the form of q can then be removed. Then

$$\begin{aligned}\phi(\underline{\tau_N}) = \cup\,\{&\text{OccType}(D_M, m)|\exists n \in \text{Nodes}(D_N), SLRP(m, n) \neq \emptyset \\ &\text{or } \exists n' \text{ such that Mark}(n') \text{ and } RP(m, n') \neq \emptyset\}\end{aligned}$$

Moreover, no $(_i$ edge has a target $m \in \text{Nodes}(D_M)$ node of the domain associated with the result of expression M (seen in Lemma D.1). Then Theorem D.14 can be applied. Therefore

$$\begin{aligned}\phi(\underline{\tau_N}) = \cup\,\{&\text{OccType}(D_M, m)|\exists n \in \text{Nodes}(D_N) \text{ reachable from } m \text{ by} \\ &\text{ordinary or summary edges or Propagate}'(m) = 0\}\end{aligned}$$

If $n \in \text{Nodes}(D_N)$ is reachable from m by ordinary or summary edges, then $\text{Propagate}'(m) \leq \text{Label}(n) \leq \text{Label}(\text{TopNode}(D_N))$ (all nodes of a domain have a label smaller than its top node, since either they are all created at the same time by Type and have the same label, or the top node has been added after the other nodes in the application case by Build, and it has a greater label than the other nodes). Conversely, if $\text{Propagate}'(m) \leq \text{Label}(\text{TopNode}(D_N))$, then there exists a path $q \in SOP(m, n')$ with $\text{Label}(n') \leq \text{Label}(\text{TopNode}(D_N))$. Assume that this path $q = n_1 \xrightarrow{\iota} n_2 ... n_{p-1} \xrightarrow{\iota} n_p$ does not contain any node of $\text{Nodes}(D_N)$. By property 1 of Theorem D.14, $\exists q_i \in SLRPEL(n_i, n_{i+1})$, and the targets of q_i are not in $\text{Nodes}(D_N)$. Then by Lemma D.4, all targets of q_i have been created during the analysis of N. Then if path q is not empty, $n' = n_p$ has been created during the analysis of N and $\text{Label}(n') > \text{Label}(\text{TopNode}(D_N))$. If q is empty, $n' = m$ has been created during the analysis of the `let` expression and we also have $\text{Label}(n') > \text{Label}(\text{TopNode}(D_N))$. Contradiction in both cases. Therefore q contains a node $n \in \text{Nodes}(D_N)$, which is therefore reachable by ordinary or summary edges. Finally,

$$\phi(\underline{\tau_N}) = \cup\{\text{OccType}(D_M, m)|\text{Propagate}'(m) \leq \text{Label}(\text{TopNode}(D_N))\}.$$

$\text{OccType}(D_M, \text{TopNode}(\rho[\![x]\!])) = \{n_0\}$, and $\forall m \neq \text{TopNode}(\rho[\![x]\!]), n_0 \notin \text{OccType}(D_M, m)$ therefore

$$n_0 \notin \phi(\underline{\tau_N}) \Leftrightarrow \text{Propagate}'(\text{TopNode}(\rho[\![x]\!])) > \text{Label}(\text{TopNode}(D_N)).$$

□

Appendix E

Proofs of escape analysis for Java

E.1 Correctness of alias analysis

We prove the correctness of our alias analysis with respect to a small step semantics similar to that of [Qia98]. The main differences are that we do not separate stack and local variables, and we do not consider subroutines, uninitialized objects. Our semantics is described in Figures E.1 and E.2. ***Locations*** $\ell \in \mathit{Loc}$ represent objects allocated in the heap (a new location is created for each object). ***Values*** $o \in \mathit{Val}$ are either objects represented by their location or integers or the special value `null`. The ***heap*** associates with each object (or class for static fields) and each field $f \in \mathit{Field}$ (or array index $n \in \mathbb{N}$) its contents. $L \in \mathit{LocVar}$ yields for each local variable of the current method its contents. The ***Java stack*** J represents the stack of the callers of the current method (for each caller, J contains the value of the local variables and of the program counter just after the call). $TY \in \mathit{OTypes}$ represents the type of each object allocated on the heap. This type information is necessary to define the semantics of virtual method calls. All this information is summarized in the state $\Xi \in \mathit{State}$, and finally the execution trace T indicates the state at each step of the program execution.

The method really called by a virtual call is computed by $\mathrm{getMeth}((C, m', t), C') = (C', m', t)$ if the method (C', m', t) exists, otherwise $\mathrm{getMeth}((C, m', t), C') = \mathrm{getMeth}((C, m', t), C'')$ where C'' is the superclass of C'. The class C' must be a subclass of C. The program counter of the following instruction is: $\mathrm{next}(pc) = \mathrm{next}((m, a)) = (m, a+1)$. $\Xi[H]$ is the state Ξ where the heap has been replaced by H. We have similar notations for other parts of the state.

Definition E.1 (Locations represented by a path) Let $\mathrm{loca} : (\mathit{Val} \cup \{\mathrm{static}\}) \times \mathit{State} \times \mathit{Path} \to \mathcal{P}(\mathit{Loc})$ such that $\mathrm{loca}(o, \Xi, p)$ is the set of locations of the part of value o represented by path p in state Ξ (this set contains at most one element).

Let H be the heap in state Ξ.

- $\mathrm{loca}(o, \Xi, p) = \emptyset$ if $o \notin \mathit{Loc} \cup \{\mathrm{static}\}$.
- $\mathrm{loca}(\ell, \Xi, \epsilon) = \{\ell\}$ if $\ell \in \mathit{Loc}$.
- $\mathrm{loca}(\ell, \Xi, f.p) = \mathrm{loca}(H(\ell)(f), \Xi, p)$ if $\ell \in \mathit{Loc}$ and $f \in \mathit{Field}$. If $H(\ell)(f)$ is not defined (the f field does not exist in the object), then $\mathrm{loca}(\ell, \Xi, f.p) = \emptyset$.
- $\mathrm{loca}(\ell, \Xi, n.p) = \mathrm{loca}(H(\ell)(n), \Xi, p)$ if $\ell \in \mathit{Loc}$, $n \in \mathbb{N}$. If $H(\ell)(n)$ is not defined, $\mathrm{loca}(\ell, \Xi, n.p) = \emptyset$.

$\ell \in \mathit{Loc}$	Set of locations (countable set)
$o \in \mathit{Val}$	$\mathit{Loc} \cup [\![-2^{31}, 2^{31}-1]\!] \cup \{\texttt{null}\}$
$H \in \mathit{Heap}$	$(\mathit{Loc} \cup \mathit{Class}) \to (\mathit{Field} \cup \mathbb{N}) \to \mathit{Val}$
$L \in \mathit{LocVar}$	$\mathit{Var} \to \mathit{Val}$
$J \in \mathit{JavaStack}$	$(\mathit{LocVar} \times \mathit{PC})^*$
$\mathit{TY} \in \mathit{OTypes}$	$\mathit{Loc} \to \mathit{RefType}$
$\Xi \in \mathit{State}$	$\mathit{Heap} \times \mathit{LocVar} \times \mathit{PC} \times \mathit{JavaStack} \times \mathit{OTypes}$
$T \in \mathit{Trace}$	$\mathit{Date} \to \mathit{State}$

Figure E.1: Domains for the small step semantics

P(pc)	Transition
entry	$\Rightarrow \Xi[pc = (\texttt{main}, 0), L, J = [\,]]$
$v_1 = v_2$	$\Xi[pc, L] \Rightarrow \Xi[\text{next}(pc), L[v_1 \mapsto L(v_2)]]$
$v_1 = (t)v_2$	$\Xi[pc, L] \Rightarrow \Xi[\text{next}(pc), L[v_1 \mapsto L(v_2)]]$
$v_1 = \phi(v_2, v_3)$	$\Xi[pc, L] \Rightarrow \Xi[\text{next}(pc), L[v_1 \mapsto L(v_2)$ or $L(v_3)$[a]$]]$
$v_1.f = v_2$	$\Xi[pc, L, H] \Rightarrow \Xi[\text{next}(pc), H[L(v_1) \mapsto H(L(v_1))[f \mapsto L(v_2)]]$
$v_2 = v_1.f$	$\Xi[pc, L, H] \Rightarrow \Xi[\text{next}(pc), L[v_2 \mapsto H(L(v_1))(f)]]$
$C.f = v$	$\Xi[pc, L, H] \Rightarrow \Xi[\text{next}(pc), H[C \mapsto H(C)[f \mapsto L(v)]]$
$v = C.f$	$\Xi[pc, L, H] \Rightarrow \Xi[\text{next}(pc), L[v \mapsto H(C)(f)]]$
$v_1[v_2] = v_3$	$\Xi[pc, L, H] \Rightarrow \Xi[\text{next}(pc), H[L(v_1) \mapsto H(L(v_1))[L(v_2) \mapsto L(v_3)]]$
$v_1 = v_2[v_3]$	$\Xi[pc, L, H] \Rightarrow \Xi[\text{next}(pc), L[v_1 \mapsto H(L(v_2))(L(v_3))]]$
$v = \texttt{null}$	$\Xi[pc, L] \Rightarrow \Xi[\text{next}(pc), L[v \mapsto \texttt{null}]]$
$v = \texttt{new}\ C$	$\Xi[pc, L, H, TY] \Rightarrow \Xi[\text{next}(pc), L[v \mapsto o], H[o \mapsto \{f \mapsto \texttt{null}$[b]$\}], TY[o \mapsto C]]$ o is a new location, f is any field of the C class or a superclass of C.
$v = \texttt{new}\ t[w]$	$\Xi[pc, L, H, TY] \Rightarrow \Xi[\text{next}(pc), L[v \mapsto o], H[o \mapsto \{n \mapsto \texttt{null}$[c]$\}], TY[o \mapsto t\texttt{[]}\,]]$ o is a new location, $n \in [\![0, L(w)-1]\!]$
$w = v_0.m'(v_1, ..., v_j)$	$\Xi[pc, J, L, TY] \Rightarrow \Xi[(\text{getMeth}(m', TY(o)), 0), (\text{next}(pc), L) : J, \{p_i \mapsto v_i\}]$ The p_i are the names of formal parameters of getMeth$(m', TY(o))$
$\texttt{return}\ v$	$\Xi[pc, (pc', L') : J, L] \Rightarrow \Xi[pc', J, L'[w \mapsto L(v)]]$ w is the variable in which we store the result of the corresponding method call.
$\texttt{if}$ (...) $\texttt{goto}\ a$	$\Xi[pc = (m, x)] \Rightarrow \Xi[(m, a)]$ or $\Xi[\text{next}(pc)]$

[a] Take $L(v_2)$ if v_2 has been defined after v_3 in the execution trace, or v_3 has never been defined. Take $L(v_3)$ otherwise.

[b] This assumes that field f contains objects or arrays. The initial value is 0 for integers.

[c] This assumes that t is an object or array type. The initial value is 0 for integers.

Figure E.2: Small step semantics

- $\mathrm{loca}(\mathrm{static}, \Xi, (C, f, t).p) = \mathrm{loca}(H(C)(C, f, t), \Xi, p)$. $\mathrm{loca}(\mathrm{static}, \Xi, \epsilon) = \mathrm{loca}(\mathrm{static}, \Xi, n.p) = \emptyset$ where $n \in \mathbb{N}$.

Definition E.2 (Index) We define indices by $Idx = Var \cup \{\mathrm{Old}(n, v) | n \in \mathbb{N}, v \in Var\} \cup \{\mathrm{static}\}$. Indices will be used to designate local variables in a state or an abstract state. The local variable v in the nth element of the Java stack will be represented by $\mathrm{Old}(n, v)$. The local variable $v \in Var$ in the method currently executed will be represented by v itself. static is used for static fields.

The value associated with an index is defined by: $\mathrm{Val} : State \times Idx \to (Val \cup \{\mathrm{static}\})$ is $\mathrm{Val}(\Xi[L], v) = L(v)$, $\mathrm{Val}(\Xi[J], \mathrm{Old}(l, v)) = L_l(v)$ with $J(l) = (L_l, pc_l)$, $\mathrm{Val}(\Xi, \mathrm{static}) = \mathrm{static}$.

The abstract value associated with an index $\mathrm{Val}^{\#} : State^{\#} \times Idx \to (OName \cup \{\emptyset, \mathrm{static}\})$ is defined by $\mathrm{Val}^{\#}((L, J), v) = L(v)$ and $\mathrm{Val}^{\#}((L, J), \mathrm{Old}(l, v)) = J(l)(v)$, $\mathrm{Val}^{\#}((L, J), \mathrm{static}) = \mathrm{static}$.

Definition E.3 (Correctness) Let $T \in Trace$ be an execution trace, $T^{\#} \in Trace^{\#}$ an abstract trace, $R \in DateAlias$ the alias relation, and $d \in Date$ a date. The relation $\mathrm{corr}(T, T^{\#}, R, d)$ means that the alias relation R and the abstract trace $T^{\#}$ computed in Section 7.3 are correct with respect to the corresponding execution trace T at date d. It is defined as follows.

Let $\Xi = T(d)$ be the concrete state at date d. Let $\Xi_1 = T(d_1)$ and $\Xi_2 = T(d_2)$ be two states (with $d_1 \leq d$, $d_2 \leq d$), i_1 and $i_2 \in Idx$ two indices. Let $o_1 = \mathrm{Val}(T(d_1), i_1)$ and $o_2 = \mathrm{Val}(T(d_2), i_2)$ the two corresponding values, $N_1 = \mathrm{Val}^{\#}(T^{\#}(d_1), i_1)$ and $N_2 = \mathrm{Val}^{\#}(T^{\#}(d_2), i_2)$ the corresponding abstract values. Then $\mathrm{corr}(T, T^{\#}, R, d)$ if and only if for all $i_1, i_2 \in Idx$, $p_1, p_2 \in Path$, $d_1 \leq d$, $d_2 \leq d$,

$$\mathrm{loca}(o_1, \Xi, p_1) \cap \mathrm{loca}(o_2, \Xi, p_2) \neq \emptyset \Rightarrow (N_1.p_1 \sim N_2.p_2) \in R(d). \tag{E.1}$$

Essentially, if two paths $N_1.p_1$ and $N_2.p_2$ may represent the same location, then they are aliased.

Alias analysis has been defined in Figure 7.4.

Theorem E.4 (Correctness) $\forall d, \mathrm{corr}(T, T^{\#}, R, d)$.

Proof: By induction on d. Let $\Xi = T(d)$, $\Xi' = T(d+1)$.

- **Cases in which the heap is not modified.** We only have to find the correspondence between the designation of objects at date d and at date d'. This correspondence can be formalized by a function $\psi : (Idx \times Path) \to (Idx \times Path)$ yielding the designation at date d from the designation at date $d+1$. $\psi(\mathrm{static}, p) = (\mathrm{static}, p)$.

In the cases $v_3 = v_4$ and $v_3 = (t)v_4$, $\psi(v_3, p) = (v_4, p)$, $\psi(i, p) = (i, p)$ if $i \neq v_3$.

In the case $v_3 = \phi(v_4, v_5)$, if v_4 has been defined after v_5, $\psi(v_3, p) = (v_4, p)$, otherwise, $\psi(v_3, p) = (v_5, p)$. $\psi(i, p) = (i, p)$ if $i \neq v_3$.

In the case $v_3 = v_4.f$, $\psi(v_3, p) = (v_4, f.p)$, $\psi(i, p) = (i, p)$ if $i \neq v_3$.

In the case $v_3 = C.f$, $\psi(v_3, p) = (\mathrm{static}, f.p)$, $\psi(i, p) = (i, p)$ if $i \neq v_3$.

In the case $v_3 = v_4[v_5]$, $\psi(v_3, p) = (v_4, \mathrm{Val}(\Xi, v_5).p)$, $\psi(i, p) = (i, p)$ if $i \neq v_3$.

In the case $w = v_0.m'(v_1, ..., v_n)$: $\psi(p_i, p) = (v_i, p)$ where $p_0, ..., p_n$ are the formal parameters of m', $\psi(\mathrm{Old}(0, v), p) = (v, p)$, $\psi(\mathrm{Old}(n+1, v), p) = (\mathrm{Old}(n, v), p)$.

In the case `return` v: $\psi(w,p) = (v,p)$ where w is the variable in which the result is stored, $\psi(v,p) = (\mathrm{Old}(0,v),p)$, $\psi(\mathrm{Old}(n,v),p) = (\mathrm{Old}(n+1,v),p)$.

In the case `if` (...) `goto` a, $\psi(i,p) = (i,p)$.

If $(i^\circ, p^\circ) = \psi(i,p)$, we can easily check on the semantics that $\mathrm{loca}(\mathrm{Val}(T(d{+}1),i),\Xi',p) = \mathrm{loca}(\mathrm{Val}(T(d),i^\circ),\Xi,p^\circ)$. And we check on the definition of the alias analysis that either $\mathrm{Val}^\#(T^\#(d),i^\circ).p^\circ = \mathrm{Val}^\#(T^\#(d+1),i).p$ or the alias $\mathrm{Val}^\#(T^\#(d),i^\circ).p^\circ \sim \mathrm{Val}^\#(T^\#(d+1),i).p$ is added at date d.

Let $(i_1{}^\circ, p_1{}^\circ) = \psi(i_1,p_1)$, $d_1{}^\circ = d$ if $d_1 = d+1$, $(i_1{}^\circ, p_1{}^\circ) = (i_1,p_1)$, $d_1{}^\circ = d_1$ otherwise, $(i_2{}^\circ, p_2{}^\circ) = \psi(i_2,p_2)$, $d_2{}^\circ = d$ if $d_2 = d+1$, $(i_2{}^\circ, p_2{}^\circ) = (i_2,p_2)$, $d_2{}^\circ = d_2$ otherwise.

$$\begin{aligned}&\mathrm{loca}(\mathrm{Val}(T(d_1),i_1),\Xi',p_1) \cap \mathrm{loca}(\mathrm{Val}(T(d_2),i_2),\Xi',p_2)\\ &\quad= \mathrm{loca}(\mathrm{Val}(T(d_1{}^\circ),i_1{}^\circ),\Xi,p_1{}^\circ) \cap \mathrm{loca}(\mathrm{Val}(T(d_2{}^\circ),i_2{}^\circ),\Xi,p_2{}^\circ) \neq \emptyset\end{aligned}$$

implies

$$(\mathrm{Val}^\#(T^\#(d_1{}^\circ),i_1{}^\circ).p_1{}^\circ \sim \mathrm{Val}^\#(T^\#(d_1{}^\circ),i_2{}^\circ).p_2{}^\circ) \in R(d)$$

by induction hypothesis. In $R(d{+}1)$, we have the alias $\mathrm{Val}^\#(T^\#(d_1{}^\circ),i_1{}^\circ).p_1{}^\circ \sim \mathrm{Val}^\#(T^\#(d_1),$ $i_1).p_1$ and similarly for i_2, either because $\mathrm{Val}^\#(T^\#(d_1{}^\circ),i_1{}^\circ).p_1{}^\circ = \mathrm{Val}^\#(T^\#(d_1),i_1).p_1$ or because the alias $\mathrm{Val}^\#(T^\#(d),i_1{}^\circ).p_1{}^\circ \sim \mathrm{Val}^\#(T^\#(d+1),i_1).p_1$ is added at date d. Then $(\mathrm{Val}^\#(T^\#(d_1),i_1).p_1 \sim \mathrm{Val}^\#(T^\#(d_2),i_2).p_2) \in R(d+1)$ which proves the correctness.

• **Case** $v_3.f = v_4$. Let $o_3 = \mathrm{Val}(\Xi,v_3)$, $o_4 = \mathrm{Val}(\Xi,v_4)$, $N_3 = \mathrm{Val}^\#(T^\#(d),v_3)$, $N_4 = \mathrm{Val}^\#(T^\#(d),v_4)$.

Assume that $\mathrm{loca}(o_1,\Xi',p_1) \cap \mathrm{loca}(o_2,\Xi',p_2) \neq \emptyset$. If $\mathrm{loca}(o_1,\Xi,p_1) \cap \mathrm{loca}(o_2,\Xi,p_2) \neq \emptyset$, the correctness at date d implies correctness at date $d+1$. Otherwise, at least one of $\mathrm{loca}(o_1,\Xi,p_1)$ or $\mathrm{loca}(o_2,\Xi,p_2)$ is changed when we replace Ξ by Ξ'. For example, assume that $\mathrm{loca}(o_1,\Xi,p_1) \neq \mathrm{loca}(o_1,\Xi',p_1)$ and $\mathrm{loca}(o_2,\Xi,p_2) = \mathrm{loca}(o_2,\Xi',p_2)$ (the other cases when $\mathrm{loca}(o_2,\Xi,p_2) \neq \mathrm{loca}(o_2,\Xi',p_2)$ or both are different are similar). In this case, the computation of $\mathrm{loca}(o_1,\Xi',p_1)$ uses the value of $H(o_3)(f)$, therefore $p_1 = q_1.f.q_2.f....f.q_n$, with $o_3 \in \mathrm{loca}(o_1,\Xi',q_1.f....q_i)$ $(i < n)$, and all uses of $H(o_3)(f)$ have been marked above by an f in the path, that is $\mathrm{loca}(o_4,\Xi',q_i) = \mathrm{loca}(o_4,\Xi,q_i)$ $(i > 1)$, $\mathrm{loca}(o_1,\Xi',q_1) = \mathrm{loca}(o_1,\Xi,q_1)$. Since $o_3 \in \mathrm{loca}(o_1,\Xi',q_1) = \mathrm{loca}(o_1,\Xi,q_1)$, by correctness at date d, $(N_1.q_1 \sim N_3) \in R(d)$. If $1 < i < n$, $o_3 \in \mathrm{loca}(o_4,\Xi',q_i) = \mathrm{loca}(o_4,\Xi,q_i)$, therefore $(N_4.q_i \sim N_3) \in R(d)$. $\mathrm{loca}(o_4,\Xi,q_n) \cap \mathrm{loca}(o_2,\Xi,p_2) \neq \emptyset$ therefore $(N_4.q_n \sim N_2.p_2) \in R(d)$. Moreover, we have added $N_3.f \sim N_4$, therefore $(N_3.f \sim N_4) \in R(d+1)$. By transitivity and right regularity of $R(d+1)$, $(N_1.q_1.f.q_2.f....f.q_n \sim N_2.p_2) \in R(d+1)$ hence the result.

• **Case** $C.f = v_3$. $\mathrm{loca}(\mathrm{Val}(T(d_1),i_1),\Xi,p_1) \neq \mathrm{loca}(\mathrm{Val}(T(d_1),i_1),\Xi',p_1)$ only if $i_1 = \mathrm{static}$ and $p_1 = f.p'$. In this case, $\mathrm{loca}(\mathrm{Val}(T(d_1),i_1),\Xi',p_1) = \mathrm{loca}(\mathrm{Val}(T(d),v_3),\Xi,p')$. Let us define $i_1{}^\circ = v_3$, $p_1{}^\circ = p'$, $d_1{}^\circ = d$ if $i_1 = \mathrm{static}$ and $p_1 = f.p'$. Otherwise, $i_1{}^\circ = i_1$, $p_1{}^\circ = p_1$, $d_1{}^\circ = d_1$. Then $\mathrm{loca}(\mathrm{Val}(T(d_1),i_1),\Xi',p_1) = \mathrm{loca}(\mathrm{Val}(T(d_1{}^\circ),i_1{}^\circ),\Xi,p_1{}^\circ)$. We define $i_2{}^\circ, d_2{}^\circ$ and $p_2{}^\circ$ similarly. Then

$$\begin{aligned}&\mathrm{loca}(\mathrm{Val}(T(d_1),i_1),\Xi',p_1) \cap \mathrm{loca}(\mathrm{Val}(T(d_2),i_2),\Xi',p_2)\\ &\quad= \mathrm{loca}(\mathrm{Val}(T(d_1{}^\circ),i_1{}^\circ),\Xi,p_1{}^\circ) \cap \mathrm{loca}(\mathrm{Val}(T(d_2{}^\circ),i_2{}^\circ),\Xi,p_2{}^\circ) \neq \emptyset\end{aligned}$$

implies

$$(\mathrm{Val}^\#(T^\#(d_1{}^\circ), i_1{}^\circ).p_1{}^\circ \sim \mathrm{Val}^\#(T^\#(d_2{}^\circ), i_2{}^\circ).p_2{}^\circ) \in R(d)$$

by induction hypothesis. We have added the alias static.$f \sim \mathrm{Val}^\#(T^\#(d), v_3)$ at date d, therefore $(\mathrm{Val}^\#(T^\#(d_1), i_1).p_1 \sim \mathrm{Val}^\#(T^\#(d_1{}^\circ), i_1{}^\circ).p_1{}^\circ) \in R(d+1)$. We have a similar property for index 2 instead of 1, and therefore $(\mathrm{Val}^\#(T^\#(d_1), i_1).p_1 \sim \mathrm{Val}^\#(T^\#(d_2), i_2).p_2) \in R(d+1)$ which proves the correctness.

- **Case** $v_3[v_5] = v_4$**.** similar to $v_3.f = v_4$.

- **Case** $v =$ `null`**.** Two cases have to be distinguished. First case: i_1 and i_2 do not correspond to the variable v at date $d+1$, i.e. we do not have $i_1 = v$ and $d_1 = d+1$, or $i_2 = v$ and $d_2 = d+1$. In this case, $\mathrm{loca}(\mathrm{Val}(T(d_1), i_1), \Xi', p_1) = \mathrm{loca}(\mathrm{Val}(T(d_1), i_1), \Xi, p_1)$ and similarly for index 2 instead of 1. Then the correctness at date $d+1$ comes immediately from the correctness at date d.

  Second case: i_1 or i_2 correspond to the variable v at date $d+1$, for example i_1 does. Then $i_1 = v$, $d_1 = d+1$, and $\mathrm{loca}(\mathrm{Val}(T(d+1), v), \Xi', p_1) = \emptyset$ and the correctness is obvious (the intersection is always empty).

- **Cases** $v =$ `new` C**,** $v =$ `new` $t[w]$**.** Two cases have to be distinguished. First case: i_1 and i_2 do not correspond to the variable v at date $d+1$. Then the correctness at date $d+1$ comes immediately from the correctness at date d, as in the $v =$ `null` case.

  Second case: i_1 or i_2 correspond to the variable v at date $d+1$, for example i_1 does. Then $i_1 = v$, $d_1 = d+1$. The location of the created object is new, therefore $\mathrm{loca}(\mathrm{Val}(T(d+1), v), \Xi', p_1) \cap \mathrm{loca}(\mathrm{Val}(T(d_2), i_2), \Xi', p_2) \neq \emptyset$ implies $d_2 = d+1$, $i_2 = v$ and $p_1 = p_2 = \epsilon$. Let $N = \mathrm{Val}^\#(T^\#(d+1), v)$. We have the alias $(N.\epsilon \sim N.\epsilon) \in R(d+1)$ by reflexivity of $R(d+1)$. This yields the correctness.

$\square$

Theorem 7.3 (Stack allocation) *Consider one execution of method m, in execution trace T, and assume that this execution of m terminates at date r.*

Consider a `new` *in method m, allocating object o. Let Res be the object name associated with the result of m, P_l the object names associated with the parameters of m (the names of these objects in the caller of m), N the object name of o. Let $Esc = \{\mathrm{static}.p' | p' \in Path\} \cup \{Res.p' | p' \in Path\} \cup \cup_l \{P_l.p' | p' \in Path\}$.*

If $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, o can be stack allocated in m in the considered execution trace T and in the execution of m that ends at date r.

Proof: The above correctness theorem, combined with $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, proves that, at the end of method m, the object o is not reachable from the parameters, the result of m or from static fields. So it is in fact not reachable, and can be stack allocated. $\square$

E.2 Correctness proof for analysis $\mathbb{E}$

Lemma 7.12 *Let m be a method, $v \in Var$ a local variable of m whose definition site is reachable (i.e. may be executed). Let $c, c_0, ..., c_j \in Ctx_{\mathbb{E}}$ such that $\forall i \in [\![0, j]\!], c_i \supseteq \mathbb{E}(m)(p_i)(c,$*

$c_0, ..., c_j)$ *where* $p_0, ..., p_j$ *are the parameters of* m. *Then*

$$\mathbb{E}_S(m)(v)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j).$$

Proof: We consider the control-flow graph G of m, that is the graph whose nodes are instructions, and which contains an edge $i_1 \to i_2$ if and only if i_2 may be executed immediately after i_1. For a method call $w = v_0.m'(v_1, ..., v_n)$, we have an edge from the instruction before the call towards the beginning of all methods that may be called, and an edge from the return of these methods to the call instruction $w = v_0.m'(v_1, ..., v_n)$.

We prove the following property by induction on k: for all variables v whose definition site can be reached from the starting point of m by a path of at most k edges in the control-flow graph, $\mathbb{E}_S(m)(v)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

This is done by cases on the definition of v.

- **Parameters** $v = p_i$. $\mathbb{E}_S(m)(p_i)(c, c_0, ..., c_j) = c_j \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ by hypothesis.

- **Case** $v = v_2$. v_2 must have been defined before v, therefore its definition site can be reached by a path of less than k edges and by induction hypothesis, $\mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j)$. Then $\mathbb{E}_S(m)(v)(c, c_0, ..., c_j) = \mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

- **Case** $v = \phi(v_2, v_3)$. At least v_2 or v_3 must have been defined before v, assume v_2 has been defined before v. Then its definition site can be reached by a path of less than k edges and by induction hypothesis, $\mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j)$. Then $\mathbb{E}_S(m)(v)(c, c_0, ..., c_j) \supseteq \mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

- **Case** $v = v_2.f$. v_2 must have been defined before v, therefore its definition site can be reached by a path of less than k edges and by induction hypothesis, $\mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j)$. Then $\mathbb{E}_S(m)(v)(c, c_0, ..., c_j) = f^{-1}.\mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \supseteq f^{-1}.\mathbb{E}(m)(v_2)(c, c_0, ..., c_j) \supseteq f^{-1}.f.\mathbb{E}(m)(v)(c, c_0, ..., c_j) = \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

- **Case** $v = C.f$. $\mathbb{E}_S(m)(v)(c, c_0, ..., c_j) = \mathit{Path} \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

- **Case** $v = v_2[v_3]$. Similar to $v = v_2.f$ (replace f with $\mathbb{N}$).

- **Case** $v =$ `new` C/`new` $t[w]$/`null`. $\mathbb{E}_S(m)(v) = \mathbb{E}(m)(v)$ by the equation defining $\mathbb{E}_S$.

- **Case** $v = (t)v_2$. Idem $v = v_2$.

- **Case** $v = v_0.m'(v_1, ..., v_n)$. v_i must be defined before v, therefore its definition site can be reached by a path of less than k edges and by induction hypothesis, $c'_i = \mathbb{E}_S(m)(v_i)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_i)(c, c_0, ..., c_j)$. Therefore $c'_i \supseteq \rho'(m')(i)(\mathbb{E}(m)(v)(c, c_0, ..., c_j), \mathbb{E}_S(m)(v_0)(c, c_0, ..., c_j), ..., \mathbb{E}_S(m)(v_n)(c, c_0, ..., c_j)) = \rho'(m')(i)(c', c'_0, ..., c'_n)$ where $c' = \mathbb{E}(m)(v)(c, c_0, ..., c_j)$. Moreover, $\rho'(m')(i) \geq \rho(m')(i) \geq \mathbb{E}(m')(p'_i)$ where $p'_0, ..., p'_n$ are the parameters of m'. Hence the c'_i satisfy the condition: $\forall i \in [\![0, n]\!], c'_i \supseteq \mathbb{E}(m')(p'_i)(c', c'_0, ..., c'_n)$.

On the other hand, $\rho'_{\mathrm{S}}(m') \geq \rho_{\mathrm{S}}(m') \geq \mathbb{E}_{\mathrm{S}}(m')(w)$ where w is the returned variable. The variable w must have been defined before v, hence its definition site can be reached by a path of less than k edges (in m') and by induction hypothesis, $\mathbb{E}_{\mathrm{S}}(m')(w)(c', c'_0, ..., c'_n) \supseteq \mathbb{E}(m')(w)(c', c'_0, ..., c'_n)$. Therefore $\mathbb{E}_{\mathrm{S}}(m)(v)(c, c_0, ..., c_j) = \rho'_{\mathrm{S}}(m')(c', c'_0, ..., c'_n) \supseteq \mathbb{E}(m')(w)(c', c'_0, ..., c'_n) \supseteq c' = \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

□

Theorem 7.13 (Correctness) $\mathbb{E}$ *and* $\mathbb{E}_{\mathrm{S}}$ *are correct:* $\mathrm{corr}_{\mathbb{E}\mathrm{S}}$.

Proof: We show the following properties:

- $\forall d, \mathrm{corr}_{\mathrm{S}}(0, d), \mathrm{corr}_{\mathbb{E}}(0, d)$. Obvious, since $R^0(r/d/s) = \emptyset$.

- $(\forall d, \mathrm{corr}_{\mathrm{S}}(k, d) \wedge \mathrm{corr}_{\mathbb{E}}(k, d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathbb{E}}(k+1, d))$. By backward induction on d. Let $d' = d + 1$.

Case `return` v. Let $Res = \mathrm{Val}^{\#}(T^{\#}(d), v)$. We have $\mathrm{escap}^{k+1}(r/d/s)(Res)(c, c_0, ..., c_j) = c$. For names $N \neq Res$, $\mathrm{escap}^{k+1}(r/d/s)(N)(c, c_0, ..., c_j) = \emptyset$ (the first alias of the chain $N.p \sim ... \sim \pi$ considered in the definition of $R^{k+1}(r/d/s)$ should be after d and before r, and this is impossible since $d = r$. Therefore, only the empty alias chain $\pi \sim \pi$ has to be considered. Since N is different from the parameter names P_l and from Res, it has no alias with elements of Esc).

Hence $\mathbb{E}^{k+1}(d)(v)(c, c_0, ..., c_j) = c = \mathit{first}_{\mathbb{E}}(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$, and for $v \neq v'$, $\mathbb{E}^{k+1}(d)(v')(c, c_0, ..., c_j) = \emptyset \subseteq \mathbb{E}(m)(v')(c, c_0, ..., c_j)$, which is the expected correctness.

Cases $v_1 = v_2.f$, $v_1 = v_2[v_3]$, $v_1 = v_2$, $v_1 = (t)v_2$, $v_1 = \phi(v_2, v_3)$. Let $N_1 = \mathrm{Val}^{\#}(T^{\#}(d'), v_1)$, $N_2 = \mathrm{Val}^{\#}(T^{\#}(d'), v_2)$. The created alias is $N_1 \sim N_2.\delta_2$ where $\delta_2 = f$ in the $v_1 = v_2.f$ case, $\delta_2 = \mathbb{N}$ in the $v_1 = v_2[v_3]$ case, $\delta_2 = \epsilon$ in the last three cases. In the $v_1 = \phi(v_2, v_3)$ case, we assume that v_2 has been defined after v_3, otherwise, simply swap v_2 and v_3.

For $N \neq N_1, N_2$, the name N has the same aliases in $R^{k+1}(r/d/s)$ and in $R^{k+1}(r/d'/s)$, therefore

$$\mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d'/s)(N). \tag{E.2}$$

For $N = N_2$, let

$$\begin{aligned} X &= \mathrm{escap}^{k+1}(r/d/s)(N_2)(c, c_0, ..., c_j) \\ &= \{p \mid \exists \pi \in Esc(c, c_0, ..., c_j), (N_2.p \sim \pi) \in R^{k+1}(r/d/s)\}. \end{aligned}$$

We consider the first alias of the chain $N_2.p \sim ... \sim \pi$ used to prove that $(N_2.p \sim \pi) \in R^{k+1}(r/d/s)$. Since this alias dates from after d, either it is $N_2.\delta_2.p' \sim N_1.p'$ or it dates from after d'. Also notice that N_1 is created at d'. Therefore all aliases of N_1 date from after d' except $N_1 \sim N_2.\delta_2$ but this alias cannot be used as second alias in this chain since this chain would then be $N_2.\delta_2.p' \sim N_1.p' \sim N_2.\delta_2.p' \sim ...$ contradicting the condition that the π_i must be pairwise distinct in the definition of $R^{k+1}(r/d/s)$. Therefore, if the first alias of the chain is $N_2.\delta_2.p' \sim N_1.p'$, then the second alias dates from after d'. Therefore

$$\begin{aligned} X &= \{\delta_2.p' | \exists \pi \in Esc(c, c_0, ..., c_j), (N_1.p' \sim \pi) \in R^k(r/d'/s)\} \\ &\quad \cup \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (N_2.p \sim \pi) \in R^{k+1}(r/d'/s)\} \\ X &= \delta_2.\mathrm{escap}^k(r/d'/s)(N_1)(c, c_0, ..., c_j) \cup \mathrm{escap}^{k+1}(r/d'/s)(N_2)(c, c_0, ..., c_j) \end{aligned}$$

Then

$$\mathrm{escap}^{k+1}(r/d/s)(N_2) = \delta_2.\mathrm{escap}^k(r/d'/s)(N_1) \sqcup \mathrm{escap}^{k+1}(r/d'/s)(N_2). \tag{E.3}$$

Let us assume that $(c, c_0, ..., c_j)$ satisfy the property:

$$\forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j). \tag{E.4}$$

For $v \neq v_1, v_2$, $\mathrm{Val}^\#(T^\#(d), v) = \mathrm{Val}^\#(T^\#(d'), v)$ and by (E.2), $\mathbb{E}^{k+1}(d)(v) = \mathbb{E}^{k+1}(d')(v)$. The correctness at date d' then implies the correctness at date d.

For $v = v_2$, by (E.3), $\mathbb{E}^{k+1}(d)(v_2) = \delta_2.\mathbb{E}^k(d')(v_1) \sqcup \mathbb{E}^{k+1}(d')(v_2)$.

$$\begin{aligned} \mathbb{E}^{k+1}(d)(v_2)(c, c_0, ..., c_j) &= \delta_2.\mathbb{E}^k(d')(v_1)(c, c_0, ..., c_j) \sqcup \mathbb{E}^{k+1}(d')(v_2)(c, c_0, ..., c_j) \\ &\subseteq \delta_2.\mathbb{E}(m)(v_1)(c, c_0, ..., c_j) \sqcup \mathbb{E}(m)(v_2)(c, c_0, ..., c_j) \end{aligned}$$

by correctness at date d'. We have the equation $\mathbb{E}(m)(v_2) \geq \delta_2.\mathbb{E}(m)(v_1)$, hence $\mathbb{E}^{k+1}(d)(v_2)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j)$ which is the expected correctness.

For $v = v_1$, $N = \mathrm{Val}^\#(T^\#(d), v_1)$ disappears at date d' (since each object name is only present in one variable at a time, and N is overwritten by the new name N_1 at d'). Then $\mathrm{escap}^{k+1}(d)(v_1) = \mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d'/s)(N) = \emptyset$ since N does not exist any more at d', no alias of N can therefore be created after d'. Then any value of $\mathbb{E}(m)(v_1)$ is correct at date d.

Cases $v_1.f = v_2$ and $v_1[v_3] = v_2$. Let $N_1 = \mathrm{Val}^\#(T^\#(d), v_1)$, $N_2 = \mathrm{Val}^\#(T^\#(d), v_2)$. The created alias is $N_1.\delta_1 \sim N_2$ where $\delta_1 = f$ in the $v_1.f = v_2$ case, $\delta_1 = \mathbb{N}$ in the $v_1[v_3] = v_2$ case. For $N \neq N_1, N_2$, the name N has the same aliases in $R^{k+1}(r/d'/s)$ as in $R^{k+1}(r/d/s)$, then

$$\mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d'/s)(N). \tag{E.5}$$

For $N = N_1$, let

$$\begin{aligned} X &= \mathrm{escap}^{k+1}(r/d/s)(N_1)(c, c_0, ..., c_j) \\ &= \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^{k+1}(r/d/s)\}. \end{aligned}$$

Since the first alias of the chain $N_1.p \sim ... \sim \pi$ used to prove that $(N_1.p \sim \pi) \in R^{k+1}(r/d/s)$ dates from after d, either it is $N_1.\delta_1 \sim N_2$, or it dates from after d'

$$\begin{aligned} X &\subseteq \{\delta_1.p' | \exists \pi \in Esc(c, c_0, ..., c_j), (N_2.p' \sim \pi) \in R^k(r/s/s)\} \\ &\quad \cup \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^{k+1}(r/d'/s)\} \\ X &\subseteq \delta_1.\mathrm{escap}^k(r/s/s)(N_2)(c, c_0, ..., c_j) \cup \mathrm{escap}^{k+1}(r/d'/s)(N_1)(c, c_0, ..., c_j) \end{aligned}$$

Hence

$$\text{escap}^{k+1}(r/d/s)(N_1) \leq \delta_1.\text{escap}^k(r/s/s)(N_2) \sqcup \text{escap}^{k+1}(r/d'/s)(N_1). \tag{E.6}$$

We now assume (E.4).

For $v \neq v_1, v_2$, $\text{Val}^\#(T^\#(d), v) = \text{Val}^\#(T^\#(d'), v)$ and by (E.5), $\mathbb{E}^{k+1}(d)(v) = \mathbb{E}^{k+1}(d')(v)$. The correctness at date d' then implies the correctness at date d.

For $v = v_1$, by (E.6), $\mathbb{E}^{k+1}(d)(v_1) \leq \delta_1.\mathbb{E}^k_S(d)(v_2) \sqcup \mathbb{E}^{k+1}(d')(v_1)$. By correctness of $\mathbb{E}_S$ at date d and iteration k and by correctness of $\mathbb{E}$ at date d' and iteration $k+1$,

$$\begin{aligned}\mathbb{E}^{k+1}(d)(v_1)(c, c_0, ..., c_j) &\subseteq \delta_1.\mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \cup \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)\\ &\subseteq \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)\end{aligned}$$

since we have the equation $\mathbb{E}(m)(v_1) \geq \delta_1.\mathbb{E}_S(m)(v_2)$. This is the expected correctness.

The case $v = v_2$, $N = N_2$ is similar to $v = v_1$, $N = N_1$ (just swap N_1 and N_2, v_1 and v_2 and replace δ_1 with δ_1^{-1}).

Cases $v_1 = C.f$**,** $v_1 =$ `new` C**,** $v_1 =$ `new` $t[v_2]$**,** $v_1 =$ `null`**.** For all names N existing at date d, the aliases of N are the same at dates d and d' (since $N \neq \text{Val}^\#(T^\#(d'), v_1)$), therefore for $N \neq \text{Val}^\#(T^\#(d), v_1)$, $\text{escap}^{k+1}(r/d/s)(N) = \text{escap}^{k+1}(r/d'/s)(N)$. For $v \neq v_1$, $\mathbb{E}^{k+1}(d)(v) = \mathbb{E}^{k+1}(d')(v)$. Hence the correctness at date d comes from the correctness at date d'.

For $v = v_1$, $N_1 = \text{Val}^\#(T^\#(d), v_1)$ disappears at date d', hence no alias of this name is created after d: $\text{escap}^{k+1}(r/d/s)(N_1) = \emptyset$, and any escape value $\mathbb{E}(m)(v_1)$ is correct for v_1 at date d.

Case $C.f = v_1$**.** Let $N_1 = \text{Val}^\#(T^\#(d), v_1)$. If $N \neq N_1$, the aliases of N at dates d and d' are the same, and

$$\text{escap}^{k+1}(r/d/s)(N) = \text{escap}^{k+1}(r/d''/s)(N). \tag{E.7}$$

Then the correctness at date d'' implies the correctness at date d for the variables v different from v_1.

The alias $N_1 \sim \text{static}.f$ is created at date d, then $\text{escap}^{k+1}(r/d/s)(N_1)(c, c_0, ..., c_j) \subseteq$ *Path* (equality for $k \geq 1$) and $\mathbb{E}^{k+1}(d)(v_1)(c, c_0, ..., c_j) \subseteq$ *Path* (again equality for $k \geq 1$). $\mathbb{E}(m)(v_1)(c, c_0, ..., c_j) = \mathit{Path}$ is then correct and also the least possible value.

Case `if` (...) `goto` a**.** No alias is created at date d', therefore $\text{escap}^k(r/d/s)(N) = \text{escap}^k(r/d'/s)(N)$. Moreover, $\text{Val}^\#(T^\#(d), v) = \text{Val}^\#(T^\#(d'), v)$, therefore the correctness at date d comes immediately from the correctness at date d'.

Case $w = v_0.m'(v_1, ..., v_n)$**.** Let d be the date just before the method call, $d' = d+1$ just after, and d'' after the corresponding `return`. Let $P'_l = \text{Val}^\#(T^\#(d), v_l)$ be the object names of the parameters of m', $P''_l = \text{Val}^\#(T^\#(d'), p'_l)$ be the object names of the parameters of m' inside m'. During the call, the alias $P'_l \sim P''_l$ is created. Let Res' be the name of the result of m'. Let $Esc'(c', c'_0, ..., c'_n) = \{\text{static}.p' | p' \in \mathit{Path}\} \cup \{Res'.p' | p' \in c\} \cup \cup_l \{P'_l.p' | p' \in c_l\}$ be the Esc associated with m'.

If N is not one of the P'_l, N is not accessible during the evaluation of m', hence no alias of N is created between d and d'', and

$$\text{escap}^{k+1}(r/d/s)(N) = \text{escap}^{k+1}(r/d''/s)(N). \tag{E.8}$$

Then the correctness at date d'' implies the correctness at date d for the variables v different from v_l.

On the other hand, let

$$\begin{aligned} X &= \text{escap}^{k+1}(r/d/s)(P'_l)(c, c_0, ..., c_j) \\ &= \{p | \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (P'_l.p \sim \pi) \in R^{k+1}(r/d/s)\}. \end{aligned}$$

Either the first alias of the chain is $P'_l \sim P''_l$ or it has been created after d'' (since P'_l is not accessible during the runtime of m'). In the first case, the alias chain can be decomposed into $P'_l.p \sim P''_l.p \sim ... \sim \pi' \sim ... \sim \pi$, where all aliases of the $P''_l.p \sim \pi'$ part have been created during the runtime of m' and the first alias of the chain $\pi' \sim \pi$ has been created outside of m' (or $\pi' = \pi$). Then $(P''_l.p \sim \pi') \in R^k(d'' - 1/d'/d)$. Let $c' = \text{escap}^k(r/d''/s)(\mathit{Res}')(c, c_0, ..., c_j)$, $c'_i = \text{escap}^k(r/s/s)(P'_i)(c, c_0, ..., c_j)$. Three cases have to be distinguished for $\pi' \sim \pi$:

- If $\pi' = \mathit{Res}'.p'$, the first alias of the chain $\pi' \sim \pi$ cannot have been created before the call to m' since Res' did not exist yet. Therefore, it has been created after this call, and $(\pi' \sim \pi) \in R^k(r/d''/s)$. Then $p' \in \text{escap}^k(r/d''/s)(\mathit{Res}')(c, c_0, ..., c_j) = c'$, and $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$.
- If $\pi' = P'_i.p'$, $(\pi' \sim \pi) \in R^k(r/s/s)$. Then $p' \in \text{escap}^k(r/s/s)(P'_i)(c, c_0, ..., c_j) = c'_i$, and $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$.
- If $\pi' = \text{static}.p'$, $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$.

In all cases, $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$ and $(P''_l.p \sim \pi') \in R^k(d'' - 1/d'/d)$ therefore $p \in \text{escap}^k(d'' - 1/d'/d)(P''_l)(c', c'_0, ..., c'_n)$ and

$$\begin{aligned} &\text{escap}^{k+1}(r/d/s)(P'_l) \leq \text{escap}^k(d'' - 1/d'/d)(P''_l) \circ \\ &\quad (\text{escap}^k(r/d''/s)(\mathit{Res}'), \text{escap}^k(r/s/s)(P'_0), ..., \text{escap}^k(r/s/s)(P'_n)) \sqcup \text{escap}^{k+1}(r/d''/s)(P'_l). \end{aligned} \tag{E.9}$$

Then $\mathbb{E}^{k+1}(d)(v_l) \leq \mathbb{E}^k(d')(p'_l) \circ (\mathbb{E}^k(d'')(w), \mathbb{E}^k_S(d)(v_0), ..., \mathbb{E}^k_S(d)(v_n)) \sqcup \mathbb{E}^{k+1}(d'')(v_l)$. Let $c'' = \mathbb{E}(m)(w)(c, c_0, ..., c_j)$, $c''_i = \mathbb{E}_S(m)(v_i)(c, c_0, ..., c_j)$, and assume (E.4). The correctness at iteration k yields $\mathbb{E}^k(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(w)(c, c_0, ..., c_j) = c''$, $\mathbb{E}^k_S(d)(v_i)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m)(v_i)(c, c_0, ..., c_j) = c''_i$. Then

$$\mathbb{E}^{k+1}(d)(v_l)(c, c_0, ..., c_j) \subseteq \mathbb{E}^k(d')(p'_l)(c'', c''_0, ..., c''_n).$$

By correctness at iteration k, $\mathbb{E}^k(d')(p'_l)(c'', c''_0, ..., c''_n) \subseteq \mathbb{E}(m')(p'_l)(c'', c''_0, ..., c''_n)$ (since by Lemma 7.12, $c''_i = \mathbb{E}_S(m)(v_i)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_i)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m')(p'_i)(c'', c''_0, ..., c''_n)$). Therefore,

$$\mathbb{E}^{k+1}(d)(v_l)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m')(p'_l)(c'', c''_0, ..., c''_n).$$

Then using the equations $\mathbb{E}(m)(v_l) \geq \rho(m')(l) \circ (\mathbb{E}(m)(w), \mathbb{E}_S(m)(v_0), ..., \mathbb{E}_S(m)(v_n))$ and $\rho(m')(l) \geq \mathbb{E}(m')(p'_l)$, we get $\mathbb{E}^{k+1}(d)(v_l)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v_l)(c, c_0, ..., c_j)$, which is the required correctness.

- $(\forall d, \mathrm{corr}_{\mathbb{E}}(k,d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathrm{S}}(k,d))$. By forward induction on d. Let $d' = d+1$.

Entry of a method. Let $P'_l = \mathrm{Val}^\#(T^\#(s+1), p_l)$ be the names of the parameters of m, created during the call. The created alias is $P_l \sim P'_l$. Let

$$\begin{aligned} X &= \mathrm{escap}^k(r/s/s)(P'_l)(c, c_0, ..., c_j) \\ &= \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (P'_l.p \sim \pi) \in R^k(r/s/s)\}. \end{aligned}$$

Either the first alias of the chain used to prove $(P'_l.p \sim \pi) \in R^k(r/s/s)$ is $P'_l \sim P_l$, or it dates from after $s+1$. Moreover the only alias of P_l created between s and r is $P_l \sim P'_l$, and it cannot be used again in this chain since the π_i have to be pairwise distinct in the definition of $R^k(r/s/s)$. Then, if the first alias is $P'_l \sim P_l$, $\pi = P_l.p$. Hence,

$$\begin{aligned} X &= \{p | \exists P_l.p \in Esc(c, c_0, ..., c_j)\} \cup \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (P'_l.p \sim \pi) \in R^k(r/s+1/s)\} \\ &= c_l \cup \mathrm{escap}^k(r/s+1/s)(P'_l)(c, c_0, ..., c_j). \end{aligned}$$

Hence

$$\mathrm{escap}^k(r/s/s)(P'_l) = P_{\mathbb{E}}(p_l) \sqcup \mathrm{escap}^k(r/s+1/s)(P'_l). \tag{E.10}$$

Then $\mathbb{E}^k_{\mathrm{S}}(s+1)(p_l) = P_{\mathbb{E}}(p_l) \sqcup \mathbb{E}^k(s+1)(p_l)$. We have by hypothesis, $c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j) \supseteq \mathbb{E}^k(s+1)(p_l)(c, c_0, ..., c_j)$ by correctness of $\mathbb{E}$. Therefore $\mathbb{E}^k_{\mathrm{S}}(s+1)(p_l)(c, c_0, ..., c_j) \subseteq c_l = \mathbb{E}_{\mathrm{S}}(m)(p_l)(c, c_0, ..., c_j)$ which is the correctness.

Cases $v_1 = v_2.f$, $v_1 = v_2[v_3]$, $v_1 = v_2$, $v_1 = (t)v_2$, $v_1 = \phi(v_2, v_3)$. Let $N_1 = \mathrm{Val}^\#(T^\#(d'), v_1)$, $N_2 = \mathrm{Val}^\#(T^\#(d'), v_2)$. The created alias is $N_1 \sim N_2.\delta_2$ where $\delta_2 = f$ in the $v_1 = v_2.f$ case, $\delta_2 = \mathbb{N}$ in the $v_1 = v_2[v_3]$ case, $\delta_2 = \epsilon$ in the last three cases. In the $v_1 = \phi(v_2, v_3)$ case, we assume that v_2 has been defined after v_3, otherwise, simply swap v_2 and v_3.

For $v \neq v_1$, the correctness comes immediately from the induction hypothesis on d, since $\mathbb{E}^k_{\mathrm{S}}(d')(v) = \mathbb{E}^k_{\mathrm{S}}(d)(v)$.

For $v = v_1$, let $N_1 = \mathrm{Val}^\#(T^\#(d'), v_1)$ be the new name, let $N_2 = \mathrm{Val}^\#(T^\#(d), v_2)$.

$$\begin{aligned} &\mathrm{escap}^k(r/s/s)(N_1)(c, c_0, ..., c_j) \\ &\quad = \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^k(r/s/s)\} \\ &\quad \subseteq \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (N_2.\delta_2.p \sim \pi) \in R^{k-1}(r/s/s)\} \\ &\quad\quad \cup \{p | \exists \pi \in Esc(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^k(r/d'/s)\} \end{aligned}$$

since if the first alias of the chain used to show that $(N_1.p \sim \pi) \in R^k(r/d'/s)$ dates from before d', then we use alias $N_1 \sim N_2.\delta_2$ followed by $N_2.\delta_2.p \sim \pi$. Then

$$\mathrm{escap}^k(r/s/s)(N_1) \leq \delta_2^{-1}.\mathrm{escap}^{k-1}(r/s/s)(N_2) \sqcup \mathrm{escap}^k(r/d'/s)(N_1). \tag{E.11}$$

Therefore

$$\mathbb{E}^k_{\mathrm{S}}(d')(v_1) \leq \delta_2^{-1}.\mathbb{E}^{k-1}_{\mathrm{S}}(d)(v_2) \sqcup \mathbb{E}^k(d')(v_1).$$

Therefore, assuming (E.4), by correctness of $\mathbb{E}_S$ at date d and iteration $k-1$ and correctness of $\mathbb{E}$ at date d' and iteration k,

$$\begin{aligned}\mathbb{E}_S^k(d)(v_1)(c, c_0, ..., c_j) &\subseteq \delta_2^{-1}.\mathbb{E}_S(m)(v_2)(c, c_0, ..., c_j) \cup \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)\\ &\subseteq \mathbb{E}_S(m)(v_1)(c, c_0, ..., c_j) \cup \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)\end{aligned}$$

since $\mathbb{E}_S(m)(v_1) = \delta_2^{-1}.\mathbb{E}_S(m)(v_2)$. By Lemma 7.12, $\mathbb{E}_S(m)(v_1)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)$. This gives the correctness.

Case $v_1 = C.f$**.** For $v \neq v_1$, the correctness comes immediately from the induction hypothesis on d, since $\mathbb{E}_S^k(d')(v) = \mathbb{E}_S^k(d)(v)$.

We have $\mathbb{E}_S^k(d')(v_1)(c, c_0, ..., c_j) = \text{escap}^k(r/s/s)(\text{Val}^\#(T^\#(d'), v_1))(c, c_0, ..., c_j) \subseteq \mathit{Path}$ (equality for $k > 1$). $\mathbb{E}_S(m)(v_1)(c, c_0, ..., c_j) = \mathit{Path}$ is correct (and also the best possible value).

Cases $v_1.f = v_2$**,** $v_1[v_2] = v_3$**,** $C.f = v$**,** `if (...) goto` a**.** Obvious by induction hypothesis on d ($\text{Val}^\#(T^\#(d'), v) = \text{Val}^\#(T^\#(d), v)$ and $R^k(r/s/s)$ does not depend on d).

Cases $v_1 =$ `new` C**,** $v_1 =$ `new` $t[v_2]$**,** $v_1 =$ `null`**.** For $v \neq v_1$, we have $\text{Val}^\#(T^\#(d), v) = \text{Val}^\#(T^\#(d'), v)$, therefore the correctness at date d' comes immediately from the correctness at date d. For $v = v_1$, let $N_1 = \text{Val}^\#(T^\#(d'), v_1)$. Since N_1 is new at date d', all aliases of N_1 are created after d', therefore:

$$\text{escap}^k(r/s/s)(N_1) = \text{escap}^k(r/d'/s)(N_1). \tag{E.12}$$

Then $\mathbb{E}_S^k(d')(v_1) = \mathbb{E}^k(d')(v_1)$. By correctness of $\mathbb{E}$, assuming (E.4), $\mathbb{E}^k(d')(v_1)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)$. By the equation $\mathbb{E}_S(m)(v_1) \geq \mathbb{E}(m)(v_1)$, we have $\mathbb{E}_S^k(d')(v_1)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m)(v_1)(c, c_0, ..., c_j)$ which is the required correctness.

Case `return` v**.** Assume that the corresponding call is $w = v_0.m'(v_1, ..., v_n)$. Let d be the date just before the method call, $d' = d+1$ just after, and d'' after the corresponding `return`. Let $P'_l = \text{Val}^\#(T^\#(d), v_l)$ be the names of the parameters of m'. Let Res' be the name of the result of m'. Let $\mathit{Esc}'(c', c'_0, ..., c'_n) = \{\text{static}.p' | p' \in \mathit{Path}\} \cup \{\mathit{Res}'.p' | p' \in c\} \cup \cup_l \{P'_l.p' | p' \in c_l\}$ be the Esc associated with m'.

For variables v different from w, $\text{Val}^\#(T^\#(d''), v) = \text{Val}^\#(T^\#(d), v)$ and the correctness at date d'' comes immediately from the correctness at date d.

For w, let

$$\begin{aligned}X &= \text{escap}^k(r/s/s)(\mathit{Res}')(c, c_0, ..., c_j)\\ &= \{p | \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (\mathit{Res}'.p \sim \pi) \in R^{k+1}(r/s/s)\}.\end{aligned}$$

The first alias of the chain used to prove $(\mathit{Res}'.p \sim \pi) \in R^{k+1}(r/s/s)$ has either been created after d'' or between d and d''. In the latter case, the alias chain is of the form $\mathit{Res}'.p \sim ... \sim \pi' \sim ... \sim \pi$ where all aliases of the $\mathit{Res}'.p \sim \pi'$ part have been created during the runtime of m' and the first alias of the chain $\pi' \sim \pi$ has been created outside of m' (or $\pi' = \pi$). Then $(\mathit{Res}'.p \sim \pi') \in R^k(d'' - 1/d/d)$. Let $c' = \text{escap}^{k-1}(r/d''/s)(\mathit{Res}')(c, c_0, ..., c_j)$, $c'_i = \text{escap}^{k-1}(r/s/s)(P'_i)(c, c_0, ..., c_j)$. We prove like in the case of a method call in the backward part that $\pi' \in Esc'(c', c'_0, ..., c'_n)$. Therefore $p \in \text{escap}^k(d'' - 1/d/d)(Res')(c', c'_0, ..., c'_n)$ and

$$\begin{aligned}&\text{escap}^k(r/s/s)(Res') \leq \text{escap}^k(d'' - 1/d/d)(Res') \circ \\ &\quad (\text{escap}^k(r/d''/s)(Res'), \text{escap}^k(r/s/s)(P'_0), ..., \text{escap}^k(r/s/s)(P'_n)) \sqcup \text{escap}^k(r/d''/s)(Res').\end{aligned} \tag{E.13}$$

Therefore,

$$\mathbb{E}^k_S(d'')(w) \leq \mathbb{E}^k_S(d'' - 1)(v) \circ (\mathbb{E}^k(d'')(w), \mathbb{E}^k_S(d)(v_0), ..., \mathbb{E}^k_S(d)(v_n)) \sqcup \mathbb{E}^k(d'')(w).$$

Let $c'' = \mathbb{E}(m)(w)(c, c_0, ..., c_j)$, $c''_i = \mathbb{E}_S(m)(v_i)(c, c_0, ..., c_j)$ and assume (E.4). Using the correctness of $\mathbb{E}_S$ at date d and of $\mathbb{E}$, we have $\mathbb{E}^k(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(w)(c, c_0, ..., c_j) = c''$, $\mathbb{E}^k_S(d)(v_i)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m)(v_i)(c, c_0, ..., c_j) = c''_i$. Therefore

$$\mathbb{E}^k_S(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}^k_S(d'' - 1)(v)(c'', c''_0, ..., c''_n).$$

By correctness of $\mathbb{E}_S$ at date $d'' - 1$, $\mathbb{E}^k_S(d'' - 1)(v)(c'', c''_0, ..., c''_n) \subseteq \mathbb{E}_S(m')(v)(c'', c''_0, ..., c''_n)$ (using Lemma 7.12 as in the method call case of the backward part). Then

$$\mathbb{E}^k_S(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m')(v)(c'', c''_0, ..., c''_n).$$

Then, using the equations $\mathbb{E}_S(m)(w) = \rho'_S(m') \circ (\mathbb{E}(m)(w), \mathbb{E}_S(m)(v_0), ..., \mathbb{E}_S(m)(v_n))$ and $\rho'_S(m') \geq \mathbb{E}_S(m')(v)$, we have $\mathbb{E}^k_S(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m)(w)(c, c_0, ..., c_j)$ which is the correctness.

• $\forall k, \text{corr}_{\mathbb{E}}(k, d) \Rightarrow \text{corr}_{\mathbb{E}}(\infty, d)$, $\forall k, \text{corr}_S(k, d) \Rightarrow \text{corr}_S(\infty, d)$. This is obvious since we have $\cup_{k \in \mathbb{N}} R^k(r/d/s) = R^\infty(r/d/s)$.

The result follows by induction on k. □

E.3 Inlining

Theorem 8.2 (Inlining) *Assume that method m_0 calls m_1 at pc_1, which calls m_2 at pc_2, ..., which calls m_n at pc_n, and that in method m_n, at pc, an object o of type τ is allocated, and stored in variable v. Then, if $\phi(\top[\tau_{-1}], ..., \top[\tau_j]) < \top[\tau]$ where*

$$\mathbb{L}_1(m_n)(v) \circ \mathbb{L}_{1C}(pc_n) \circ ... \circ \mathbb{L}_{1C}(pc_1) = (\phi, (\tau_{-1}, ..., \tau_j), \tau)$$

the object o can be stack allocated if we inline m_1, ..., m_n in m_0.

We prove the corresponding result for analysis $\mathbb{E}$. The result for analysis $\mathbb{L}_1$ can then easily be inferred using the properties of the Galois connection and the concretization.

Assume that we call a method m' with $j + 1$ parameters at pc in method m: $r = p_0.m'(p_1, ..., p_j)$. We define:

$$\mathbb{E}_C(pc) = (\mathbb{E}(m)(r), \mathbb{E}_S(m)(p_0), ..., \mathbb{E}_S(m)(p_j))$$

We prove that, if $\epsilon \notin \mathbb{E}(m_n)(v) \circ \mathbb{E}_C(pc_n) \circ ... \circ \mathbb{E}_C(pc_1)(Path, ..., Path)$ then the object o can be stack allocated if we inline m_1, ..., m_n in m_0.

There are essentially two proofs of this result. The first proof is by simulating the analysis of the method m_0 in which m_1, ..., m_n have been inlined, and applying Theorem 7.14.

The second proof shows that the object o is not reachable any more at the end of m (even without inlining m_1, ..., m_n). It is then an obvious consequence that o can be stack allocated if the inlining is performed. The advantage of this second proof is that it could also be applied to preallocation, and that most of the proof can be applied to synchronization elimination (in the next section).

We detail this second proof. First, the correctness theorem of our alias instrumentation is extended:

Theorem E.9 *We consider one execution trace T, and one execution of the analyzed method m_0 in this execution trace. Assume that this execution of m_0 terminates at date r.*

Let Res be the object name associated with the result of m_0, and P_l the object names associated with the parameters of method m_0 (the names of these objects in the caller of m_0). Let $Esc = \{\text{static}.p' | p' \in Path\} \cup \{Res.p' | p' \in Path\} \cup \cup_l \{P_l.p' | p' \in Path\}$.

Let us consider the `new` *at address pc in method m_n. Let N be the name associated with the new object o created by this* `new`*. If $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, then o can be stack allocated, after inlining of calls to m_1, ..., m_n.*

Proof: Indeed, let Ξ be the state at the return from method m_0, *res* its result, par_i its parameters. By correctness of R, we have $o \notin \text{loca}(\text{static}, \Xi, p) \cup \text{loca}(res, \Xi, p) \cup \cup_i \text{loca}(par_i, \Xi, p)$ for all p, hence o is not reachable after the return from m_0, therefore it can be stack allocated in m_0. □

If m is one of the methods $m_0, ..., m_n$ of the call chain, let *Res* be the object name associated with the result of the m, and P_l the object names associated with the parameters of m (in the caller of m). We define $Esc_m(c, c_0, ..., c_j) = \{\text{static}.p' | p' \in Path\} \cup \{Res.p' | p' \in c\} \cup \cup_l \{P_l.p' | p' \in c_l\}$.

Then we prove the following scope extension lemma:

Lemma E.10 *Let N_1 be the name of a parameter or of the result of a method m' called by m at pc. r is the date at the end of m, r' at the end of m', s is the date of the call to m.*

Also assume that if N_1 is the name of a parameter of m', $(N_1.p_1 \sim \pi) \in R(r)$, and if N_1 is the name of the result of m', $(N_1.p_1 \sim \pi) \in R(r/r'/s)$. Also assume that $(c_0, ..., c_j) \geq (\rho(m)(0), ..., \rho(m)(j))(c, c_0, ..., c_j)$. If $\pi \in Esc_m(c, c_0, ..., c_j)$ then $N_1.p_1 \in Esc_{m'} \circ \mathbb{E}_C(pc)(c, c_0, ..., c_j)$.

Proof: Assume that the call is $w = v_0.m'(v_1, ..., v_n)$.

Two cases have to be distinguished: if N_1 is the name of the parameter v_k of m', by correctness of $\mathbb{E}_S$, $\{p | \exists \pi \in Esc_m(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^\infty(r/s/s)\} \subseteq \mathbb{E}_S(m)(v_k)(c, c_0, ..., c_j)$. Since N_1 has been created after the beginning of m, the aliases of N_1 date from after s, and are therefore the same in $R(r)$ et $R^\infty(r/s/s)$. Therefore $p_1 \in \mathbb{E}_S(m)(v_k)(c, c_0, ..., c_j)$, hence $N_1.p_1 \in Esc_{m'} \circ \mathbb{E}_C(pc)(c, c_0, ..., c_j)$.

If N_1 is the name of the result w of method m', by correctness of $\mathbb{E}$, $\{p | \exists \pi \in Esc_m(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^\infty(r/r'/s)\} \subseteq \mathbb{E}(m)(w)(c, c_0, ..., c_j))\}$. Therefore $p_1 \in \mathbb{E}(m)(w)(c, c_0, ..., c_j))$ and $N_1.p_1 \in Esc_{m'} \circ \mathbb{E}_C(pc)(c, c_0, ..., c_j)$. □

The theorem of correctness of inlining is then easy to prove:

Theorem E.11 (E.11 (Inlining)) *Assume that method m_0 calls m_1 at pc_1, which calls m_2 at pc_2, ..., which calls m_n at pc_n, and that in method m_n, at pc, an object o of type τ is allocated, and stored in variable v. Then, if*

$$\epsilon \notin \mathbb{E}(m_n)(v) \circ \mathbb{E}_{\mathrm{C}}(pc_n) \circ ... \circ \mathbb{E}_{\mathrm{C}}(pc_1)(Path, ..., Path),$$

the object o can be stack allocated if we inline m_1, ..., m_n in m_0.

Proof: Consider one execution trace T of the program. In this execution trace, consider one execution of the considered call chain: m_0 calls m_1, ..., m_n. Let s_k be the date at the beginning of the execution of m_k, r_k at the end of m_k. Let N' be the object name of an object allocated at pc, at date d, during this execution of m_n.

By contradiction, assume that $\exists \pi \in Esc, (N'.\epsilon \sim \pi) \in R(r_0)$. The idea is to cut the alias chain $N'.\epsilon \sim \pi$, to separate the aliases created in each method of the call chain. Aliases created in method m_i are taken into account by $\mathbb{E}_{\mathrm{C}}(pc_{i+1})$.

More precisely, we choose π_k in the chain $(N'.\epsilon \sim ... \sim \pi_n \sim ... \sim \pi_{n-1} \sim ... \sim \pi) \in R(r_0)$ such that all aliases of the chain $N'.\epsilon \sim ... \sim \pi_k$ have been created in the date interval $]s_k, r_k]$, and the first alias after π_k has been created outside of this interval.

Then $\pi_k = N_k.p_k$ and N_k is the name of a parameter or of the result of method m_k, since these names are the only ones known both inside and outside $]s_k, r_k]$. We have $(N'.\epsilon \sim \pi_k) \in R(r_k)$. Moreover, if N_{k+1} is the name of the result of m_{k+1}, $(\pi_{k+1} \sim \pi_k) \in R(r_k/r_{k+1}/s_k)$. Indeed, the name of the result has been created during the execution of m_{k+1}, hence it is not known before s_{k+1}. Its aliases not in $]s_{k+1}, r_{k+1}]$ have therefore been created after r_{k+1}. By hypothesis, $\pi \in Esc_{m_0}(Path, ..., Path)$. Applying n times Lemma E.10, we infer $\pi_1 \in Esc_{m_1} \circ \mathbb{E}_{\mathrm{C}}(pc_1)(Path, ..., Path)$, ..., $\pi_n \in Esc_{m_n} \circ \mathbb{E}_{\mathrm{C}}(pc_n) \circ ... \circ \mathbb{E}_{\mathrm{C}}(pc_1)(Path, ..., Path)$. Then by correctness of $\mathbb{E}$, $\{p | (N'.p \sim \pi_n) \in R^\infty(r_n/d/s_n)\} \subseteq \mathbb{E}(m_n)(v) \circ \mathbb{E}_{\mathrm{C}}(pc_n) \circ ... \circ \mathbb{E}_{\mathrm{C}}(pc_1)(Path, ..., Path)$ since the name N' only appears in v. Moreover, since the name N' is new at date d, $R(r_n)$ and $R^\infty(r_n/d/s_n)$ have the same aliases for N', therefore $\{p | (N'.p \sim \pi_n) \in R(r_n)\} \subseteq \mathbb{E}(m_n)(v) \circ \mathbb{E}_{\mathrm{C}}(pc_n) \circ ... \circ \mathbb{E}_{\mathrm{C}}(pc_1)(Path, ..., Path)$ and therefore does not contain ϵ, contradiction.

Then the considered object can be stack allocated in the execution trace T, in the considered executions of m_0, ..., m_n. Moreover, this is true for any execution trace, and any executions of m_0, ..., m_n, hence the expected result. □

E.4 Global synchronization analysis

Theorem 8.5 *If, in method m, a synchronized method m' is called by $w = v_0.m'(v_1, ..., v_j)$ and $\mathbb{L}_{1\mathrm{S}}(m)(v_0) = (\phi, (\tau_{-1}, ..., \tau_j), \tau'_0)$, and $\phi(\mathbb{L}_{1\mathrm{sync}}(m)) < \top[\tau'_0]$, then the object in v_0 on which m' is called is thread local, and the synchronization can be eliminated.*

Proof: If an object is never reachable from a static field or from a thread object passed as a parameter to the `Thread.start` method, then it will be local to the thread in which it has been created. Since `Thread.start` is a native method, we can set its escape information ourselves, and we consider that its parameter escapes, as if it were stored in a static field.

Using a technique similar to the theorem of correctness of inlining, we prove that the object on which m' is called is never aliased to a path static.p.

Let us translate the hypothesis into analysis $\mathbb{E}$: we define: $\mathbb{E}_{\text{sync}}(m) \in Ctx_{\mathbb{E}}^{j+2}$ such that if m may call m' at pc, then $\mathbb{E}_{\text{sync}}(m') \geq \mathbb{E}_{\text{C}}(pc)(\mathbb{E}_{\text{sync}}(m))$ and $\mathbb{E}_{\text{sync}}(m) \geq (\rho_{\text{S}}(m), \rho(m)(0), ..., \rho(m)(j))(\mathbb{E}_{\text{sync}}(m))$. We have $\epsilon \notin \mathbb{E}_{\text{S}}(m)(v_0)(\mathbb{E}_{\text{sync}}(m))$.

Consider one execution trace T of the program. Assume that in this execution trace, the program terminates at date r. We have to show that $(N'.\epsilon \sim \text{static}.p) \notin R(r)$ where N' is the name of the considered object. Let us consider an execution of m such that the call to method m' is reached by the call chain m_0 calls m_1 at pc_1, which calls m_2 at pc_2 ... which calls $m_n = m$ at pc_n. Let s_k be the date at the call to m_k, and r_k the date at the end of m_k in this execution of m.

By contradiction, assume that $(N'.\epsilon \sim \text{static}.p) \in R(r)$. We choose π_k in the chain $(N'.\epsilon \sim ... \sim \pi_n \sim ... \sim \pi_{n-1} \sim ... \sim \text{static}.p) \in R(r)$ such that all aliases of the chain $N'.\epsilon \sim ... \sim \pi_k$ have been created in the interval $]s_k, r_k]$, and the first alias after π_k has been created outside of this interval.

Then $\pi_k = N_k.p_k$ and N_k is the name of a parameter or of the result of method m_k, since these names are the only ones known both inside and outside $]s_k, r_k]$. We have $(N'.\epsilon \sim \pi_k) \in R(r_k)$. Moreover, if N_{k+1} is the name of the result of m_{k+1}, $(\pi_{k+1} \sim \pi_k) \in R(r_k/r_{k+1}/s_k)$. Indeed, the name of the result has been created during the execution of m_{k+1}, hence it is not known before s_{k+1}. Its aliases not in $]s_{k+1}, r_{k+1}]$ have therefore been created after r_{k+1}. By definition of Esc, $\text{static}.p \in Esc_{m_0}(\mathbb{E}_{\text{sync}}(m_0))$. Applying n times Lemma E.10, we can infer $\pi_1 \in Esc_{m_1} \circ \mathbb{E}_{\text{C}}(pc_1)(\mathbb{E}_{\text{sync}}(m_0)) \subseteq Esc_{m_1}(\mathbb{E}_{\text{sync}}(m_1))$, ..., $\pi_n \in Esc_{m_n} \circ \mathbb{E}_{\text{C}}(pc_n)(\mathbb{E}_{\text{sync}}(m_{n-1})) \subseteq Esc_{m_n}(\mathbb{E}_{\text{sync}}(m_n))$. Then, by correctness of $\mathbb{E}_{\text{S}}$, $\{N'.p | (N'.p \sim \pi_n) \in R(r_n/s_n)\} \subseteq \{N'.p | p \in \mathbb{E}_{\text{S}}(m)(v_0)(\mathbb{E}_{\text{sync}}(m_n))\}$. Moreover, since the name N' is created after the beginning of m_n, i.e. after date s_n (all names used in a method are created in this method or are results of methods it calls), N' has the same aliases in $R(r_n)$ and in $R(r_n/s_n)$, therefore $\{N'.p | (N'.p \sim \pi_n) \in R(r_n)\} \subseteq \{N'.p | p \in \mathbb{E}_{\text{S}}(m)(v_0)(\mathbb{E}_{\text{sync}}(m_n))\}$ and therefore does not contain $N'.\epsilon$. Contradiction. This yields the result in the considered execution of m.

If the program does not terminate, let r be any date. Change the program P such that it terminates at date r (more concretely, add a date counter, and test at each instruction whether date r has been reached. If yes, return immediately from the method, returning `null`, or 0 depending on the result type). The escape analysis obviously yields the same results on P and on the modified program P'. On P', the proof above shows that $(N'.\epsilon \sim \text{static}.p) \notin R(r)$. Since P and P' have obviously the same aliases at or before date r, this is also true for P. Since this is true for any r, N' is never reachable from a static field.

Then, in any execution trace T, in any executions of methods m_0, ..., m_n, the object in v_0 is thread-local. This is the expected result. □

Bibliography

[Abr86] S. Abramsky. Strictness Analysis and Polymorphic Invariance. In *Proc. Programs as Data Objects*, volume 217 of *Lecture Notes on Computer Science*, pages 1–23. Springer Verlag, 1986.

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes on Computer Science*, pages 19–38, Venice, Italy, September 1999. Springer Verlag.

[ADG+99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 207–222, Denver, Colorado, November 1999.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 174–185, La Jolla, California, June 1995.

[App87] Andrew W. Appel. Garbage Collection can be faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, January 1987.

[AS96] Andrew W. Appel and Zhong Shao. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming*, 6(1):47–74, January 1996.

[Bac97] David F. Bacon. *Fast and Efficient Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California, Berkeley, October 1997.

[BH99] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 35–46, Denver, Colorado, November 1999.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 258–268, Montreal, Canada, May 1998.

[Bla96] Bruno Blanchet. Garbage Collection statique. Rapport de DEA, INRIA, Rocquencourt, September 1996.

[Bla98] Bruno Blanchet. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*, pages 25–37, San Diego, California, January 1998. ACM.

[Bla99] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java™. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34, Denver, Colorado, November 1999.

[BS98] Anindya Banerjee and David A. Schmidt. Stackability in the Simply-Typed Call-By-Value Lambda Calculus. *Science of Computer Programming*, 31(1):47–73, May 1998.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183, St. Petersburg Beach, Florida, 21-24 January 1996.

[BV99] Boris Bokowski and Jan Vitek. Confined Types. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 82–96, Denver, Colorado, November 1999.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 29-31 January 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependance Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGS+99] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 1–19, Denver, Colorado, November 1999.

[Cha88] David R. Chase. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 1–10, Atlanta, Georgia, 22-24 June 1988. ACM Press.

[CLR94] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

[CM95] Guy Cousineau and Michel Mauny. *Approche fonctionnelle de la programmation.* Ediscience (Collection Informatique), Paris, 1995. Also available in English.

[CM98] Guy Cousineau and Michel Mauny. *The functional Approach to Programming.* Cambridge University Press, Cambridge, 1998. Aussi disponible en français.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler Optimizations for Improving Data Locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 252–262, October 1994.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Université Scientifique et Médicale de Grenoble, 21 March 1978.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Seventeenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 157–168, San Francisco, California, January 1990.

[Deu94] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, 20-24 June 1994. ACM Press.

[Deu97] Alain Deutsch. On the Complexity of Escape Analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'97)*, pages 358–371, Paris, France, January 1997.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[Dol95] Damien Doligez. *Conception, réalisation et certification d'un glaneur de cellules concurrent.* PhD thesis, Université Paris VII, May 1995.

[DR98] Pedro C. Diniz and Martin C. Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, March 1998.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An Optimizing Compiler for Java. *Software – Practice and experience*, 30(3):199–232, March 2000.

[GA95] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache Performance of Fast-Allocating Programs. In *SIGPLAN/SIGARCH?WG2.8 Conf. on Functional Programming and Computer Architecture (FPCA'95)*, pages 293–305. ACM Press, June 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification.* Addison-Wesley, 1996.

[GS00] David Gay and Bjarne Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In David A. Watt, editor, *Compiler Construction, 9th International Conference, CC'2000*, volume 1781 of *Lecture Notes on Computer Science*, pages 82–93. Springer Verlag, March 2000.

[Gui00] Romain Guider. *Analyse statique de programmes Java. Application à la parallélisation.* PhD thesis, Université de Nice - Sophia Antipolis, September 2000.

[Han95] John Hannan. A Type-based Analysis for Stack Allocation in Functional Languages. In *Proceedings of the Second International Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes on Computer Science*, pages 172–188. Springer Verlag, September 1995.

[Han98] John Hannan. A Type-based Escape Analysis for Functional Languages. *Journal of Functional Programming*, 8(3):239–273, May 1998.

[Har89] W. Harrison. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation*, 2:176–396, 1989.

[HDT87] Susan Horwitz, Alan Demers, and Tim Teitelbaum. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Informatica*, 24(6):679–694, 1987.

[Hed88] Lucy Hederman. Compile Time Garbage Collection Using Reference Count Analysis. Technical Report Rice COMP TR88-75, Rice University, Houston, Texas, August 1988.

[Hic93] James Hicks. Experiences with Compiler-Directed Storage Reclamation. In *Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, Copenhagen, Denmark, 9-11 June 1993. ACM Press.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. Technical Report TR-1283, University of Wisconsin, Madison, WI, August 1995.

[Hud86] Paul Hudak. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *Proceedings of the 1986 ACM Conference on LISP and functional programming*, pages 351–363, Cambridge, Massachusetts, 4-6 August 1986.

[Hug88] John Hughes. Backward Analysis of Functional Programs. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation.* Elsevier Science Publishers B.V. (North Holland), 1988.

[Hug92] Simon Hughes. Compile-Time Garbage Collection for Higher-Order Functional Languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.

[Int99] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 1999.

[ISY88] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, October 1988.

[JM82] N. D. Jones and S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *9th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 66–74, Albequerque, New Mexico, 1982.

[JW90] Simon B. Jones and Michael White. Is Compile Time Garbage Collection Worth the Effort. In *Workshop on Functional Programming*, Glasgow, 1990. Springer Verlag.

[Knu77] Donald E. Knuth. A Generalization of Dijkstra's Algorithm. *Information Processing Letters*, 6(1):1–5, February 1977.

[KU76] John B. Kam and Jeffrey D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[Ler92] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

[Liu98] Yanhong A. Liu. Dependence analysis for recursive data. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 206–215, Chicago, Illinois, May 1998.

[LW93] Xavier Leroy and Pierre Weis. *Manuel de référence du langage Caml*. InterEditions, Paris, 1993. See also `http://caml.inria.fr` for information about the Caml language.

[LY96] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 1996.

[McD98] Charles E. McDowell. Reducing garbage in Java. *ACM Sigplan Notices*, 33(9):84–86, September 1998.

[MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *3rd Usenix Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pages 1–20, Portland, Oregon, June 1997.

[Moh95a] Markus Mohnen. Efficient Closure Utilisation by Higher-Order Inheritance Analysis. In *Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes on Computer Science*, pages 261–278, Glasgow, UK, 1995. Springer Verlag.

[Moh95b] Markus Mohnen. Efficient Compile-Time Garbage Collection for Arbitrary Data Structure. In *Symposium on Programming Language Implementation and Logic Programming (PLILP'95)*, volume 982 of *Lecture Notes on Computer Science*, pages 241–258, Utrecht, The Netherlands, 1995. Springer Verlag.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQeen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[OK99] Tamiya Onodera and Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 223–237, Denver, Colorado, November 1999.

[Pau96] Larry C. Paulson. *ML for the Working Programmer (2nd edition)*. Cambridge University Press, 1996.

[PG92] Young Gil Park and Benjamin Goldberg. Escape Analysis on Lists. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 116–127, San Francisco, California, 17-19 July 1992.

[Qia98] Zhenyu Qian. A Formal Specification of Java^{tm} Virtual Machine Instructions for Objects, Methods and Subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java(tm)*, volume 1523 of *Lecture Notes on Computer Science*. Springer Verlag, 1998.

[RM88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'88)*, pages 285–293, San Diego, California, 13-15 January 1988.

[RSH94] Thomas Reps, Mooly Sagiv, and Susan Horwitz. Interprocedural Dataflow Analysis via Graph Reachability. Technical Report TR94–14, University of Copenhagen, Copenhagen, Denmark, April 1994.

[Ruf00] Erik Ruf. Effective Synchronization Removal for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, Vancouver, Canada, June 2000.

[Sam97] Samsung Electronics. *21164 Alpha Microprocessor, Hardware Reference Manual*, 1997.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, Copenhagen, Denmark, October 1991.

[SF96] Manuel Serrano and Marc Feeley. Storage Use Analysis and its Applications. In *1996 ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 50–61, Philadelphia, Pennsylvania, 24-26 May 1996.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, June 1988.

[SL96] André Seznec and Fabien Lloansi. Etude des architectures des microprocesseurs MIPS R10000, UltraSPARC et PentiumPro. Technical Report 1024, IRISA, May 1996.

[Sun97a] Sun Microsystems. *microSPARC-IIep User's Manual*, April 1997. `http://www.sun.com/microelectronics/microSPARC-IIep/`.

[Sun97b] Sun Microsystems. *UltraSPARC-IIi User's Manual*, October 1997. `http://www.sun.com/microelectronics/UltraSPARC-IIi/`.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tar79] Robert Endre Tarjan. Applications of Path Compression on Balanced Trees. *Journal of the ACM*, 26(4):690–715, October 1979.

[TT93] Mads Tofte and Jean-Pierre Talpin. A theory of Stack Allocation in Polymorphically Typed Languages. Technical Report 93/15, Departement of Computer Science, Copenhagen University, 9 July 1993.

[W+98] Michael Weiss et al. TurboJ: a Java Bytecode-to-Native Compiler. In *Proceedings of Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474 of *Lecture Notes on Computer Science*, pages 119–130, Montreal, Canada, June 1998. Springer Verlag.

[WL99] Pierre Weis and Xavier Leroy. *Le langage Caml (Second edition)*. Dunod, Paris, 1999.

[WR99] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 187–206, Denver, Colorado, November 1999.

[Wri95] Andrew K. Wright. Simple Imperative Polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, December 1995.

[WSR00] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In David A. Watt, editor, *Compiler Construction, 9th International Conference, CC 2000*, volume 1781 of *Lecture Notes on Computer Science*, pages 1–17. Springer Verlag, March 2000.

Index

$\bot$, least element, 26
$\sqcap$, lower bound, 26
$\top$, greatest element, 26
$\sqcup$, upper bound, 26

α, abstraction, 27
abstract trace, 149
abstraction, 27
Access, set of accessors, 37
access path, 37, 147
accessor, 37
Alias, set of alias relations, 148
annotated type, 56
anti-tree, 80
app, application accessor, 37
assignment instruction, 29

β, type variable, 29

C, class (Java part), 144
C, constructor (ML part), 29
c, context, 38, 152
$C?$, test function, 29
chain, 25
$\overline{C_i}$, destructor, 29
$\overline{C_i}(v) \leftarrow w$, assignment instruction, 29
Class, set of classes, 144
class declaration, 144
closed, 44, 52, 62
closure, 30
complete lattice, 26
concretization, 27
Constr, set of constructors, 30
$\mathrm{Constr}_A(\delta, \tau)$, construction, 49
construction, 38, 152
constructor, 29
context, 38, 152
    construction, 38, 152
    restriction, 39, 152
context transformer, 39, 153
continuous, 25
control expression, 85
copy collector, 95
CPO, 25
Ctx_A, set of contexts, 38, 152

δ, accessor, 37
d, date, 148
Date, set of dates, 148
date, 149
DateAlias, set of dated alias relations, 148
dated alias relation, 149
Destr, set of destructors, 30
destructor, 29

E, escape analysis for ML, 40
$\mathbb{E}$, escape analysis for Java, 151
ϵ, empty path, 37, 147
e, semantic environment, 209
edge
    call, 98
    ordinary, 98
    return, 98
    summary, 111
Env, set of environments, 209
$Env_{\mathbb{E}}$, set of abstract environments, 154
Env_E, set of abstract environments, 40
environment
    abstract, 40, 154
    parameter, 212
    semantics, 209
    typing, 32
Exp_E, set of abstractions of expressions, 40
explicit deallocation, 200
exploding objects, 200
expression, 30

F, escape analysis for ML, 77
f, field, 144
fetch-on-write, 135

Field, set of fields, 144
first-order, 257
free variables
  type, 32
  type scheme, 32
FV, free variables, 26

G^{τ}_{A}, generalization (escape analysis), 48
Γ, typing environment, 32
γ, concretization, 27
Galois connection, 27
Gen, typing generalization, 32
generalization
  escape analysis, 48
  typing, 32
generations, 95
Gr, graph of a type, 54
graph
  isomorphism, 53
  morphism, 53
  of a type, 53, 54

H, heap, 286
heap, 285
height, 68, 160

I^{τ}_{A}, instantiation (escape analysis), 48
ι, empty edge label, 98
Id^{τ}_{A}, identity, 49
Idx, set of indices, 287
IFDS problems, 98
incremental, 95
Ind, set of parameter indices, 40, 154
index, 287
inherit, 143
Ins^{τ}_{p}, path insertion, 60
instantiation
  escape analysis, 48
  typing, 32
IVP, interprocedurally valid paths, 100

J, Java stack, 148, 286
java stack, 285
JavaStack, set of java stacks, 286
$JavaStack^{\#}$, set of abstract java stacks, 148

L, escape analysis for ML, 68
L, local variables (Java part), 148, 286
$\mathbb{L}$, escape analysis for Java, 160
Λ, subset of *Lbl*, 209
ℓ, location, 208, 286
λ, label of a value, 208
L', escape analysis for ML, 251
$\mathbb{L}_1$, escape analysis for Java, 168
L_1, escape analysis for ML, 254
Label, set of labels, 208
label, 208
labelled value, 208
LabelVal, set of labelled values, 208
Lbl, set of elementary labels, 208
lfp, least fixed point, 25
Loc, set of locations, 208
location, 208, 285
LocVar, set of local variables, 286
$LocVar^{\#}$, set of abstract local variables, 148

M, expression, 30
$\xrightarrow{\mathrm{m}}$, monotone functions, 26
m, method, 144
major heap, 95
mark and sweep collector, 95, 187
Method, set of methods, 144
minor heap, 95

N, object name, 147
NofP, node of path, 56
non-expansive, 33
non-monotone, 67

o, value, 286
object name, 147
OName, set of object names, 148
OTypes, object types, 286

p, access path, 38, 147
Path, set of access paths, 37, 147
path
  access, 37, 147
  realizable, 103
  valid, 99
PC, set of program counters, 148
pc, program counter, 148
polymorphic invariance, 45
preallocation, 200
prefetching, 135

q, path in super-graph, 99

R, dated alias relation, 148
ρ, abstract environment, 40, 154
ϱ, environment, 81
realizable path, 103
RefType, set of reference types, 144
region allocation, 201
RemPath, remaining path, 60
$\mathrm{Restr}_A(\delta, \tau)$, restriction, 49
restriction, 39, 152
$\rho_\top$, escape information for an unknown function, 42
roots, 95
RP, realizable paths, 104

σ, substitution, 32
s, store, 209
s_∞, infinite step function, 77
Ξ, state, 286
SLIVP, same level interprocedurally valid paths, 99
SLRP, same level realizable paths, 103
SOP, path containing summary and ordinary edges, 279
State, set of states, 286
$State^\#$, set of abstract states, 148
storable value, 208
Store, set of stores, 209
store, 209
subclass, 143
substitution, 32
subtyping, 143
super-graph, 98
  exploded, 103
SV, set of storable values, 208

T, escape analysis for ML, 56
T, execution trace (Java part), 149, 286
$T^\#$, abstract execution trace (Java part), 148
τ, type, 28, 144
τ_{N}, name of type constructor, 29
τ_{S}, type scheme, 32
tail call, 85
$\mathrm{test}_A^{\tau,\tau'}$, step function, 49
test function, 29
$\top[\tau]$, height of a type, 160
$\top_A[\tau]$, height of a type, 68
Tr_A, set of context transformers, 40
Trace, set of traces, 286
$Trace^\#$, set of abstract traces, 148
TY, types, 286
Type, set of types, 30, 144
type, 28
type constructor, 29
type declaration, 29
type graph, 54
Types, types labelling a context, 61

U, label for updated values, 208

v, value (ML part), 208
v, variable (Java part), 144
Val, set of values, 208, 286
$\mathrm{Val}^\#$, abstract value of a variable, 148
Val_E, set of abstractions of values, 40
valid path, 99
value, 208, 285
  labelled, 208
  storable, 208
Var, set of variables, 30, 144
variable, 30, 144

write allocate, 135
write around, 135
write buffer, 135

x, variable (ML part), 30

Abstract

Escape analysis is a static analysis that aims at determining whether the lifetime of values exceeds their static scope or not. If not, the values can be stack allocated. This thesis extends the previous works on escape analysis by fully handling two realistic languages for the first time: Objective Caml and Java. Our contributions are both theoretical and practical.

The main theoretical contribution is the correctness proof of the analyses from a semantics of the analyzed language, using the techniques of abstract interpretation. For ML, the imperative operations (assignments), polymorphism, higher-order functions, recursive types are handled. Two representations of the escape abstract values are proposed: one, very efficient, uses integers, the other, more precise but also more costly, uses annotated types. The comparison between these two representations indicates which one yields the best tradeoff. For Java, we propose a new technique to take into account assignments more precisely than in ML, while keeping a fast analysis.

The analyses have been fully implemented in compilers, to study two applications of escape analysis: stack allocation and synchronization elimination. (This last optimization does not apply to ML.) A detailed experimental study of the effects of these optimizations has been performed on large programs. This study has shown that stack allocation improves data locality, decreases the GC (garbage collector) workload and the allocation time, and that the effects of stack allocation highly depend on the kind of GC and of cache. The analysis algorithms have been chosen to obtain particularly efficient analyses. The experiments have shown that the representation using integers yields the best cost/precision tradeoff.

Keywords

Static analysis, abstract interpretation, optimization, escape analysis, stack allocation, synchronization elimination, Java, ML.

Résumé

L'analyse d'échappement est une analyse statique visant à déterminer si la durée de vie des données dépasse leur portée statique. Si elle ne la dépasse pas, les données peuvent être allouées en pile. Cette thèse étend les travaux précédents sur l'analyse d'échappement en traitant pour la première fois complètement deux langages réalistes : Objective Caml et Java. Nos contributions sont à la fois théoriques et pratiques.

Le principal apport théorique est la preuve de correction des analyses à partir d'une sémantique du langage analysé, en utilisant les techniques de l'interprétation abstraite. Pour ML, les opérations impératives (affectations), le polymorphisme, les fonctions d'ordre supérieur, les types récursifs sont traités. Deux représentations des valeurs abstraites d'échappement sont proposées : l'une, très efficace, utilise des entiers et l'autre, plus précise mais plus coûteuse, des types annotés. La comparaison entre ces deux représentations permet de trouver laquelle offre le meilleur compromis. Pour Java, nous proposons une nouvelle technique permettant de prendre en compte les affectations de façon plus précise que pour ML, tout en conservant une analyse rapide.

Une implantation complète dans des compilateurs a été effectuée, afin d'étudier deux applications de l'analyse d'échappement : l'allocation en pile et l'élimination des synchronisations (cette dernière application ne concerne pas ML). Les effets de ces optimisations font l'objet d'une étude expérimentale détaillée sur de gros programmes. Cette étude a montré que l'allocation en pile améliore la localité des données, diminue la charge du GC (glaneur de cellules) et le temps d'allocation, et que ses effets dépendent considérablement du type de GC et de cache utilisé. Les algorithmes utilisés ont été choisis de façon à fournir des analyses particulièrement efficaces. L'expérience a montré que la représentation utilisant des entiers est celle qui donne le meilleur rapport coût/précision.

Mots-clés

Analyse statique, interprétation abstraite, optimisation, analyse d'échappement, allocation en pile, élimination des synchronisations, Java, ML.