Prolongement de stage de DEA
Garbage Collection statique

BLANCHET Bruno
`Bruno.Blanchet@ens.fr`

sous la direction d'Alain DEUTSCH
`Alain.Deutsch@inria.fr`

Octobre 1997

Remerciements

Je tiens à remercier toute l'équipe de l'INRIA pour son accueil chaleureux et en particulier M. Alain Deutsch qui a été très disponible pour m'encadrer tout au long de mon stage, et qui m'a apporté à la fois son aide pour résoudre les problèmes techniques et ses idées sur le sujet du stage.

Je voudrais aussi remercier MM. Xavier Leroy et Damien Doligez pour leur aide dans la correction du programme et Mme Josy Baron pour sa diligence dans les démarches administratives.

Table des matières

Chapitre 1

Introduction

L'objectif de ce stage est d'étudier et d'implémenter une optimisation de l'allocation des données dans les langages à *garbage collector* (GC). Le GC récupère automatiquement la place occupée par les données qui ne sont plus référencées au cours de l'exécution des programmes. Cependant ce mécanisme prend du temps. Le but du projet est d'allouer autant que possible les données en pile, pour les libérer sans faire appel au GC. Le travail peut se décomposer en deux phases : on commence par effectuer une analyse par interprétation abstraite [CC77, CC79] pour déterminer les données qui peuvent effectivement être allouées en pile, c'est-à-dire celles dont la durée du vie ne dépasse pas la portée lexicale. Cette analyse est appelée analyse d'échappement. Ensuite, on transforme le programme en conséquence pour que les données voulues soient effectivement allouées en pile. Le langage analysé est Caml Special Light (CSL), mais il est prévu d'adapter le programme à Objective Caml qui est la nouvelle version de CSL.

Les résultats expérimentaux montrent qu'on obtient ainsi une optimisation de 5 à 20 % sur beaucoup de programmes. L'amélioration obtenue dépend de façon importante des caractéristiques du processeur utilisé (en particulier de la taille du cache mémoire). C'est pourquoi on a effectué une comparaison entre différentes machines : SparcStation 5, Alpha, Pentium Pro.

Ces essais montrent que l'affirmation d'A. Appel [App87] selon laquelle la *garbage collection* peut être plus rapide que l'allocation en pile, n'est valable que si la quantité de mémoire disponible est nettement supérieure à celle vraiment nécessaire. Ceci n'est pas le cas en général car malgré l'augmentation rapide de la quantité de mémoire disponible sur les ordinateurs, elle reste limitée par rapport à la taille utile qui augmente elle aussi dans les mêmes proportions.

1.1 Travaux associés

Les travaux les plus proches du nôtre sont ceux de Park et Goldberg [PG92] : ils effectuent une analyse d'échappement sur les listes. Nous implémentons cette analyse avec une complexité en $\mathcal{O}(n \log^2 n)$, selon la méthode montrée par A. Deutsch [Deu97], et nous la généralisons aux types inductifs, avec aussi des constructions impératives et un traitement (certes approximatif) des fonctions d'ordre supérieur. Nous avons aussi étendu les stratégies de transformation de programme en exploitant les résultats de l'analyse sémantique, et nous avons effectué de nombreuses expérimentations pour en déterminer avec précision les résultats.

M. Mohnen [Moh95b, Moh95a] effectue une analyse assez voisine de la nôtre, mais son algorithme a une complexité quadratique, et le langage analysé est du premier ordre sans opérations impératives.

S. Hughes [Hug92] introduit déjà une représentation avec des niveaux entiers pour l'échappement des données, mais utilise une analyse en deux étapes (génération et héritage). Il analyse un langage d'ordre supérieur purement fonctionnel, mais l'optimisation qu'il effectue est différente de la nôtre : il n'alloue pas en pile, mais mémorise les adresses qui devront être libérées pour éviter de les rechercher avec le GC. Le travail le plus proche de ce-dernier est celui de Inoue, Seki et Yagi [ISY88], qui eux récupèrent seulement le sommet des listes, mais donnent des résultats expérimentaux intéressants.

Une autre optimisation de l'allocation a été proposée par M. Tofte et J.-P. Talpin [TT93], et expérimentée dans [BTV96] : l'allocation en régions. L'étude a été poursuivie dans [AFL95]. Toutes les données sont allouées dans des régions de taille non fixée au départ en général, mais pour lesquelles on a une borne sur la durée de vie. On peut alors libérer les régions sans utiliser de GC. Cette analyse résoud un problème plus général que la nôtre mais au prix d'une complexité nettement plus grande, et en fait sur beaucoup de programmes les opportunités d'allocation en pile sont plus nombreuses que celles d'allocation en régions.

A. Deutsch introduit une analyse d'alias [Deu94] qui donnerait des informations beaucoup plus précises, mais qui est malheureusement trop complexe pour être appliquée à de gros programmes.

Une autre technique d'analyse utilisable pour déterminer les possibilités d'allocation en pile est l'utilisation d'un compteur de référence, proposée dans [Hud86]. P. Hudak l'applique ici à la mise à jour en place et uniquement pour des tableaux fonctionnels. L. Hederman [Hed88] reprend cette technique pour l'appliquer à l'allocation en pile.

M. Serrano et M. Feeley [SF96] décrivent une technique d'analyse d'échappement intéressante, la "Storage Use Analysis" (analyse d'utilisation de la mémoire), qui malgré sa complexité en $\mathcal{O}(n^4)$ est rapide à exécuter et donne des optimisations importantes. Malheureusement, ce travail mêle deux optimisations indépendantes, l'allocation en pile et l'*unboxing* de tableaux. Cette implémentation, extrêmement proche des travaux de Jones et Muchnick [JM82], Harrison [Har89], Deutsch [Deu90] et Shivers [Shi88], a été réalisée sur le compilateur Bigloo 1.8.

J. Hannan [Han95] utilise des types annotés pour décrire les informations d'échappement. Les résultats obtenus sont nettement moins bons que ceux de notre analyse (par exemple, dans $\lambda x.\lambda y.(x + y)$, seul le paramètre passé pour y pourra jamais être mis en pile, car lors d'une application partielle x s'échappe). De plus, il donne uniquement des règles d'inférence, et pas d'algorithme de calcul des types annotés.

Chapitre 2

Analyse d'échappement

Cette analyse s'inspire de celle définie dans [PG92], mais on l'étend et on l'implémente en temps $\mathcal{O}(n \log^2 n)$ où n est la taille du programme.

2.1 Quelques notations

Je ne rappelle ici que les notations mathématiques générales. Les autres seront définies au fur et à mesure des besoins.

Les fonctions sont manipulées ainsi :

- $\{x_1 \mapsto y_1, ..., x_n \mapsto y_n\}$ désigne la fonction qui à x_i associe y_i pour i dans $[1, n]$.
- $f[x_1 \mapsto y_1, ..., x_n \mapsto y_n]$ désigne le prolongement de f qui à x_i associe y_i pour i dans $[1, n]$. Si f était déjà définie en certains de ces points, la nouvelle valeur remplace l'ancienne.
- $f_{|E}$ désigne la restriction de f à l'ensemble E.
- $\mathrm{Dom}(f)$ est le domaine de définition de f

Les listes sont manipulées comme suit :

- S^* désigne l'ensemble des listes dont les éléments sont dans S.
- $[]$ est la liste vide.
- $[p_1, \ldots, p_n]$ est la liste d'éléments $p_1, \ldots, p_n$.
- $p_1 : l$ désigne la liste l à laquelle on a concaténé en tête l'élément p_1.
- $l(i)$ désigne le i^{e} élément de la liste l.

$FV(M)$ est l'ensemble des variables libres de M.

Dans un treillis, la borne supérieure est $\sqcup$ et la borne inférieure $\sqcap$.

2.2 Le langage analysé

2.2.1 Syntaxe

L'analyse a été réalisée sur le langage Caml Special Light. Pour la description théorique du système, on se contente ici d'un sous-ensemble restreint de ce langage. Il comprend les types suivants :

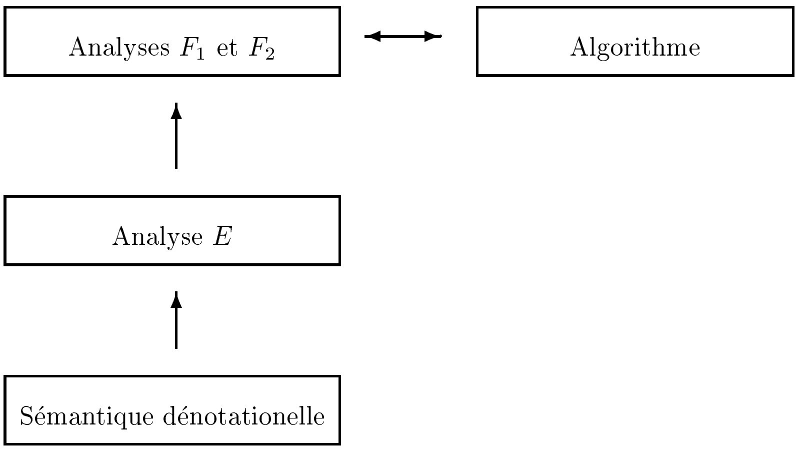
$$\tau = \tau_1 \to \tau_2 \mid \tau\ \texttt{list} \mid \tau_1 * \tau_2 \mid \tau\ \texttt{ref} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{unit}$$

Les entiers se traitent comme les booléens, les enregistrements et les n-uplets comme les paires. Les champs mutables se traitent en s'inspirant des références. On dispose des expressions :

$e =$ `v`
 | `let x=`e_1 `in` e_2
 | `let rec f(x)=`e_1 `in` e_2
 | $e_1\ e_2$
 | `fun x` $\rightarrow e$
 | `true` | `false` | `0` | `1` | ...
 | e_1 `::` e_2 | `[]`
 | `hd` | `tl` | `null`
 | `ref` | `!` | `:=`
 | `(`e_1`,`e_2`)`
 | `fst` | `snd`
 | e_1`;`e_2
 | `if` e_1 `then` e_2 `else` e_3

L'opérateur d'affectation `:=` est infixe. On généralise facilement le `let rec` donné ici au cas des fonctions mutuellement récursives.

On associe à ce langage plusieurs sémantiques, en commençant par une sémantique exacte puis en l'approchant par des sémantiques abstraites comme indiqué sur la figure ci-dessous. La sémantique la plus abstraite est équivalente à l'algorithme utilisé.



2.2.2 Sémantique

On utilise ici une sémantique dénotationnelle avec état et non-terminaison. On utilise les domaines :

- Variables : $x \in \mathit{Var}$.
- Locations ou cellules mémoire (ensemble dénombrable) : $\ell \in \mathit{Loc}$.
- Valeurs : $\mathit{Val} = \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z} \cup \mathit{Loc}$.
- $\mathit{SV} = (\mathit{Val} \cup \mathit{Val} \times \mathit{Val} \cup (\mathit{Env} \rightarrow \mathit{Exp}) \times \mathit{Env} \times \mathit{Var})$, $\mathit{Store} = (\mathit{Loc} \times \mathit{SV})^*$ est l'ensemble des états. On note $s[\ell \mapsto v]$ pour $(\ell, v) : s$ et $s(\ell)$ est tel que $(\ell, s(\ell))$ est le premier élément de s de la forme $(\ell, \_)$. On fait donc comme si $\mathit{Store} = \mathit{Loc} \rightarrow \mathit{SV}$ mais on utilise une liste au lieu d'une vraie fonction pour pouvoir retrouver les modifications faites à un état, ce qui sera utile dans la définition de l'analyse. Ceci est analogue à la définition des traces d'exécution d'Hoare [Hoa78] comme suites de tests et d'affectations, mais on n'a pas besoin de conserver les tests ici. Un état associe à chaque cellule son contenu. Ce contenu peut être une valeur pour les références, un couple de valeurs pour les couples et les listes (non vides) ou une fermeture pour les fonctions. Les états sont en général notés $s, s' \ldots$

- Expressions : $Exp = Store \to (Val \times Store)_\perp$.
- Environnments : $Env = Var \to (Store \times Val)$. On stocke non seulement la valeur, mais l'état dans lequel elle a été créée. On définit $e[\![x]\!] = v$ si $e(x) = (s, v)$.

On peut spécifier plus précisément le domaine des valeurs pour chaque type :

$Val[\![\texttt{bool}]\!] = \{\text{true}, \text{false}\}$
$Val[\![\texttt{int}]\!] = \mathbb{Z}$
$Val[\![\texttt{unit}]\!] = \{\text{NIL}\}$
$\ell \in Val[\![\tau_1 \to \tau_2]\!]$ si $\ell \in Loc$ et $s(\ell) = (f, e, x), x \in Var, e \in Env, f \in Env[\![x \in \tau_1]\!] \to Exp[\![\tau_2]\!]$
$\ell \in Val[\![\tau_1 * \tau_2]\!]$ si $\ell \in Loc$ et $s(\ell) \in Val[\![\tau_1]\!] \times Val[\![\tau_2]\!]$
$\ell \in Val[\![\tau\ \texttt{ref}]\!]$ si $\ell \in Loc$ et $s(\ell) \in Val[\![\tau]\!]$
$\ell \in Val[\![\tau\ \texttt{list}]\!]$ si $\ell \in Loc$ et $s(\ell) \in Val[\![\tau]\!] \times Val[\![\tau\ \texttt{list}]\!]$ ou $\ell = \text{NIL}$ pour la liste vide.
$Exp[\![\tau]\!] = Store \to (Val[\![\tau]\!] \times Store)_\perp$
$Env[\![x \in \tau]\!] = \{e \in Env | e(x) \in Val[\![\tau]\!]\}$

La fonction sémantique a le domaine suivant : $[\![M]\!] : Env \to Exp$.
Soient les abréviations :

- up est l'injection de $Val \times Store$ dans $(Val \times Store)_\perp$.
- let $[x] = h$ in $g = \begin{cases} \perp & \text{si } h = \perp, \\ g[x'/x] & \text{si } h = \text{up}(x'). \end{cases}$
- $\text{newloc}(s, v) = \text{up}(\ell, s[\ell \mapsto v])$ où ℓ est une nouvelle cellule, qui sera initialisée à la valeur v.
- let $(\ell_1, \ell_2) = s(\ell)$ in g indique que ℓ_1 désigne la première composante du couple $s(\ell)$, et ℓ_2 la deuxième. Dans le cas où ℓ est une liste (qui est aussi un couple si la liste est non vide), cette notation n'est autorisée que si ℓ est non vide, sinon il se produit une erreur à l'exécution. Pour simplifier, le cas d'erreur n'est pas considéré dans la sémantique.

et les équations sémantiques sont rassemblées sur la figure 2.1

2.3 Première version de l'analyse

Cette analyse peut être appliquée telle que à n'importe quel langage fonctionnel, même non typé. C'est une analyse très précise, donc elle peut être utilisée comme base pour beaucoup d'analyses moins précises que l'on peut déduire par interprétation abstraite. Le coût de ceci est qu'elle ne peut pas être implémentée directement car elle est trop complexe. On devra donc effectuer une deuxième étape d'approximation (dans la section suivante).

C'est une analyse par interprétation abstraite. Les domaines abstraits associés aux données sont des chemins d'accès qui représentent les parties utilisées (figure 2.2) :

$$Chem = \text{g}.Chem | \text{d}.Chem | \text{app}.Chem | \top | \perp$$

où $\top$ indique que toute la donnée est utilisée; $\perp$ indique que rien n'est utilisé; g indique que la partie gauche d'un couple, ou la tête d'une liste, ou le contenu d'une référence est utilisé; d indique que la partie droite d'un couple ou la queue d'une liste est utilisée. Enfin, app indique que la fonction est appliquée à un argument.

On définit les restrictions des chemins par :

- $\text{app}.c_{|\to} = c$, $c_{|\to} = \perp$ si c n'est pas de la forme $\text{app}.c'$.
- $\top.|g = \top$, $\text{g}.c_{|g} = c$, $c_{|g} = \perp$ dans les autres cas. On définit de même $|d$.

$[\![M]\!] : \mathit{Env} \to \mathit{Exp}$
$[\![x]\!]es = \mathrm{up}(e[\![x]\!], s)$
$[\![M\ N]\!]es = \mathrm{let}\ [(f', s')] = [\![M]\!]es\,\mathrm{in}\ \ \mathrm{let}\ [(v, s'')] = [\![N]\!]es'\ \mathrm{in}\ \ \mathrm{let}\ (m, e', x) = s'(f')\ \mathrm{in}\ \ m(e'[x \mapsto v])s''$
$[\![\mathtt{fun}\ x \to M]\!]es = \mathrm{newloc}(s, ([\![M]\!], e_{|FV(\mathtt{fun}\ x \to M)}), x))$ [a]
$[\![\mathtt{let}\ x = M\ \mathtt{in}\ N]\!]es = \mathrm{let}\ [(v, s')] = [\![M]\!]es\,\mathrm{in}\ [\![N]\!](e[x \mapsto v])s'$
$[\![\mathtt{let\ rec}\ f(x) = M\ \mathtt{in}\ N]\!]es = [\![N]\!](e[f \mapsto (s', \ell)])s'$ où $\ell \notin \mathrm{Dom}(s),\ s' = s[\ell \mapsto (\phi, e_{|FV(\mathtt{fun}\ x \to M)}, x)]$
et $\phi = \mathrm{lfp}(\lambda\phi_1.\lambda e_1.\lambda s_1.[\![M]\!](e_1[f \mapsto (s_1', \ell)])s_1')$ où s_1' est l'état s_1 où $s_1(\ell)$ a été remplacé par (ϕ_1, e_1, x) [b]
$[\![\mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ P]\!]es = \mathrm{let}\ [(x, s')] = [\![M]\!]es\,\mathrm{in}\ \ \mathrm{if}\ x\ \mathrm{then}\ [\![N]\!]es'\ \mathrm{else}\ [\![P]\!]es'$
$[\![M{::}N]\!]es = \mathrm{let}\ [(x, s')] = [\![M]\!]es\,\mathrm{in}\ \ \mathrm{let}\ [(y, s'')] = [\![N]\!]es'\ \mathrm{in}\ \ \mathrm{newloc}(s'', (x, y))$
$[\![(M, N)]\!]es = \mathrm{let}\ [(l_1, s')] = [\![M]\!]es\,\mathrm{in}\ \ \mathrm{let}\ [(l_2, s'')] = [\![N]\!]es'\ \mathrm{in}\ \ \mathrm{newloc}(s'', (l_1, l_2))$
$[\![M; N]\!]es = \mathrm{let}\ [(l, s')] = [\![M]\!]es\,\mathrm{in}\ [\![N]\!]es'$

Etat initial : $s_0 = \{\mathrm{ref} \mapsto (\lambda e.\lambda s'.\,\mathrm{newloc}(s', e[\![x]\!]), \emptyset, x), \mathrm{deref} \mapsto (\lambda e.\lambda s'.\,\mathrm{up}(s'(e[\![x]\!]), s'), \emptyset, x),$
$\mathrm{affect} \mapsto (\lambda e.\lambda s'.\,\mathrm{newloc}(s', (\lambda e'.\lambda s''.\,\mathrm{up}(\mathrm{NIL}, s''[e'[\![m]\!] \mapsto e'[\![n]\!]]), e, n)), \emptyset, m),$
$\mathrm{fst} \mapsto (\lambda e.\lambda s'.\,\mathrm{let}\ (h, t) = s'(e[\![x]\!])\ \mathrm{in}\ \ \mathrm{up}(h, s'), \emptyset, x),$
$\mathrm{snd} \mapsto (\lambda e.\lambda s'.\,\mathrm{let}\ (h, t) = s'(e[\![x]\!])\ \mathrm{in}\ \ \mathrm{up}(t, s'), \emptyset, x),$
$\mathrm{null} \mapsto (\lambda e.\lambda s'.(e[\![x]\!] = \mathrm{NIL}), \emptyset, x)\}$
Environnement initial : $e_0 = \{\mathtt{ref} \mapsto (s_0, \mathrm{ref}), ! \mapsto (s_0, \mathrm{deref}), := \mapsto (s_0, \mathrm{affect}), \mathtt{true} \mapsto (s_0, \mathrm{true}),$
$\mathtt{false} \mapsto (s_0, \mathrm{false}), n \mapsto (s_0, n)$ si $n \in \mathbb{Z}$, $\mathtt{[]} \mapsto (s_0, \mathrm{NIL}), \mathtt{hd} \mapsto (s_0, \mathrm{fst}),$
$\mathtt{tl} \mapsto (s_0, \mathrm{snd}), \mathtt{null} \mapsto (s_0, \mathrm{null}), \mathtt{fst} \mapsto (s_0, \mathrm{fst}), \mathtt{snd} \mapsto (s_0, \mathrm{snd})\}.$

[a] On crée des fermetures minimales, donc on ne stocke dans l'environnement de la fermeture que les variables libres de $\mathtt{fun}\ x \to M$.
[b] On n'écrit pas $s_1' = s_1[\ell \mapsto (\phi_1, e_1, x)]$ car on ne souhaite pas considérer ces modifications comme des affectations.

FIG. 2.1 – *Sémantique*

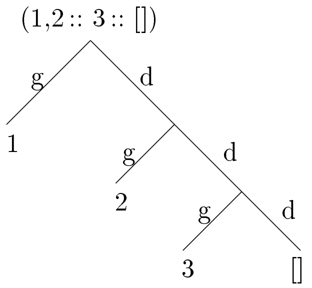


FIG. 2.2 – *Chemins. Par exemple, 2 est désigné par le chemin* d.g.$\top$

Les contextes associés aux données sont des ensembles non vides de chemins $Ctx \subseteq \mathcal{P}(Chem)$. En effet, un contexte vide correspondrait intuitivement à une expression non évaluée. Ici, à cause de l'appel par valeur, les expressions sont évaluées même si leur résultat n'est pas utile, et cette évaluation peut causer de l'échappement à cause des opérations impératives. Soit $\mathcal{P}^*(Chem) = \mathcal{P}(Chem) - \emptyset$. Ceci est un sup-demi-treillis, mais pas un treillis. Pour préserver la borne inférieure, il faut se restreindre à des contextes contenant au moins un élément fixé, $\perp$.

Si $\phi : Chem \to Chem$ est une fonction qui à p associe $\mathrm{g}.p, \mathrm{d}.p, \mathrm{app}.p, p_{|g}, p_{|d}, p_{|\to}$, on définit $\phi^P : \mathcal{P}^*(Chem) \to \mathcal{P}^*(Chem)$ par $\phi^P(c) = \{\phi(p) | p \in c\}$ et on veut définir $\mathrm{Ferm} : \mathcal{P}^*(Chem) \to Ctx$ and $\phi^C : Ctx \to Ctx$ tels que $\mathrm{Ferm} \circ \phi^P = \phi^C \circ \mathrm{Ferm}$, $c \subseteq \mathrm{Ferm}(c)$, $\perp \in \mathrm{Ferm}(c)$, $\mathrm{Ferm}(a \cup b) = \mathrm{Ferm}(a) \cup \mathrm{Ferm}(b)$, $\mathrm{Ferm} \circ \mathrm{Ferm} = \mathrm{Ferm}$ et $\mathrm{Ferm}(c)$ est aussi petit que possible (pour avoir des contextes aussi précis que possible).

$$
\begin{array}{ccccc}
Chem & \longrightarrow & \mathcal{P}^*(Chem) & \xrightarrow{\mathrm{Ferm}} & Ctx \\
{\scriptstyle\phi}\downarrow & & {\scriptstyle\phi^P}\downarrow & & {\scriptstyle\phi^C}\downarrow \\
Chem & \longrightarrow & \mathcal{P}^*(Chem) & \xrightarrow{\mathrm{Ferm}} & Ctx
\end{array}
$$

On peut montrer que la solution est

- $\mathrm{Ferm}(c) = c \cup \{c_1.\perp | c_1.c_2 \in c\}$
- $c_{|\to} = \{c' | \mathrm{app}.c' \in c\} \cup \{\perp\}$.
- $c_{|g} = \{c' | \mathrm{g}.c' \in c \text{ ou } c' = \top \text{ si } \top \in c \text{ ou } c' = \perp\}$. On définit de même $|d$
- $\mathrm{g}.c = \{\mathrm{g}.c' | c' \in c\} \cup \{\perp\}$. On définit $\mathrm{d}.c$ de la même façon.
- $\mathrm{app}.c = \{\mathrm{app}.c' | c' \in c\} \cup \{\perp\}$.

La propriété $\mathrm{Ferm} \circ \phi^P = \phi^C \circ \mathrm{Ferm}$ montre qu'il suffit d'appliquer l'opérateur Ferm à la fin du calcul des contextes. On l'omettra dans la suite. Le chemin c et le contexte $\mathrm{Ferm}(\{c\})$ seront considérés comme identiques.

Les domaines utilisés sont les suivants :

- Contextes : $c \in Ctx = \mathrm{Ferm}(\mathcal{P}^*(Chem))$.
- Indices de paramètres : $i \in Ind = \mathbb{N}$.
- Le domaine abstrait d'une valeur ou d'une expression est $Exp^{\#} = Val^{\#} = Ctx \to (Var \cup Ind) \to Ctx$
- Environnement : $\rho \in Env^{\#} = Var \to Val^{\#}$.
- Fonction sémantique abstraite : $E[\![M]\!] : Env^{\#} \to Exp^{\#}$.
- Environnement de paramètres : $e_p \in Env_p = (Store \times Val)^*$ On spécifie en plus de la valeur du paramètre l'état dans lequel il est lié.

Définition 2.1 (Cellules accessibles) Je note $\ell \downarrow_s$ l'ensemble des cellules mémoire accessibles depuis la cellule ℓ dans l'état s. On étend cette définition aux valeurs par $v \downarrow_s = \emptyset$ si $v \in \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z}$ et aux environnements par $e \downarrow_s = \cup\{e[\![x]\!] \downarrow_s \mid x$ variable définie dans $e\}$. On note $\mathrm{loc}(s)$ l'ensemble des cellules existantes dans l'état s, $\mathrm{loc}(e_p) = \cup_{i, e_p(i) = (s_p, v_p)} v_p \downarrow_{s_p}$, $\mathrm{loc}(s, e_p) = \mathrm{loc}(s) \cup \mathrm{loc}(e_p)$.

Définition 2.2 (Ordre sur les états) On dit qu'un état s' est après s si sA^*s' où A^* est la fermeture réflexive et transitive de la relation A définie comme suit : $sAs' \Leftrightarrow s' = s[\ell \mapsto v]$ où $\ell \in Loc$; si $\ell \in \mathrm{Dom}(s)$, ℓ est mutable (c'est-à-dire a été créée par une référence); $v \in \mathrm{val}(s) \cup \mathrm{val}(s) \times \mathrm{val}(s) \cup (Env \to Exp) \times (Var \to \mathrm{val}(s)) \times Var$ et $\mathrm{val}(s) = \mathrm{loc}(s) \cup \{\mathrm{true}, \mathrm{false}, \mathrm{NIL}\} \cup \mathbb{Z}$. Intuitivement, s' peut être obtenu après s pendant l'exécution d'une programme (correct).

Définition 2.3 (Cellules stockées) Soit $a : \mathit{Store} \to \mathcal{P}(\mathit{Loc})$ telle que $a(s)$ désigne l'ensemble des cellules apparues dans un membre droit d'affectation depuis l'état s_0 au début de l'exécution du programme jusqu'à l'état s. Plus précisément, $a([]) = \emptyset$, $a(s[\ell \mapsto v]) = a(s) \cup \ell \downarrow_{s'}$ si $\ell \in \mathrm{loc}(s)$ (une affectation) et $a(s[\ell \mapsto v]) = a(s)$ sinon (une allocation). Ces cellules comprennent toutes les locations des structures stockées (pas seulement leur sommet). Alors $a(s)$ est clos par $\downarrow_s$.

On note $a(s, s') = a(s') - a(s)$.

Remarque : Pour pouvoir définir cette notion à partir de l'état s uniquement, il faut supposer qu'il conserve l'histoire de ses modifications. C'est pour cela qu'on a défini les états comme des listes d'association et non comme des fonctions. Sinon, on peut aussi utiliser une sémantique à petits pas et définir cette notion à partir de la trace d'exécution.

Lemme 2.4 *$a(s, s'') = a(s, s') \cup a(s', s'')$ si s'' est après s' qui est après s.*

Preuve : Evident. □

Définition 2.5 (Concrétisation des contextes)

Soit cell : $\mathit{Store} \times \mathit{Val} \times \mathit{Ctx} \times \mathit{Env}_p \to \mathcal{P}(\mathit{Loc})$ telle que $\mathrm{cell}(s, v, c, e_p)$ représente l'ensemble des cellules qui s'échappent quand on conserve les cellules de contexte c de la valeur v dans l'état s. L'environnement e_p détermine les paramètres qui seront passés à la fonction v (il est vide si v n'est pas une fonction).

Soit cell_e : $\mathit{Store} \times \mathit{Exp} \times \mathit{Ctx} \times \mathit{Env}_p \to \mathcal{P}(\mathit{Loc})$ telle que $\mathrm{cell}(s, f, c, e_p)$ représente l'ensemble des cellules qui s'échappent quand on évalue l'expression de sémantique f dans l'état s et qu'on conserve les cellules de contexte c du résultat.

On définit cell par cas comme suit si c est un chemin :

- $\mathrm{cell}(s, v, c, e_p) = \cup_{s' \text{ après } s}\, \mathrm{cell}_0(s', v, c, e_p)$. Ceci est utile pour assurer que si on estime l'échappement dans un état donné, l'analyse restera correcte dans un état futur (par exemple, quand on utilisera l'expression analysée).

- Si $c = \bot$, $\mathrm{cell}_0(s, v, c, []) = \emptyset$.

- Si $c = \top$,

$$\mathrm{cell}_0(s, v, c, []) = v \downarrow_s - a(s) \tag{2.1}$$

  Ceci représente l'ensemble des cellules dénotées par ℓ dans l'état s, à ceci près qu'on retire $a(s)$ pour compenser l'effet des opérations impératives.

- Si $c = \mathrm{g}.c_1$, $s(\ell) = (\ell_1, \ell_2)$ (couples ou listes) ou $s(\ell) = \ell_1$ (références),

$$\mathrm{cell}_0(s, \ell, c, e_p) = \mathrm{cell}(s, \ell_1, c_1, e_p) \tag{2.2}$$

  De même si $c = \mathrm{d}.c_2$.

- si $c = \mathrm{app}.c_1$, $s(\ell) = (m, e', x)$ (fermetures) :

$$\mathrm{cell}_0(s, \ell, c, (s, p_1) : e_p) = \mathrm{cell}_e(s, m(e'[x \mapsto p_1]), c_1, e_p) \tag{2.3}$$

  Si l'état s ne correspond pas à celui indiqué pour le paramètre p_1 dans l'environnement de paramètres, le résultat est $\emptyset$.

- si le chemin demandé ne correspond pas à la donnée (le chemin commence par d pour une référence, par g ou d pour une fermeture, par app pour autre chose qu'une fermeture), le résultat est $\emptyset$.

- si $f\ s = \bot$, $\mathrm{cell}_e(s, f, c, e_p) = \emptyset$;

– si $f\ s = \mathrm{up}(v, s')$, $\mathrm{cell}_e(s, f, c, e_p) = (a(s, s') \cup \mathrm{cell}(s', v, c, e_p)) \cap (\mathrm{loc}(s, e_p))$.

Si c est un ensemble de chemins, on fait simplement l'union :

$$\mathrm{cell}(s, \ell, c, e_p) = \cup\{\mathrm{cell}(s, \ell, c_1, e_p) | c_1 \in c\}$$

Intuitivement, on descend dans les niveaux de deux façons différentes : pour des structures de données, on suit simplement les pointeurs, tandis que pour des fonctions, on leur applique un paramètre pris dans l'environnement de paramètres. On arrête la descente quand le chemin est $\top$ ou $\bot$.

Lemme 2.6 *Si s' est après s dans la trace d'exécution,* $\mathrm{cell}(s', v, c, e_p) \subseteq \mathrm{cell}(s, v, c, e_p)$.

Preuve : Évident :

$$\begin{aligned}\mathrm{cell}(s', v, c, e_p) &= \cup_{s'' \text{ après } s'} \mathrm{cell}_0(s'', v, c, e_p)\\ &\subseteq \cup_{s'' \text{ après } s} \mathrm{cell}_0(s'', v, c, e_p)\\ &\subseteq \mathrm{cell}(s, v, c, e_p)\end{aligned}$$

□

Lemme 2.7 *Si* $[\![M]\!]es = \mathrm{up}(\ell, s')$, $(a(s, s') \cup \ell \downarrow_{s'}) \cap \mathrm{loc}(s) \subseteq e \downarrow_s$. *Ceci exprime que seules les cellules accessibles depuis l'environnement peuvent être accédées.*

Preuve : En observant la sémantique, on peut remarquer que toutes les cellules qui apparaissent dans un calcul sont celles rendues par $e[\![x]\!]$ où $x \in \mathit{Var}$, $s(l)$ où ℓ est une cellule et $\mathrm{newloc}(s, x)$. Toutes ces cellules sont donc dans $L(e, s) = e \downarrow_s \cup (\mathit{Loc} - \mathrm{loc}(s))$. Mais cet ensemble de cellules ne dépend que de e et s, et les seules opérations qui modifient e ou s sont $e' = e[x \mapsto \ell]$, $s' = s[\ell \mapsto v]$ ($\mathrm{up}(\ell, s') = \mathrm{newloc}(s, v)$ est un cas particulier du précédent) où $\ell \in L(e, s)$, $v \in V(e, s) \cup V(e, s) \times V(e, s) \cup (\mathit{Env} \to \mathit{Exp}) \times \{e_{|E}\} \times \mathit{Var}$ et $V(e, s) = L(e, s) \cup \{\mathrm{NIL}, \mathrm{true}, \mathrm{false}\} \cup \mathbb{Z}$. Alors $L(e', s') \subset L(e, s)$. Donc toutes les cellules qui apparaissent sont dans $L(e, s)$ ce qui est le résultat voulu. □

Lemme 2.8 $\mathrm{cell}(s, v, c, e_p) \subseteq v \downarrow_s \cup \mathrm{loc}(e_p) - a(s)$ *et de même pour* cell_0.

Preuve : Par récurrence sur c.

–

$$\begin{aligned}\mathrm{cell}(s, v, c, e_p) &= \cup_{s'} \mathrm{cell}_0(s', v, c, e_p)\\ &\subseteq \cup_{s'} v \downarrow_{s'} \cup \mathrm{loc}(e_p) - a(s') \text{ par hypothèse sur } \mathrm{cell}_0\\ &\subseteq v \downarrow_s \cup \mathrm{loc}(e_p) - a(s)\end{aligned}$$

car, si une cellule est dans $v \downarrow_{s'}$, mais pas dans $v \downarrow_s$, elle a été stockée (dans v) entre les états s et s', donc $v \downarrow_{s'} - a(s') \subseteq v \downarrow_s - a(s)$.

– Le seul autre cas non trivial est celui de l'application :

$$\begin{aligned}\mathrm{cell}_0(s, \ell, \mathrm{app}.c, (s, p_1) : e_p) &= (a(s, s') \cup \mathrm{cell}(s', v, c, e_p)) \cap \mathrm{loc}(s, e_p)\\ &\subseteq (a(s, s') \cup (v \downarrow_{s'} \cup \mathrm{loc}(e_p) - a(s'))) \cap \mathrm{loc}(s, e_p)\\ &\quad \text{par hypothèse de récurrence sur } c\\ &\subseteq (e[x \mapsto p_1]) \downarrow_s \cup \mathrm{loc}(e_p) - a(s)\\ &\quad \text{par le lemme 2.7}\\ &\subseteq \ell \downarrow_s \cup \mathrm{loc}((s, p_1) : e_p) - a(s)\end{aligned}$$

□

Lemme 2.9 *Si c ne commence pas par une application et v n'est pas mutable, ou bien si c ne contient pas d'application, l'union sur s' est inutile et* $\mathrm{cell} = \mathrm{cell}_0$. *En particulier,* $\mathrm{cell}(s, v, \top, []) = v \downarrow_s - a(s)$

Preuve :

- Si $c = \top$,
$$\text{cell}(s, v, \top, []) = \cup_{s' \text{ après } s} v \downarrow_{s'} - a(s')$$
Or, si une cellule est dans $v \downarrow_{s'}$, mais pas dans $v \downarrow_s$, elle a été stockée (dans v) entre les états s et s', donc $v \downarrow_{s'} - a(s') \subseteq v \downarrow_s - a(s)$, ce qui donne le résultat voulu.
- Si $c = \bot$, $\text{cell}(s, v, \bot, []) = \cup_{s' \text{ après } s} \emptyset = \emptyset$.
- Si $c = \text{g}.c'$, si $s'(v) = (\ell, \ell')$ ou $s'(v) = \ell$, $\text{cell}(s, v, \text{g}.c', e_p) = \cup_{s' \text{ après } s} \text{cell}(s', \ell, c', e_p) = \text{cell}(s, \ell, c', e_p)$ d'après le lemme 2.6. Si v n'a pas été modifiée entre s et s', $s(v) = s'(v) = (\ell, \ell')$ ou ℓ, donc on a le résultat voulu. Si c ne contient pas d'application, et que v a été modifiée entre s et s', $\text{cell}(s', \ell, c', e_p) \subseteq \ell \downarrow_{s'} - a(s') = \emptyset$ (lemme 2.8).

□

Définition 2.10 Soit dest : $(Store \times Val) \times Ctx \to \mathcal{P}(Loc)$ définie par
$$\text{dest}((s, v), c) = \cup_{e_p \in Env_p} \text{cell}(s, v, c, e_p) - \text{loc}(e_p)$$

Remarque : Si v est non-fonctionnelle, la soustraction des cellules de e_p et l'union sur e_p sont inutiles (car e_p est toujours vide).

Lemme 2.11 $\text{dest}((s, v), c) \subseteq v \downarrow_s$

Preuve : Le résultat se déduit immédiatement du lemme 2.8. □

Lemme 2.12 $\text{dest}((s, v), a) \cup \text{dest}((s, v), b) = \text{dest}((s, v), a \sqcup b)$

Preuve : Ceci résulte immédiatement de la définition de cell sur un ensemble de chemins comme union des valeurs de cell sur chaque chemin. □

Théorème 2.13 *Considérons une instance d'une variable x dans une construction de la forme :* `let` x `=` M `in` N *ou* `let rec` $x(y)$ `=` M `in` N*. Soit s_x l'état au début de l'évaluation de N, e l'environnement passé à N. Supposons que la cellule ℓ au sommet de x soit nouvelle au moment de la définition de x. Si $\ell \notin \text{cell}_e(s_x, [\![N]\!]e, \top, [])$, alors ℓ est allouable en pile.*

Preuve : Si $[\![N]\!]e = \text{up}(r, s'_x)$, on a :
$$\begin{aligned}\text{cell}_e(s_x, [\![N]\!]e, \top, []) &= (a(s_x, s'_x) \cup \text{cell}(s'_x, r, \top, [])) \cap \text{loc}(s_x) \\ &= (a(s_x, s'_x) \cup r \downarrow_{s'_x}) \cap \text{loc}(s_x) - a(s_x)\end{aligned}$$
Comme $\ell \notin a(s_x)$ et $\ell \in \text{loc}(s_x)$, $\ell \notin a(s_x, s'_x) \cup r \downarrow_{s'_x}$. ℓ n'est pas accessible par le résultat r car $\ell \notin r \downarrow_{s'_x}$, et ℓ n'est pas accessible par les autres variables de l'environnement car elle n'a pas subi d'affectation. Donc ℓ n'est accessible que par l'intermédiaire de x dans l'état s'_x. Quand on supprime x, ℓ n'est donc plus accessible et peut être libérée. □

Définition 2.14 (Correction) On définit la correction en plusieurs étapes :

- **Pour une valeur v :** ϕ est une abstraction correcte de la sémantique d'une valeur v dans l'environnement e si
$$\begin{aligned}&\text{corr}_v : Env \times Store \times Val \times Val^{\#} \to \{\text{vrai}, \text{faux}\} \\ &\text{corr}_v(e, s, v, \phi) \Leftrightarrow \forall c \in Ctx, \forall e_p \in Env_p, \\ &\quad \text{cell}(s, v, c, e_p) \subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), \phi x) \cup \cup_{i \in Ind} \text{dest}(e_p(i), \phi i)\end{aligned} \tag{2.4}$$
où $e(x)$ est un couple (état au moment de la définition de x, valeur de x). Ceci est une sorte de sémantique instrumentée où on a remplacé $e[x \mapsto \ell]$ par $e[x \mapsto (s, \ell)]$ pour mémoriser aussi l'état au moment de la création de la variable.

- **Pour une expression** M **:** ϕ est une abstraction correcte de la sémantique $f = [\![M]\!]$ d'un terme M dans l'environnement e si

$$\begin{aligned}
&\mathrm{corr}_e : \mathit{Env} \times \mathit{Store} \times \mathit{Exp} \times \mathit{Exp}^{\#} \to \{\mathrm{vrai}, \mathrm{faux}\} \\
&\mathrm{corr}_e(e, s, f, \phi) \Leftrightarrow \forall c \in \mathit{Ctx}, \forall e_p \in \mathit{Env}_p, \\
&\quad \mathrm{cell}_e(s, f, c, e_p) \subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), \phi x) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e_p(i), \phi i)
\end{aligned} \tag{2.5}$$

  Intuitivement, on marque la partie $\mathrm{cell}_e(s, f, c, e_p)$ du résultat réprésentée par le contexte c et les cellules stockées, qui sont considérées comme s'échappant, et on cherche les cellules marquées dans les variables et paramètres.

- On dit qu'une fonction $f \in \mathit{Val}^{\#} = \mathit{Exp}^{\#}$ est ρ-transitive si $fcx \sqcup \rho[\![y]\!]\bot x \sqsupseteq \rho[\![y]\!](fcy)x$ si y est dans la portée lexicale de x. Ceci exprime que f tient compte des échappements qui se produisent par des déclarations de variables intermédiaires. Par exemple, dans l'expression `let x =` M `in let y =` N `in` P, si $f = E[\![P]\!]\rho$ est l'analyse de P, et si on garde la partie représentée par le contexte c du résultat, l'échappement de `y` sera représenté par $c_1 = fc\mathtt{y}$, cet échappement va causer l'échappement $c_2 = \rho[\![\mathtt{y}]\!]c_1\mathtt{x}$ de `x`. D'autre part, ce qui s'échappe de `x` est ce qui s'échappe à travers des affectations pendant l'évaluation de N, $\rho[\![\mathtt{y}]\!]\bot\mathtt{x}$, union ce qui s'échappe par le résultat, $fc\mathtt{x}$. La ρ transitivité affirme que cet échappement prend en compte l'échappement c_2 à travers `y`.

- **Pour l'analyse entière :** L'analyse est correcte si elle donne des résultats corrects quand on lui passe un environnement correct, c'est-à-dire si :

$$\begin{aligned}
&\mathrm{corr} : (\mathit{Env} \to \mathit{Exp}) \times (\mathit{Env}^{\#} \to \mathit{Exp}^{\#}) \to \{\mathrm{vrai}, \mathrm{faux}\} \\
&\mathrm{corr}(f, \phi) \Leftrightarrow \forall s, \forall e \in \mathit{Env}, \forall \rho \in \mathit{Env}^{\#}, \forall y \in \mathrm{Dom}(e), \mathrm{corr}_v(e, s, e[\![y]\!], \rho[\![y]\!]) \text{ et } \rho[\![y]\!] \text{ est} \\
&\quad \rho\text{-transitive } \Rightarrow \mathrm{corr}_e(e, s, fe, \phi\rho) \text{ et } \phi\rho \text{ est } \rho\text{-transitive}
\end{aligned} \tag{2.6}$$

  où $\phi\rho c$ est définie sur $\mathrm{Dom}(e) \cup \mathit{Ind}$. Le but sera de montrer que $\mathrm{corr}([\![M]\!], E[\![M]\!])$.

Exemple 2.15 Par exemple,

```
let x = (true :: []) :: [] in
let y = hd x in
  y
```

Soient e et s l'environnement e et l'état lors de l'évaluation de `y`. Soient x_1 et $x_2 = y$ les cellules telles que $e[\![\mathtt{x}]\!] = x_1$, $e[\![\mathtt{y}]\!] = y$, $s(x_1) = (x_2, \mathrm{NIL})$, $s(x_2) = \mathrm{true}, \mathrm{NIL})$. On cherche $\phi\top\mathtt{x}$ pour que $\mathrm{corr}_v(e, s, y, \phi)$.

$\mathrm{cell}(s, y, \top, []) = \{x_2\}$, $\mathrm{dest}((s, x_1), \mathrm{g}.\top) = \{x_2\}$ et $\mathrm{dest}((s, x_1), \mathrm{g.g}.\top) = \emptyset$ donc $\phi\top\mathtt{x} = \mathrm{g}.\top$.

Définition 2.16 (Concrétisation) On peut se placer dans le cadre de la structure de concrétisation de [CC92].

On utilise comme treillis concret $\mathcal{P}(\mathit{Exp})$ muni de l'inclusion, et comme treillis abstrait $\mathit{Exp}^{\#}$ muni de l'ordre $\sqsubseteq$ point par point sur les fonctions.

On étend alors la relation de correction corr au treillis concret par $\mathrm{corr}(e, \phi) \Leftrightarrow \forall f \in e, \mathrm{corr}(f, \phi)$.

On définit la fonction de concrétisation :

$$\begin{aligned}
&\gamma : \mathit{Exp}^{\#} \to \mathcal{P}(\mathit{Exp}) \\
&\gamma(\phi) = \{f \mid \mathrm{corr}(f, \phi)\}
\end{aligned} \tag{2.7}$$

autrement dit $\gamma(\phi)$ est le plus grand élément correct du treillis concret.

γ est croissante et $\gamma(\lambda cx.\mathit{Chem}) = \mathit{Exp}$.

Définition 2.17 (Complétude (forte)) L'analyse est complète si elle est la plus petite qui soit correcte, c'est-à-dire :

$$\mathrm{corr}([\![M]\!], \phi) \Rightarrow E[\![M]\!] \sqsubseteq \phi \tag{2.8}$$

ou encore

$$[\![M]\!] \in \gamma(\phi) \Rightarrow E[\![M]\!] \sqsubseteq \phi \tag{2.9}$$

Ceci est une définition très forte de la complétude, qui n'est pas réalisée ici pour trois raisons :

- Le `if` : dans l'expression `if true then [] else x`, l'analyse dit que `x` peut s'échapper ce qui ne sera en fait jamais le cas.
- Les fonctions d'ordre supérieur : dans l'expression `map (fun x -> []) l`, l'analyse ne trouve pas que les éléments de `l` ne s'échappent pas.
- Les fonctions stockées dans des structures de données : dans `(hd [fun x -> true]) z`, `z` ne s'échappe pas, mais l'analyse ne le trouve pas. On a un problème analogue avec : `(! (ref (fun x -> true))) z`, mais ici il est encore plus délicat car une affectation intermédiaire peut changer la fonction stockée dans la référence.

Remarque : L'existence d'un plus petit élément à l'ensemble des contextes corrects n'est pas évidente. Par exemple, si on a un couple $x = (a, a)$, et que a s'échappe, g.$\top$ et d.$\top$ sont deux éléments minimaux incomparables de l'ensemble des contextes corrects. Cependant, si on regarde l'expression qui utilise x, pour une valeur de x différente de (a, a), on saura quelle partie du couple s'échappe effectivement. En particulier, le plus petit élément existe quand il n'y a initialement aucun alias (plusieurs chemins menant à la même cellule). La difficulté est de se convaincre que l'échappement décrit dans ce cas particulier convient aussi dans le cas général. C'est pourquoi on n'a pas défini d'abstraction $\alpha : \mathcal{P}(\mathit{Exp}) \to \mathit{Exp}^{\#}$.

Définition 2.18 (Complétude) L'analyse est complète si elle est la plus petite qui soit correcte et compositionnelle (c'est-à-dire que l'analyse d'une expression dépend de l'analyse des sous-expressions, mais pas des sous-expressions elles-mêmes). Considérons une construction du langage $M = C(M_1, ..., M_n)$. Sa sémantique est $[\![M]\!] = f_C([\![M_1]\!], ..., [\![M_n]\!])$ et l'analyse correspondante est $E[\![M]\!] = \phi_C(E[\![M_1]\!], ..., E[\![M_n]\!])$

$$\begin{aligned}(\forall M_i, \mathrm{corr}(f_C([\![M_1]\!], ..., [\![M_n]\!])]\!], \phi(E[\![M_1]\!], ..., E[\![M_n]\!])))\\ \Rightarrow \forall M_i, \phi_C(E[\![M_1]\!], ..., E[\![M_n]\!]) \sqsubseteq \phi(E[\![M_1]\!], ..., E[\![M_n]\!])\end{aligned} \tag{2.10}$$

ou dans la formulation avec concrétisation, ϕ_C est correcte vis à vis de f_C si et seulement si $\gamma \circ f_C \sqsubseteq \phi_C \circ (\gamma, ..., \gamma)$. La complétude exprime que ϕ_C est minimale pour cette propriété, c'est-à-dire : $\gamma \circ f_C \sqsubseteq \phi \circ (\gamma, ..., \gamma) \Rightarrow \phi_C \sqsubseteq \phi$.

On note $f = \{x \mapsto f(x), ...\}$ une fonction $f : (\mathit{Var} \cup \mathit{Ind}) \to \mathit{Ctx}$ définie point par point. f vaut $\perp$ aux points où elle n'est pas explicitement définie. $\{\}$ désigne donc la fonction constante égale à $\perp$.

Définition 2.19 On définit l'analyse E sur la figure 2.3 : La valeur $\rho[\![y]\!]$ associée à une variable libre y d'une expression M d'un programme $P = C[M]$ pour l'analyse $E[\![M]\!]\rho$ est celle qu'elle prend dans l'analyse de P au moment où on évalue $E[\![M]\!]$. La correction de cette valeur est prouvée pour pouvoir appliquer l'hypothèse d'induction à M. Par exemple, $\rho[\![y]\!] = \rho_\top$ si y est définie par `fun` $y \to M$

On définit E pour des ensembles de chemins par additivité :

$$E[\![M]\!]\rho c x = \cup\{E[\![M]\!]\rho c' x | c' \in c\}$$

On peut remarquer que, comme les restrictions des contextes sont les extensions par additivité des restrictions des chemins, la plupart des formules données pour des chemins se prolongent sans changement aux contextes. Seules les formules qui distinguent plusieurs cas suivant le chemin c doivent être adaptées en effectuant une borne supérieure sur tous les cas qui se trouvent dans le contexte.

$$E[\![M]\!] : \mathit{Env}^\# \to \mathit{Exp}^\#$$
$$E[\![y]\!]\rho c = \rho[\![y]\!]c$$
$$E[\![M\ N]\!]\rho c = \{i \mapsto \phi(i+1) \text{ si } i \in \mathit{Ind}, x \mapsto \phi(x) \sqcup \psi(x) \text{ si } x \in \mathit{Var}\}$$
où $\phi = E[\![M]\!]\rho(\text{app}.c)$ et $\psi = E[\![N]\!]\rho(\phi(1))$

$$E[\![\mathtt{fun}\ y \to M]\!]\rho c = \begin{cases} \bigsqcup_{z \in FV(\mathtt{fun}\ y \to M)} \rho[\![z]\!]\top & \text{si } c = \top, \\ \bot & \text{si } c = \bot, \\ \{1 \mapsto \phi(y), i \mapsto \phi(i-1) \text{ si } i > 1, x \mapsto \phi(x) \text{ si } x \in \mathit{Var}\ \} & \text{sinon.} \end{cases}$$

où $\phi = E[\![M]\!]\rho[y \mapsto \rho_\top](c_{|\to})_{|\overline{\{y\}}}$ et $\rho_\top(c) = \{i \mapsto \top, y \mapsto c\}$.

$$E[\![\mathtt{let}\ y = M\ \mathtt{in}\ N]\!]\rho c = E[\![N]\!]\rho[y \mapsto E[\![M]\!]\rho \sqcup \lambda c.\{y \mapsto c\}]c_{|\overline{\{y\}}} \sqcup E[\![M]\!]\rho\bot_{|\mathit{Var}}$$
$$E[\![\mathtt{let\ rec}\ f(y) = M\ \mathtt{in}\ N]\!]\rho c = E[\![N]\!](\text{lfp}(\lambda\rho_1.\rho[f \mapsto E[\![\mathtt{fun}\ y \to M]\!]\rho_1 \sqcup \lambda c.\{f \mapsto c\}])c_{|\overline{\{f\}}}$$
$$E[\![\mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ P]\!]\rho c = E[\![M]\!]\rho\bot_{|\mathit{Var}} \sqcup E[\![N]\!]\rho c \sqcup E[\![P]\!]\rho c$$
$$E[\![M::N]\!]\rho c = E[\![M]\!]\rho(c_{|g}) \sqcup E[\![N]\!]\rho(c_{|d})$$
$$E[\![(M,N)]\!]\rho c = E[\![M]\!]\rho(c_{|g}) \sqcup E[\![N]\!]\rho(c_{|d})$$
$$E[\![M;N]\!]\rho c = E[\![M]\!]\rho\bot_{|\mathit{Var}} \sqcup E[\![N]\!]\rho c$$

$$\rho[\![\mathtt{ref}]\!]c = \begin{cases} \{\} & \text{si } c = \bot \text{ ou } \top, \\ \{1 \mapsto c_{|\to|g}, i \mapsto \top \text{ si } i > 1\} & \text{sinon.} \end{cases}$$

$$\rho[\![:=]\!]c = \begin{cases} \{\} & \text{si } c = \bot, \text{app}.\bot \text{ ou } \top \\ \{1 \mapsto \top\} & \text{si } c = \text{app}.\top, \\ \{1 \mapsto \bot, 2 \mapsto \top\} & \text{si } c = \text{app.app}.\bot \text{ ou app.app}.\top. \end{cases}$$

$$\rho[\![a]\!]c = \{\} \text{ où } a \in \{\mathtt{true}, \mathtt{false}, []\} \cup \mathbb{Z}$$

$$\rho[\![\mathtt{hd}]\!]c = \begin{cases} \{\} & \text{si } c = \bot \text{ ou } \top, \\ \{1 \mapsto \text{g}.c_{|\to}, i \mapsto \top \text{ si } i > 1\} & \text{sinon.} \end{cases} \quad \text{De même pour } \mathtt{fst} \text{ et } \mathtt{!}.$$

$$\rho[\![\mathtt{tl}]\!]c = \begin{cases} \{\} & \text{si } c = \bot \text{ ou } \top, \\ \{1 \mapsto \text{d}.c_{|\to}, i \mapsto \top \text{ si } i > 1\} & \text{sinon.} \end{cases} \quad \text{De même pour } \mathtt{snd}.$$

$$\rho[\![\mathtt{null}]\!]c = \{\}$$

FIG. 2.3 – *Analyse E*

Exemple 2.20 Considérons l'exemple suivant qui utilise la fonction `map` :

```
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 ...
```

Soit M l'expression `let rec map = ... in map g a`. Soit ρ_0 l'environnement qui ne contient que les variables prédefinies.

$$\rho = \rho_0[\mathtt{a} \mapsto (E[\![\mathtt{1 :: 2 :: 3 :: []}]\!]\rho_0 \sqcup \lambda c.\{\mathtt{a} \mapsto c\}), \mathtt{g} \mapsto (E[\![\mathtt{fun\ x\ ->\ x+1}]\!]\rho_0 \sqcup \lambda c.\{\mathtt{g} \mapsto c\})]$$

$$E[\![M]\!]\rho c = E[\![\mathtt{map\ g\ a}]\!](\mathrm{lfp}(\lambda\rho_1.\rho[\mathtt{map} \mapsto E[\![\mathtt{fun\ f\ l\ ->\ if\ \ldots}]\!]\rho_1 \sqcup \lambda c.\{\mathtt{map} \mapsto c\}]))c$$

avec

$$\begin{aligned}
E[\![\mathtt{fun\ f\ l\ ->\ if\ \ldots}]\!]\rho_1 c 1 &= E[\![\mathtt{fun\ l\ ->\ if\ \ldots}]\!]\rho_1[\mathtt{f} \mapsto \rho_\top](c_{|\to})\mathtt{f} \\
&= E[\![\mathtt{if}\ldots]\!]\rho_1[\mathtt{f} \mapsto \rho_\top, \mathtt{l} \mapsto \rho_\top](c_{|\to|\to})\mathtt{f} \\
&= E[\![\mathtt{null\ l}]\!]\rho_2 \bot \mathtt{f} \sqcup E[\![\mathtt{[]}]\!]\rho_2(c_{|\to|\to})\mathtt{f} \\
&\quad \sqcup E[\![\mathtt{(f\ (hd\ l))} :: \mathtt{(map\ f\ (tl\ l))}]\!]\rho_2(c_{|\to|\to})\mathtt{f} \\
&= \emptyset \sqcup \emptyset \sqcup E[\![\mathtt{(f\ (hd\ l))}]\!]\rho_2(c_{|\to|\to|g})\mathtt{f} \sqcup E[\![\mathtt{(map\ f\ (tl\ l))}]\!]\rho_2(c_{|\to|\to|d})\mathtt{f} \\
&= E[\![\mathtt{f}]\!]\rho_2(\mathrm{app}.c_{|\to|\to|g})\mathtt{f} \sqcup E[\![\mathtt{hd\ l}]\!]\rho_2(E[\![\mathtt{f}]\!]\rho_2((\mathrm{app}.c_{|\to|\to|g})1)\mathtt{f} \\
&\quad \sqcup E[\![\mathtt{map}]\!]\rho_2 c'\mathtt{f} \sqcup E[\![\mathtt{f}]\!]\rho_2(E[\![\mathtt{map}]\!]\rho_2 c' 1)\mathtt{f} \sqcup E[\![\mathtt{hd\ l}]\!]\rho_2(E[\![\mathtt{map}]\!]\rho_2 c' 2)\mathtt{f} \\
&= (\mathrm{app}.c_{|\to|\to|g}) \sqcup \emptyset \sqcup \rho_2[\![\mathtt{map}]\!]c'\mathtt{f} \sqcup \rho_2[\![\mathtt{f}]\!](\rho_2[\![\mathtt{map}]\!]c'1)\mathtt{f} \sqcup \emptyset \\
&= (\mathrm{app}.c_{|\to|\to|g}) \sqcup \rho_2[\![\mathtt{map}]\!]c'\mathtt{f} \sqcup \rho_2[\![\mathtt{map}]\!]c'1
\end{aligned}$$

où $\rho_2 = \rho_1[\mathtt{f} \mapsto \rho_\top, \mathtt{l} \mapsto \rho_\top]$ et $c' = \mathrm{app.app}.c_{|\to|\to|d}$. De même,

$$\begin{aligned}
E[\![\mathtt{fun\ f\ l\ ->\ if\ \ldots}]\!]\rho_1 c 2 &= E[\![\mathtt{if}\ldots]\!]\rho_2(c_{|\to|\to})\mathtt{l} \\
&= E[\![\mathtt{null}]\!]\rho_2 \bot \mathtt{l} \sqcup E[\![\mathtt{[]}]\!]\rho_2(c_{|\to|\to})\mathtt{l} \sqcup (E[\![\mathtt{f\ (hd\ l)}]\!]\rho_2(c_{|\to|\to|g})\mathtt{l} \\
&\quad \sqcup E[\![\mathtt{map\ f\ (tl\ l)}]\!]\rho_2(c_{|\to|\to|d})\mathtt{l} \\
&= E[\![\mathtt{hd\ l}]\!]\rho_2(E[\![\mathtt{f}]\!]\rho_2(\mathrm{app}.c_{|\to|\to|g})1)\mathtt{l} \sqcup E[\![\mathtt{map}]\!]\rho_2 c\mathtt{l} \\
&\quad \sqcup E[\![\mathtt{tl\ l}]\!]\rho_2(E[\![\mathtt{map}]\!]\rho_2 c' 2)\mathtt{l} \\
&= E[\![\mathtt{hd\ l}]\!]\rho_2 \top \mathtt{l} \sqcup \rho_2[\![\mathtt{map}]\!]c'\mathtt{l} \sqcup E[\![\mathtt{tl\ l}]\!]\rho_2(\rho_2[\![\mathtt{map}]\!]c2)\mathtt{l} \\
&= \{\mathrm{g}.\top\} \sqcup \rho_2[\![\mathtt{map}]\!]c'1 \sqcup \mathrm{d}.\rho_2[\![\mathtt{map}]\!]c'2
\end{aligned}$$

On a aussi : $E[\![\mathtt{fun\ f\ l\ ->\ if\ \ldots}]\!]\rho_1 c\mathtt{map} = c' \sqcup \rho_2[\![\mathtt{map}]\!]c'\mathtt{map}$ et $E[\![\mathtt{fun\ f\ l\ ->\ if\ \ldots}]\!]\rho_1 c x = \emptyset$ si $x \in \mathit{Var} - \{\mathtt{map}\}$. Un calcul de point fixe montre alors que

$$\rho_1[\![\mathtt{map}]\!]c = \{\mathtt{map} \mapsto c, 1 \mapsto \mathrm{app}.c_{|\to|\to|d^*|g}, 2 \mapsto \mathrm{d}^*.\mathrm{g}.\top\}$$

(d^* désigne l'ensemble des mots constitués d'un nombre quelconque de d) donc

$$E[\![M]\!]\rho\{\top\}\mathtt{a} = \rho_1[\![\mathtt{map}]\!]\{\mathrm{app.app}.\top\}2 = \mathrm{d}^*.\mathrm{g}.\top$$

et

$$E[\![M]\!]\rho\{\top\}\mathtt{g} = \rho_1[\![\mathtt{map}]\!]\{\mathrm{app.app}.\top\}1 = \{\mathrm{app}.\top\}$$

donc on peut mettre le sommet de `a` et la fermeture `g` en pile (ici, on peut aussi mettre `map` en pile mais dans un vrai programme, `map` est une fonction de bibliothèque et donc une déclaration globale qui n'est jamais mise en pile).

Lemme 2.21 *Si* $\forall y, \rho[\![y]\!]$ *est* ρ*-transitive,* $E[\![M]\!]\rho$ *aussi.*

Preuve : Par induction sur M :

- Cas y : par hypothèse sur ρ.
- Cas $M\ N$: si x est une variable,

$$\begin{aligned}
E[\![M\ N]\!]\rho c x \sqcup \rho[\![y]\!]\bot x &= E[\![M]\!]\rho(\mathrm{app}.c)x \sqcup E[\![N]\!]\rho(E[\![M]\!]\rho(\mathrm{app}.c)1)x \sqcup \rho[\![y]\!]\bot x \\
&\supseteq \rho[\![y]\!](E[\![M]\!]\rho(\mathrm{app}.c)y)x \sqcup \rho[\![y]\!](E[\![N]\!]\rho(E[\![M]\!]\rho(\mathrm{app}.c)1)y)x \\
&\qquad \text{par hypothèse de récurrence sur } M \text{ et sur } N \\
&\supseteq \rho[\![y]\!](E[\![M]\!]\rho(\mathrm{app}.c)y \sqcup E[\![N]\!]\rho(E[\![M]\!]\rho(\mathrm{app}.c)1)y)x \\
&\supseteq \rho[\![y]\!](E[\![M\ N]\!]\rho c y)x
\end{aligned}$$

- Cas $\texttt{fun}\ y \to M$: Montrons tout d'abord que $\rho_\top$ est ρ-transitive. En effet, $\rho[\![z]\!](\rho_\top c z)x \subseteq \rho_\top c x \sqcup \rho[\![z]\!]\bot x$ car si $z \neq y$, $\rho_\top c z = \bot$ donc on a le résultat et si $z = y$, $\rho[\![z]\!] = \rho_\top$ et $\rho_\top c z = c$ donc on a encore le résultat.

  Alors

$$\begin{aligned}
E[\![\texttt{fun}\ y \to M]\!]\rho c x \sqcup \rho[\![z]\!]\bot x &= E[\![M]\!]\rho[y \mapsto \rho_\top](c_{|\to})x \sqcup \rho[\![z]\!]\bot x \\
&\supseteq \rho[\![z]\!](E[\![M]\!]\rho[y \mapsto \rho_\top](c_{|\to})z)x \text{ par hypothèse de récurrence sur } M \\
&\supseteq \rho[\![z]\!](E[\![\texttt{fun}\ y \to M]\!]\rho c z)x
\end{aligned}$$

- Les autres cas se traitent de manière analogue.

□

Lemme 2.22 $\mathrm{corr}_v(e, s, v, \phi) \Rightarrow \mathrm{corr}_v(e, s', v, \phi)$ *si* s' *est après* s *dans la trace d'exécution.*

Preuve : Par le lemme 2.6, $\mathrm{cell}(s', v, c, e_p) \subseteq \mathrm{cell}(s, v, c, e_p)$. Ceci donne l'implication voulue sur corr_v en utilisant la définition de corr_v. □

Théorème 2.23 *L'analyse* E *est correcte, et* e *est cohérent avec* ρ *sur les variables prédéfinies :* $\mathrm{corr}_v(e, s, e[\![y]\!], \rho[\![y]\!])$.

Preuve : On fait évidemment la preuve par induction sur la syntaxe et par cas. J'omets de traiter les cas de non terminaison, qui sont évidents : rien ne s'échappe dans ces cas-là.

On fait la démonstration dans le cas où c est un chemin, on en déduit la correction dans le cas où c est un contexte :

$$\begin{aligned}
\mathrm{cell}_e(s, [\![M]\!]e, c, e_p) &= (a(s,s') \cup \mathrm{cell}(s', \ell, c, e_p)) \cap \mathrm{loc}(s, e_p) \\
&= (a(s,s') \cup \cup_{c' \in c} \mathrm{cell}(s', \ell, c', e_p)) \cap \mathrm{loc}(s, e_p) \\
&= \cup_{c' \in c}(a(s,s') \cup \mathrm{cell}(s', \ell, c', e_p)) \cap \mathrm{loc}(s, e_p) \text{ car } c \neq \emptyset \text{ puisque } \bot \in c \\
&\subseteq \cup_{c' \in c} \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho c' x) \cup \cup_{i \in Ind} \mathrm{dest}(e_p(i), E[\![M]\!]\rho c' i) \\
&\subseteq \cup_{x \in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho c x) \cup \cup_{i \in Ind} \mathrm{dest}(e_p(i), E[\![M]\!]\rho c i)
\end{aligned}$$

On note $A = \mathrm{cell}_e(s, [\![M]\!]e, c, e_p)$ quand on traite le cas de l'expression M, ou $A = \mathrm{cell}(s, v, c, e_p)$ quand on traite celui de la valeur v.

- Cas y :

$$[\![y]\!]es = \mathrm{up}(e[\![y]\!], s)$$

$$\begin{aligned}A &= (a(s,s) \cup \mathrm{cell}(s, e[\![y]\!], c, e_p)) \cap \mathrm{loc}(s, e_p)\\ &\subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), \rho[\![y]\!]cx) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), \rho[\![y]\!]ci) \text{ d'après (2.4)}\end{aligned}$$

donc

$$\mathrm{corr}_e(e, s, [\![y]\!], \rho[\![y]\!])$$

– Cas $M\ N$:

$$\begin{aligned}[\![M\ N]\!]es = \text{let } [(f', s')] &= [\![M]\!]es \text{ in let } [(v, s'')] = [\![N]\!]es' \text{ in}\\ &\text{let } (m, e', y) = s'(f') \text{ in } m(e'[y \mapsto v])s''\\ &= \mathrm{up}(l, s''')\end{aligned}$$

$$A = (a(s, s''') \cup \mathrm{cell}(s''', \ell, c, e_p)) \cap \mathrm{loc}(s, e_p)$$

D'après le lemme 2.4,

$$\begin{aligned}A &= (\mathrm{cell}(s''', \ell, c, e_p) \cup a(s, s') \cup a(s', s'') \cap a(s'', s''')) \cap \mathrm{loc}(s, e_p)\\ &= (\mathrm{cell}(s'', f', \mathrm{app}.c, (s'', v) : e_p) \cup a(s, s') \cup a(s', s'')) \cap \mathrm{loc}(s, e_p) \text{ d'après (2.3)}\\ &\subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho(\mathrm{app}.c)x) \cup \cup_{i>1} \mathrm{dest}(e_p(i-1), E[\![M]\!]\rho(\mathrm{app}.c)i) \cup (a(s', s'')\\ &\quad \cup \mathrm{dest}((s'', v), E[\![M]\!]\rho(\mathrm{app}.c)1)) \cap \mathrm{loc}(s') \text{ par hypothèse de récurrence sur } M\\ &\subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho(\mathrm{app}.c)x) \cup \cup_{i>1} \mathrm{dest}(e_p(i-1), E[\![M]\!]\rho(\mathrm{app}.c)i)\\ &\quad \cup \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho(E[\![M]\!]\rho(\mathrm{app}.c)1)x)\end{aligned}$$

car $(a(s', s'')\cup\mathrm{cell}(s'', v, c', e_p))\cap\mathrm{loc}(s', e_p) \subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho c'x)\cup\cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![N]\!]\rho c'i)$ par hypothèse de récurrence sur N, donc

$$(a(s', s'') \cup \mathrm{dest}((s'', v), c')) \cap \mathrm{loc}(s') \subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho c'x)$$

Alors, comme $\mathrm{dest}(e(x), c)$ croît en c,

$$\mathrm{corr}_e(e, s, [\![M\ N]\!], \lambda c.\lambda x. \begin{cases} E[\![M]\!]\rho(\mathrm{app}.c)x \sqcup E[\![N]\!]\rho(E[\![M]\!]\rho(\mathrm{app}.c)1)x & \text{si } x \in \mathit{Var}\\ E[\![M]\!]\rho(\mathrm{app}.c)(x+1) & \text{si } x \in \mathit{Ind}\end{cases})$$

– Cas $\mathtt{fun}\ y \to M$:

$$[\![\mathtt{fun}\ y \to M]\!]es = \mathrm{newloc}(s, ([\![M]\!], e_{|FV(\mathtt{fun}\ y\to M)}, y)) = \mathrm{up}(\ell, s')$$

Si $c = \perp$:

$$A = a(s, s') \cup \mathrm{cell}(s', \ell, \perp, []) = \emptyset$$

Si $c = \top$:

$$\begin{aligned}A &= (a(s, s') \cup \mathrm{cell}(s', \ell, \top, [])) \cap \mathrm{loc}(s)\\ &= \ell\downarrow_{s'} \cap \mathrm{loc}(s) - a(s') \text{ d'après le lemme 2.9}\\ &= \bigcup_{z\in FV(\mathtt{fun}\ y\to M)} z\downarrow_s - a(s)\\ &= \bigcup_{z\in FV(\mathtt{fun}\ y\to M)} \mathrm{cell}(s, z, \top, [])\\ &\subseteq \bigcup_{z\in FV(\mathtt{fun}\ y\to M)} \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), \rho[\![z]\!]\top x) \text{ par cohérence de } \rho\end{aligned}$$

Sinon :

$$\begin{aligned}A &= (a(s, s') \cup \mathrm{cell}(s', \ell, c, (s'', p_1) : e_p)) \cap \mathrm{loc}(s, (s'', p_1) : e_p)\\ &\subseteq \mathrm{cell}_e(s'', [\![M]\!](e_{|FV(\mathtt{fun}\ y\to M)}[y \mapsto p_1], c_{|\to}, e_p) \text{ d'après (2.3)}\end{aligned}$$

où $s''(\ell) = ([\![M]\!], e_{|FV(\texttt{fun}\ y \to M)}, y)$. On a la cohérence entre p_1 et $\rho_\top$: en effet,

$$\begin{aligned}\text{cell}(s'', p_1, c, e_p) &\subseteq \text{cell}(s'', p_1, c, e_p) - \text{loc}(e_p) \cup \cup_{i, e_p(i) = (s_p, v_p)} v_p \downarrow_{s_p} \\ &\subseteq \text{dest}((s'', p_1), c) \cup \cup_{i \in Ind} \text{dest}(e_p(i), \top)\end{aligned}$$

donc on peut appliquer l'hypothèse de récurrence à M

$$\begin{aligned}A \subseteq\ & \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!](\rho[y \mapsto \rho_\top])(c_{|\to})x) \cup \text{dest}((s'', p_1), E[\![M]\!](\rho[y \mapsto \rho_\top])(c_{|\to})y) \\ & \cup \cup_{i \in Ind} \text{dest}(e_p(i), E[\![M]\!](\rho[y \mapsto \rho_\top])(c_{|\to})i)\end{aligned}$$

donc

$$\text{corr}_e(e, s, [\![\texttt{fun}\ y \to M]\!], \lambda c.\lambda x. \begin{cases} \bigsqcup_{z \in FV(\texttt{fun}\ y \to M)} \rho[\![z]\!]\top x & \text{si } c = \top, \\ \perp & \text{si } c = \perp, \\ E[\![M]\!](\rho[y \mapsto \rho_\top])(c_{|\to})x & \text{sinon, si } x \in Var. \\ E[\![M]\!](\rho[y \mapsto \rho_\top])(c_{|\to})y & \text{sinon, si } x = 1. \\ E[\![M]\!](\rho[y \mapsto \rho_\top])(c_{|\to})(x-1) & \text{sinon, si } x > 1. \end{cases})$$

– Cas $\texttt{let}\ y = M\ \texttt{in}\ N$:

$$[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]es = \text{let } [(v, s')] = [\![M]\!]es \text{ in } [\![N]\!](e[y \mapsto v])s' = \text{up}(\ell'', s'')$$

$$\begin{aligned}A &= (a(s, s'') \cup \text{cell}(s'', \ell'', c, e_p)) \cap \text{loc}(s, e_p) \\ &= (a(s, s') \cup a(s', s'') \cup \text{cell}(s'', \ell'', c, e_p)) \cap \text{loc}(s, e_p) \text{ d'après le lemme 2.4} \\ &\subseteq (a(s, s') \cap \text{loc}(s)) \cup (a(s', s'') \cup \text{cell}(s'', \ell'', c, e_p)) \cap \text{loc}(s', e_p) \cap \text{loc}(s, e_p) \\ &\subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!]\rho\perp x) \cup \cup_{x \in \text{Dom}(e')} \text{dest}(e'(x), E[\![N]\!]\rho' cx) \cap \text{loc}(s, e_p) \\ &\quad \cup \cup_{i \in Ind} \text{dest}(e_p(i), E[\![N]\!]\rho' ci)\end{aligned}$$

par hypothèse de récurrence appliquée à M et à N, avec $\rho' = \rho[y \mapsto E[\![M]\!]\rho \sqcup \lambda c.\{y \mapsto c\}]$ et $e' = e[y \mapsto v]$. En effet, montrons que l'environnement ρ' utilisé pour l'analyse de N est correct, c'est-à-dire $\text{corr}_v(e', s', e'[\![y]\!], \rho'[\![y]\!])$: $\text{cell}(s', v, c, e_p) \subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!]\rho cx) \cup \cup_{i \in Ind} \text{dest}(e_p(i), E[\![M]\!]\rho ci)$ par hypothèse de récurrence sur M donc

$$\begin{aligned}\text{cell}(s', v, c, e_p) \cap \text{loc}(s, e_p) &\subseteq (a(s, s') \cup \text{cell}(s', v, c, e_p)) \cap \text{loc}(s, e_p) \\ &\subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!]\rho cx) \cup \cup_{i \in Ind} \text{dest}(e_p(i), E[\![M]\!]\rho ci)\end{aligned}$$

et $\text{cell}(s', v, c, e_p) - \text{loc}(e_p) \subseteq \text{dest}((s', v), c)$ donc

$$\text{cell}(s', v, c, e_p) \subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!]\rho cx) \cup \text{dest}((s', v), c) \cup \cup_{i \in Ind} \text{dest}(e_p(i), E[\![M]\!]\rho ci)$$

Ceci permet d'appliquer l'hypothèse de récurrence à N.
Le terme en $x = y$ est majoré par ρ-transitivité :

$$\begin{aligned}\text{dest}(e'(y), c') \cap \text{loc}(s, e_p) &= \text{dest}((s', v), c') \cap \text{loc}(s, e_p) \\ &\subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![M]\!]\rho c' x) \text{ par correction de } E[\![M]\!] \\ &= \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), \rho'[\![y]\!]c' x) \\ &\subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![N]\!]\rho' cx \sqcup \rho'[\![y]\!]\perp x) \text{ par } \rho\text{-transitivité.} \\ &\subseteq \cup_{x \in \text{Dom}(e)} \text{dest}(e(x), E[\![N]\!]\rho' cx \sqcup E[\![M]\!]\rho\perp x)\end{aligned}$$

Alors, comme $\text{dest}(e(x), c)$ croît en c (lemme 2.12),

$$\text{corr}_e(e, s, [\![\texttt{let}\ y = M\ \texttt{in}\ N]\!], \lambda c.E[\![N]\!](\rho[y \mapsto E[\![M]\!]\rho \sqcup \lambda c.\{y \mapsto c\}])c \sqcup E[\![M]\!]\rho\perp)$$

- Cas $\texttt{let rec}\ f(y) = M\ \texttt{in}\ N$:

$$[\![\texttt{let rec}\ f(x) = M\ \texttt{in}\ N]\!]es = [\![N]\!](e[f \mapsto (s', \ell)])s'$$

où $\ell \notin \mathrm{Dom}(s)$, $s' = s[\ell \mapsto (\phi, e_{|FV(\texttt{fun}\ x \to M)}, x)]$ et $\phi = \mathrm{lfp}(\lambda\phi_1.\lambda e_1.\lambda s_1.[\![M]\!](e_1[f \mapsto (s'_1, \ell)])s'_1)$ où s'_1 est l'état s_1 où $s_1(\ell)$ a été remplacé par (ϕ_1, e_1, x).

Soit $\rho_2 = \mathrm{lfp}(\lambda\rho_1.\rho[f \mapsto E[\![\texttt{fun}\ y \to M]\!]\rho_1 \sqcup \lambda c.\{f \mapsto c\}])$.

On effectue une induction de point fixe, en considérant des exécutions fictives dans lesquelles on arrête le calcul de point fixe à la n^{e} itération, puis en passant à la limite :

  - Cas initial : $f_1^0 = \bot$, $s^0 = s[\ell \mapsto (f_1^0, e, y)]$, $\rho_2^0 = \rho[f \mapsto \lambda c.\{f \mapsto c\}]$ alors

$$\mathrm{cell}(s^0, \ell, c, e'_p) \subseteq \mathrm{dest}((s', \ell), c) = \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \rho_2^0[\![f]\!]cx)$$

    où $e_1 = e[f \mapsto \ell]$.

  - Induction : on montre par récurrence que

$$\mathrm{cell}(s^n, \ell, c, e'_p) \subseteq \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \rho_2^n[\![f]\!]cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), \rho_2^n[\![f]\!]ci)$$

    où $f_1^{n+1} = \lambda e_1.\lambda s.[\![M]\!](e_1[f \mapsto \ell])s^n$, $s^{n+1} = s[\ell \mapsto (f_1^{n+1}, e, y)]$, $\rho_2^{n+1} = \rho[f \mapsto E[\![\texttt{fun}\ y \to M]\!]\rho_2^n \sqcup \lambda c.\{f \mapsto c\}]$. En effet, l'hypothèse de récurrence donne la cohérence entre $s^n(\ell)$ et $\rho_2^n[\![f]\!]$ en utilisant la formule (2.4), ce qui permet d'appliquer l'hypothèse d'induction sur l'expression à $\texttt{fun}\ y \to M$ et on obtient alors :

$$\begin{aligned}(a(s, s^{n+1}) \cup \mathrm{cell}(s^{n+1}, \ell, c, e'_p)) \cap \mathrm{loc}(s, e'_p) \subseteq\ & \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), E[\![\texttt{fun}\ y \to M]\!]\rho_2^n cx) \\ & \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), E[\![\texttt{fun}\ y \to M]\!]\rho_2^n ci)\end{aligned}$$

    en notant que $\rho_2^{n+1}[\![f]\!] = E[\![\texttt{fun}\ y \to M]\!]\rho_2^n \sqcup \lambda c.\{f \mapsto c\}$,

$$\mathrm{cell}(s^{n+1}, \ell, c, e'_p) \cap \mathrm{loc}(s, e_p) \subseteq \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \rho_2^{n+1}[\![f]\!]cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), \rho_2^{n+1}[\![f]\!]ci)$$

    et $\mathrm{cell}(s^{n+1}, \ell, c, e'_p) - \mathrm{loc}(e_p) \subseteq \mathrm{dest}((s', \ell), c) \subseteq \mathrm{dest}(e_1(f), \rho_2^{n+1}[\![f]\!]c)$ donc

$$\mathrm{cell}(s^{n+1}, \ell, c, e'_p) \subseteq \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \rho_2^{n+1}[\![f]\!]cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), \rho_2^{n+1}[\![f]\!]ci)$$

  - Passage à la limite :

$$\begin{aligned}\mathrm{cell}(s', \ell, c, e'_p) &= \cup_n \mathrm{cell}(s^n, \ell, c, e'_p) \\ &\subseteq \cup_n \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \rho_2^n[\![f]\!]cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), \rho_2^n[\![f]\!]ci) \\ &\subseteq \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \sqcup_n \rho_2^n[\![f]\!]cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), \sqcup_n \rho_2^n[\![f]\!]ci)\end{aligned}$$

    d'après le lemme 2.12

$$\subseteq \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), \rho_2[\![f]\!]cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e'_p(i), \rho_2[\![f]\!]ci)$$

On obtient donc la cohérence entre ℓ et $\rho_2[\![f]\!]$ (2.4), donc on peut appliquer l'hypothèse de récurrence sur N.

$$\begin{aligned}A &= (a(s, s'') \cup \mathrm{cell}(s'', \ell', c, e_p)) \cap \mathrm{loc}(s, e_p) \\ &= (a(s', s'') \cup \mathrm{cell}(s'', \ell', c, e_p)) \cap \mathrm{loc}(s', e_p) \\ &\subseteq \cup_{x \in \mathrm{Dom}(e_1)} \mathrm{dest}(e_1(x), E[\![N]\!]\rho_2 cx) \cup \cup_{i \in \mathit{Ind}} \mathrm{dest}(e_p(i), E[\![N]\!]\rho_2 ci)\end{aligned}$$

car on ne considère pas comme des affectations les modifications effectuées sur la fermeture entre les états s et s'. Comme dans le cas `let`, le terme en $x = f$ est majoré par ρ-transitivité :

$$\begin{aligned}\mathrm{dest}(e_1(f), c') &= \mathrm{dest}((s', \ell), c') \\ &\subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), \rho_2[\![f]\!]c'x) \\ &\subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho_2 cx \sqcup \rho_2[\![f]\!]\bot x) \text{ par } \rho\text{-transitivité.} \\ &\subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho_2 cx)\end{aligned}$$

Alors $A \subseteq \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho_2 cx) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![N]\!]\rho_2 ci)$ donc

$\mathrm{corr}_e(e, s, [\![\texttt{let rec } f(y) = M \texttt{ in } N]\!], \lambda c.E[\![N]\!]((\mathrm{lfp}(\lambda\rho_1.\rho[f \mapsto E[\![\texttt{fun } y \to M]\!]\rho_1 \sqcup \lambda c.\{f \mapsto c\}]))c)$

- Cas `if` M `then` N `else` P :

$$\begin{aligned}[\![\texttt{if } M \texttt{ then } N \texttt{ else } P]\!]es &= \text{let } [(x, s')] = [\![M]\!]es \text{ in if } x \text{ then } [\![N]\!]es' \text{ else } [\![P]\!]es' \\ &= \mathrm{up}(\ell'', s'')\end{aligned}$$

$$\begin{aligned}A &= (a(s, s'') \cup \mathrm{cell}(s'', \ell'', c, e_p)) \cap \mathrm{loc}(s, e_p) \\ &\subseteq (\mathrm{cell}(s'', \ell'', c, e_p) \cup a(s, s') \cup a(s', s'')) \cap \mathrm{loc}(s, e_p) \text{ d'après le lemme 2.4} \\ &\subseteq [\cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho cx) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![N]\!]\rho ci) \\ &\quad \cup \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![P]\!]\rho cx) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![P]\!]\rho ci)] \cup \cup_{x\in\mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho\bot x)\end{aligned}$$

par hypothèse de récurrence appliquée à M, N et P

Comme $\mathrm{dest}((s, v), c)$ croît en c (lemme 2.12),

$$\mathrm{corr}_e(e, s, [\![\texttt{if } M \texttt{ then } N \texttt{ else } P]\!], \lambda c.E[\![M]\!]\rho\bot \sqcup E[\![N]\!]\rho c \sqcup E[\![P]\!]\rho c)$$

- Cas $M{::}N$:

$$\begin{aligned}[\![M{::}N]\!]es &= \text{let } [(h, s')] = [\![M]\!]es \text{ in let } [(t, s'')] = [\![N]\!]es' \text{ in newloc}(s'', (h, t)) \\ &= \mathrm{up}(\ell, s''')\end{aligned}$$

$$A = (a(s, s''') \cup \mathrm{cell}(s''', \ell, c, e_p)) \cap \mathrm{loc}(s, e_p)$$

On distingue alors deux cas :

  - Si $c = \top$, $\mathrm{cell}(s''', \ell, \top, []) \cap \mathrm{loc}(s) = (\ell \cup h\downarrow_{s'''} \cup t\downarrow_{s'''}) \cap \mathrm{loc}(s)$. Comme ℓ est une nouvelle cellule, $\ell \notin \mathrm{loc}(s)$, donc

  $$\mathrm{cell}(s''', \ell, c, e_p) \cap \mathrm{loc}(s, e_p) = (\mathrm{cell}(s''', h, c_{|g}, e_p) \cup \mathrm{cell}(s''', t, c_{|d}, e_p)) \cap \mathrm{loc}(s, e_p).$$

  - Si $c = \mathrm{g}.c_1$, par définition de cell (2.2),

  $$\begin{aligned}\mathrm{cell}(s''', \ell, c, e_p) \cap \mathrm{loc}(s, e_p) &= \cup_{s_4 \text{ après } s'''} \mathrm{cell}(s_4, h, c_1, e_p) \cap \mathrm{loc}(s, e_p) \\ &= (\mathrm{cell}(s''', h, c_{|g}, e_p) \cup \mathrm{cell}(s''', t, c_{|d}, e_p)) \cap \mathrm{loc}(s, e_p)\end{aligned}$$

  car cell décroît au cours de l'exécution (lemme 2.6) et $\mathrm{cell}(s''', t, c_{|d}, e_p) = \emptyset$. On traite de même le cas $c = \mathrm{d}.c_1$.

  - Le cas $c = \bot$ est évident :

  $$\mathrm{cell}(s''', \ell, c, e_p) \cap \mathrm{loc}(s, e_p) = \emptyset = (\mathrm{cell}(s''', h, c_{|g}, e_p) \cup \mathrm{cell}(s''', t, c_{|d}, e_p)) \cap \mathrm{loc}(s, e_p).$$

En appliquant de plus le lemme 2.4,

$$\begin{aligned}
A &\subseteq (a(s,s') \cup a(s',s'') \cup a(s'',s''') \\
&\quad \cup \mathrm{cell}(s''',h,c_{|g},e_p) \cup \mathrm{cell}(s''',t,c_{|d},e_p)) \cap \mathrm{loc}(s,e_p) \\
&\subseteq (a(s,s') \cup \mathrm{cell}(s',h,c_{|g},e_p)) \cap \mathrm{loc}(s,e_p) \cup (a(s',s'') \cup \mathrm{cell}(s'',t,c_{|d},e_p)) \cap \mathrm{loc}(s',e_p) \\
&\subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho(c_{|g})x) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![M]\!]\rho(c_{|g})i) \\
&\quad \cup \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho(c_{|d})x) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![N]\!]\rho(c_{|d})i)
\end{aligned}$$

en utilisant l'hypothèse de récurrence sur M et N et le lemme 2.6, donc

$$\mathrm{corr}_e(e,s,[\![M{::}N]\!], \lambda c.E[\![M]\!]\rho(c_{|g}) \sqcup E[\![N]\!]\rho(c_{|d}))$$

- Cas (M,N) : analogue à $M{::}N$.
- Cas $M;N$:

$$[\![M;N]\!]es = \text{let } [(l,s')] = [\![M]\!]es \text{ in } [\![N]\!]es' = \mathrm{up}(\ell',s'')$$

$$\begin{aligned}
A &= (a(s,s'') \cup \mathrm{cell}(s'',\ell',c,e_p)) \cap \mathrm{loc}(s,e_p) \\
&\subseteq (a(s,s') \cup a(s',s'') \cup \mathrm{cell}(s'',\ell',c,e_p)) \cap \mathrm{loc}(s,e_p) \text{ d'après le lemme 2.4} \\
&\subseteq \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![M]\!]\rho\bot x) \cup \cup_{x\in \mathrm{Dom}(e)} \mathrm{dest}(e(x), E[\![N]\!]\rho c x) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i), E[\![N]\!]\rho c i)
\end{aligned}$$

par récurrence sur M et N, donc en utilisant encore le lemme 2.12,

$$\mathrm{corr}_e(e,s,[\![M;N]\!], \lambda c.E[\![M]\!]\rho\bot \sqcup E[\![N]\!]\rho c)$$

- Cas `ref` :

$$e[\![\texttt{ref}]\!] = \mathrm{ref} \text{ où } s(\mathrm{ref}) = (\lambda e.\lambda s'.\, \mathrm{newloc}(s', e[\![x]\!]), \emptyset, x)$$

  - Cas $c = \top$ ou $c = \bot$:

$$\mathrm{cell}(s,\mathrm{ref},\bot,[]) = \emptyset$$
$$\mathrm{cell}(s,\mathrm{ref},\top,[]) = \{\mathrm{ref}\} = \mathrm{dest}((s,\mathrm{ref}),\top)$$

  - Cas $c = \mathrm{app}.c'$: Soit $\mathrm{newloc}(s', e[\![x]\!]) = \mathrm{up}(\ell, s'')$

$$\begin{aligned}
A &= \mathrm{cell}(s,\mathrm{ref},c,(s',p_1):e_p) \\
&\subseteq a(s',s'') \cup \mathrm{cell}(s'',\ell,c',e_p) \text{ d'après (2.3)} \\
&\subseteq \mathrm{cell}(s'',\ell,c',e_p) \\
&\subseteq \cup_{s''' \text{ après } s''} \mathrm{cell}_0(s''',r',c'_{|g},e_p)
\end{aligned}$$

  car ℓ étant mutable, $p_1 = s''(\ell)$ peut-être remplacé par $r' = s'''(\ell)$ dans un état futur s'''. Si $r' \neq p_1$, d'après le lemme 2.8,

$$\mathrm{cell}_0(s''',r',c'_{|g},e_p) \subseteq \mathrm{loc}(e_p) \cup r' \downarrow_{s'''} - a(s''')$$

  $r' \downarrow_{s'''} \subseteq a(s''')$ donc

$$\mathrm{cell}_0(s''',r',c'_{|g},e_p) \subseteq \mathrm{loc}(e_p)$$

  Finalement, $A \subseteq \mathrm{cell}(s''',p_1,c'_{|g},e_p) \cup \mathrm{loc}(e_p)$
  $A \subseteq \mathrm{dest}((s'',p_1),c'_{|g}) \cup \cup_{i\in Ind} \mathrm{dest}(e_p(i),\top)$

Alors

$$\mathrm{corr}_v(e,s,\mathrm{ref},\lambda c. \begin{cases} \{\} & \text{si } c = \bot \text{ ou } \top, \\ \{1 \mapsto c_{|\to|g}, i \mapsto \top \text{ si } i > 1\} & \text{sinon.} \end{cases})$$

– Cas ! :

$$e[\![!]\!] = \text{deref où } s(\text{deref}) = (\lambda e.\lambda s'.\,\text{up}(s'(e[\![x]\!]), s'), \emptyset, x)$$

– Cas $c = \top$ ou $c = \bot$:

$$\text{cell}(s, \text{deref}, \top, []) = \{\text{deref}\} = \text{dest}((s, \text{deref}), \top)$$

$$\text{cell}(s, \text{deref}, \bot, []) = \emptyset$$

– Cas $c = \text{app}.c'$:

$$\begin{aligned} A &= \text{cell}(s, \text{deref}, c, (s', p_1) : e_p) \\ &\subseteq \text{cell}(s', s'(p_1), c', e_p) \text{ d'après (2.3)} \\ &\subseteq \text{cell}(s', p_1, \text{g}.c', e_p) \\ &\subseteq \text{dest}((s', p_1), \text{g}.c') \cup \cup_{i \in Ind}\, \text{dest}(e_p(i), \top) \end{aligned}$$

Alors,

$$\text{corr}_v(e, s, \text{deref}, \lambda c. \begin{cases} \{\} & \text{si } c = \bot \text{ ou } \top, \\ \{1 \mapsto \text{g}.c_{|\to}, i \mapsto \top \text{ si } i > 1\} & \text{sinon.} \end{cases})$$

– Cas := :

$$e[\![:=]\!] = \text{affect où } s(\text{affect}) = (\lambda e.\lambda s'.\,\text{newloc}(s', (\lambda e'.\lambda s''.\,\text{up}(\text{NIL}, s''[e'[\![m]\!] \mapsto e'[\![n]\!]]), e, n)), \emptyset, m)$$

– Cas $c = \bot$ ou $c = \top$:

$$\text{cell}(s, \text{affect}, \top, []) = \{\text{affect}\} = \text{dest}((s, \text{affect}), \top)$$

$$\text{cell}(s, \text{affect}, \bot, []) = \emptyset$$

– Cas $c = \text{app}.\bot$: $\text{cell}(s, \text{affect}, \text{app}.\bot, [(s', p_1)]) \subseteq \text{cell}(s'', \ell, \bot, []) \cup a(s', s'')$ d'après (2.3), où $\text{newloc}(s', (\lambda e'.\lambda s''.\,\text{up}((), s''[e'[\![m]\!] \mapsto e'[\![n]\!]]), e, n)) = \text{up}(\ell, s'')$. Alors

$$\text{cell}(s, \text{affect}, c, [(s', p_1)]) = \emptyset.$$

– Cas $c = \text{app}.\top$ avec les mêmes notations que pour $c = \text{app}.\bot$:

$$\begin{aligned} A &= \text{cell}(s, \text{affect}, \text{app}.\top, [(s', p_1)]) \\ &\subseteq \text{cell}(s'', \ell, \top, []) \cup a(s', s'') \text{ d'après (2.3)} \\ &\subseteq a(s', s'') \cup \ell \downarrow_{s''} - a(s'') \\ &\subseteq p_1 \downarrow_{s'} - a(s') \\ &\subseteq \text{dest}((s', p_1), \top) \end{aligned}$$

– Cas $c = \text{app}.\text{app}.c'$:

$$A = \text{cell}(s, \text{affect}, c, [(s', p_1), (s''', p_2)])$$

En appliquant 2 fois (2.3),

$$\begin{aligned} A &\subseteq \text{cell}(s'''', \text{NIL}, c', []) \cup a(s', s'') \cup a(s''', s'''') \\ &\subseteq a(s''', s'''') \\ &\subseteq p_2 \downarrow_{s'''} - a(s''') \\ &\subseteq \text{dest}((s''', p_2), \top) \end{aligned}$$

Alors,

$$\mathrm{corr}_v(e, s, \mathrm{affect}, \lambda c. \begin{cases} \{\} & \text{si } c = \bot, \mathrm{app}.\bot \text{ ou } \top \\ \{1 \mapsto \top\} & \text{si } c = \mathrm{app}.\top, \\ \{1 \mapsto \bot, 2 \mapsto \top\} & \text{si } c = \mathrm{app.app}.\bot \text{ ou } \mathrm{app.app}.\top. \end{cases})$$

- Cas `true` : $e[\![\texttt{true}]\!] = \mathrm{true}$ et $\mathrm{cell}(s, \mathrm{true}, c, []) = \emptyset$ donc $\mathrm{corr}_v(e, s, \mathrm{true}, \lambda c.\bot)$
- Cas `false`, $\mathbb{Z}$ et `[]` : comme `true`.
- Cas `hd`, `tl`, `null`, `fst` et `snd` : analogues à `!`.

□

Dans le cas :

```
let rec f(x) = ... z := f ...
in f(3)
```

$\mathrm{dest}((s, \texttt{f}), \mathrm{app}.\top)$ comprend toutes les cellules de `f`, donc on ne peut pas utiliser directement le critère $\mathrm{corr}_e(e, s, [\![\texttt{f(3)}]\!], E[\![\texttt{f(3)}]\!]\rho)$ pour décider que `f` ne s'échappe pas si $\top \notin E[\![\texttt{f(3)}]\!]\rho\top\texttt{f}$.

Le même problème se produit quand on autorise des structures récursives définies par `let rec` : la cellule du sommet peut fort bien être accédée par l'intermédiaire d'un chemin différent de $\top$.

On peut cependant prouver le résultat suivant.

Théorème 2.24 (Correction de l'analyse E) *Considérons l'expression :* `let` x `=` M `in` N *ou* `let rec` $x(y)$ `=` M `in` N*. Soit ℓ la cellule au sommet de x. Supposons que la création de cette cellule soit la dernière opération de M. Supposons que $\top \notin E[\![N]\!]\rho\, \mathit{Chem}\, x$. Alors ℓ est allouable en pile.*

Preuve : Soit s_x l'état avant évaluation de N, et e l'environnement dans lequel N est évalué.

Si x est défini par `let`, $\mathrm{dest}((s_x, \ell), c) \subseteq \cup_{e_p}(a(s_1, s'_1) \cup \dots \cup a(s_n, s'_n) \cup r' \downarrow_{s'_n} - \mathrm{loc}(e_p))$ où s_i est l'état du i^{e} paramètre de e_p, s'_i est l'état après application de ce paramètre à la valeur courante, et r' est le résultat de la descente dans x par le chemin c. Les affectations ne peuvent concerner que des cellules accessibles pendant le calcul des applications, c'est-à-dire des paramètres passés ou des cellules contenues dans l'environnement des fermetures éventuellement contenues dans x, et donc pas la cellule ℓ, donc $\ell \notin a(s_x, s'_n) - \mathrm{loc}(e_p)$. D'autre part, si $c \neq \top$, $\ell \notin r' \downarrow_{s'}$, donc finalement, si $c \neq \top$, $\ell \notin \mathrm{dest}((s_x, \ell), c)$.

Si x est défini par un `let rec`, x lui-même peut faire référence à ℓ. Par ρ-transitivité (lemme 2.21), $E[\![N]\!]\rho\, \mathit{Chem}\, x \sqcup \rho[\![x]\!]\bot x \supseteq \rho[\![x]\!](E[\![N]\!]\rho\, \mathit{Chem}\, x)x$ avec $\rho[\![x]\!] = \mathrm{lfp}(\lambda\rho_x.E[\![\texttt{fun}\ y \to M]\!]\rho[x \mapsto \rho_x] \sqcup \lambda c.\{x \mapsto c\})$. Alors $\rho[\![x]\!]\bot x = \bot$, donc $\top \notin \rho[\![x]\!](E[\![N]\!]\rho\, \mathit{Chem}\, x)x$. Soit $c \in E[\![N]\!]\rho\, \mathit{Chem}\, x$. $\top \notin \rho[\![x]\!]cx$, donc $\top \notin \mathrm{lfp}(\lambda\rho_x.E[\![\texttt{fun}\ y \to M]\!]\rho[x \mapsto \rho_x] \sqcup \lambda c.\{x \mapsto c\})cx$.

De façon analogue à la preuve de correction dans le cas `let rec`, considérons des exécutions fictives où on ne fait que n itérations dans le calcul du point fixe, pour montrer par induction que $\ell \notin \mathrm{dest}((s_x^n, \ell), c)$, si $c \neq \top$.

- Si $n = 0$, x est la fonction qui ne termine jamais, donc rien ne s'échappe : $\mathrm{dest}((s_x^0, \ell), c) = \emptyset$ car $c \neq \top$.
- Pour $n + 1$,

$$\begin{aligned} \mathrm{dest}((s_x^{n+1}, \ell), c) \subseteq{} & \cup_{z \in \mathit{Var}-\{x\}} \mathrm{dest}(e^n(z), E[\![\texttt{fun}\ y \to M]\!]\rho^n cz) \\ & \cup \mathrm{dest}(e^n(x), E[\![\texttt{fun}\ y \to M]\!]\rho^n cx \sqcup c) \end{aligned}$$

  par correction de l'analyse pour le terme `fun` $y \to M$ (ceci se déduit de l'étape d'induction effectuée dans la preuve de correction de `let rec`). Par hypothèse de récurrence, comme $\top \notin E[\![\texttt{fun}\ y \to M]\!]\rho^n cx \sqcup c$, $\ell \notin \mathrm{dest}((s_x^n, \ell), E[\![\texttt{fun}\ y \to M]\!]\rho^n cx \sqcup c)$, et ℓ n'est pas dans les autres variables de l'environnement, car elle est créée au moment de la création de x, donc $\ell \notin \mathrm{dest}((s_x^{n+1}, \ell), c)$.

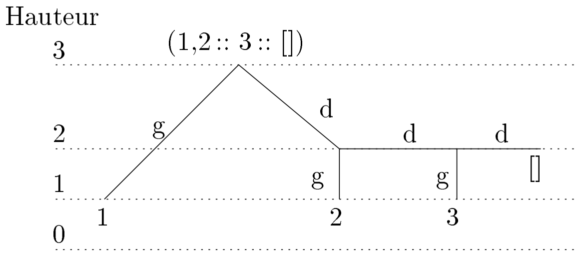


FIG. 2.4 – *Hauteurs de types*

– Passage à la limite : $\mathrm{dest}((s_x, \ell), c) = \cup_n \mathrm{dest}((s_x^n, \ell), c)$, donc $\ell \notin \mathrm{dest}((s_x, \ell), c)$.

On a donc ici encore : $\ell \notin \mathrm{dest}((s_x, \ell), c)$ si $c \in E[\![N]\!]\rho\ \mathit{Chem}\ x$.

Alors $\ell \notin \mathrm{dest}((s_x, \ell), E[\![N]\!]\rho\ \mathit{Chem}\ x)$. De plus, $\ell \notin \mathrm{dest}(e(y), E[\![N]\!]\rho\ \mathit{Chem}\ y)$ pour $y \neq x$ car elle est créée avec x. Par correction de l'analyse (théorème 2.23),

$$\mathrm{cell}_e(s_x, [\![N]\!]e, \top, []) \subseteq \cup_{y \in \mathit{Var}} \mathrm{dest}(e(y), E[\![N]\!]\rho\ \mathit{Chem}\ y).$$

Alors $\ell \notin \mathrm{cell}_e(s_x, [\![N]\!]e, \top, [])$. D'après le théorème 2.13, ℓ est alors allouable en pile.

□

Cette analyse est évidemment invariante par polymorphisme car les types n'interviennent pas dans sa définition [Abr86].

2.4 Deuxième version plus approchée

L'analyse précédente est précise, mais sa complexité est trop importante pour permettre une implémentation efficace. On fait donc davantage d'approximations.

On définit les hauteurs de types par

$$\begin{aligned}
&\top_2[\tau] = 1 \text{ si } \tau = \texttt{bool}, \texttt{int} \text{ ou } \texttt{unit}.\\
&\top_2[\tau_1 \to \tau_2] = \top_2[\tau_2]\\
&\top_2[\tau_1 * \tau_2] = 1 + \max(\top_2[\tau_1], \top_2[\tau_2])\\
&\top_2[\tau\ \texttt{list}] = 1 + \top_2[\tau]\\
&\top_2[\tau\ \texttt{ref}] = 1 + \top_2[\tau]
\end{aligned}$$

et par $\top_1[\tau] = \begin{cases} 1 & \text{si } \tau \text{ est contient un type fonctionnel } \tau_1 \to \tau_2 \\ 0 & \text{sinon} \end{cases}$

La hauteur d'un chemin est alors définie par

$$\begin{aligned}
&\alpha_2^{\tau}(\top) = \top_2[\tau] \text{ si } \tau \text{ n'est pas fonctionnel, et } \infty \text{ sinon.}\\
&\alpha_2^{\tau}(\bot) = 0\\
&\alpha_2^{\tau_1 \to \tau_2}(\mathrm{app}.c) = \alpha_2^{\tau_2}(c)\\
&\alpha_2^{\tau_1 * \tau_2}(\mathrm{g}.c) = \alpha_2^{\tau_1}(c), \alpha_2^{\tau\ \texttt{list}}(\mathrm{g}.c) = \alpha_2^{\tau}(c), \alpha_2^{\tau\ \texttt{ref}}(\mathrm{g}.c) = \alpha_2^{\tau}(c)\\
&\alpha_2^{\tau_1 * \tau_2}(\mathrm{d}.c) = \alpha_2^{\tau_2}(c), \alpha_2^{\tau\ \texttt{list}}(\mathrm{d}.c) = \alpha_2^{\tau\ \texttt{list}}(c)\\
&\alpha_2^{\tau}(c) = 0 \text{ dans tous les autres cas (chemins interdits).}
\end{aligned}$$

On définit la hauteur d'un contexte par passage à la borne supérieure :

$$\alpha_2^{\tau}(c) = \bigsqcup_{c' \in c} \alpha_2^{\tau}(c')$$

et la fonction de concrétisation

$$\gamma_2^\tau(c') = \sqcup\{c \in \mathit{Ctx} | \alpha_2^\tau(c) \leq c'\} = \{c \in \mathit{Chem} | \alpha_2^\tau(c) \leq c'\}$$

α_2^τ et γ_2^τ forment une correspondance de Galois :

$$(\mathit{Ctx}, \subseteq) \underset{\alpha_2^\tau}{\overset{\gamma_2^\tau}{\leftrightarrows}} (\mathbb{N} \cup \{\infty\}, \leq)$$

$$\alpha_2^\tau(x) \leq y \Leftrightarrow x \subseteq \gamma_2^\tau(y)$$

et

$$\alpha_2^\tau \circ \gamma_2^\tau = id$$

On notera souvent en exposant de α_2 ou de γ_2 une expression à la place d'un type pour signifier le type de cette expression.

Soit $\mathit{Ctx}_1 = \{0,1\}$ et $\mathit{Ctx}_2 = \mathbb{N}$. On définit l'abstraction $\alpha_1^\tau(c) = (1,0)$ si $\alpha_2^\tau(c) = \infty$, $\alpha_1^\tau(c) = (0, \alpha_2^\tau(c))$ sinon. $\gamma_1^\tau(c_1, c_2) = \sqcup\{c \in \mathit{Ctx} | \alpha_1(c) \leq (c_1, c_2)\} = \{c \in \mathit{Chem} | \alpha_1(c) \leq (c_1, c_2)\}$ où l'ordre sur les couples est lexicographique. $(\alpha_1^\tau, \gamma_1^\tau)$ est une correspondance de Galois :

$$(\mathit{Ctx}, \subseteq) \underset{\alpha_1^\tau}{\overset{\gamma_1^\tau}{\leftrightarrows}} (\mathit{Ctx}_1 \times \mathit{Ctx}_2, \leq)$$

$\mathit{Val}_i = \mathit{Ctx}_i \to (\mathit{Var} \cup \mathit{Ind}) \to \mathit{Ctx}_i$ et $\mathit{Env}_i = \mathit{Var} \to \mathit{Val}_i$. On définit deux analyses approchées $F_i[\![M]\!] : \mathit{Env}_i \to \mathit{Val}_i$ sur la figure 2.5 Dans ces définitions, $x \in \mathit{Var}$ et $i \in \mathit{Ind}$. Ceci permet de distinguer facilement ce qui se passe pour les paramètres de ce qui concerne les variables. Soit $\top_{1x} = \top_1[\tau]$ si x est de type τ. On notera $\top_{1,x}$ pour plus de clarté quand x est un indice de paramètre numérique. On utilise les mêmes notations pour $\top_2$.

L'analyse F_1 prend en compte les variables qui s'échappent à travers des fermetures, alors que F_2 donne une information plus précise pour les autres variables. Une seule analyse n'est pas suffisante, car le niveau d'une varaible dans une fermeture peut être plus grand que le niveau de la fermeture, donc la fonction d'échappement correspondante ne serait pas inférieure et notre algorithme rapide ne fonctionnerait pas (voir section 3.3).

Exemple 2.25 Pour la fonction `map : (int ->int) ->int list ->int list` (exemple 2.20),

$$F_1[\![\texttt{fun f l -> if ...}]\!]\rho_1 c =$$
$$\{1 \mapsto c \sqcup \rho_1[\![\texttt{map}]\!]c\,\texttt{f} \sqcup \rho_1[\![\texttt{map}]\!]c\,1, 2 \mapsto c \sqcup \rho_1[\![\texttt{map}]\!]c\,\texttt{l} \sqcup \rho_1[\![\texttt{map}]\!]c\,2, \texttt{map} \mapsto c \sqcup \rho_1[\![\texttt{map}]\!]c\,\texttt{map}\}$$
$$F_2[\![\texttt{fun f l -> if ...}]\!]\rho_2 c =$$
$$\{1 \mapsto (c \sqcap 1) \sqcup \rho_2[\![\texttt{map}]\!]c\,\texttt{f} \sqcup \rho_2[\![\texttt{map}]\!]c\,1, 2 \mapsto 1 \sqcup \rho_2[\![\texttt{map}]\!]c\,\texttt{l} \sqcup \rho_2[\![\texttt{map}]\!]c\,2, \texttt{map} \mapsto c \sqcup \rho_2[\![\texttt{map}]\!]c\,\texttt{map}\}$$

On peut alors calculer les points fixes:

$$\rho_1[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c, 2 \mapsto c\}$$
$$\rho_2[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c \sqcap 1, 2 \mapsto 1\}$$

Lemme 2.26 *Si x est une variable,* $F_1[\![M]\!]\rho_1 1x \geq \sqcup_{z \in FV(M)} \rho_1[\![z]\!]1x$

Preuve : Par induction sur M. Le point clé est de noter que dans la définition de $F_1[\![M]\!]\rho_1 cx$, la valeur du contexte passée à une sous-expression de M est au moins c, donc ici c'est 1. □

On pourrait montrer que

$$\alpha_1(E[\![M]\!]\rho\gamma_1(c_1, c_2)x) \leq (F_1[\![M]\!]\rho_1 c_1 x, F_2[\![M]\!]\rho_2 c_2 x)$$

si

$$\alpha_1(\rho[\![y]\!]\gamma_1(c_1, c_2)x) \leq (\rho_1[\![y]\!]c_1 x, \rho_2[\![y]\!]c_2 x)$$

$F_1[\![M]\!] : Env_1 \to Val_1$
$F_1[\![y]\!]\rho_1 c = \rho_1[\![y]\!]c$
$F_1[\![M\ N]\!]\rho_1 c = \{i \mapsto \phi(i+1), x \mapsto \phi(x) \sqcup \psi(x)\}$ avec $\phi = F_1[\![M]\!]\rho_1 c$ et $\psi = F_1[\![N]\!]\rho_1(\phi(1) \sqcup c)$.
$F_1[\![\texttt{fun}\ y \to M]\!]\rho_1 c = \{1 \mapsto \phi(y), i \mapsto \phi(i-1)$ si $i > 1, x \mapsto \phi(x)\}$
  où $\phi = F_1[\![M]\!]\rho_1[y \mapsto \rho_{1\top}]c$ et $\rho_{1\top} c = \{i \mapsto \top_{1i} \sqcup c, y \mapsto c\}$.
$F_1[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]\rho_1 c = F_1[\![N]\!]\rho_1[y \mapsto F_1[\![M]\!]\rho_1 \sqcup \lambda c.\{y \mapsto c\}]c \sqcup F_1[\![M]\!]\rho_1 c_{|\,Var}$
$F_1[\![\texttt{let rec}\ f(y) = M\ \texttt{in}\ N]\!]\rho_1 c = F_1[\![N]\!]\rho_3 c \sqcup \rho_3(f)c_{|\,Var}$
  avec $\rho_3 = \mathrm{lfp}(\lambda\rho_3.\rho_1[f \mapsto F_1[\![\texttt{fun}\ y \to M]\!]\rho_3 \sqcup \lambda c.\{f \mapsto c\}])$.
$F_1[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c_{|\,Var} \sqcup F_1[\![N]\!]\rho_1 c \sqcup F_1[\![P]\!]\rho_1 c$
$F_1[\![M{::}N]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c \sqcup F_1[\![N]\!]\rho_1 c$
$F_1[\![(M,N)]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c \sqcup F_1[\![N]\!]\rho_1 c$
$F_1[\![M;N]\!]\rho_1 c = F_1[\![M]\!]\rho_1 c_{|\,Var} \sqcup F_1[\![N]\!]\rho_1 c$
$\rho_1[\![:=]\!]c = \{1 \mapsto c, 2 \mapsto \top_{1,2} \sqcup c\}$
$\rho_1[\![a]\!]c = \{\}$ où $a \in \{\texttt{true}, \texttt{false}, []\} \cup \mathbb{Z}$.
$\rho_1[\![f]\!]c = \{1 \mapsto c, i \mapsto \top_{1i} \sqcup c\}$ où f vaut `hd`, `tl`, `fst`, `snd`, `ref` ou `!`.
$\rho_1[\![\texttt{null}]\!]c = \{1 \mapsto c\}$

$F_2[\![M]\!] : Env_2 \to Val_2$
$F_2[\![y]\!]\rho_2 c = \rho_2[\![y]\!]c$
$F_2[\![M\ N]\!]\rho_2 c = \{i \mapsto \phi(i+1), x \mapsto \phi(x) \sqcup \psi(x)\}$ avec $\phi = F_2[\![M]\!]\rho_2 c$ et $\psi = F_2[\![N]\!]\rho_2(\phi(1))$.
$F_2[\![\texttt{fun}\ y \to M]\!]\rho_2 c = \{1 \mapsto \phi(y), i \mapsto \phi(i-1)$ si $i > 1, x \mapsto \phi(x)\}$
  où $\phi = F_2[\![M]\!]\rho_2[y \mapsto \rho_{2\top}]c$ et $\rho_{2\top} c = \{i \mapsto \top_{2i}, y \mapsto c\}$.
$F_2[\![\texttt{let}\ y = M\ \texttt{in}\ N]\!]\rho_2 c = F_2[\![N]\!]\rho_2[y \mapsto F_2[\![M]\!]\rho_2 \sqcup \lambda c.\{y \mapsto c\}]c \sqcup F_2[\![M]\!]\rho_2 0_{|\,Var}$
$F_2[\![\texttt{let rec}\ f(y) = M\ \texttt{in}\ N]\!]\rho_2 c = F_2[\![N]\!](\mathrm{lfp}(\lambda\rho_4.\rho_2[f \mapsto F_2[\![\texttt{fun}\ y \to M]\!]\rho_4 \sqcup \lambda c.\{f \mapsto c\}])c$
$F_2[\![\texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P]\!]\rho_2 c = F_2[\![M]\!]\rho_2 0_{|\,Var} \sqcup F_2[\![N]\!]\rho_2 c \sqcup F_2[\![P]\!]\rho_2 c$
$F_2[\![M{::}N]\!]\rho_2 c = F_2[\![M]\!]\rho_2(c \sqcap \top_{2M}) \sqcup F_2[\![N]\!]\rho_2 c$
$F_2[\![(M,N)]\!]\rho_2 c = F_2[\![M]\!]\rho_2(c \sqcap \top_{2M}) \sqcup F_2[\![N]\!]\rho_2(c \sqcap \top_{2N})$
$F_2[\![M;N]\!]\rho_2 c = F_2[\![M]\!]\rho_2 0_{|\,Var} \sqcup F_2[\![N]\!]\rho_2 c$
$\rho_2[\![\texttt{ref}]\!]c = \{1 \mapsto c \sqcap \top_{2,1}, i \mapsto \top_{2i}$ si $i > 1\}$
$\rho_2[\![:=]\!]c = \{2 \mapsto \top_{2,2}\}$
$\rho_2[\![a]\!]c = \{\}$ si $a \in \{\texttt{true}, \texttt{false}, []\} \cup \mathbb{Z}$.
$\rho_2[\![f]\!]c = \{1 \mapsto c, i \mapsto \top_{2i}$ si $i > 1\}$ si f vaut `hd`, `tl`, `fst`, `snd` ou `!`.
$\rho_2[\![\texttt{null}]\!]c = \{\}$

FIG. 2.5 – *Analyses F_1 et F_2*

mais le lemme 2.26 donne une propriété plus forte quand $F_1[\![M]\!]\rho_1 1x = 0$: x n'est pas libre dans M. On peut alors montrer le théorème de correction suivant :

Théorème 2.27 *Les analyses $f_1 \in Ctx_1 \to Ctx_1$ et $f_2 \in Ctx_2 \to Ctx_2$ qui donnent l'information d'échappement sur une variable de type τ' dans un terme de type τ sont dites correctes par rapport à $f \in Ctx \to Ctx$ si*

$$C(\tau[\tau'], f, f_1, f_2) \Leftrightarrow \begin{cases} \text{Si } f_1 1 = 0,\ f\ Chem \subseteq \gamma_2^{\tau'}(0) \\ \text{Si } f_1 0 = 0,\ f(\gamma_2^{\tau}(c)) \subseteq \gamma_2^{\tau'}(f_2 c) \end{cases}$$

Le premier cas traite la situation où une fermeture s'échappe dans le résultat (le contexte passé à l'analyse f_1 est 1). Dans ce cas, l'analyse f_1 est juste une approximation grossière qui dit que toutes les variables libres s'échappent, donc si f_1 donne 0 comme résultat, on sait que la variable n'est pas libre et rien ne s'en échappe. Le deuxième cas traite la situation où aucune fermeture ne s'échappe dans le résultat. Dans ce cas, l'analyse f_1 a pour résultat 0 si on peut prouver qu'aucune fermeture ne s'échappe de la variable, et quand $f_1 0 = 0$, l'analyse f_2 donne l'information d'échappement précise.

Alors,

$$\begin{aligned} &\forall y, \forall x, C(\tau_y[\tau_x], \lambda c.\rho[\![y]\!]cx, \lambda c_1.\rho_1[\![y]\!]c_1 x, \lambda c_2.\rho_2[\![y]\!]c_2 x) \Rightarrow \\ &\forall x, C(\tau_M[\tau_x], \lambda c.E[\![M]\!]\rho cx, \lambda c_1.F_1[\![M]\!]\rho_1 c_1 x, \lambda c_2.F_2[\![M]\!]\rho_2 c_2 x) \end{aligned}$$

Comme l'environnement initial vérifie $\forall y, \forall x, C(\tau_y[\tau_x], \lambda c.\rho[\![y]\!]cx, \lambda c_1.\rho_1[\![y]\!]c_1 x, \lambda c_2.\rho_2[\![y]\!]c_2 x)$, on en déduit que $\forall x, C(\tau_M[\tau_x], \lambda c.E[\![M]\!]\rho cx, \lambda c_1.F_1[\![M]\!]\rho_1 c_1 x, \lambda c_2.F_2[\![M]\!]\rho_2 c_2 x)$, ce qui prouve la correction des analyses F_1 et F_2 :

$$\begin{cases} \text{Si } F_1[\![M]\!]\rho_1 1x = 0,\ E[\![M]\!]\rho\ Chem\, x \subseteq \gamma_2^{\tau_x}(0) \\ \text{Si } F_1[\![M]\!]\rho_1 0x = 0,\ E[\![M]\!]\rho(\gamma_2^{\tau_M}(c))x \subseteq \gamma_2^{\tau_x}(F_2[\![M]\!]\rho_2 cx) \end{cases}$$

Preuve : On montre tout d'abord les résultats suivants, avec $I_i : Ctx_i \to Ctx_i, I_i c = c$:

$$C(\tau'[\tau''], f, f_1, f_2) \text{ et } C(\tau[\tau'], g, g_1, g_2) \Rightarrow C(\tau[\tau''], f \circ g, f_1 \circ (g_1 \sqcup I_1), f_2 \circ g_2) \tag{2.11}$$
$$C(\tau[\tau'], f, f_1, f_2) \text{ et } C(\tau[\tau'], g, g_1, g_2) \Rightarrow C(\tau[\tau'], f \sqcup g, f_1 \sqcup g_1, f_2 \sqcup g_2) \tag{2.12}$$
$$\text{Si } k \subseteq \gamma_2^{\tau'}(0),\ C(\tau[\tau'], \lambda c.k, \lambda c_1.0, \lambda c_2.0) \tag{2.13}$$
$$\text{Sinon, } C(\tau[\tau'], \lambda c.k, \lambda c_1.c_1 \sqcup k_1, \lambda c_2.k_2) \text{ où } (k_1, k_2) = \alpha_1^{\tau'}(k) \tag{2.14}$$
$$C(\tau[\tau], \lambda c.c, I_1, I_2) \tag{2.15}$$
$$C(\tau[\tau'], \lambda c.\mathrm{g}.c, I_1, I_2) \tag{2.16}$$
$$C(\tau[\tau'], \lambda c.c_{|g}, I_1, \lambda c.c \sqcap \top_2[\tau']) \tag{2.17}$$

Si $hc = \begin{cases} fc & \text{si } c \in A \\ gc & \text{si } c \in B \end{cases}$ pour $c \in Chem$, la définition correspondante pour $c \in Ctx$ est $hc = \cup_{c' \in A \cap c} fc' \cup \cup_{c' \in B \cap c} gc'$. On peut se contenter de traiter le premier terme en utilisant (2.12). Posons donc : $hc = \cup_{c' \in A \cap c} fc'$, $h_1 c_1 = \begin{cases} f_1 c_1 & \text{si } c_1 = 1 \text{ ou } A \cap \gamma_2^{\tau}(\mathbb{N}) \neq \emptyset \\ 0 & \text{sinon} \end{cases}$, $h_2 c_2 = \begin{cases} f_2 c_2 & \text{si } A \cap \gamma_2^{\tau}(c_2) \neq \emptyset \\ 0 & \text{sinon} \end{cases}$.
Alors

$$C(\tau[\tau'], f, f_1, f_2) \Rightarrow C(\tau[\tau'], h, h_1, h_2) \tag{2.18}$$

$$\begin{aligned} &\forall z, \forall x, C(\tau_z[\tau_x], \lambda c.\rho[\![z]\!]cx, \lambda c_1.\rho_1[\![z]\!]c_1 x, \lambda c_2.\rho_2[\![z]\!]c_2 x) \text{ et} \\ &(\forall z, \forall x, C(\tau_z[\tau_x], \lambda c.\rho[\![z]\!]cx, \lambda c_1.\rho_1[\![z]\!]c_1 x, \lambda c_2.\rho_2[\![z]\!]c_2 x) \\ &\qquad \Rightarrow \forall x.C(\tau[\tau_x], \lambda c.f\rho cx, \lambda c_1.f_1\rho_1 c_1 x, \lambda c_2.f_2\rho_2 c_2 x)) \\ &\Rightarrow \forall z, \forall x, C(\tau_z[\tau_x], \lambda c.\,\mathrm{lfp}(\lambda\rho'.\rho[\mathtt{f} \mapsto f\rho' \sqcup \lambda c.\{\mathtt{f} \mapsto c\}])(z)cx, \\ &\quad \lambda c_1.\,\mathrm{lfp}(\lambda\rho'.\rho_1[\mathtt{f} \mapsto f_1\rho' \sqcup \lambda c.\{\mathtt{f} \mapsto c\}])(z)cx.\lambda c_2.\,\mathrm{lfp}(\lambda\rho'.\rho_2[\mathtt{f} \mapsto f_2\rho' \sqcup \lambda c.\{\mathtt{f} \mapsto c\}])(z)c_2 x) \end{aligned} \tag{2.19}$$

– Preuve de (2.11) :

Si $f_1((g_1 \sqcup I_1)1) = f_1 1 = 0$, $f(g\ Chem) \subseteq f\ Chem \subseteq \gamma_2^{\tau''}(0)$.

Si $f_1((g_1 \sqcup I_1)0) = f_1(g_1 0) = 0$, soit $g_1 0 = 1$ et $f_1 1 = 0$ donc comme dans le cas précédent, $f(g\ Chem) \subseteq f\ Chem \subseteq \gamma_2^{\tau''}(0) \subseteq \gamma_2^{\tau''}(f_2(g_2 c))$, soit $g_1 0 = 0$ et $f_1 0 = 0$ et on a $f(\gamma_2^{\tau'}(c)) \subseteq \gamma_2^{\tau''}(f_2 c)$ et $g(\gamma_2^{\tau}(c)) \subseteq \gamma_2^{\tau'}(g_2 c)$, donc $f(g(\gamma_2^{\tau}(c))) \subseteq f(\gamma_2^{\tau'}(g_2 c)) \subseteq \gamma_2^{\tau''}(f_2(g_2 c))$.

– (2.16) résulte de $\mathrm{g}.\gamma_2^{\tau}(c) \subseteq \gamma_2^{\tau'}(c)$ (où τ' vaut $\tau * \tau_2$, τ `ref` ou τ `list`). En effet, supposons $p_0 \in \gamma_2^{\tau}(c)$. Alors $\alpha_2^{\tau'}(\mathrm{g}.p_0) = \alpha_2^{\tau}(p_0) \subseteq c$ d'où le résultat voulu.

– (2.17) résulte de $\gamma_2^{\tau}(c)_{|g} \subseteq \gamma_2^{\tau'}(c \sqcap \top_2[\tau'])$. En effet, supposons $p_0 \in \gamma_2^{\tau}(c)_{|g}$ alors $\mathrm{g}.p_0 \in \gamma_2^{\tau}(c)$ donc $\alpha_2^{\tau}(\mathrm{g}.p_0) = \alpha_2^{\tau'}(p_0) \leq c$. De plus, $\alpha_2^{\tau'}(p_0) \leq \top_2[\tau']$ car la hauteur d'une sous-structure de donnée est inférieure à celle de la structure entière. Alors $\alpha_2^{\tau'}(p_0) \leq c \sqcap \top_2[\tau']$, ce qui donne le résultat voulu.

– Preuve de (2.18) :

Si $h_1 1 = f_1 1 = 0$, $f\ Chem \subseteq \gamma_2^{\tau'}(0)$ d'où $\cup_{c' \in A} fc' \subseteq \gamma_2^{\tau'}(0)$.

Si $h_1 0 = 0$, soit $A \cap \gamma_2^{\tau}(\mathbb{N}) = \emptyset$, et $\cup_{c' \in A \cap \gamma_2^{\tau}(c)} fc' = \perp \subseteq \gamma_2^{\tau'}(h_2(c))$, soit $A \cap \gamma_2^{\tau}(\mathbb{N}) \neq \emptyset$ et $f_1 0 = 0$ donc $f(\gamma_2^{\tau}(c)) \subseteq \gamma_2^{\tau'}(f_2 c)$ alors $h(\gamma_2^{\tau}(c)) = \cup_{c' \in A \cap \gamma_2^{\tau}(c)} fc' \subseteq \gamma_2^{\tau'}(h_2 c)$.

– La preuve de (2.19) se fait par une induction transfinie, en utilisant (2.11), facile malgré la lourdeur des notations.

Les autres cas sont plus simples et laissés au lecteur. On montre des propriétés analogues à (2.16) et (2.17) pour les autres constructions et restrictions. On peut alors traduire étape par étape l'analyse E dans les analyses F_1 et F_2. On obtient facilement les analyses annoncées, sauf dans le cas des fermetures, où on obtient :

$$F_1[\![\texttt{fun}\ y \to M]\!]\rho_1 c = \begin{cases} \sqcup_{z \in FV(\texttt{fun}\ y \to M)} \rho_1[\![z]\!]1 & \text{si } c_1 = 1 \\ 0 & \text{si } c_1 = 0 \end{cases} \sqcup \{1 \mapsto \phi(y), i \mapsto \phi(i-1) \text{ si } i > 1, x \mapsto \phi(x)\}$$

Le lemme 2.26 permet de montrer que le premier terme est majoré par le second et d'en déduire la forme annoncée.

□

Théorème 2.28 (Correction des analyses F_1 et F_2) *Considérons l'expression :* `let` x `=` M `in` N *ou* `let rec` $x(y)$ `=` M `in` N. *Soit l la cellule au sommet de x. Supposons que la création de cette cellule soit la dernière opération de M. Supposons de plus que :*

$$\begin{cases} \textit{si ni } N \textit{ ni } x \textit{ ne sont fonctionnels,} & F_1[\![N]\!]\rho_1 0x = 0 \textit{ et } F_2[\![N]\!]\rho_2 \top_{2N} x < \top_{2x} \\ \textit{si } N \textit{ est fonctionnel,} & F_1[\![N]\!]\rho_1 1x = 0 \\ \textit{si } N \textit{ est non fonctionnel et } x \textit{ est fonctionnel,} & F_1[\![N]\!]\rho_1 0x = 0 \end{cases} \tag{2.20}$$

Alors l est allouable en pile.

Preuve : La condition (2.20) combinée au théorème 2.27 montre que $E[\![N]\!]\rho\ Chem\, x \subseteq a$, où $\top \notin a$. En effet, si x est fonctionnel, $a = \gamma_2^{x}(b)$ ne contient pas de type fonctionnel. Sinon, $a = \gamma_2^{x}(b)$ avec $b < \top_{2x}$ donc les données référencées par a ont des types sous-types stricts du type de x. Alors $\top \notin E[\![N]\!]\rho\ Chem\, x$.

Le théorème 2.24 permet alors de conclure. □

2.5 Traitement du polymorphisme

On analyse d'abord toutes les expressions en les considérant comme monomorphes, c'est-à-dire que les variables de type sont supposées de type atomique : $\top_2[\beta] = 1, \alpha_2^\beta(c_1.\bot) = 0, \alpha_2^\beta(c_1.\top) = 1$ si β est une variable de type. On peut alors en déduire une analyse approchée pour les instanciations de cette expression, mais attention, ceci peut entraîner une perte de précision par rapport à l'analyse directe de l'instanciation en question. Par exemple, si $\tau = 'a * 'b$ et qu'on l'instancie par $\sigma = \{'a \mapsto$ `bool list list`$, 'b \mapsto$ `bool`$\}$, si on sait que le niveau 1 de τ s'échappe, ce qui correspond à $'a$ ou $'b$, on dira que le niveau 3 de $\sigma\tau$ s'échappe, alors qu'en fait si $'b$ seul s'échappe, l'analyse directe aurait indiqué que le niveau 1 de $\sigma\tau$ s'échappe, ce qui est nettement plus précis.

Mais chose plus surprenante, l'analyse obtenue par instanciation peut aussi être plus précise que l'analyse directe. Par exemple,

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

Ici, l'analyse donne $F_2[\![\texttt{phi}]\!]\rho c1 = c \sqcup \top_2[\tau]$ si `y` est de type τ. Supposons qu'on effectue une analyse sur la version générale $\tau = \beta$. Alors $F_2[\![\texttt{phi}]\!]\rho c1 = c \sqcup 1$ et l'instanciation laisse cette valeur (car on instancie relativement au type de `x` indépendamment de celui de `y`). Par contre, l'analyse directe avec $\tau =$ `int list` par exemple donne $F_2[\![\texttt{phi}]\!]\rho c1 = c \sqcup 2$ ce qui est moins précis.

Finalement, l'analyse obtenue par instanciation et l'analyse directe sont incomparables. Autrement dit, les analyses F_1 et F_2 ne sont pas polymorphiquement invariantes [Abr86]. On peut cependant prouver la correction par rapport à l'analyse E.

Définition 2.29 Soit σ une substitution fixée sur les variables de type. Les fonctions d'instanciation et de généralisation sont respectivement :

$$I_2^\tau(c') = \begin{cases} \sqcup\{\top_2[\sigma\tau'] \mid \tau' \text{ sous-expression de } \tau, \top_2[\tau'] \le c'\} & \text{si } c' > 0 \\ 0 & \text{si } c' = 0 \end{cases}$$

$$G_2^\tau(c) = \begin{cases} \sqcup\{\top_2[\tau'] \mid \tau' \text{ sous-expression de } \tau, \top_2[\sigma\tau'] \le c\} \sqcup 1 & \text{si } c > 0 \\ 0 & \text{si } c = 0 \end{cases}$$

où τ est le type associé aux contextes qu'on instancie ou qu'on généralise, et la définition des sous-expressions diffère de la définition habituelle pour le type fonction : ce sont uniquement les sous-expressions du type du résultat. Elles ne forment pas une correspondance de Galois. Pour l'analyse F_1, l'instanciation est $I_1^\tau(c, b) = 1$ si σ substitue certaines variables de type de τ par des fonctions et $b \neq 0$, et $I_1^\tau(c, b) = c$ sinon. La généralisation est $G_1^\tau(c) = c$.

Lemme 2.30 *$\alpha_2^{\sigma\tau} \le I_2^\tau \circ \alpha_2^\tau$ quand $\alpha_2^{\sigma\tau} \neq \infty$. Ceci exprime que $I_2^\tau(c)$ correspond à au moins autant de chemins dans le type $\sigma\tau$ que c dans le type τ.*

Preuve : Soit $c \in \mathit{Chem}$. Si $c = c_1.\bot$, $\alpha_2^{\sigma\tau}(c) = 0$ donc on a le résultat. Si $c = c_1.\top$, soit il existe une sous-expression τ_0 de τ telle que $\alpha_2^\tau(c_1.\top) = \alpha_2^{\tau_0}(\top)$. Alors,

$$\begin{aligned} \alpha_2^{\sigma\tau}(c_1.\top) &= \alpha_2^{\sigma\tau_0}(\top) \\ &= \top_2[\sigma\tau_0] \\ &\le I_2^\tau(\top_2[\tau_0]) \text{ par définition de } I_2 \\ &\le I_2^\tau(\alpha_2^{\tau_0}(\top)) \\ &\le I_2^\tau(\alpha_2^\tau(c_1.\top)) \end{aligned}$$

ce qui est le résultat voulu. Soit $\alpha_2^\tau(c_1.\top) = \alpha_2^\beta(c_1'.\top) = 1$. Alors,

$$\begin{aligned}\alpha_2^{\sigma\tau}(c_1.\top) &= \alpha_2^{\sigma\beta}(c_1'.\top)\\ &\leq \top_2[\sigma\beta] \text{ car } \alpha_2^{\sigma\beta}(c_1'.\top) \neq \infty\\ &\leq I_2^\tau(\top_2[\beta]) \text{ par définition of } I_2\\ &\leq I_2^\tau(1)\\ &\leq I_2^\tau(\alpha_2^\tau(c_1.\top))\end{aligned}$$

ce qui est encore le résultat voulu. Le passage aux ensembles de chemins est facile car α_2 est additive et I_2 est croissante. $\square$

Théorème 2.31 (Instanciation) *Soit* $f = \lambda c.E[\![M]\!]\rho cx$.

$$\begin{aligned}&C(\tau[\tau_x], f, \lambda c.(c \sqcap a_1) \sqcup b_1, \lambda c.(c \sqcap a_2) \sqcup b_2)\\ &\quad\Rightarrow C(\sigma\tau[\sigma\tau_x], f, \lambda c.(c \sqcap a_1) \sqcup I_1^{\tau_x}(b_1, b_2), \lambda c.(c \sqcap I_2^{\tau_x}(a_2)) \sqcup I_2^{\tau_x}(b_2))\end{aligned}$$

Ceci est utile dans l'implémentation car toutes les fonctions manipulées sont de cette forme (on le vérifie facilement sur les équations, car cette forme est stable par composition, par minimum et par maximum). Ceci prouve la correction des fonctions d'instanciation I_2^τ *et* I_1^τ.

Preuve : On a donc par hypothèse :

$$\begin{cases}\text{Si } a_1 \sqcup b_1 = 0,\ f\ \mathit{Chem} \subseteq \gamma_2^{\tau_x}(0)\\ \text{Si } b_1 = 0,\ f(\gamma_2^\tau(c)) \subseteq \gamma_2^{\tau_x}((c \sqcap a_2) \sqcup b_2)\end{cases}$$

Il s'agit de montrer :

- Si $a_1 \sqcup I_1^{\tau_x}(b_1, b_2) = 0$, $f\ \mathit{Chem} \subseteq \gamma_2^{\sigma\tau_x}(0)$.
  Or $a_1 = b_1 = 0$ donc $f\ \mathit{Chem} \subseteq \gamma_2^{\tau_x}(0) = \gamma_2^{\sigma\tau_x}(0)$.
- Si $I_1^{\tau_x}(b_1, b_2) = 0$, $f(\gamma_2^{\sigma\tau}(c)) \subseteq \gamma_2^{\sigma\tau_x}((c \sqcap I_2^{\tau_x}(a_2)) \sqcup I_2^{\tau_x}(b_2))$.
  Or $b_1 = 0$ donc $f(\gamma_2^\tau(c)) \subseteq \gamma_2^{\tau_x}((c \sqcap a_2) \sqcup b_2)$. On procède alors par induction sur la forme de $f = \lambda c.E[\![M]\!]\rho cx$ pour montrer que, si $b_1 = 0$ et σ ne substitue pas de variable de type de τ_x par des fonctions ou que $b_2 = 0$, $f(\gamma_2^\tau(c)) \subseteq \gamma_2^{\tau_x}((c \sqcap a_2) \sqcup b_2) \Rightarrow f(\gamma_2^{\sigma\tau}(c)) \subseteq \gamma_2^{\sigma\tau_x}((c \sqcap I_2^{\tau_x}(a_2)) \sqcup I_2^{\tau_x}(b_2))$. On peut toujours mettre f sous la forme $f(c) = \bigsqcup_{i \in I} c_i.c_{|c_i'} \sqcup c_1$ où $c_i \in (\text{app}|\text{g}|\text{d})^*$, $c_i' \in (\rightarrow |\text{g}|\text{d})^*$, $c_1 \in \mathit{Ctx}$ et où l'union peut être infinie (on le voit facilement car cette forme est stable par composition, borne supérieure, restriction et construction de chemins et par point fixe).
  - Si $f(c) = c_1$, il suffit de montrer que $c_1 \subseteq \gamma_2^{\tau_x}(b_2) \Rightarrow c_1 \subseteq \gamma_2^{\sigma\tau_x}(I_2^{\tau_x}(b_2))$ donc que $\gamma_2^{\tau_x}(b_2) \subseteq \gamma_2^{\sigma\tau_x} \circ I_2^{\tau_x}(b_2)$. Si $b_2 = 0$ c'est évident. Sinon, σ ne substitue pas de variable de type de τ_x par des fonctions, tous les chemins de $\gamma_2^{\tau_x}(b_2)$ sont non fonctionnels dans τ_x donc aussi dans le type $\sigma\tau_x$, donc $\alpha_2^{\sigma\tau_x}(\gamma_2^{\tau_x}(b_2)) \neq \infty$. Alors le lemme 2.30 donne : $\alpha_2^{\sigma\tau_x} \circ \gamma_2^{\tau_x} \leq I_2^{\tau_x} \circ \alpha_2^{\tau_x} \circ \gamma_2^{\tau_x} = I_2^{\tau_x}$. Donc la propriété de correspondance de Galois donne le résultat voulu.
  - Si $f(c) = c_1.c_{|c_2}$, si $c_1.\gamma_2^\tau(c)_{|c_2} \subseteq \gamma_2^{\tau_x}((c \sqcap a_2) \sqcup b_2)$, considérons le type τ_0 associé aux contextes $c_{|c_2}$ et $f(c)_{|c_1}$, en prenant $c = \top_2[\tau_0]$, $a_2 \sqcup b_2 \geq \alpha_2^{\tau_x}(c_1.\mathit{NF}(\tau_0)) \geq \alpha_2^{\tau_x}(c_1.\mathit{NF}(\sigma\tau_0))$ où $\mathit{NF}(\tau)$ est l'ensemble des chemins menant à un type non fonctionnel dans τ. En effet, $\mathit{NF}(\tau_0) \supseteq \mathit{NF}(\sigma\tau_0)$. Alors

    $$I_2^{\tau_x}(a_2) \sqcup I_2^{\tau_x}(b_2) \geq I_2^{\tau_x} \circ \alpha_2^{\tau_x}(c_1.\mathit{NF}(\sigma\tau_0)) \geq \alpha_2^{\sigma\tau_x}(c_1.\mathit{NF}(\sigma\tau_0))$$

    d'après le lemme 2.30, donc

    $$\begin{aligned}\gamma_2^{\sigma\tau_x}((c \sqcap I_2^{\tau_x}(a_2)) \sqcup I_2^{\tau_x}(b_2)) &\supseteq \gamma_2^{\sigma\tau_x}(c \sqcap \alpha_2^{\sigma\tau_x}(c_1.\mathit{NF}(\sigma\tau_0)))\\ &\supseteq \gamma_2^{\sigma\tau_x}(c) \sqcap c_1.\mathit{NF}(\sigma\tau_0)\\ &\supseteq c_1.\gamma_2^{\sigma\tau}(c)_{|c_2}\end{aligned}$$

car

$$\alpha_2^{\sigma\tau_x}(c_1.\gamma_2^{\sigma\tau}(c)_{|c_2}) \leq \alpha_2^{\sigma\tau_0}(\gamma_2^{\sigma\tau}(c)_{|c_2}) \leq \alpha_2^{\sigma\tau}(\gamma_2^{\sigma\tau}(c)) = c$$

– Si $f(c) = \bigsqcup_{i\in I} f_i(c)$: facile en utilisant la croissance de I_2.

Ceci donne le résultat voulu.

□

Exemple 2.32 La fonction `map` est de type $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`. L'analyse de la version polymorphe donne :

$$\rho_1[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c, 2 \mapsto c\}$$
$$\rho_2[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c \sqcap 1, 2 \mapsto 1\}$$

Quand on instancie $\alpha = \tau_1, \beta = \tau_2$, où τ_1 et τ_2 ne sont pas des types fonctionnels :

$$\rho_1[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c, 2 \mapsto c\}$$
$$\rho_2[\![\texttt{map}]\!]c = \{\texttt{map} \mapsto c, 1 \mapsto c \sqcap \top_2[\tau_2], 2 \mapsto \top_2[\tau_1]\}$$

Lemme 2.33 $\alpha_2^\tau \leq G_2^\tau \circ \alpha_2^{\sigma\tau}$. *Ceci exprime que $G_2^\tau(c)$ correspond à au moins autant de chemins dans le type τ que c dans le type $\sigma\tau$.*

Preuve : Soit $c \in$ *Chem*. Si $c = c_1.\bot$, $\alpha_2^\tau(c) = 0$ donc on a les résultat voulu. Si $c = c_1.\top$, soit il existe τ_0 une sous-expression de τ telle que $\alpha_2^\tau(c_1.\top) = \alpha_2^{\tau_0}(\top)$. Alors,

$$\begin{aligned}\alpha_2^\tau(c_1.\top) &= \alpha_2^{\tau_0}(\top)\\ &= \top_2[\tau_0]\\ &\leq G_2^\tau(\top_2[\sigma\tau_0]) \text{ par définition de } G_2\\ &\leq G_2^\tau(\alpha_2^{\sigma\tau_0}(\top))\\ &\leq G_2^\tau(\alpha_2^{\sigma\tau}(c_1.\top))\end{aligned}$$

ce qui est le résultat voulu. Sinon, c correspond à la pénétration dans une variable de type de τ, donc $\alpha_2^\tau(c) = 1 \leq G_2^\tau \circ \alpha_2^{\sigma\tau}(c)$. Le passage aux ensembles de chemins est facile car α_2 est additive et G_2 est croissante. □

Théorème 2.34 (Généralisation)

$$C(\tau[\tau_x], f, f_1, f_2) \Rightarrow C(\sigma\tau[\tau_x], f, f_1, f_2 \circ G_2^\tau)$$

Ceci montre la correction des fonctions de généralisation $G_1 = \mathrm{id}$ et G_2. Ceci est utile en particulier quand on instancie une variable y de type τ dans le type $\sigma\tau$ (cas $f = \lambda c.E[\![y]\!]\rho c x$). On posera alors

$$F_1[\![\Gamma, y : \tau \vdash y : \sigma\tau]\!]\rho_1 = \rho_1 \textit{ et } F_2[\![\Gamma, y : \tau \vdash y : \sigma\tau]\!]\rho_2 c x = \rho_2(G_2^\tau(c))x$$

Preuve : On a par hypothèse,

$$\begin{cases}\text{Si } f_1 1 = 0,\ f\ \mathit{Chem} \subseteq \gamma_2^{\tau_x}(0)\\ \text{Si } f_1 0 = 0,\ f(\gamma_2^\tau(c)) \subseteq \gamma_2^{\tau_x}(f_2 c)\end{cases}$$

On veut prouver que

– Si $\rho_1 1x = 0$, $f\ \mathit{Chem} \subseteq \gamma_2^{\tau_x}(0)$: c'est dans les hypothèses.

– Si $\rho_1 0x = 0$, $f(\gamma_2^{\sigma\tau}(c)) \subseteq \gamma_2^{\tau_x}(f_2(G_2^\tau(c)))$. On a par hypothèse, $f(\gamma_2^\tau(G_2^\tau(c))) \subseteq \gamma_2^{\tau_x}(f_2(G_2^\tau(c)))$, il suffit alors de prouver que $\gamma_2^{\sigma\tau}(c) \subseteq \gamma_2^\tau(G_2^\tau(c))$ ce qu'on déduit du lemme 2.33.

□

Chapitre 3

Implémentation de l'analyse

3.1 Description générale de l'algorithme

Le programme d'analyse fonctionne en deux passes. Il prend en argument l'arbre de syntaxe abstraite du programme à analyser. La première passe le transforme en un arbre étiqueté avec toutes les informations fournies par l'analyse d'échappement. La seconde passe prend cet arbre étiqueté et décide des transformations à effectuer dans le programme pour l'optimiser.

3.2 Calcul des hauteurs des types

Comme on a vu dans la section 2.4, on associe à chaque type τ, sa hauteur $\top_2[\tau]$. Cette hauteur doit vérifier la propriété suivante : si une valeur v_1 de type τ_1 est à l'intérieur de v_2 de type τ_2, $\top_2[\tau_1] \leq \top_2[\tau_2]$. Alors, dans le cas de types récursifs comme les listes ou certains types définis par l'utilisateur, toutes les cellules d'une même composante fortement connexe du type doivent avoir la même hauteur. Entre composantes fortement connexes distinctes, on ajoute 1 à la hauteur.

Exemple 3.1 Considérons la déclaration de type suivante :

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

$\top_2[\texttt{term}] = \top_2[\texttt{term list}] = \top_2[\texttt{term} * \texttt{term}] = \top_2[(\texttt{term} * \texttt{term})\texttt{list}] = \top_2[\texttt{head}] = 2$ et $\top_2[\texttt{head list}] = 1 + \top_2[\texttt{head}] = 3$ car ce type n'est pas dans la même composante fortement connexe.

On peut remarquer que cette définition est compatible avec la définition précédente de $\top_2$ pour le type `list` :

```
type 'a list = Nil | Cons of 'a * 'a list
```

De plus, cette contrainte ne peut pas toujours être respectée : on ne sait pas quelles sont les valeurs à l'intérieur des fermetures. Dans ce cas, on dit que le type considéré ne préserve pas les niveaux, et il est traité comme les types fonctionnels dans la section 2.4. D'autre part, si τ est un type abstrait, $\top[\tau] = 1$ (mais τ préserve les niveaux, car on ne peut rien extraire de τ).

Un autre cas qui pose des difficultés est celui des types récursifs non monotones : dans la définition d'un constructeur de type polymorphe, on peut faire appel à ce même constructeur avec des paramètres différents, par exemple

α `t = None | Some of` α `*` α `list t`

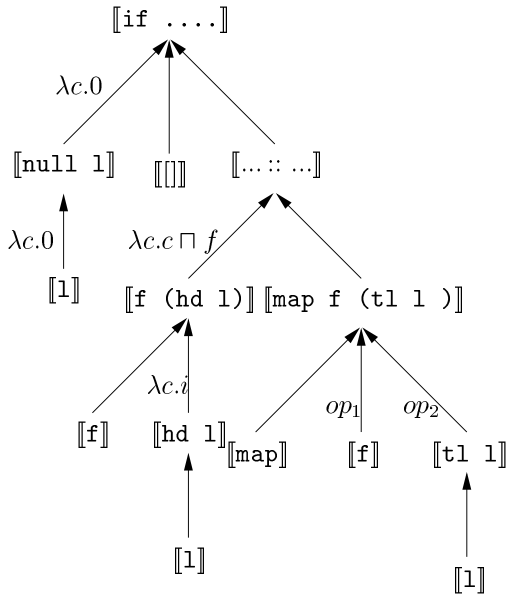


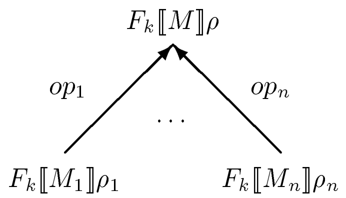
FIG. 3.1 – *Arbre construit pour l'exemple 2.20. L'opération identité a été omise.* op_1 *et* op_2 *sont les inconnues qui représentent l'échappement des paramètres de la fonction* `map`

La détection de cycle ne suffit alors pas car on fait appel à une infinité de types différents α `list` ... `list t`. Ce cas est détecté et traité séparément : le type est déclaré ne pas préserver les niveaux, et sa hauteur vaut 1. De toute façon, un tel type n'a pas d'utilité pratique, car on ne peut pas écrire d'itérateur dessus. Le cas où on fait appel dans la définition de `t` à plusieurs instances de `t` mais à un nombre fini est traité de même que le cas infini par souci de simplicité, bien qu'on puisse ici faire mieux (ces types sont rarement utiles).

Ce calcul étant assez lent, on l'optimise en traduisant le langage de type d'origine vers des types étiquetés avec leur hauteur. Ceci évite d'avoir à recalculer la hauteur plusieurs fois pour le même type.

3.3 Construction des équations sémantiques

On représente les équations sémantiques par un anti-arbre dont les nœuds sont les occurrences de $F_k[\![M]\!]\rho$ et les arêtes sont telles que $F_k[\![M]\!]\rho = \sqcup_i F_k[\![M_i]\!]\rho_i \circ op_i$ si $F_k[\![M]\!]\rho$ est lié par des arêtes étiquetées op_i à $F_k[\![M_i]\!]\rho_i$. On voit facilement sur la définition de F_1 et F_2 que toutes les opérations à considérer sont de la forme $op = \lambda c.(c \sqcap f) \sqcup i$. On peut alors représenter ces opérations par les couples (f, i).



On définit la composition des opérations sur les couples associés :

$$(f_1, i_1) \circ (f_2, i_2) = (f_1 \sqcap f_2, (i_2 \sqcap f_1) \sqcup i_1)$$

Si certaines de ces valeurs sont inconnues, il faut créer de nouvelles inconnues pour représenter l'opération composée et émettre des équations qui traduisent la formule ci-dessus.

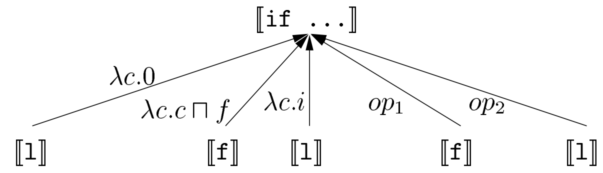


FIG. 3.2 – *Figure 3.1 après compression de chemin*

L'anti-arbre est muni de deux opérations *eval* et *link* comme dans l'article de Tarjan [Tar79]. $eval(n)$ calcule la composée des opérations qui étiquettent les arêtes du nœud n à la racine de l'anti-arbre en effectuant de la compression de chemin : tous les nœuds sur le chemin de n à la racine sont liés directement à la racine après $eval(n)$ et les étiquettes des arêtes sont correctement mises à jour. $link(n_1, n_2, op)$ lie le nœud n_1 au nœud n_2 avec l'opération op (n_1 doit être une racine d'anti-arbre, car chaque nœud n'a qu'un père).

On représente le calcul de point fixe en utilisant des opérations op_i inconnues, que l'on détermine par la méthode de résolution décrite dans la section suivante.

On maintient la relation $F_k[\![M]\!]\rho = \sqcup_{x \in FV(M), n \in \delta(x)} \rho[\![x]\!] \circ eval(n)$ quand on analyse M, où $\delta(x)$ est l'ensemble des nœuds représentant des occurences de x. Ceci permet de calculer F_k. L'échappement d'une variable x est $\sqcup_{n \in \delta(x)} eval(n)$ où les *eval* doivent remonter jusqu'à l'expression qui lie x comme on peut facilement le déduire de la définition des étiquettes. Pour effectuer ce calcul, on mémorise pour chaque variable la liste de ses nœuds occurences, le calcul lui-même est retardé jusqu'à la remontée dans le lieur de x.

L'échappement d'une expression M dans une de ses sur-expressions N est la composition des opérations du nœud n qui représente M à celui qui représente N. On la calcule en effectuant $eval(n)$ dans l'analyse de N. Pour chaque expression, on souhaite calculer son échappement dans l'expression immédiatement englobante, qui est simplement l'étiquette de l'arête qui lie n à son père, mais aussi jusqu'à la plus grande expression où on peut faire remonter les `let` de M (dite expression de contrôle). Cette expression est la première expression englobante de la forme :

```
fun y -> []
if [] then [] else []
let rec x = [] in M
[] && []
[] or []
match [] with p -> []
try [] with p -> []
M <- []
[]; []
while [] do [] done
for i=M to N do [] done
[] when []
```

Autrement dit, si un `let` est dans une expression `[]` de la liste ci-dessus, il n'en sort pas.

Cette définition est quelque peu heuristique. Elle doit surtout veiller à ne jamais provoquer d'évaluations supplémentaires ou changer le nombre d'évaluations du fait du déplacement d'un `let`. D'autre part, il est inutile de remonter davantage un `let` quand on sait que le contexte d'échappement ne changera plus (cas du membre droit de `<-`). Ce contexte d'échappement permet de savoir si on va pouvoir placer une valeur en pile : elle ne doit pas s'échapper de la plus grande expression de contrôle. L'échappement vis-à-vis de l'expression immédiatement englobante servira à déterminer plus précisément la durée de vie à donner à cette valeur.

On calcule aussi un autre contexte : l'échappement dans les appels terminaux. Le calcul est identique au précédent sauf que lors d'un appel terminal, on considère que tous les paramètres s'échappent. On ajoute pour cela à chaque opération op un troisième élément i_t en plus du couple (f, i) car la valeur de f est la même que pour l'analyse normale. La loi de composition devient sur les triplets :

$$(f_1, i_1, i_{t1}) \circ (f_2, i_2, i_{t2}) = (f_1 \sqcap f_2, (i_2 \sqcap f_1) \sqcup i_1, (i_{t2} \sqcap f_1) \sqcup i_{t1})$$

Cet échappement permet de déterminer si on peut libérer les valeurs avant l'appel terminal, ce qui permet de conserver les optimisations des appels terminaux.

Exemple 3.2 Reprenons l'exemple de la fonction `map` et supposons que l'expression `map g a` est en position terminale. Alors dans ce formalisme à triplets, les analyses sont codées comme suit :

$$F_1[\![\texttt{map g a}]\!]\rho_1 = \{\texttt{g} \mapsto (0,0,1), \texttt{a} \mapsto (0,0,0), \texttt{map} \mapsto (0,0,1)\}$$

$$F_2[\![\texttt{map g a}]\!]\rho_2 = \{\texttt{g} \mapsto (\top_{2\texttt{g}}, 0, \top_{2\texttt{g}}), \texttt{a} \mapsto (\top_{2\texttt{a}} - 1, 0, \top_{2\texttt{a}}), \texttt{map} \mapsto (\top_{2\texttt{map}}, 0, \top_{2\texttt{map}})\}$$

L'analyse F_1 exprime que les fermetures de `map` et `g` s'échappent dans l'appel terminal car le dernier élément du triplet, i_t, vaut 1 pour ces deux variables. Ce sont les seules fermetures qui s'échappent. L'analyse F_2 exprime que le résultat de `g`, `a` et le résultat de `map` s'échappent dans l'appel terminal (car i_t vaut la hauteur de chacune de ces variables). Pour `g` et `map`, F_1 vient compléter comme on l'a vu pour indiquer que la fermeture s'échappe. D'autre part, le résultat de `g`, les éléments de la liste `a` et le résultat de `map` s'échappent quand le résultat de `map g a` s'échappe (valeur de f) tandis que rien ne s'échappe si le résultat ne s'échappe pas (valeur de i) ce qui logique car ce programme ne contient aucune construction impérative.

3.4 Résolution

On utilise une généralisation de l'algorithme de Dijkstra des plus courts chemins dans un graphe, donnée par Knuth dans [Knu77].

Cet algorithme utilise une file de priorité qui peut être implémentée efficacement avec un tas binaire, ou mieux un tas de Fibonacci, comme expliqué dans [CLR94].

On cherche le plus petit point fixe d'un système d'équations $Y = \sqcup_i g_i(X_1, \ldots, X_k)$ où les g_i sont des fonctions inférieures : $g_i(x_1, \ldots, x_k) \leq \min(x_1, \ldots, x_k)$. Soit G la grammaire dont les non-terminaux sont les inconnues X_i, et les règles de production $Y \longrightarrow g_i(X_1, \ldots, X_k)$. Soit T l'alphabet terminal, formé des symboles de fonction g, des parenthèses et de la virgule et $L(Y) = \{\alpha \mid \alpha \in T^* \text{ et } Y \longrightarrow^* \alpha\}$ le langage engendré par Y. Soit $val(\alpha)$ la valeur de la chaîne α. Le plus petit point fixe cherché est alors : $m(Y) = \max\{val(\alpha) \mid \alpha \in L(Y)\}$.

L'algorithme est le suivant :
Soit P la file de priorité.
Au départ P contient pour chaque non terminal la clé 0.
Pour chaque équation $Y = g$ (constante), mettre à jour clé$(Y) = \max(\text{clé}(Y), g)$.
Tant que P est non vide,

- Extraire le non-terminal Y de clé maximale dans P. On a alors $m(Y) = \text{clé}(Y)$
- Pour chaque non-terminal qui est encore dans P, mettre à jour

  $$\text{clé}(Y') = \max(\text{clé}(Y'), g(\text{clé}(X_1), \ldots, \text{clé}(X_k)))$$

  pour chaque équation $Y' = g(X_1, \ldots, X_k)$ telle que $X_1, \ldots, X_k$ ne sont pas dans P et Y est un des $X_1, \ldots, X_k$.

La preuve de correction de cet algorithme est dans [Knu77]. Chaque équation est utilisée exactement une fois, quand on sort de P le dernier non-terminal argument de g. L'idée est alors de maintenir pour chaque non-terminal la liste des équations qui le contiennent, avec un compteur indiquant le nombre de non-terminaux de cette équation qui sont dans P. On applique l'équation quand le compteur s'annule.

Supposons que les performances de la file de priorité sont données par T_{max} pour le temps d'extraction du maximum et T_{maj} pour le temps de mise à jour d'une clé. La complexité est alors :

- Temps des extractions de maximum = nombre d'inconnues (noté i) $\times T_{max}$

– Temps de calcul des valeurs de mise à jour = temps total du calcul des équations (noté t)

– Temps des mises à jour des clés = nombre d'équations (noté e) $\times T_{maj}$

– Temps de mise à jour des compteurs = longueur totale des équations = $\mathcal{O}(t)$

d'où la complexité totale $C = \mathcal{O}(i \times T_{max} + e \times T_{maj} + t)$ soit pour une file de priorité implémentée par un tas binaire : $T_{max} = \mathcal{O}(\log i)$, $T_{maj} = \mathcal{O}(\log i)$ donc $C = \mathcal{O}(e \times \log i + t)$ et pour un tas de Fibonacci (en temps amorti) : $T_{max} = \mathcal{O}(\log i)$, $T_{maj} = \mathcal{O}(1)$ donc $C = \mathcal{O}(i \times \log i + t)$. Etant donné la taille des systèmes à résoudre (rarement plus de 10 équations), et le fait qu'en général le nombre d'équations est du même ordre que celui des inconnues, on a estimé que le gain de complexité de l'implémentation avec un tas de Fibonacci ne valait pas la peine, d'autant plus que la constante est plus grande. On a donc utilisé un tas binaire.

L'opération d'instanciation décrite à la section 2.5 ne respecte pas la condition de Knuth : elle n'est pas inférieure. Pour pouvoir utiliser quand même l'algorithme de Knuth, on commence par découper le système en composantes fortement connexes, par l'algorithme classique qui utilise deux parcours en profondeur (voir par exemple [CLR94]). On résout alors séparément chaque composante avec l'algorithme de Knuth. A l'intérieur d'une composante, on approche l'instanciation par la fonction constante égale au sommet du type. Ceci fait perdre en précision, mais est surement correct. Entre les composantes, on peut par contre effectuer l'opération d'instanciation la plus précise.

3.5 Transformation de programme

On utilise une version un peu modifiée de CSL, qui comprend une expression `letstack` $x = M$ `in` N équivalente de `let` $x = M$ `in` N sauf que le dernier constructeur de x sera alloué sur la pile et libéré à la fin de l'exécution de N. On a ajouté une expression `letstack'` $x = M$ `in` N qui alloue aussi le dernier constructeur de x sur la pile, mais le libère avant l'appel terminal de N, ce qui permet de conserver les optimisations des appels terminaux.

Exemple 3.3 Pris dans un programme de complétion de Knuth-Bendix :

```
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
  ...
```

Ici, `left_reducible` et `right_reduce` sont libérés avant l'appel récursif terminal à `kbrec`, ce qui évite une augmentation inutile de la taille de la pile.

On transforme alors une expression de la forme `let` $x = M$ `in` N en `letstack` $x = M$ `in` N si le sommet de x est allouable en pile, c'est-à-dire en utilisant le théorème 2.28, si on a la condition (2.20) ce qui est déterminé par l'analyse d'échappement. Pour pouvoir effectuer cette transformation, il faut de plus que l'expression soit d'un type allouable en pile (pas entier, booléen, caractère ou `unit`) et qu'elle ait en tête un allocateur : un constructeur de type, ou une primitive standard qui effectue une allocation. Suivant les options de la ligne de commande, on refuse de mettre un `letstack` qui détruit un appel terminal récursif, ou un appel terminal quelconque. Dans ce cas, seul le `letstack'` est possible, si l'échappement le permet. En effet, si on place un `letstack` en position terminale dans une boucle récursive, la taille de la pile augmente à chaque tour de boucle alors qu'avec l'optimisation des appels terminaux, elle n'aurait pas augmenté. Ceci peut faire échouer le programme et doit donc être évité (voir l'article de Chase [Cha88] pour plus de détails sur la sureté des allocations en pile).

Une structure `let` qui définit plusieurs variables est découpée en plusieurs `let`s successifs pour pouvoir allouer certaines variables seulement en pile si les autres s'échappent (ce découpage est évidemment impossible dans le cas du `let rec`).

La transformation de programme peut aussi ajouter des lets pour se trouver dans la situation ci-dessus. Par exemple, on peut transformer $M_0 M_1 \dots M_n$ en

$$\begin{aligned}
\texttt{let}\ x_0 &= M_0\ \texttt{in} \\
\texttt{let}\ x_n &= M_n\ \texttt{in} \\
&\dots \\
\texttt{let}\ x_{i+1} &= M_{i+1}\ \texttt{in} \\
\texttt{letstack}\ x_i &= M_i\ \texttt{in} \\
x_0 M_1 \dots M_{i-1} x_i \dots x_n &
\end{aligned}$$

suivant les indications données par l'analyse d'échappement : on peut effectuer cette transformation pour x_i si la condition (2.20) est vérifiée après ajout des `let` (ce qu'on peut très bien vérifier sans faire explicitement la transformation : on vérifie que le i^{e} paramètre de M_0 ne s'échappe pas). Il faut aussi mettre des `let` pour les x_j avec $j > i$ pour respecter l'ordre d'évaluation : x_j est évalué avant x_i.

Pour augmenter les opportunités d'allocation, on effectue aussi de l'*inlining* automatique des petites fonctions qui allouent des données, ainsi, on amène l'allocateur au niveau du `let` et on peut allouer ces données en pile. Le codage en ligne n'est effectué que s'il permet vraiment une allocation en pile supplémentaire. Sinon, on conserve la version non en ligne, pour éviter de trop transformer le programme initial. Par exemple,

```
let f x = [x];; hd (f 3)
```

devient

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

puis

```
let x = 3 in letstack %t1 = [x] in hd %t1.
```

La transformation de programme prend en paramètre une expression à transformer et rend l'expression transformée et l'ensemble des `let` à mettre en dehors de l'expression courante. Ces `let` sont collectés dans un arbre dont les arêtes sont étiquetées avec des transformateurs de contexte qui définissent l'échappement des `let` jusqu'à l'expression englobante. Ces transformateurs ont été calculés pour chaque expression par la phase d'analyse. On effectue de la compression de chemin dans l'arbre en utilisant encore l'algorithme de Tarjan, pour mettre à jour les transformateurs de contexte au fur et à mesure de la remontée dans l'arbre de syntaxe abstraite du programme. Dès que le dernier `let` (dans l'ordre d'évaluation) ne s'échappe plus de l'expression courante, on le supprime de l'antiarbre et on l'écrit dans le programme. Ceci permet de placer les `letstack` de façon à ce que les données mises en pile aient la durée de vie la plus courte possible, car on n'attend pas d'avoir remonté jusqu'à l'expression englobante la plus haute pour placer le `letstack`.

Exemple 3.4 Soit la fonction suivante :

```
let rec tak (x,y,z) =
  if x>y then
    tak
     (letstack %t1=(x-1, y, z) in tak %t1,
      letstack %t2=(y-1, z, x) in tak %t2,
      letstack %t3=(z-1, x, y) in tak %t3)
  else
      z
```

`%t1`, `%t2` et `%t3` sont libérés chacun avant l'allocation du suivant, ce qui réduit la taille de la pile (l'expérience confirme que les données allouées en pile occupent 3 fois moins de place avec ce programme qu'en mettant tous les `letstack` au début.

3.6 Complexité de l'analyse

Soit n la taille du programme. Le nombre d'*eval* et de *link* effectués est en $\mathcal{O}(n)$, donc d'après le théorème 1 de [Tar79], la longueur des compressions de chemin est en $\mathcal{O}(n \log n)$. Le nombre d'équations et d'inconnues générées est donc en $\mathcal{O}(n \log n)$, ce qui donne un temps de résolution en $\mathcal{O}(n \log^2 n)$ (le calcul des composantes fortement connexes est linéaire, donc dominé par l'algorithme de Knuth).

Le calcul des hauteurs des types est dominé par la taille des types (plus précisément par la somme des tailles des déclarations de types).

La transformation de programme est en $\mathcal{O}(n \log n)$ selon le même raisonnement sur l'algorithme de Tarjan.

Finalement, la complexité de l'analyse est $\mathcal{O}(n \log^2 n + t)$ où t est le temps de calcul des hauteurs des types.

Quand l'*inlining* est actif, la complexité augmente évidemment. Si I est la taille maximale d'une fonction inlinée (peut être fixée par l'utilisateur), la taille du programme résultant est au plus $n' = \mathcal{O}(nI)$, et le temps d'analyse est $\mathcal{O}(nI + n' \log n' + n \log^2 n)$ (on post-transforme juste le programme inliné. Toutes les opérations effectuées pour l'*inlining* sont linéaires si on considère comme constant le temps de recherche dans une table d'identificateurs).

3.7 Quelques détails techniques

L'analyse de chaque module est effectuée indépendamment. Pour pouvoir profiter des informations d'échappement calculées dans les modules importés, le programme écrit ces informations dans les fichiers `.cmx` à la fin de l'analyse d'un module, puis les récupère quand on utilise le module.

Le GC de Caml Special Light est un GC à deux générations, la jeune génération est traitée par un GC à recopie, alors que la vieille utilise un GC à marquage balayage [Dol95].

On a dû modifier le *Garbage Collector* pour l'adapter à la gestion de données en pile, et rendre cette gestion la moins coûteuse possible : en particulier, les données en pile ne sont parcourues lors des cycles de GC mineur que si elles sont encore vivantes au moment du cycle. Cette méthode est plus performante que de considérer que les données en pile sont des racines, car cela évite de recopier des données du tas mineur pointées par les données en pile, alors qu'en fait ces données sont peut-être inutiles.

Pour pouvoir suivre le fonctionnement de l'analyseur, on dispose d'options en ligne de commande pour afficher les informations d'échappement calculées pour les fonctions, le code optimisé du programme, les systèmes d'équations résolus, les temps passés dans les différentes phases de l'analyse. On peut également spécifier d'effectuer ou non certaines transformations de programme (*inlining*, allocations qui suppriment l'optimisation de récursion terminale par exemple).

Chapitre 4

Evaluation des performances obtenues

4.1 Machines testées

Les tests ont été effectués sur 4 machines différentes pour pouvoir observer l'impact des différences entre les processeurs. Voici les caractéristiques essentielles de ces machines :

Machines testées	Cache primaire	Cache secondaire	Autre cache	SPECint95
Alpha DEC 3000/300	8kb(I) + 8kb(D)	512kb(I+D)		Non connue
Alpha DEC 500/400	8kb(I) + 8kb(D)	96kb (I+D)	2Mb(I+D)	12
Pentium Pro 200MHz	8kb(I) + 8kb(D)	256kb(I+D)		8,20
Sparc 5 110MHz	16kb(I) + 8kb(D)	Aucun		1,59

4.2 Les programmes de test du compilateur

Le compilateur est fourni avec 10 programmes de test classiques qui nous ont servi pour nos essais :

Programmes utilisés	
taku	Calculs sur les entiers, avec allocation de n-uplets
reynolds2	Recherche dans un arbre binaire, avec création de fermetures
reynolds3	Recherche dans un arbre binaire, avec création de couples
boyer	Traitements sur les termes
kb	Algorithme de complétion de Knuth-Bendix
nucleic	Calculs flottants
fft	Transformée de Fourier rapide
sieve	Crible d'Erathostène
quicksort	Tri rapide

L'analyseur dispose de nombreuses options pour régler son comportement aussi précisément que possible. On a testé les programmes dans plusieurs configurations pour apprécier l'effet de ces options :

Codage des versions des programmes	
Tous	Préserve tous les appels terminaux
Réc.	Préserve seulement les appels terminaux récursifs
Aucun	Ne préserve aucun appel terminal

L'inlining ne change rien sauf pour kb et nucleic, donc on ne testera qu'une version pour les autres programmes. On nommera kb-inl et nucleic-inl les versions inlinées de kb et nucleic. L'allocation en pile a un effet négligeable sur fft, sieve et quicksort, qui ne seront donc pas testés dans la suite.

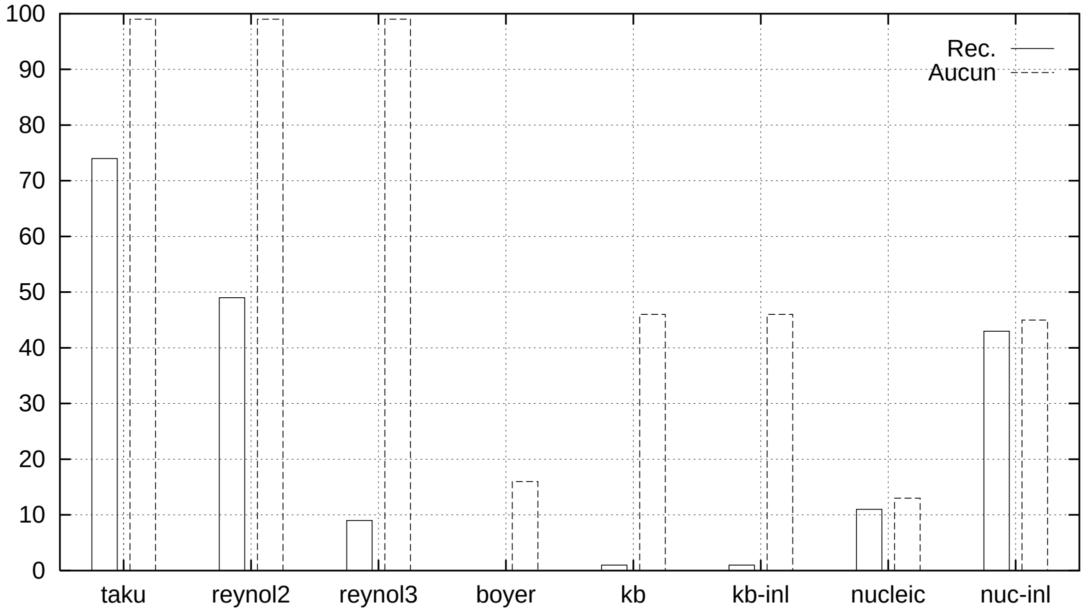


FIG. 4.1 – *Pourcentage de mémoire allouée en pile*

On se rend bien compte de l'effet qui sera obtenu à l'exécution en consultant le graphique de la figure 4.1. Il donne dans chaque cas le pourcentage de mémoire réellement allouée en pile pendant l'exécution. Toutes les versions où on ne préserve pas la récursion terminale donnent des résultats identiques entre elles ; les autres sont identiques à la version où on préserve la récursion terminale, sauf pour nucleic.

Diminution de taille mémoire en %				Taille
Préserve	Tous	Réc.	Aucun	(mots)
taku	74	74	99	12723344
reynolds2	49	49	99	10487623
reynolds3	9	9	99	31460427
boyer	0	0	16	1238977
kb	1	1	46	8325579
kb-inl	1	1	46	8325579
nucleic	11	11	13	3829681
nucleic-inl	31	43	45	3829681

Il est logique de permettre plus de letstack quand on ne préserve pas la récursion terminale, mais cette transformation n'est pas sûre, car on risque une saturation de la mémoire, même si cela ne se produit pas sur les programmes de test. D'autre part, détruire l'optimisation de récursion terminale pour permettre l'allocation en pile peut se révéler dans certains cas désavantageux. C'est pourquoi dans la suite on s'intéressera à la fois aux versions qui soit préservent les appels terminaux récursifs, soit ne préservent aucun appel terminal.

Ce graphique donne déjà une idée assez précise des résultats attendus : l'effet sera faible pour boyer et kb si on préserve l'optimisation de la récursion terminale et pour nucleic. On peut attendre des améliorations plus importantes pour nucleic avec l'inlining ce qui montre l'intérêt de l'inlining pour l'allocation en pile. En effet, on ne peut allouer une donnée en pile que si son allocateur est syntaxiquement présent. L'inlining des fonctions qui allouent leur résultat permet de faire apparaître un allocateur qui était encapsulé dans une fonction. Nucleic est le seul programme qui profite vraiment de l'inlining. Quelques fonctions sont aussi inlinées dans kb, mais cela n'améliore pas nettement les performances, car ces fonctions ne manipulent pas un volume de données suffisant. Les meilleurs résultats devraient être obtenus pour taku, reynolds2 et reynolds3.

Le tableau suivant donne le pourcentage d'augmentation de la taille de la pile dû à l'allocation en pile. La dernière colonne indique la taille initiale de la pile, pour référence. Ces tests ont été effectués sur Sparc 5, mais ils dépendent peu de la machine considérée. Seuls l'alignement des données dans la pile et le nombre de registres mis temporairement en pile peuvent influer dessus.

Augmentation de taille de la pile (%)				Taille
Préserve	Tous	Réc.	Aucun	(octets)
taku	25	25	29	996
reynolds2	69	71	71	692
reynolds3	45	45	107	668
boyer	0	0	1	1772
kb	58	58	165	85812
kb-inl	74	74	181	85812
nucleic	0	0	40	2236
nucleic-inl	3	3	44	2236

Ce tableau montre l'intérêt de la préservation de la récursion terminale : sur le programme kb, elle permet de limiter l'augmentation de la taille de la pile à moins de 60%, contre plus de 160% sinon. On obtient des résultats du même ordre sur taku et reynolds3, et on évite une augmentation de 40% de la taille de la pile sur nucleic et nucleic-inline. La taille de la pile reste invariante sur boyer car même si on alloue des données en pile, cela ne se fait pas au moment où la taille de la pile est à son maximum.

Les tableaux suivants donnent les diminutions de temps d'exécution sur Sparc 5, dans toutes les conditions d'exécution testées. La dernière colonne indique le temps sans allocation en pile en millisecondes pour référence. Les conditions d'exécution ont leurs valeurs par défaut : ratio de GC de 30%, tas du tas mineur 32768 mots. Les graphiques des figures 4.2 à 4.9 présentent une partie de ces informations de façon plus frappante visuellement. Les barres "Temps total" représentent le pourcentage de diminution du temps d'exécution; les barres "Temps de GC" représentent la part de ce pourcentage due au GC et les barres "let" représentent la part due aux lets seuls (on n'a pas du tout d'allocation en pile dans ce cas. On a seulement placé des let là où il y a des letstack dans la version avec allocation en pile).

Diminution du temps d'exécution (%)				Temps
Préserve	Tous	Réc.	Aucun	(ms)
taku	36	37	44	2020
reynolds2	4	5	17	8681
reynolds3	3	3	23	5814
boyer	2	-1	7	1014
kb	8	10	21	3745
kb-inl	8	12	21	3760
nucleic	1	3	3	1628
nucleic-inl	-2	5	3	1635

Les résultats obtenus ont bien le profil attendu étant donné le pourcentage de mémoire allouée en pile. L'effet est négligeable sur boyer quand on préserve la récursion ou les appels terminaux (l'incertitude sur les mesures de temps peut être évaluée à 1 à 2% environ, bien qu'on ait effectué une moyenne sur 30 exécutions pour améliorer la précision). A l'opposé, taku donne les meilleurs résultats quand on ne préserve pas la récursion terminale, puisque son temps d'exécution est presque divisé par deux. Plus généralement, la non-préservation de la récursion terminale a des effets positifs quand elle permet davantage d'allocation en pile. On peut être un peu déçu par les résultats sur nucleic, mais on constate bien que l'inlining améliore les performances de l'allocation en pile. Par contre, à lui seul (sans allocation en pile) il ralentit l'exécution, ce qui explique les mauvais résultats de la courbe "let" sur nucleic-inl. Pour les autres programmes, l'effet des lets seuls, tel qu'il apparaît sur les graphiques, est assez négligeable comme on pouvait le penser.

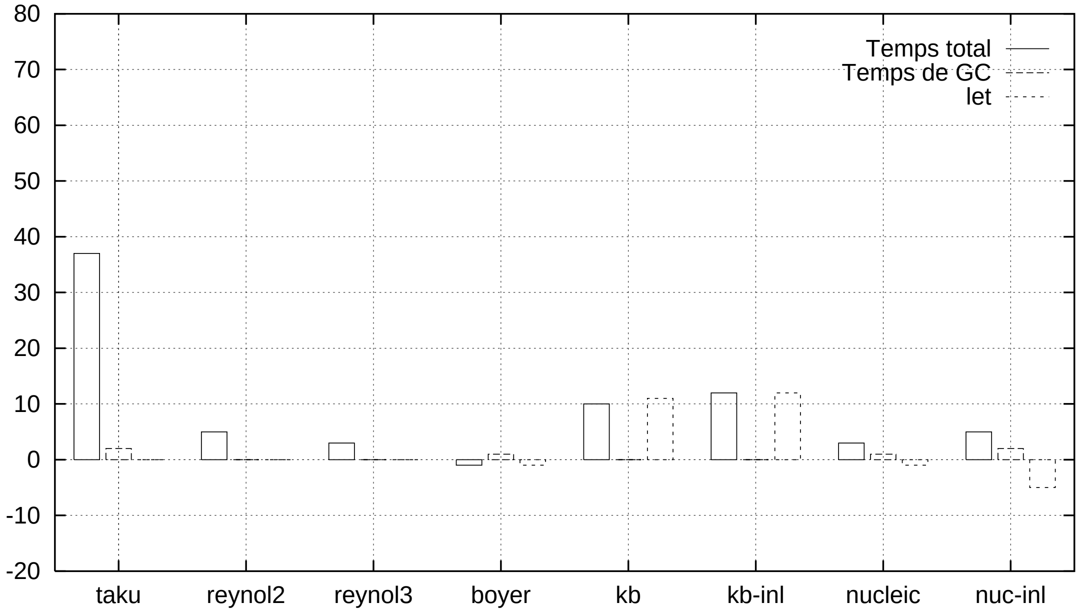


FIG. 4.2 – *Test effectué sur Sparc 5, avec préservation de la récursion terminale. Les conditions d'exécution ont leurs valeurs par défaut : ratio de GC de 30%, taille du tas mineur 32768 mots.*

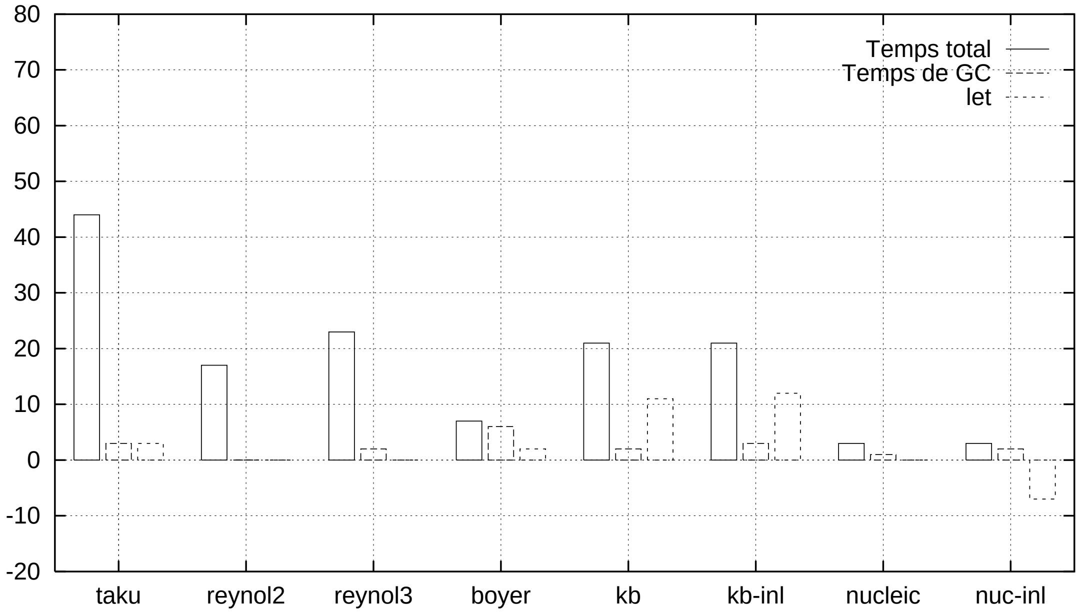


FIG. 4.3 – *Test effectué sur Sparc 5, en ne préservant aucun appel terminal*

Le placement optimal des letstack et les letstack' sont utiles car ils permettent d'augmenter de façon importante les opportunités d'allocation en pile, ce qui améliore les résultats obtenus (voir [Bla96] pour une description détaillée de l'effet de ces deux améliorations). Elles permettent aussi de diminuer la taille nécessaire en pile en optimisant la durée de vie des données.

Les tests montrent que l'effet obtenu est à peu près indépendant des paramètres du GC (voir [Bla96] pour des résultats numériques précis). Ceci peut sembler étonnant au premier abord, mais en fait les données concernées par l'allocation en pile sont pour la plupart des données à courte durée de vie, qui ont un coût nul pour le GC puisqu'elles ne sont pas parcourues lors du GC mineur. L'avantage de l'allocation en pile est de retarder les cycles de GC puisqu'il y a moins de données allouées dans le tas, mais cela dépend peu des paramètres du GC.

Le tableau suivant donne la part due au GC du gain sur le temps total. La dernière colonne donne le temps d'exécution en millisecondes sans allocation en pile, et avec le chronométrage du GC activé. La différence avec les tableaux précédents est due au temps de chronométrage.

Gain du au GC sur le temps total (%)				Temps
Préserve	Tous	Réc.	Aucun	(ms)
taku	3	2	3	2241
reynolds2	0	0	0	13707
reynolds3	0	0	2	8087
boyer	1	1	6	1876
kb	0	0	2	4937
kb-inl	0	0	3	4917
nucleic	0	1	1	1942
nucleic-inl	1	2	2	1912

Le gain dû au GC sur le temps total est faible, car, comme on l'a dit, le coût des données allouées en pile ne représente qu'une faible part du coût du GC pour beaucoup de programmes. Pour les programmes dans lesquels on peut allouer toutes les données en pile, il n'y a plus de GC, mais il n'occupait de toute façon qu'une faible part du temps d'exécution. L'essentiel du gain est donc dû plutôt à un temps d'allocation plus court et à un meilleur comportement en cache, grâce à une plus grande localité des données, puisque seules les données du sommet de pile sont en général utiles.

Le tableau suivant donne le pourcentage de gain sur le temps de GC. La dernière colonne indique le temps de GC quand on ne fait pas d'allocation en pile (en ms).

Diminution du temps de GC (%)				Temps
Préserve	Tous	Réc.	Aucun	de GC (ms)
taku	83	72	100	81
reynolds2	56	42	100	112
reynolds3	29	20	100	235
boyer	2	2	9	1425
kb	-1	0	6	1543
kb-inl	0	2	9	1558
nucleic	1	15	17	162
nucleic-inl	20	27	25	155

On retrouve des résultats évidemment en concordance avec le tableau précédent. L'incertitude relative de mesure est plus importante ici que sur les temps totaux car les temps mesurés sont plus petits et sont fragmentés. Comme on s'y attendait, quand on peut allouer en pile la presque totalité des données, le temps de GC s'annule (diminution de 100%).

Le programme kb présentait un phénomène de rétention des données qui diminuait les performances obtenues. En effet, certaines données étaient crues vivantes trop longtemps, car pour maintenir les données vivantes pour le GC, on considèrait comme des racines toutes les données allouées en pile. Ce comportement est sûr, mais il se peut qu'une donnée D reste dans la pile un peu plus longtemps que sa durée de vie réelle. Si un GC mineur se produit pendant ce temps, les données pointées par

D dans le tas mineur seront recopiées dans le tas majeur inutilement et cela occasionne une perte de temps. Ce problème a été résolu en modifiant le GC pour qu'il suive les références en pile lors de la recopie. Par contre, pour la phase de marquage, les données en pile sont toujours considérées comme des racines (c'est la seule méthode réalisable à cause de la disparition possible de données qui étaient dans la pile entre deux morceaux de la phase de marquage). Il faut aussi noter que la "malchance" peut être responsable d'une petite augmentation de la taille allouée dans le tas majeur. En effet, au moment d'un GC mineur, il existe toujours un petit volant de données à courte durée de vie qui sera copié dans le tas majeur parce qu'il est vivant à ce moment-là. Avec l'allocation en pile, on modifie le moment où le GC est effectué donc ces données ne sont pas les mêmes. Il se peut que leur taille soit plus importante si on fait l'allocation en pile que si on ne la fait pas.

Diminution du temps d'exécution (%) sur Pentium Pro				Temps
Préserve	Tous	Réc.	Aucun	(ms)
taku	25	23	26	488
reynolds2	1	0	1	6072
reynolds3	2	0	-6	3070
boyer	0	0	5	523
kb	0	0	2	1332
kb-inl	0	0	2	1339
nucleic	6	6	0	696
nucleic-inl	6	10	5	692

Les registres flottants du processeur Pentium sont organisés en pile, et l'allocation de registres de CSL est mal adaptée à cette organisation. De ce fait, l'ajout de lets sur des flottants introduit des moves inutiles qui occasionnent une perte de temps importante. On a donc dû limiter au maximum ces ajouts en supprimant tous les letstack qui étaient introduits inutilement sur les flottants dans des situations où le compilateur est capable de les allouer dans des registres. On évite ainsi une perte de plus de 25% sur nucleic et nucleic-inline quand on ne préserve pas la récursion terminale.

La non-préservation de la récursion terminale se révèle moins avantageuse sur Pentium Pro que sur Sparc 5 (reynolds3 par exemple est plus long à s'exécuter quand on ne préserve pas la récursion terminale). L'optimisation des appels terminaux compense ici le bénéfice de l'allocation en pile; celle-ci a donc moins d'intérêt que sur Sparc 5. On peut penser qu'une allocation en pile fait gagner plus de temps quand la taille de la donnée allouée est plus grande (car alors on économise plus de temps de GC et que le comportement en cache est amélioré pour un plus grand nombre d'accès). Il pourrait alors exister un seuil de taille de données à partir duquel l'allocation en pile devient plus avantageuse que l'optimisation des appels terminaux. Cependant ce seuil dépend de beaucoup de facteurs difficiles à cerner : le comportement en cache dépend du reste du programme; le temps gagné sur le GC dépend non seulement de la taille de la donnée considérée mais aussi des autres données, car les données à courte durée de vie ne sont pas parcourues par le GC. L'économie vient alors d'un nombre plus faible de cycles GC mineurs. Cela permettrait quand même d'expliquer que reynolds2 gagne un peu à ne pas préserver la récursion terminale, alors que reynolds3 y perd, car reynolds2 alloue des fermetures qui sont plus grandes que les couples alloués par reynolds3.

Diminution du temps d'exécution (%) sur Alpha 3000				Temps
Préserve	Tous	Réc.	Aucun	(ms)
taku	55	59	75	2696
reynolds2	-13	20	30	15470
reynolds3	2	4	47	9459
boyer	0	-2	6	1781
kb	2	4	18	4782
kb-inl	2	5	18	4793
nucleic	4	5	5	1313
nucleic-inl	6	18	20	1313

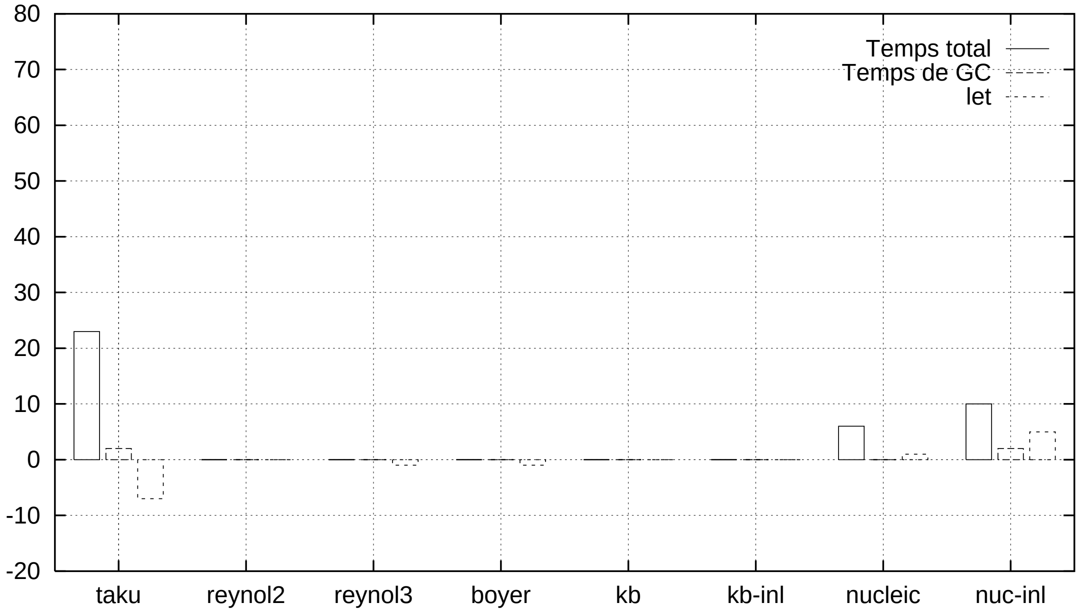


FIG. 4.4 – *Test effectué sur Pentium Pro, avec préservation de la récursion terminale*

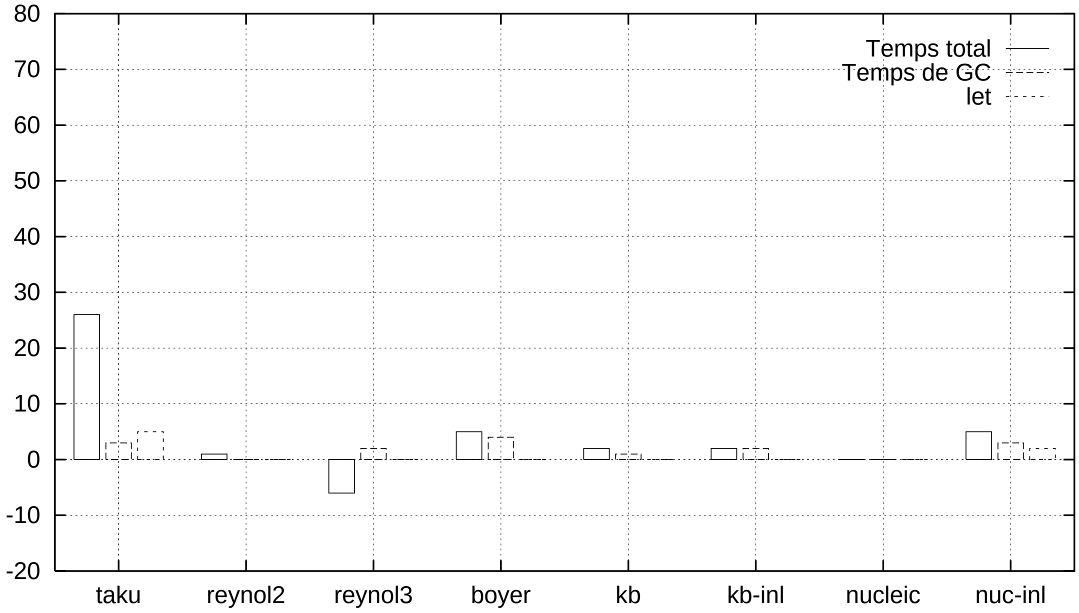


FIG. 4.5 – *Test effectué sur Pentium Pro, en ne préservant aucun appel terminal*

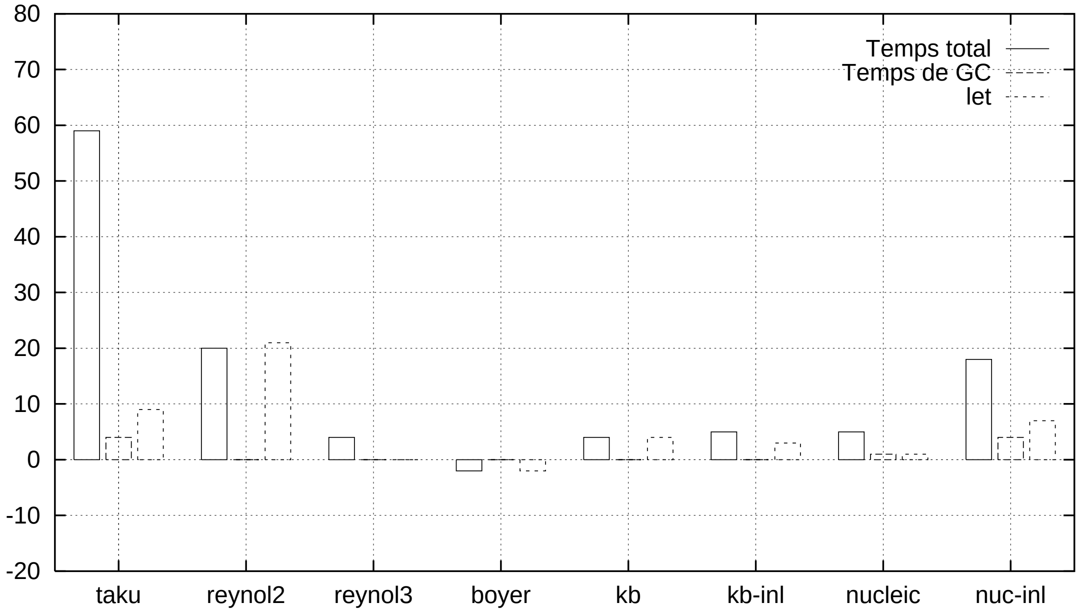


FIG. 4.6 – *Test effectué sur Alpha 3000, avec préservation de la récursion terminale*

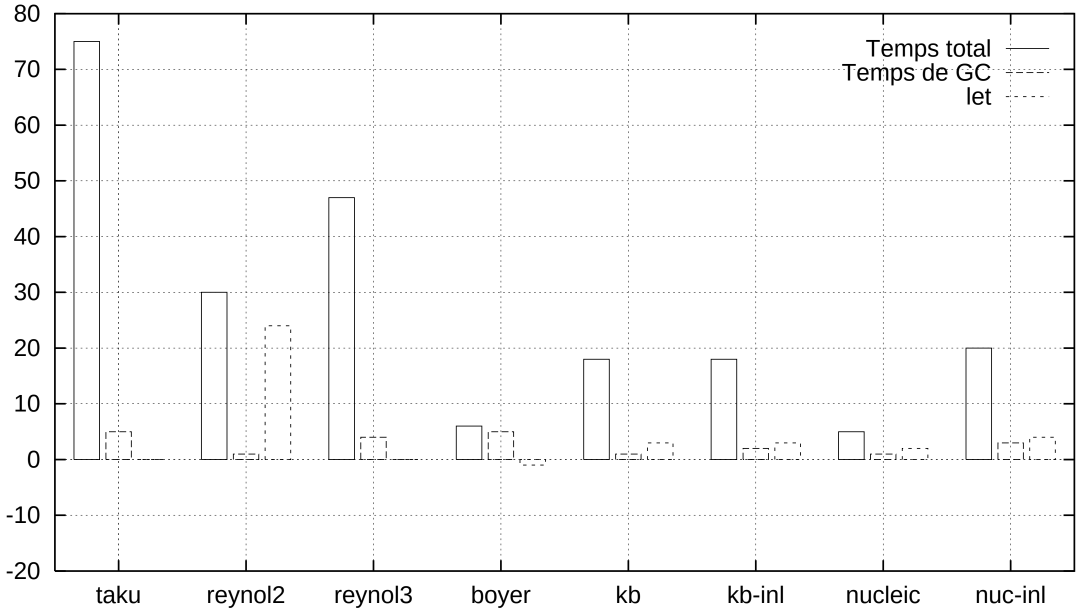


FIG. 4.7 – *Test effectué sur Alpha 3000, en ne préservant aucun appel terminal*

Sur Alpha, la non-préservation de la récursion terminale apporte un bénéfice incontestable pour tous les programmes qui allouent alors davantage en pile. Cette machine est celle qui bénéficie le plus de l'allocation en pile. Le seul résultat vraiment étrange est la perte en temps d'exécution sur reynolds2.

Diminution du temps d'exécution (%) sur Alpha 500				Temps
Préserve	Tous	Réc.	Aucun	(ms)
taku	44	43	59	475
reynolds2	3	12	17	3199
reynolds3	3	3	24	1750
boyer	0	0	6	266
kb	0	0	6	753
kb-inl	0	0	6	754
nucleic	3	3	1	198
nucleic-inl	5	7	7	206

Cette machine est la plus rapide du test. Elle profite un peu moins de l'allocation en pile que l'Alpha 3000, mais donne quand même de très bons résultats. L'incertitude de mesure est plus grande que sur les autres machines testées (environ 4%).

4.3 Le compilateur lui-même

On a fait compiler par CSL son propre code (sans allocation en pile) par plusieurs versions optimisées ou non du compilateur en code natif. Pour tous ces tests, on a autorisé les letstack' et le placement optimal des letstack.

Diminution du temps d'exécution (%)				Temps
Préserve	Tous	Réc.	Aucun	(s)
Sparc 5	1.5	-0.2	2.1	182.1
Pentium Pro	2.2	1	3.3	78.5
Alpha 3000	-2.6	1.5	1	319.6

Gain dû au GC sur le temps d'exécution (%)				Temps de GC
Préserve	Tous	Réc.	Aucun	(s)
Sparc 5	-6.3	-3.4	-5.4	83.1
Pentium Pro	0.2	1.3	2.3	32.1
Alpha 3000	0.6	1.2	1.8	98.4

Diminution de la taille allouée dans le tas (%)				Taille
Préserve	Tous	Réc.	Aucun	(Mmots)
Sparc 5	10	16	21	100
Pentium Pro	9	14	19	87
Alpha 3000	10	16	20	96

Le pourcentage de données allouées en pile est non négligeable puisqu'il atteint environ 20% mais quand même insuffisant pour procurer une amélioration importante. On retrouve toujours que la non-préservation des appels terminaux est défavorable sur Pentium Pro alors qu'elle améliore les résultats sur Alpha. Les pertes en temps de GC sur Sparc 5 sont assez inexplicables.

4.4 Coq

Coq est un programme de démonstrateur automatique, qu'on a testé avec deux versions d'allocation en pile, en lui faisant compiler son propre jeu de benchs. Dans tous ces essais, on autorise les letstack', le placement optimal des lets et l'*inlining*.

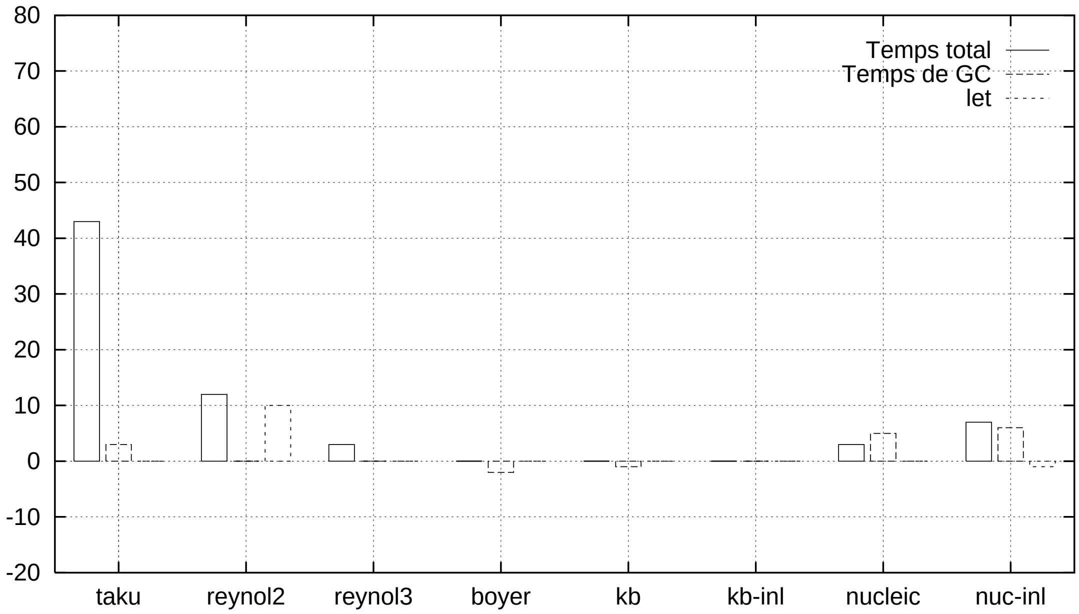


FIG. 4.8 – *Test effectué sur Alpha 500, avec préservation de la récursion terminale*

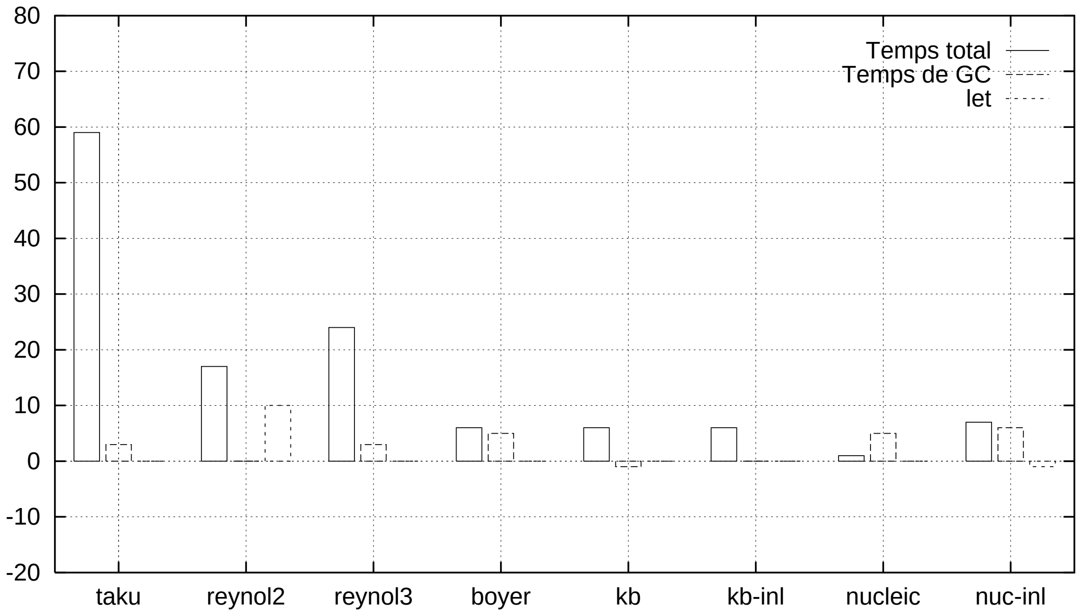


FIG. 4.9 – *Test effectué sur Alpha 500, en ne préservant aucun appel terminal*

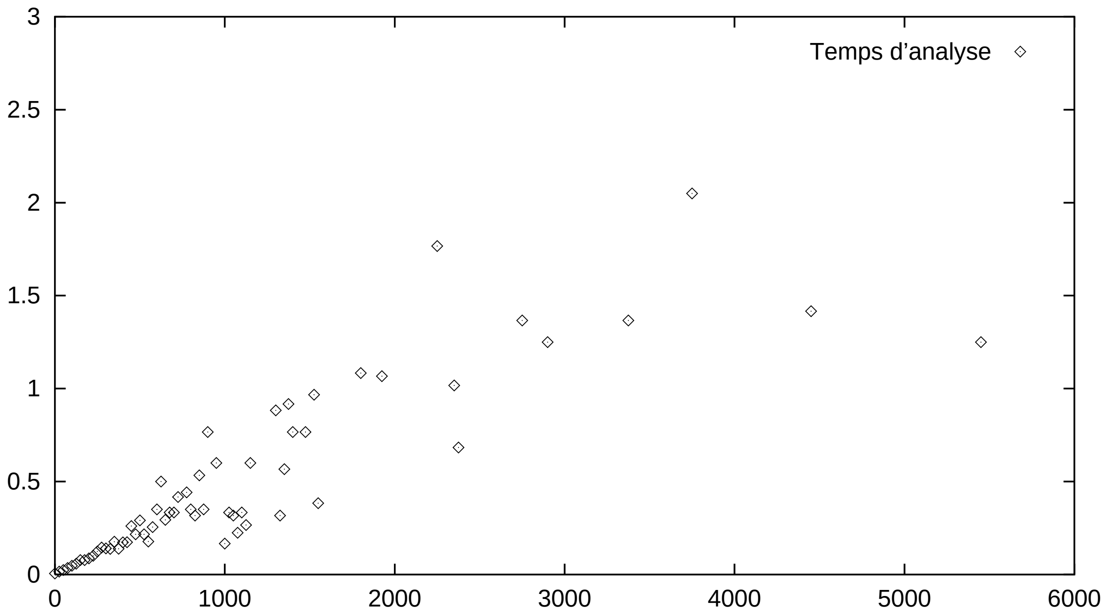


FIG. 4.10 – *Temps d'analyse en secondes en fonction de la taille de l'expression en nombre de nœuds dans l'arbre de syntaxe abstraite*

Diminution du temps d'exécution (%)			Temps
Préserve	Réc.	Aucun	(s)
Sparc 5	-2.1	3.0	5713
Pentium Pro	3.0	4.3	2339

Gain dû au GC sur le temps d'exécution (%)			Temps de GC
Préserve	Réc.	Aucun	(s)
Sparc 5	1.2	3.8	2942
Pentium Pro	1.7	2.9	1342

Au cours du test, Coq alloue 5,25 milliards de mots, et en place 17% en pile quand on préserve la récursion terminale et 25% sinon. L'inlining des petites fonctions permet d'allouer plus de données en pile (on ne peut pas en allouer plus de 11% sans inlining). Coq n'a pas pu être testé sur Alpha suite à une erreur non résolue (probablement liée au "global pointer" utilisé par l'Alpha).

4.5 Vitesse d'analyse

La vitesse de l'analyseur permet de l'utiliser pour compiler de gros programmes, Coq en est un bon exemple avec 65000 lignes. L'analyse prend 14% du temps de compilation de Coq sur Sparc 5 avec l'inlining, et la compilation est 19% plus longue au total car la compilation du code transformé est un peu plus longue que celle du code initial surtout avec l'inlining. (mais Coq ne comprend pas uniquement du code ML, donc il y a une part du travail effectué qui ne subit pas l'optimisation. Ceci montre quand même que l'analyse n'est pas trop coûteuse). La figure 4.10 montre que le temps d'analyse est quasi-linéaire. De plus, on peut analyser chaque déclaration récursive indépendamment, donc la valeur de n dans la complexité $\mathcal{O}(n \log n)$ n'est pas trop grande.

Pour compiler le compilateur lui-même, l'analyse prend 17% du temps sur Alpha 3000 mais la compilation est au total 19% plus longue. Ces chiffres sont respectivement 16% et 20% sur Sparc 5, 19% et 21% sur Pentium Pro. L'optimisation du compilateur lui-même permet de réduire ce coût. Ainsi le coût de la compilation avec optimisation sur un compilateur optimisé est 17% supérieur à celui de la compilation sans optimisation sur un compilateur non optimisé sur Alpha 3000. Cette valeur devient 19% sur Pentium Pro et sur Sparc 5.

Pour mieux évaluer le coût des différentes parties de l'analyse, on a utilisé un profileur qui a donné les résultats suivants[1] :

Répartition du temps d'analyse en %						
Programme	Génération des équations				Résolution	Post-trans- -formation
	Total	Compression de chemin	Traduction des types	Calcul des hauteurs		
taku	63	14	11	7	16	8
reynolds2	61	12	20	4	14	5
reynolds3	63	14	16	3	13	6
boyer	71	15	13	6	3	22
kb	68	14	16	7	11	9
nucleic	77	22	8	7	4	15
fft	77	16	11	3	10	7
sieve	67	11	15	4	10	10
quicksort	74	17	14	5	10	7

Ainsi, l'essentiel du temps est pris par la génération des équations sémantiques. La calcul des hauteurs des types prend un temps non négligeable, contrairement à ce qu'on pourrait penser au premier abord. Rédiger cette partie du programme de façon assez optimisée est donc un point important.

Sur certains programmes le total des 3 phases de l'analyse n'atteint pas 100%. La différence est partiellement due au prétraitement des déclarations de types, qui n'est pas pris en compte ici, et aux opérations d'initialisation de l'analyseur.

4.6 Améliorations futures

Une autre possibilité pour augmenter les opportunités d'allocation en pile serait d'effectuer de la préallocation : quand une fonction alloue son résultat, on peut allouer en pile l'espace nécessaire avant appel de la fonction puis passer à la fonction un paramètre supplémentaire qui donne l'adresse où placer le résultat. Ceci pose quelques problèmes car à cause du GC, tous les objets doivent être initialisés, et l'initialisation de l'objet avant de passer son adresse à la fonction ferait perdre du temps.

On pourrait également employer des algorithmes plus efficaces pour la compression de chemin dans les antiarbres, comme l'algorithme offline de Tarjan [Tar79]. Cependant, si cet algorithme a une complexité inférieure, $\mathcal{O}((m+n)\alpha(m+n))$ pour m compressions de chemin dans une forêt à n sommets, la constante cachée est plus grande, donc on n'observe de gain que pour des antiarbres assez grands.

Une amélioration qui concerne des détails plus techniques est la possibilité pour l'utilisateur de déclarer des informations d'échappement pour des primitives C qu'il écrit (actuellement seules les primitives de la bibliothèque standard sont prises en compte, leurs informations d'échappement étant codées en dur dans l'analyseur). On pourrait aussi réduire la taille des informations stockées dans les fichiers `.cmx` en ne conservant que les fonctions exportées dans l'interface.

Ce serait aussi intéressant d'étudier plus précisément ce qui se passe dans le cache (par exemple, le nombre d'accès à des données qui ne sont pas en cache, et le temps pris par ces accès, ou plus généralement, de simuler l'ensemble du comportement du cache).

1. Ces tests ont été réalisés sur Sparc 20 avec le compilateur écrit en bytecode, sans inlining

Chapitre 5

Conclusion

Nous avons implémenté une analyse d'échappement issue de celle de Park et Goldberg, mais nous avons considérablement réduit sa complexité par rapport aux résultats donnés dans leur article [PG92] grâce à la méthode d'A. Deutsch [Deu97]. Nous l'avons aussi étendue de façon très importante en traitant les types inductifs, les constructions impératives et (avec approximations) les fonctions d'ordre supérieur.

Les résultats obtenus par cette analyse sont tout à fait satisfaisants. L'expérimentation a montré la faisabilité de l'analyse, puisque son coût raisonnable en $\mathcal{O}(n \log^2 n)$ (en supposant les hauteurs de types précalculées) permet de l'appliquer à tous les programmes au prix d'un surcoût à la compilation tout à fait acceptable, qui ne dépasse pas 20%. Elle a montré aussi l'intérêt de l'allocation en pile, puisque qu'on obtient des gains de temps importants : souvent 5 à 20%, quelquefois plus dans des cas particulièrement favorables. Les améliorations sont plus modestes pour le compilateur et pour Coq, mais peuvent quand même atteindre 3%.

Il faut remarquer que l'effet observé dépend très peu des paramètres du GC, contrairement à ce qu'on pourrait penser a priori. Les meilleurs résultats sont obtenus en ne préservant pas la récursion terminale sur Sparc 5 et Alpha. Cette solution n'est pas sûre car elle peut provoquer une saturation de la pile, mais cela ne se produit pas dans les exemples utilisés. Par contre, sur Pentium Pro, il vaut mieux préserver la récursion terminale. En effet, l'optimisation des appels terminaux compense le gain obtenu par l'allocation en pile.

Ces résultats ont été quelque peu améliorés grâce au codage en ligne des petites fonctions pour augmenter les opportunités d'allocation en pile et grâce à une meilleure adaptation du GC au stockage de données en pile. La bonne qualité du GC de Caml Special Light réduit cependant les possibilités de gain.

Enfin, ce travail peut être adapté aux versions les plus récentes de Caml Light, Objective Caml, en vue d'une distribution.

Bibliographie

[Abr86] S. Abramsky. – Strictness Analysis and Polymorphic Invariance. *In: Proc. Programs as Data Objects*. – Springer Verlag.

[AFL95] Alexander Aiken, Manuel Fähndrich et Raph Levien. – Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. *In: PLDI95*. – San Diego, California, June 1995.

[App87] Andrew W. Appel. – Garbage Collection can be faster than Stack Allocation. *Information Processing Letters*, vol. 25, n° 4, janvier 1987, pp. 275 – 279.

[AS93] Andrew W. Appel et Zhong Shao. – An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming*, vol. 1, n° 1, janvier 1993, pp. 1 – 27.

[Bla96] Bruno Blanchet. – *Garbage Collection statique*. – rapport de DEA, Rocquencourt, INRIA, septembre 1996.

[BTV96] Lars Birkedal, Mads Tofte et 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)*. – St. Petersburg Beach, Florida, 21-24 janvier 1996.

[Bur90] G.L. Burn. – A Relationship Between Abstract Interpretation and Projection Analysis. *In: Principles of Programming Languages*. pp. 151–156. – ACM.

[CC77] Patrick Cousot et 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*, pp. 238 – 252. – Los Angeles, California, janvier 1977.

[CC79] Patrick Cousot et Radhia Cousot. – Systematic Design of Program Analysis Frameworks. *In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 269 – 282. – San Antonio, Texas, 29-31 janvier 1979.

[CC92] Patrick Cousot et Radhia Cousot. – Abstract Interpretation Frameworks. *Journal of Logic and Computation*, vol. 2, n° 4, août 1992, pp. 511–547.

[Cha88] David R. Chase. – Safety considerations for storage allocation optimizations. *In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*. pp. 1 – 10. – ACM Press.

[CLR94] Thomas H. Cormen, Charles E. Leiserson et Ronald L. Rivest. – *Introduction à l'algorithmique*. – Dunod, 1994.

[CMT94] Steve Carr, Kathryn S. McKinley et Chau-Wen Tseng. – Compiler Optimizations for Improving Data Locality. *In: Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 252 – 262.

[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*, pp. 157 – 168. – San Francisco, janvier 1990.

[Deu92] Alain DEUTSCH. – A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations. *In : Proceedings of the IEEE 1992 International Conference on Computer Languages*. pp. 2 – 13. – Oakland, California, avril 1992.

[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*. pp. 230 – 241. – 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).*

[Dol95] Damien DOLIGEZ. – *Conception, réalisation et certification d'un glaneur de cellules concurrent*. – Thèse de PhD, Université Paris VII, mai 1995.

[Han95] John HANNAN. – A Type-based Analysis for Stack Allocation in Functional Languages. *In : Proceedings of the Second International Static Analysis Symposium (SAS '95).*

[Har89] W. HARRISON. – The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation*, vol. 2, 1989, pp. 176 – 396.

[Hed88] Lucy HEDERMAN. – *Compile Time Garbage Collection Using Reference Count Analysis.* – Rapport technique n° Rice COMP TR88-75, Houston, Texas, Rice University, août 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 juin 1993.

[HMT89] Robert HARPER, Robin MILNER et Mads TOFTE. – *The Definition of Standard ML.* – LFCS Report Series n° ECS-LFCS-89-81, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, mai 1989.

[Hoa78] C.A.R. HOARE. – Some properties of predicate transformers. *Journal of the ACM*, vol. 25, n° 3, juillet 1978, pp. 461 – 480.

[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*, pp. 351 – 363.

[Hug88] John HUGHES. – Backward Analysis of Functional Programs. *In : Partial Evaluation and Mixed Computation*, éd. par D. BJØRNER, A. P. ERSHOV et N. D. JONES. – Elsevier Science Publishers B.V. (North Holland).

[Hug92] Simon HUGHES. – Compile-Time Garbage Collection for Higher-Order Functional Languages. *J. Logic Computat.*, vol. 2, n° 4, 1992, pp. 483 – 509.

[ISY88] Katsuro INOUE, Hiroyuki SEKI et Hikaru YAGI. – Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Transactions on Programming Languages and Systems*, vol. 10, n° 4, octobre 1988, pp. 555 – 578.

[JM82] N. D. JONES et S. MUCHNICK. – A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *In : Nineth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 66 – 74.

[JW90] Simon B. Jones et Michael White. – Is Compile Time Garbage Collection Worth the Effort. *In : Workshop on Functional Programming.* – Glasgow, 1990.

[Knu77] Donald E. Knuth. – A Generalization of Dijkstra's Algorithm. *Information Processing Letters*, vol. 6, n° 1, février 1977, pp. 1 – 5.

[Moh95a] Markus Mohnen. – Efficient Closure Utilisation by Higher-Order Inheritance Analysis. *In : Static Analysis Symposium (SAS'95).*

[Moh95b] Markus Mohnen. – Efficient Compile-Time Garbage Collection for Arbitrary Data Structure. *In : PLILP'95.*

[NPD87] Anne Neyrinck, Prakash Panangaden et A.J. Demers. – Computation of aliases and support sets. *In : Fourteenth Annual ACM Symp. on Principles of Programming Languages.* ACM Press, pp. 274–283.

[PG92] Young Gil Park et Benjamin Goldberg. – Escape Analysis on Lists. *In : Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 116 – 127. – San Francisco, California, 17-19 juillet 1992.

[Ram93] G. Ramalingam. – *Bounded Incremental Computation.* – Rapport technique n° 1172, Madison, University of Wisconsin, août 1993.

[RM88] Cristina Ruggieri et Thomas P. Murtagh. – Lifetime Analysis of Dynamically Allocated Objects. *In : Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '88)*, pp. 285 – 293. – San Diego, California, 13-15 janvier 1988.

[Sch78] Jerald Schwartz. – Verifying the safe Use of Destructive Operators in Applicative Programs. *In : Transformations de programmes. Actes du 3e colloque international sur la programmation*, éd. par B. Robinet, pp. 395 – 411.

[SF96] Manuel Serrano et Marc Feeley. – Storage Use Analysis and its Applications. *In : 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP).* – Philadelphia, Pennsylvania, 24-26 mai 1996.

[Shi88] O. Shivers. – Control flow analysis in Scheme. *Conference on Programming Language, Design and Implementation*, juin 1988, pp. 164 – 174.

[Tar74] Robert Tarjan. – Finding dominators in Directed Graphs. *SIAM Journal of Computing*, vol. 3, n° 1, mars 1974, pp. 62 – 89.

[Tar79] Robert Endre Tarjan. – Applications of Path Compression on Balanced Trees. *Journal of the Association for Computing Machinery*, vol. 26, n° 4, octobre 1979, pp. 690 – 715.

[TT93] Mads Tofte et Jean-Pierre Talpin. – *A theory of Stack Allocation in Polymorphically Typed Languages.* – Rapport technique n° 93/15, Departement of Computer Science, Copenhagen University, 9 juillet 1993.