Reversible versus Irreversible Machines: Experimental Results

Bruno Blanchet
DMI - CNRS URA 1327
École Normale Supérieure
45 Rue D'Ulm, 75005 Paris, France
blanchet@clipper.ens.fr

Ian Mackie*
LIX - CNRS URA 1439
École Polytechnique
91128 Palaiseau Cedex, France
mackie@lix.polytechnique.fr

9 October 1995

Abstract

The Geometry of Interaction semantics introduced by Girard provides a new way of understanding the reduction process in the λ-calculus where the computation steps are *linear* and *reversible*. Moreover, this can be trivially implemented to give a simple compilation technique: the Geometry of Interaction Machine. At the cost of introducing *global* operations (irreversible copying and discarding) Danos and Regnier have shown a correspondence with Krivine's environment machine. In this paper we report on experimental results for the respective performances of these two implementations to try to judge the relative merits of reversible versus irreversible computations.

Keywords: λ-calculus, Functional Language Compilation, Geometry of Interaction, Environment Machines.

1 Introduction

There have been many successful attempts to give implementations of functional programming languages by looking at *models* of the λ-calculus. To highlight just two, we mention categorical combinators, coming from categorical models of the λ-calculus; and SKI combinators coming from combinatory logic.

In [9] the second author gave a different kind of compilation of a simple λ-calculus based functional programming language coming directly from Girard's Geometry of Interaction semantics [7, 6]; a semantics of computation capturing the actual reduction process.

One way of understanding the Geometry of Interaction is to think of a program represented as a graph. The meaning of a program is then given by the set of *paths* in the graph. Of course not all paths, but some particular ones — the paths that survive the action of reduction. That is to say that if there is a path connecting two parts of the graph before the reduction, then they should be still connected after reduction. These are the paths which are picked out by the Geometry of Interaction. Computation can then be understood as generating all the paths in the graph that satisfy this property; hence computing the transitive closure of the graph with respect to the Geometry of Interaction.

There are many ways of realizing this process as a concrete implementation. One way is to compile a program, thought of as a graph, into fixed code (which behaves as the network). The runtime system is then realized as a token traversing this network, following the code generated from the root of the graph, thus giving a sequential data-flow model of computation. This is the method given for the Geometry of Interaction Machine [9], which we believe is the simplest method of compilation for this idea.

*Second author funded by a British Royal Society Research Fellowship.

One of the main novelties of this approach to compilation is that the basic *instructions* (or combinators) are very simple. More precisely, they are *linear* and *reversible*, in other words they do not involve the global operations of copying and discarding. As a consequence all the machinery for the execution is in one uniform framework so it is easy to study the cost and memory requirements of a computation. The significant features of this kind of implementation can be summarised as:

- Compact compilation directly into assembly language.
- Small and compact run-time system.
- Solid foundation from Girard's Linear Logic [5] and the Geometry of Interaction.

An "optimisation" for this implementation was also proposed in [9] which takes us away a little from the linear and reversible approach. The idea is based around a notion of *questions and answers* for base types. Briefly, when the token reaches a base value then it has found an *answer*; not any answer, but a specific one: the answer to the last question asked. Questions are asked by (in addition to the initial question when we start the computation) each time we pass a function requiring an argument of base type. With this knowledge it is clear that we don't have to follow the path in the graph back to where the question was asked, we could simply jump back to that place. However, this is not so straightforward since the reverse computation would change the structure of the token. However, if we had stored the token when we started asking the question, then when we jump back we just need to restore the old token.

Hence, this notion of questions and answers reduces the length of the reversible computation, but at the expense of introducing global copying and discarding. The "optimisation" was done for constants (base types of the language PCF [11]). This idea was then generalised by Danos and Regnier [4] to question and answers at function types, and they showed that the optimised computation leads to a beautiful reconstruction of Krivine's call-by-name environment machine.

This then left open an interesting question as to which, reversible or irreversible computation, is the most efficient. In this paper we try to answer this question by reporting on practical experience that has been gained by implementing both these algorithms in a uniform framework. We generate and analyse a series of benchmark results. The surprising result that we obtain is that the reversible strategy performs better. This then provides some confidence that the Geometry of Interaction approach to compilation is worth further study for more realistic implementations of functional programming languages (data-types, other strategies, etc.).

The rest of this paper is structured as follows. In the next section we give an informal overview, using an example, to explain the notions of reversible and irreversible computation. In Section 3 we recap on the compilation for the Geometry of Interaction Machine and show how the optimisation mentioned above gives the Krivine Machine. In Section 4 we give a set of benchmark results for both implementations and provide an analysis. Finally, we conclude in Section 5 and suggest further directions of study.

2 Basic Concepts

In this section we give an intuitive explanation of both the reversible and irreversible computations in the λ-calculus; focusing mostly on the reversible one since this is the least well-known of the two.

We remark on an important point that throughout this paper we focus on the *call-by-name* evaluation since this is the only strategy that exists so far for the reversible computation; we mention other evaluation strategies in the conclusions at the end of the paper.

2.1 Reversible Computation

The basic idea of a reversible computation can be explained with the help of a simple example. Consider the λ-term $(\lambda xy.xy)(\lambda z.z)n$ which is the application function applied to the identity, all applied to the constant n. There are many ways to reduce this term to it's *normal form*; all of which would somehow simulate β-reduction. For example this could be done by compiling the term into a graph representation and giving a set of rules to reduce this graph to its normal form, then *extracting* the result from the representation of the normal form. Similar ideas apply also for environment machines where this time the term is compiled into code which will manipulate an environment, followed by the execution, and again the execution process to read out the result of the computation.

In each of the above methods involves a *three* phase process: a compilation, a reduction, and a read-back. The idea of a reversible computation can be understood as providing an alternative method for the computation process. In naïve terms it is an attempt to perform the read-back (the final phase of computation mentioned above) *without* actually doing any of the reduction whatsoever. Hence, we push the computation into the read-back phase of the computation.

To see how this can be done we will compare the ideas with graph rewriting. In Figure 1 we give a possible graph representation of the above term and show a sequence of rewrites that would be required to reduce this term to its normal form.

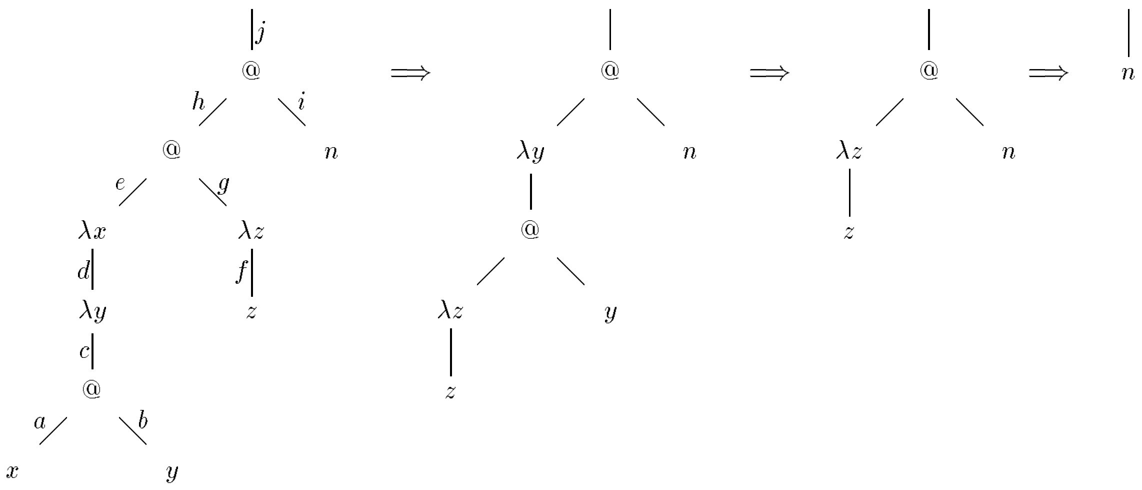


Figure 1: An example graph reduction

Of course we have simplified the graph rewriting process somewhat in that we have just shown several snapshots of the reduction; the process of doing a substitution has been left implicit. The example is also sufficiently simple in that there is no need to capture any sharing, for example. We refer the reader to the literature for a more comprehensive discussion on graph rewriting.

The result of rewriting this graph has generated a graph representation of the result n. To extract this result from the graph we require some kind of graph traversal algorithm which would start at the root (top) of the graph and follow pointers along a *path* until we reach the result. In this example we find the node n immediately, and this is the result of the computation.

The reversible algorithm puts a completely different perspective on "reduction" in that we would like to read out the answer in the original graph (before the reduction). To be more precise, the result of the computation is already in the original graph, but hidden from us by the presence of redexes in the term. What we require, and this is the essence of the Geometry of Interaction, is a way of reading out the result that is *invariant* under the reduction process.

The idea then is to start at the root of the graph representation of the program and follow pointers along the graph to extract the result, but in some way "bypassing" the redexes in the graph. The occurrences of variables in the body of the abstraction are assumed to be connected to the binding λ so that there is no need for us to mention the names of the variables explicitly in the graph, and in addition this makes it into the kind of graph that we can traverse in the desired way.

Here is the scenario for the example given. We start at the root and travel along the edge j until we meet the first application node @. At this point we have a choice: along the edge h towards the function, or along i towards the argument.

The choice that we have is basically one of evaluation order. Since we are working with call-by-name evaluation, we should therefore travel towards the function first and only look into the argument if the function requires it. Hence we travel along h towards the second application node, remembering that we came from the top of the application node. The same reasoning indicates that we should now travel along e towards the function, again remembering from which direction we came from.

At the λ node we are again faced with a choice: along d into the body of the function, or along a towards the variable. At this point we make use of the history that we remembered at the last application node. Since we came from the top of the application this indicates that we want the result of the application, hence we want the result of the body of the lambda. Therefore we proceed along edge d. At the next lambda we again need the history, this time from the first application node. Since again we came from the top we proceed into the body of the lambda along edge c.

Remark 2.1 *An important remark at this point is that we have traversed a path in the graph from the top application node to the λy node along the path deh. If we look at the graph after one step of reduction we see that these two nodes are now connected. Hence we have traversed a path that still exists after graph reduction (only that it is shorter after the reduction!). This is no coincidence and is exactly the idea we are going to formalise: capturing paths that survive the action of reduction, and this justifies our choices made during the path computation.* $\diamondsuit$

We are again at the top of an application node, and thus proceed into the function as usual, along a remembering which way we came from. Now we arrive back at the lambda node along the variable edge, and again we are faced with a choice: along d back into the body of the lambda, or along along e back towards the application. However, since we are looking for the argument for this abstraction it is pointless to return into the body. Hence the only way to go is along e, remembering that we came from the variable and not the body.

At the application node we use the history that we came from the variable of the abstraction and travel along g towards the argument where we immediately meet the abstraction. The history of the path is used again to dictate that we travel along f into the body of the function where we immediately find ourselves back at the λz node from the variable edge.

Next we travel along g back towards the application, where we arrive from the argument indicating that we are returning the argument to the function that called it. Hence we go back into the function along e, remembering that we came from the argument.

At the λx we use the history, since we came from the argument of the previous application, we proceed into the variable along a. Whence we meet the application node again. We remark this time that we have traversed a path from this application to the argument λz which we can see is also a path that exists after one step of graph reduction.

This process continues, remembering which direction we came from, and moving around the graph until we can go no further. In this example we will eventually reach the root of the graph again with the constant n that would be picked up during the computation.

A few remarks are in order:

- During this path traversal of the graph one should note that at no point did we suddenly change direction of the flow and go back on the path that we came along.
- At no point did we rewrite any part of the program — it remains in the same form all the way through the path traversal.

In the following we will make more precise the notions of "remembering" information that we mentioned during this kind of computation which make it into a deterministic algorithm which is the heart of our implementation.

In [9] we gave the reversible strategy for the language PCF which requires the use of *side effects* to model the constants which are *not*, in general, reversible.

2.2 Irreversible Computation

Having presented the reversible computation, we will now explain the irreversible approach using the same kind of reasoning so that the differences will become apparent. This computation can be thought of as an optimisation of the previous scenario.

We assume a notion of a state consisting of a stack s and an *environment* e where we can store addresses of arguments that we can jump to rather than follow the path backwards as we did previously.

We begin at the root of the graph with and environment e and traverse the edge j towards the first application. At this point we want to evaluate the function in e, and store the current environment for the evaluation of the argument (if we need to!). Hence we copy e and store this together with the address of the argument i on the stack s giving state $(e,(i,e):s)$, and travel along edge h.

At the next application we do the same operation as above, this time yielding the state $(e,(g,e):(i,e):s)$ and then traveling along edge e.

We now arrive at the abstraction, at which point we want to evaluate the body in an environment augmented with the top element of the stack so that the variable x becomes bound. This yields the state $((g,e):e,(i,e):s)$ and we travel into the body of the lambda along d.

At the next abstraction (λy) we do the same operation, this time yielding the state $((i,e):(g:e):e,s)$ and then traveling along edge c into the body.

At the application we do the familiar operation, and update the state to $(e',(b,e'):s)$, where $e'=(i,e):(g,e):e$, and we travel along a towards the variable x. We see that x is bound "two lambda's up", so that we take the second element off the environment which is (g,e), update the state with $(e,(b,e'):s)$ and branch to g. Hence discarding the first element of the environment (*i.e.*, garbage collection).

This process of storing the environment with an address, manipulating and looking up values in the state continues until we finally arrive at the constant n which, in this case, indicates that the computation is complete.

3 The Geometry of Interaction and Environment Machines

3.1 The Geometry of Interaction Machine

Here we will briefly review the Geometry of Interaction Machine compilation for the λ-calculus; we refer the reader to [9] for the full details and extensions to more realistic functional programming languages.

First we compile the terms of the λ-calculus into labelled graphs. These graphs are very similar to the standard graph representation of a λ-term, for example those given by Wadsworth [12, Chapter 4]. In fact the graphs that we use are *proof nets* from Linear Logic [5] which we draw in the orientation corresponding to familiar λ-graphs. There are several reasons why we pass through Linear Logic for the compilation, but it is not essential to the paper. The main point is that Linear Logic provides a decomposition of the λ-calculus into more primitive operations — application and abstraction are decomposed into more primitive operations — and in addition it gives a representation of a λ-term in which the copying and discarding of information is made explicit. Also, since the Geometry of Interaction was originally defined for Linear Logic it provides a convenient middle ground for studying Geometry of Interaction compilation of the λ-calculus.

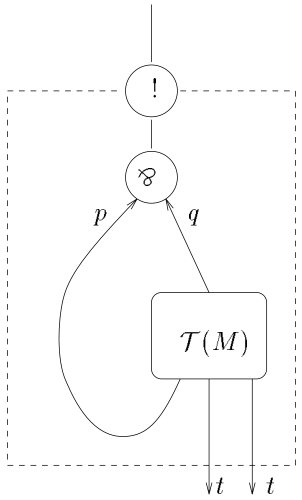
In these graphs we will label the *edges* which will allow us to write *paths* in a graph. Paths are then nothing more than a sequence of edges that we write in a, traditional, compositional way: if we have a labelled edge l_1 followed by the edge l_2 then we have a path l_2l_1. Equivalently, we also have the *reverse* of this path, written $(l_2l_1)^*$, which satisfies $(l_2l_1)^*=l_1^*l_2^*$ as expected. Hence our edges are *directed*. By imposing an algebra on these paths we can pick out the ones of interest to us — the ones which survive the action of reduction — which is given to us by the Geometry of Interaction semantics.

The translation $\mathcal{T}$ from λ-terms to labelled graphs is defined inductively as follows. We will explain the labelled graph structures that we use alongside the translations.

A *variable* x is translated into a connecting edge; no node in the graph is introduced. These are drawn simply as a piece of wire that will join nodes together:

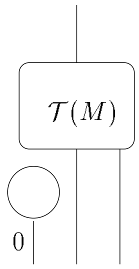
With respect to paths, we use the label 1 for this edge. This will serve as the *identity* for paths: if we connect a path ϕ to this edge then we compose to get $1\phi = \phi$. It is useful to think of this as extending a path, but without changing it.

An *abstraction* $\lambda x.M$ is encoded by first translating the body $\mathcal{T}(M)$ then building the following structure:



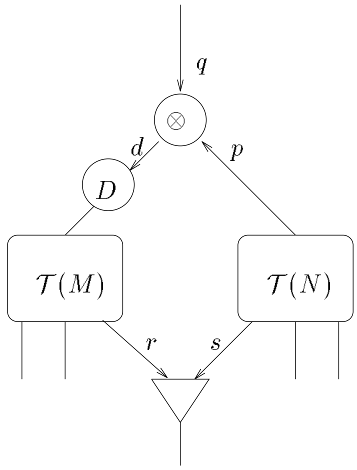
Some explanations are in order. First, we should imagine that the ⅋ (read "par") and ! (read "of course") nodes are decompositions of a λ node that we would draw in standard graph representations. The two main points of departure from standard notations is that we have explicitly connected the variable occurrence to the binding par node and placed a box structure around the body of the abstraction. In addition, we have labelled some of the edges (with a direction) that will play a crucial rôle later in the paper.

The translation of $\mathcal{T}(M)$ will in fact ensure that there is at most one occurrence of the variable x that we need to bind to the par node. If x does not occur free in M, then we make add this explicitly in the following way using a *plug* node.



It is worth pointing out that this box structure is just capturing the de Bruijn level. Since we are representing variables explicitly by connecting wires we require to know when a variable is free in the term. This is represented by the edges leaving the bottom of the box, which are called *auxiliary doors* of the box. The top of the box, at the ! node, is called the *principal door* of the box.

An *application* MN corresponds very closely with the standard graph representation of λ-calculus. We translate the terms $\mathcal{T}(M)$ and $\mathcal{T}(N)$ then build the following structure:



This time we can see the application as decomposed into two nodes: the $\otimes$ (read "tensor") and the D (read "dereliction") nodes. The main point of departure from standard graph representations is that if any variables occur in both M and N, then we connect the edges together in the *fan-in* node, drawn as a triangle. This ensures that when we make an abstraction, there is at most one occurrence of the variable being bound. There is nothing more to this than just making the join of two edges explicit in this *wiring* representation of the λ-calculus.

Context transformers (combinators)

Rather than giving the algebra on paths that is used to pick out the good ones, we prefer to present a model of the algebra directly. We refer the interested reader to the literature for the complete story; see for example [3].

Definition 3.1 *Here we present the meaning of the combinators (labels) that suffice to code computation in the λ-calculus. There are many variations, but here we give the so-called B-Model of Danos and Regnier [4] which helps to bring out the correspondence between reversible and irreversible computation.*

For each operator of the algebra (the labels) we give the transitions on an context *which can be thought of as two stacks (e, s) of an environment machine; e is usually called the* environment, *and s the* stack.

$$\begin{array}{lcllcl}
p(e,s) & = & (e, r : s) & q(e,s) & = & (e, l : s) \\
r(e, u : s) & = & (e, R \cdot s) & s(e, u : s) & = & (e, L \cdot s) \\
t(e, u : v : s) & = & (e, u \cdot v : s) & d(e,s) & = & (e, \Box : s) \\
!(e, u : s) & = & (u : e, s) & & &
\end{array}$$

For a context transformer x, we define x^* as the (partial) inverse of x. In addition, 1 is the identity and 0 is the nowhere defined context transformer. $\diamondsuit$

It should come as no surprise that these can be coded directly into assembly language; again see [9] for details. We now present the compilation, in terms of these combinators, of the λ-calculus into a simple assembly language.

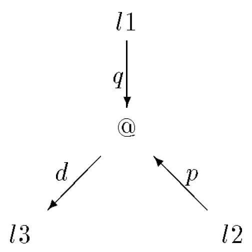
First we define the language that we will compile into. We augment the set of combinators defined above with some simple instructions that will facilitate the compilation. Since the combinators given do all the compilation, there is very little to add. In fact all that we need is some notion of branching so that we can jump around the graph. The simplest way to do this is to change, very slightly, the inverse of the p, q, r and s operations so that, in addition to popping elements from the stacks, they set a register to indicate that the operation was successful. Hence we will write instructions such as "p^* br l" to indicate that if p^* was defined, then we perform p^* before branching to the label l. We will also use the branch instruction "br " as an unconditional jump. No other instructions will be needed.

The compilation proceeds by generating code for each node in the graph. Hence a node in the graph with n edges will become a block of code with n inputs. For labels in the following, we will write l_{out} meaning the label of the node that l is connected to, provided by linking information.

This compiled code will then serve as a network for which we can use the token, the (e, s) data-structure, to compute the result of the program in a sequential way.

Variables Since variables are connecting edges together we produce no code at all; however, they provide the linking information to the compiler to indicate how the nodes are connected.

Application Recall that the application is built up from a tensor ($\otimes$) and a dereliction (D). We draw this here as an application node, but keeping the labelled edges from before.



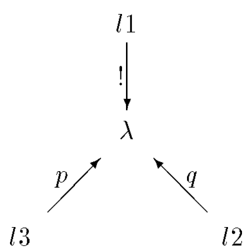
Using $l1, l2$ and $l3$ as the addresses in the graph where a token can arrive, we can generate the following block of code:

$l1:$ q d br $l3_{out}$	$l2:$ p d br $l3_{out}$
$l3:$ d^* q^* br $l1_{out}$ p^* br $l2_{out}$	

We can equally well see the above as context transitions in an environment machine. We write these as triples consisting of the address in the graph and the e and s components of the context. Hence the following four transitions capture the above compilation:

$$\begin{array}{lcl}
(l1, e, s) & \Rightarrow & (l3_{out}, e, \Box : l : s) \\
(l2, e, s) & \Rightarrow & (l3_{out}, e, \Box : r : s) \\
(l3, e, \Box : l : s) & \Rightarrow & (l1_{out}, e, s) \\
(l3, e, \Box : r : s) & \Rightarrow & (l2_{out}, e, s)
\end{array}$$

Abstraction Recall that an abstraction is built out of an ! and an $\parr$ node. Writing this as an abstraction node λ, but keeping the labelled edges from before gives:



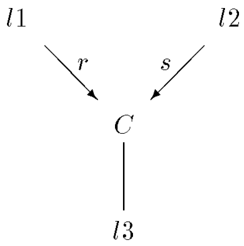
Using $l1, l2$ and $l3$ as the addresses in the graph where the token can arrive, we can generate the following block of code:

$l1:$ $!$ q^* br $l2_{out}$ p^* br $l3_{out}$	
$l3:$ p $!^*$ br $l1_{out}$	l2: q $!^*$ br $l1_{out}$

As transitions for an environment machine, we can write the following four rules:

$$
\begin{array}{lcl}
(l1, e, u : l : s) & \Rightarrow & (l2_{out}, u : e, s) \\
(l1, e, u : r : s) & \Rightarrow & (l3_{out}, u : e, s) \\
(l2, u : e, s) & \Rightarrow & (l1_{out}, e, u : l : s) \\
(l3, u : e, s) & \Rightarrow & (l1_{out}, e, u : r : s)
\end{array}
$$

Copying A fan-in node is labelled as follows, using $l1, l2$ and $l3$ as the addresses in the graph:



The following block of code with three inputs is generated for this node:

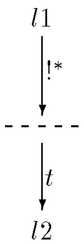
$l1:$ r br $l3_{out}$	$l2:$ s br $l3_{out}$
$l3:$ r^* br $l1_{out}$ s^* br $l2_{out}$	

Again, machine transitions can be given as follows:

$$
\begin{array}{lcl}
(l1, e, u : s) & \Rightarrow & (l3_{out}, e, L \cdot u : s) \\
(l2, e, u : s) & \Rightarrow & (l3_{out}, e, R \cdot u : s) \\
(l3, e, L \cdot u : s) & \Rightarrow & (l1_{out}, e, u : s) \\
(l3, e, R \cdot u : s) & \Rightarrow & (l2_{out}, e, u : s)
\end{array}
$$

Discarding During computation the token will never travel towards a discarding node, hence no code is produced.

Auxiliary doors The auxiliary doors of the box structure are labelled as follows, using $l1$ and $l2$ for the addresses in the graph:



We generate the following block of code with two inputs:

$l1:$ $!^*$ t br $l2_{out}$	$l2:$ t^* $!$ br $l1_{out}$

Writing the above as machine transitions give the following:

$$\begin{array}{lcl}(l1, u:e, v:s) & \Rightarrow & (l2_{out}, e, u\cdot v:s)\\ (l2, e, u\cdot v:s) & \Rightarrow & (l1_{out}, u:e, v:s)\end{array}$$

With these very simple rules we have the power to implement the λ-calculus. The size of the object code (number of lines of assembly language) is linear in the number of constructors in the original program.

Evaluation consists of starting at the root of the term with the empty context $(\Box, \Box)$, which is the *token*, and following the code generated until none of the rules apply. For a closed term, this situation arises when the token reaches a λ which is the weak head normal form. For a program of base type the computation will be completed when the token is again at the root (*cf.* the example in Section 2), assuming, of course, that the program terminates!

3.2 Reconstructing Irreversible Environment Machines

The correspondence between call-by-name environment machines and the Geometry of Interaction has been detailed by Danos and Regnier [4]. Here our only addition to that work is to show how to compile these ideas using the same notion of compilation from before. Hence we are implementing both compilers in a common framework.

We begin by giving some intuitions on how the optimisation works. The basic idea comes from the fact that during a reversible computation when we are looking up a variable in the environment we travel in the graph to correct part of the graph. After arriving, finally, at the corresponding term we first evaluate it, then we return back along *exactly* the same path that we came along. Now, if we were to store the address of the place that we came from when we started looking up a variable (and also make a copy of the token) then we could jump straight back to the place where the path would return. We would then restore the token and continue the above process.

To give the compilation for this we need to incorporate slightly more complex contexts so that the return address and environment can be stored. There is also a major simplification in the code produced since when we jump back we do not need a lot of the information that we required before to route the token back to the place of the call. In fact the simplification is so great that we no longer require the p, q, r, s combinators.

In the following we will make the above more precise thus showing a way of compiling the code for the Krivine Machine.

Auxiliary Doors

The purpose of the $t!^*$ combinators, $t!^*(u:e, v:s) = (e, u\cdot v:s)$, is to save the contents of the environment u for when return into the box at the same auxiliary door. Since we never return here in an irreversible computation we can simply throw this away (we will see shortly that the application makes a copy of the environment so nothing is lost). We can thus compile the auxiliary doors as follows:

$l1$:	discard br $l2_{out}$

where $\mathsf{discard}(u:e, s) = (e, s)$. Writing this as a machine transition gives:

$$(l1, u:e, s) \quad \Rightarrow \quad (l2_{out}, e, s)$$

We remark again that is is useful to think of the box level as being the de Bruijn index. The above operation then becomes the familiar variable lookup for an environment machine.

It is clear that this operation is non-linear and it is certainly not reversible. There is a "hidden" garbage collection phase required to discard u which in turn requires the use of some external machinery. discard is therefore *not* an atomic operation.

Application

To evaluate an application we have to evaluate the function first and the argument in the same environment. Hence we have to *copy* e before going into the function. The following compilation will do this:

$l1$	:	d
		br $l3_{out}$

where d is now a more complex operation which involves storing the address of the argument, and also making a copy of the environment. d is thus defined as follows:

$$d(e, s) = (e, (l2_{out}, e) : s)$$

Writing this as a machine transition gives:

$$(l1, e, s) \quad \Rightarrow \quad (l3_{out}, e, (l2_{out}, e) : s)$$

There is no code for arriving at the other ports of the application node since we will not return along that path. Again, note that this is not reversible and linear, and requires external machinery to perform the copying; so it is not an atomic operation.

Abstraction

If we arrive at the top of an abstraction node then we require the result of the body of the lambda; evaluated in an environment that has been updated from the stack.

The only other place that we can now arrive at an abstraction is from the variable edge, if in the evaluation of the body we need to look up the value of that variable. At this point we remove the top element of the environment, update the environment and jump to the address that we find. (The value that we are looking up will be at the top of the environment because of the coding of the auxiliary doors given above.) The compilation is thus given by:

$l1$	:	!	$l3$	:	!*
		br $l2_{out}$			br l

Where ! is exactly the same operation as before, but !* is redefined as:

$$!^*((l, e) : e', s) = (e, s)$$

where we jump to the address l in the graph. Writing this as machine transitions gives the following:

$$\begin{array}{lcl} (l1, e, u : s) & \Rightarrow & (l2_{out}, u : e, s) \\ (l3, (l, e) : e', s) & \Rightarrow & (l, e, s) \end{array}$$

In all of the above we didn't need the l, r information. In addition the code for fan-in is just the identity on contexts, hence no code is generated: we just connect the edges without using these nodes (so no need for L, R). Summarising the above compilation, using the names of subterms as addresses, gives the familiar Krivine Machine:

$$\begin{array}{llcl} \text{(app)} & (MN, e, s) & \Rightarrow & (M, e, (N, e) : s) \\ \text{(abs)} & (\lambda M, e, u : s) & \Rightarrow & (M, u \cdot e, s) \\ \text{(var1)} & (1, (M, e) \cdot e', s) & \Rightarrow & (M, e, s) \\ \text{(var2)} & (n + 1, u \cdot e, s) & \Rightarrow & (n, e, s) \end{array}$$

Having such a correspondence between the two formalisms gives us, on one hand, the possibility of using the Geometry of Interaction to prove correctness of the Krivine Machine, and on the other hand the Krivine Machine provides some intuitions about the internal workings of the Geometry of Interaction Machine.

Remark 3.2 *Our presentation is a slight variant of that given by Danos and Regnier [4] in that we don't need the multiplicative information. Technically, the only difference is that we are working with slightly longer well-balanced-paths.* ♢

We have not shown the compilation for recursion for either the reversible or irreversible machines. They are both coded the same way using a cycle in the graph exactly as it is done for graph rewriting, see for example [10]. We can code this cycle without needing to introduce any new nodes in the graph, so that the compilation of recursion can be given in terms of the nodes defined here. We refer the reader to [9] for the details. The important point to note is that we compile recursion in the same way for both implementations that we are considering here.

4 Test Data

Having presented the Geometry of Interaction Machine and the Krivine Machine in the same framework, we next address the main issues of this paper which is to compare the two implementations. We provide a series of benchmark results which give evidence that the Geometry of Interaction Machine performs better than the Krivine Machine. We then discuss some reasons why this is the case.

In Figure 2 we show some example programs that can be used on our implementation [2]. The language implemented is basically a sugared λ-calculus; it should be clear what the corresponding λ-terms are that these programs represent.

The programs given correspond to Church numerals and functions operating upon them. Such terms provide an easy way to give a thorough test of reduction since we can generate a large number of β-reductions with small terms. The examples given are taken from those used by Asperti in the study of optimal reduction [1].

Table 1 shows the results of the execution of some of the defined functions. The contents of the table include the term together with the execution times for both the Geometry of Interaction Machine and the Krivine Machine.

Table 2 shows the results of another set of test data that constitute the terms that are better suited to execution on the Krivine Machine.

In the next subsection we will explain these results in more detail.

4.1 Analysis of Results

We begin by summarising the two implementations. First, neither of the machines do any form of sharing; they are both call-by-name implementations. At first sight the Krivine Machine looks simpler, but we should note the essential differences of the two:

- The Geometry of Interaction Machine transitions are linear. Nothing extra is needed for the run-time system like copying and discarding machinery.
- The Krivine Machine transitions copy and discard environments that make them very far from atomic operations.

Therefore the cost of a computation is not captured in the Krivine Machine, whereas in the Geometry of Interaction Machine everything is included.

From the first set of benchmark results, Table 1, we can see a vast difference in the execution times of the two machines for all the programs given. The difference ranges from around 10 times to, in some cases, several hundred times faster. There are also occasions when the Krivine Machine

```
let I = fn x => x;;

let zero = fn f => fn x => x;;
let one  = fn f => fn x => f x;;
let two  = fn f => fn x => f (f x);;
let three = etc...

let Succ = fn n => fn x => fn y => (x (n x y));;
let Mult = fn n => fn m => fn x => (n (m x));;

let Pair = fn  x => fn y => fn z => (z x y);;
let Fst = fn x => fn y => x;;
let Snd = fn x => fn y => y;;

let Nextfact = fn p => let n1 = (p Fst) in
               let n2 = (Succ (p Snd)) in
                     (Pair (Mult n1 n2) n2);;

let Fact = fn n => (n Nextfact (Pair one zero) Fst);;

let Add = fn m => fn n => fn x => fn y => (m x (n x y));;

let Nextfib = fn p => let n1 = (p Fst) in
    let n2 = (p Snd) in (Pair (Add n1 n2) n1);;

let Fib = fn n => (n Nextfib (Pair zero one) Fst);;

let g = fn n => n two I tt;;
```

Figure 2: Test Data

Time (ms)	GOI Machine	Krivine Machine
Fact five I tt	383	114 183
Fact six I tt	2 516	core dump
Fib seven I tt	133	966
Fib ten I tt	666	20 716
Fib (Add three ten) I tt	7 016	1 140 000
Fib (Add six ten) I tt	33 850	core dump
g ten	216	1 416
g (Add three ten)	4 983	50 166
g (Add six ten)	38 716	430 000

Table 1: Test Data: 1

Time (ms)	GOI Machine	Krivine Machine
I I I I I I I I I I I I tt	116	0.1
I I I I I I I I I I I I I I I I I I tt	6 716	0.2
I I I I I I I I I I I I I I I I I I I I I I tt	107 583	0.2

Table 2: Test Data: 2

core dumps if a sufficiently large program is evaluated. For the Geometry of Interaction Machine, we have not been able to generate core dumps — no matter how complicated the program is, the Geometry of Interaction Machine seems quite happy to evaluate it in a very small amount of memory —, but warning: we gave up waiting for some large examples since the time was getting rather long!

In Table 2 we have constructed some (very simple) examples that show when the Krivine Machine performs better. In fact the terms given here correspond to a very easy way of generating terms (and subterms) of *very* high type. For example in the term II, the first I has type $\alpha = (A \to A) \to (A \to A)$. Another application introduces a term if type $\alpha \to \alpha$. At each step we are doubling the size of the type of the term.

The unfortunate phenomenon is that the Geometry of Interaction Machine doesn't handle terms of very high type too well. The reason is in fact that Geometry of Interaction is insensitive to η-conversion: when we evaluate II this is the same as evaluating $(\lambda xy.xy)(\lambda z.z)$ which is the η-expanded version such that the term corresponds to the type structure. When we expand the above terms in this way we generate much larger graphs, and hence the paths are considerably longer.

If we were to expand the terms for the evaluation on the Krivine Machine, then we see again that the Geometry of Interaction Machine performs better.

We are, however, not too disturbed by this fact since most of the time people don't write programs at very high types.

In summary, since the Geometry of Interaction Machine is free from copying environments and garbage collection, it runs in a much smaller space, and considerably faster than the irreversible counterpart.

5 Conclusions

In this paper we have shown that the basic Geometry of Interaction Machine, which performs a sequential call-by-name reversible strategy is more efficient in general than the corresponding non-reversible strategy coded as an environment machine.

We are not claiming that either the Krivine Machine or the Geometry of Interaction Machine are good for practical implementations, simply since both are call-by-name. However, this work has shown that the reversible approach can perform substantially better than an environment machine. This provides then a good motivation to study other evaluation orders in the Geometry of Interaction Machine in the hope that we can continue to perform better than the environment machine counterpart. Only then will we be in a position to state whether the Geometry of Interaction Machine could ever seriously be used for implementation. The language and implementation presented here consists only of the pure λ-calculus. We focused on this core language just to keep the ideas simple. However, the theory and the implementation have been developed to cover more realistic functional programming languages with constants and data-types. For more details of the theory see [8], and for the implementation and test data showing that the same results hold for more realistic languages with the addition of data-types see [2].

There are a number of directions and questions that are now open.

- Is this also the case for other evaluation strategies? For the moment we are working on call-by-value, call-by-need and optimal reduction strategies in this framework. Preliminary investigations suggest that our results still hold for call-by-value and call-by-need, but the case for comparing optimal reduction strategies (implemented as Interaction Nets rather than as an environment machine) is less obvious to us how to compile both methods in a comparable way.
- If the previous set of experiments proves successful the next question to ask is how does this compare with other kinds of implementation, for example Turner Combinators, graph reduction, and to investigate performance issues relative to production level compilers such as Standard ML and Caml-light.

The implementation that was used to generate the results in this paper is available from URL `http://lix.polytechnique.fr/~mackie/goi`.

References

[1] Andrea Asperti. $\delta \circ !\epsilon = 1$. In Jieh Hsiang, editor, *Proceedings of the 6th Rewriting Techniques and Applications Conference (RTA'95)*, LNCS, pages 102 – 116. Springer-Verlag, 1995.

[2] Bruno Blanchet. Implémentation de la géométrie de l'intéraction, 1995. Rapport de Stage d'Informatique, École Polytechnique.

[3] Vincent Danos. *La Logique Linéaire appliquée à l'étude de divers processus de Normalisation (principalement du λ-calcul)*. PhD thesis, Université Paris VII, June 1990.

[4] Vincent Danos and Laurent Regnier. Reversible and irreversible computations: GOI and λ-machines, 1994.

[5] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[6] Jean-Yves Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, Amsterdam, 1989. North Holland.

[7] Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and André Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.

[8] Ian Mackie. Compiling via the Geometry of Interaction, 1995. Submitted for publication.

[9] Ian Mackie. The Geometry of Interaction Machine. In *Proceedings, 22nd ACM Symposium on Principles of Programming Languages*, pages 198–208, January 1995.

[10] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[11] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[12] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, Oxford University, 1971.