# The Geometry of Interaction Machine

Ian Mackie*
Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ England
im@doc.ic.ac.uk

**Abstract**

We investigate implementation techniques arising directly from Girard's Geometry of Interaction semantics for Linear Logic, specifically for a simple functional programming language (PCF). This gives rise to a very simple, compact, compilation schema and run-time system. We analyse various properties of this kind of computation that suggest substantial optimisations that could make this paradigm of implementation not only practical, but potentially more efficient than extant paradigms.

## 1 Introduction

There are a number of established implementation techniques for functional programming languages. Broadly speaking these are based on graph rewriting [Wad71], stack manipulation, for example the SECD machine [Lan64], and buffered data-flow [FH88]. In this paper we present the *Geometry of Interaction Machine*—an implementation technique based on *sequential*, rather than buffered, data-flow. The significant features of this approach include:

- Compact object code.
- Small and compact run-time system.
- Sound semantic foundation arising from Linear Logic [Gir87] and the Geometry of Interaction [Gir89b, Gir89a].

Semantics has played a crucial rôle in the past by providing a tool for proving (static) properties of programs and implementations. A new paradigm in semantics developed by Jean-Yves Girard under the name *Geometry of Interaction* can now allow the study of the dynamics of computation. His motivations were to provide mathematical tools for the study of the cut-elimination process in Linear Logic proof structures; but the fundamental ideas are applicable to computation in general. The prominent features of his programme can be summarised as follows.

- The semantics is *syntax free* in that it avoids the *bureaucratic* issues that come with the notion of substitution (*cf.* the $\lambda$-calculus).
- The semantics is chiefly a study of the dynamics of the cut-elimination process (hence $\beta$-reduction) using denotational tools. The model is termed *dynamic* since it captures the essential computational content.
- The semantics is operational in the sense that there are a set of prescribed rules specifying how computation should proceed at each step.

It is hoped that this new understanding of the dynamics of computation could lead to radically new implementation techniques for programming languages, we hope that this paper provides a contribution towards this goal. In particular the dynamics give a highly decomposed notion of a reduction step such that the operational handling of information is *linear* and even *invertible* — something that is certainly not true about standard implementations of $\beta$-reduction. We shall demonstrate in this paper that this idea can be trivially implemented.

The philosophy behind this semantics is a notion of information flow through a program. The meaning of a program can be regarded as a set of information flows which are called *paths* in the program. As reduction proceeds one would expect that the information flows are invariant—no new information flows can be created, and none can be lost. The meaning of a program is given by a set of paths, and computation is given by a calculating the transitive closure of these paths. Girard has a useful analogy of thinking of these paths as electrical circuits—as reduction proceeds the electrical connectivity is preserved.

We refer the reader to the work in [ADLR94, DR93] for detailed work on the notion of paths in the $\lambda$-calculus; and [Mac94] for the extension to cover PCF.

The applications of this theory have been substantial. It has been the basis of the solution to a number of outstanding problems in Computer Science, for example the Full abstraction for PCF [AJM94], and it has shed new insight into existing problems, for example optimal reduction for the $\lambda$-calculus [GAL92], etc.

The purpose of this paper is to extend the Geometry of Interaction interpretation to a simple functional programming language (PCF) and show how simply we can implement this semantics directly. One of the main hopes of the Geometry of Interaction is to obtain a deeper understanding of the dynamics of computation and, hopefully, to give new

*Research supported by an UK SERC Studentship and ESPRIT Basic Research Action 6454 "CONFER".

insights into implementation techniques and optimisations. We see this paper as a contribution to this work in that we:

- Provide an extension of the Geometry of Interaction interpretation to cover a simple functional programming language (PCF).
- Give a concrete implementation of this semantics by compiling it directly into assembly language.
- Provide some optimisations of this compiler technology arising from properties of the semantic framework.

Additionally, this compilation technique provides a very clear *computational* understanding of Girard's Geometry of Interaction.

### Related work

This work is about implementing Girard's Geometry of Interaction. Two related pieces of work are Gonthier et al. for their work on *Optimal reduction* and Danos and Regnier for their work on *Virtual reduction*. Both of these pieces of work are related to "computing paths".

- Optimal reduction: The algorithm for implementing optimal reduction presented in [GAL92] uses the Geometry of Interaction as a proof technique to show that all paths are preserved under reduction. One can see each local rewrite rule of there system as computing a set of paths.
- Virtual reduction: The work of Danos and Regnier [DR93] looks at computing paths in a more direct way, and can be seen as a decomposition of the work on optimal reduction but still rewriting the graph, hence again their computation is using the Geometry of Interaction to show invariance of reductions.

In this paper we *compute the execution path directly*, by performing a simple token pushing semantics. In particular we do *not* rewrite the structure of the graph—we provide a compilation of programs directly into assembly language.

Finally we remark that there is also some recent work of Danos and Regnier that is in the same spirit of this work, but we are unable to comment on the relationship at the present time.

### Overview

The rest of this paper is organised as follows. First, we recall the syntax of PCF and show an encoding into Linear Logic proof structures. Section 3 gives the theoretical foundation of the implementation of PCF. In Section 4 we give the main contribution by showing how we can effectively implement this semantics in a very simple way in assembly language for a simple machine. We will explain the concept of a path alongside the compilation that we will perform. Section 5 discusses efficiency and we show how we can improve the implementation by a time-space trade off. We conclude our ideas in Section 6 by suggesting some applications of this kind of implementation and discuss some further directions of this work.


### Acknowledgement

I am most grateful to Vincent Danos, Mark Dawson, Simon Gay, Radha Jagadeesan and Pasquale Malacaria for discussions on this work, and to the anonymous referees for their detailed comments.


## 2 PCF in Linear Logic proof structures

We first recall the syntax of PCF, (Programming Language for Computable Functions) [Plo77], which we take to be standard. We have a set of variables $x : \sigma$, a set of constants $c : \sigma$, an abstraction $\lambda x.M$ and application $MN$. The constants that we shall use are natural numbers: $n$ : **nat**, booleans: **tt**, **ff** : **bool**; together with arithmetic functions over these constants: **succ**, **pred** : **nat** $\to$ **nat** and **iszero** : **nat** $\to$ **bool**, a conditional **cond** : **bool** $\to \sigma \to \sigma \to \sigma$, and a fix point combinator $Y : (\sigma \to \sigma) \to \sigma$.

Linear Logic provides a decomposition of the functional arrow $A \to B$ into $!(A \multimap B)$:

- The type reflects the fact that functional application is a two phase process — a *linear* ($\multimap$) part that suggests a single usage of the argument, and a non-linear part reflected by the modality ! (the *exponential*) which allows multiple usage of the argument.
- This in turn induces a decomposition of the $\lambda$-calculus in which both the *abstraction* and *application* are decomposed into more primitive operations.
- Information is made explicit in the proof structure—the previously hidden *structural rules* of Weakening (discarding an argument) and Contraction (copying an argument) are made explicit.
- Finally, it provides a new way of representing terms in the calculus—Proof Nets—which have strong connections with standard graph representations of the $\lambda$-calculus.

There are several codings of the $\lambda$-calculus into Linear Logic proof net structures in the literature [Dan90, GAL92]. Here we will very briefly recall one translation, (which is based on embedding the intuitionistic arrow $A \to B$ into $!(A \multimap B)$), and give the extension of this translation to cover the constants of PCF.

### Variables

The variable $x$ is translated into an *axiom* link, which is represented simply by a piece of "connecting wire", drawn simply as:



### Abstraction and Application

The abstraction $\lambda x.M$ and the application $MN$ are translated into the following nets as shown in Figure 1. Intuitively we can see these as standard graphs for the $\lambda$-calculus where we have decomposed abstraction ($\lambda$) into a par ($\parr$) and the exponential (!). The exponential of linear logic requires a notion of a *box*, which we have drawn around the abstraction connecting the ! node and the ? nodes. We call the top of the box ! the *principal door*, and the free wires that exit the box from the bottom are called the *auxiliary doors* which we draw as (?) nodes.

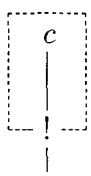
Application (@) is decomposed into a Tensor ($\otimes$) and Dereliction ($D$). The Dereliction is connected to the translated function $M$ by a *cut* link (drawn as a piece of connecting wire), and the result of the application (the top of the Tensor) is connected to the rest of the structure by an axiom link.

The main point of difference with standard graphs is that the binding variable is connected to its occurrence in the term. If no such variable exists then we represent this as a terminal node to the net, called a Weakening node. Multiple occurrences of a variable are made explicit using the triangular Contraction (sharing) node.

We have also labelled some of the edges of the structure that will be used for the Geometry of Interaction interpretation that we will present shortly. All other edges are assumed to have a label 1.

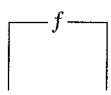
### Constants at base type

Constants at base type, $c \in \{\overline{n}, \mathbf{tt}, \mathbf{ff}\}$, are represented as terminals of the net structure. For this translation we require that the constants are boxed values which we draw as follows:

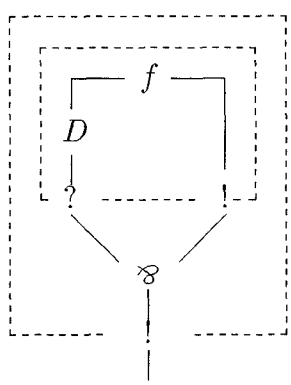



### Arithmetic functions

The constant functions, $f \in \{\mathbf{succ}, \mathbf{pred}, \mathbf{iszero}\}$, are represented in a similar way to axiom links, *i.e.* a piece of connecting wire, but labelled with the corresponding function:



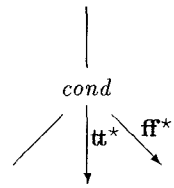

The orientation of this should be thought of as taking inputs from the left hand side, and the result on the right hand side. Under the chosen translation, it is necessary to package these up with the constructors of linear logic. Specifically, we need to do the following:




Hence we can see this structure as a boxed Dereliction, and then a par and a box (hence a $\lambda$) has been added to close up the structure.

### Conditional

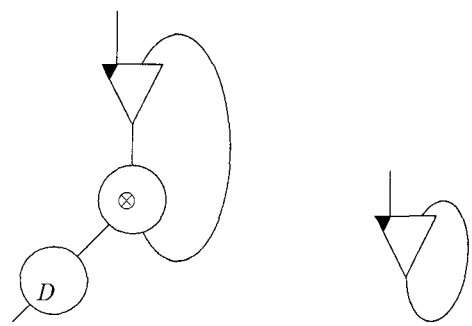
The conditional of PCF is represented as the following graph structure:




where we label the true and false branches with labels $\mathbf{tt}^\star$ and $\mathbf{ff}^\star$ respectively. The leftmost argument is the boolean test, and the result is at the top of the structure. Again we are required to package this constant up in Linear Logic connectives. The idea is the same as the arithmetic functions in that we close the construction up using the codings for $\lambda$.

### Recursion

We code recursion without the need to introduce any new nodes by generating a cyclic structure that explicitly "ties the knot". We can now get the coding of the PCF constant $\Omega$ as $YI$. The following diagram shows the graph representations of $Y$ and $\Omega$.




This corresponds exactly to the coding of recursion in graph reduction, see [Pey87]. The path computation semantics is therefore given in terms of the rules for Tensor, Dereliction and Contraction.

## 3 The Geometry of Interaction interpretation

The proof net structures have been annotated with *directed* labels $p, q, r, s, t, d$ which are the generators of the dynamic algebra $\Lambda^*$ used for the Geometry of Interaction. We will call the extension to cover PCF constants $\Lambda^*_{\mathsf{pcf}}$ [Mac94].

$\Lambda^*$ is a single sorted $\Sigma$ algebra. We write $x$ and $y$ for the variables, and all other symbols $0, 1, p, q, r, s, t, d : \Sigma$ are constants of the theory. There is a associative multiplication operator $\cdot : \Sigma \times \Sigma \to \Sigma$, (that we will omit), which has unit 1 and absorbing element 0. The theory is equipped with an involution $* : \Sigma \to \Sigma$, which is used to label edges as we travel in the opposite direction, and an exponential operator $! : \Sigma \to \Sigma$, that is used to label *all* the labels that are inside a box structure. The following equations define the properties that we require. We present the algebra, which is essentially taken from [Dan90].

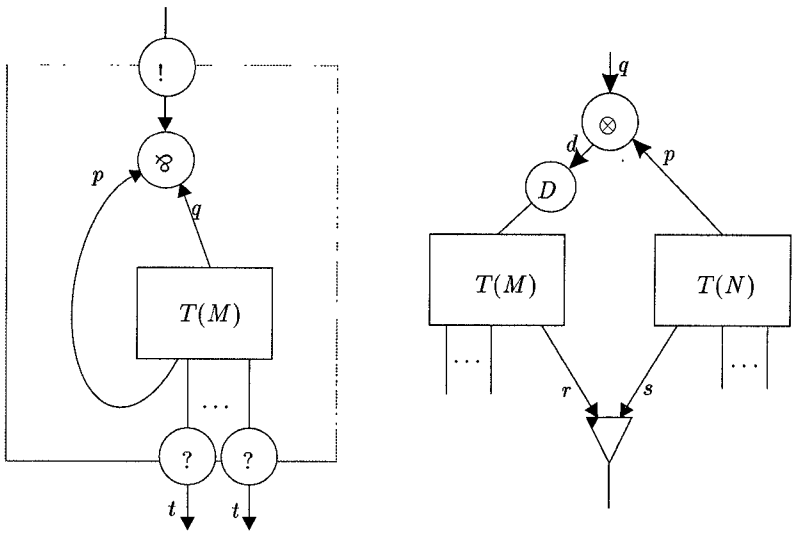



Figure 1: Coding the $\lambda$-calculus into Linear Logic proof structures

$$\begin{array}{ll} 0^* = !(0) = 0 & 1^* = !(1) = 1 \\ 0x = x0 = 0 & 1x = x1 = x \\ !(x)^* = !(x^*) & (xy)^* = y^*x^* \\ (x^*)^* = x & !(x)!(y) = !(xy) \end{array}$$

**Annihilation:** There are six constants to consider: $p$ and $q$ are the multiplicative coefficients, $r$ and $s$ are the Contraction coefficients, $d$ is the Dereliction coefficient, and finally, $t$ is the *auxiliary door* coefficient.

$$\begin{array}{lll} p^*p = q^*q = 1 & q^*p = p^*q = 0 & d^*d = 1 \\ r^*r = s^*s = 1 & s^*r = r^*s = 0 & t^*t = 1 \end{array}$$

**Communication:** The exponential coefficients interact with the exponential morphism as defined by:

$$\begin{array}{ll} !(x)r = r!(x) & !(x)s = s!(x) \\ !(x)t = t!!(x) & !(x)d = dx \end{array}$$

The additional constants which we need for PCF are: natural numbers $n$, which we shall write as $\overline{n}$ to avoid confusion with the constants 0 and 1; and boolean constants $\mathbf{tt}, \mathbf{tt}^\star$ and $\mathbf{ff}, \mathbf{ff}^\star$, together with function constants **succ**, **pred** and **iszero**. Hence we are overloading notation by using the same names for constants in $\Lambda^*_{\mathsf{pcf}}$ and PCF.

We have the following annihilation equations:

$$\begin{array}{lcllcl} \mathbf{tt}^\star\mathbf{tt} = \mathbf{ff}^\star\mathbf{ff} & = & 1 & \mathbf{ff}^\star\mathbf{tt} = \mathbf{tt}^\star\mathbf{ff} & = & 0 \\ \textbf{iszero}\ \overline{0} & = & \mathbf{tt} & \textbf{iszero}\ \overline{n+1} & = & \mathbf{ff} \\ \textbf{pred}\ \overline{0} & = & \overline{0} & \textbf{pred}\ \overline{n+1} & = & \overline{n} \\ \textbf{succ}\ \overline{n} & = & \overline{n+1} & & & \end{array}$$

The whole concept of this semantic interpretation is to use this theory to talk about which paths are the "good ones" in the proof net structures.

Proof nets are graphs $G = (V, E)$ in the usual sense. The *vertices* $V$ are the logical symbols (for example $\otimes$ and $\parr$ in the multiplicative fragment) and the *edges* $E$ are the connecting links that join these symbols to their premises and conclusions.

There is a natural notion of a *walk* in a graph which is nothing more than a travel around the structure. A walk in a graph is specified uniquely by a sequence of vertices.

Here we take a slightly different perspective on these standard notions.

- First we label the *edges* of a proof net rather than the vertices. The labels that we use are elements of the algebra $\Lambda^*_{\mathsf{pcf}}$.
- Consequently the walks in a proof net are specified by a sequence of *edges* rather than vertices.

We are now in a position to define the notion of a path in a proof net structure.

**Definition 3.1** *A path $\phi$ in a proof net structure is a walk along the labelled edges such that $\Lambda^*_{\mathsf{pcf}} \vdash \phi \neq 0$.*

Intuitively, we can think of the algebra restricting the way that we can walk over the structure. The Geometry of Interaction interpretation is nothing more than a mechanism for picking out all the "good paths" in the term which can be characterised as being the paths that *survive* the action of reduction. Hence the Geometry of Interaction semantics can be seen as a decomposed way of looking at reduction.

There is a particular (unique) path that exists in a PCF program at base type that starts and finishes at the root of the graph. This will be called the *execution path*.

We will use a model of this algebra based on simple data structures that will be the basis of the run-time system of our implementation. A context is a triple $(\mathcal{M}, \mathcal{E}, \mathcal{D})$ where $\mathcal{M}$ is the multiplicative stack, $\mathcal{E}$ is the exponential tree, and $\mathcal{D}$ is the data value (a single element stack). These are defined by the following grammar:

$$\begin{array}{lll} \mathcal{M} & ::= & l:\mathcal{M} \mid r:\mathcal{M} \mid \Box \\ \mathcal{E} & ::= & L:\mathcal{E} \mid R:\mathcal{E} \mid <\mathcal{E},\mathcal{E}> \mid \Box \\ \mathcal{D} & ::= & \overline{n} \mid \mathbf{tt} \mid \mathbf{ff} \mid \Box \end{array}$$

where we write $\Box$ for the empty stack.

We define the carrier of the model to be the partial injective functions on $(\mathcal{M},\mathcal{E},\mathcal{D})$.

The operations are interpreted as:

- $[\![\cdot]\!]$ is defined to be the composition of partial injective functions.
- $[\![f^*]\!]$ is defined to be the inverse partial injective function $[\![f]\!]^{-1}$.
- $[\![!]\!]$ is defined as:

$$[\![!(f)]\!](m,<e_1,e_2>,v) = \begin{array}{l}\text{let } (m',e_2',v)=[\![f]\!](m,e_2,v) \\ \text{in } (m',<e_1,e_2'>,d)\end{array}$$

The generators of the algebra are interpreted as follows:

**Constants**

$[\![1]\!]$ is the identity and $[\![0]\!]$ is the nowhere defined context transformer on $(\mathcal{M},\mathcal{E},\mathcal{D})$.

**Multiplicatives**

$$\begin{array}{lll} [\![p]\!](m,e,v) & = & (l:m,e,v) \\ [\![q]\!](m,e,v) & = & (r:m,e,v) \end{array}$$

**Exponentials**

$$\begin{array}{lll} [\![r]\!](m,<e_1,e_2>,v) & = & (m,<L:e_1,e_2>,v) \\ [\![s]\!](m,<e_1,e_2>,v) & = & (m,<R:e_1,e_2>,v) \\ [\![t]\!](m,<e_1,<e_2,e_3>>,v) & = & (m,<<e_1,e_2>,e_3>,v) \\ [\![d]\!](m,e,v) & = & (m,<\Box,e>,v) \end{array}$$

**PCF Constants**

$$\begin{array}{lll} [\![\mathbf{tt}]\!](m,e,\Box) & = & (m,e,\mathbf{tt}) \\ [\![\mathbf{tt}^*]\!](m,e,\mathbf{tt}) & = & (m,e,\Box) \\ [\![\mathbf{ff}]\!](m,e,\Box) & = & (m,e,\mathbf{ff}) \\ [\![\mathbf{ff}^*]\!](m,e,\mathbf{ff}) & = & (m,e,\Box) \\ [\![\mathbf{iszero}]\!](m,e,0) & = & (m,e,\mathbf{tt}) \\ [\![\mathbf{iszero}]\!](m,e,n+1) & = & (m,e,\mathbf{ff}) \\ [\![\mathbf{pred}]\!](m,e,0) & = & (m,e,0) \\ [\![\mathbf{pred}]\!](m,e,n+1) & = & (m,e,n) \\ [\![\overline{n}]\!](m,e,\Box) & = & (m,e,n) \\ [\![\mathbf{succ}]\!](m,e,n) & = & (m,e,n+1) \end{array}$$

**Proposition 3.2** *The context transformers on $(\mathcal{M},\mathcal{E},\mathcal{D})$ yield a sound model of the algebra $\Lambda^*_{\mathsf{pcf}}$.*

## 4 A sequential data-flow machine

The basic notion considered here is that of *data-flow*. More specifically, pushing a *single* token around a fixed network.

- The network is a proof net structure; generated by the given translation of PCF in Section 2.
- The token is the *context* data structure $(\mathcal{M},\mathcal{E},\mathcal{D})$ that we used as a model of the dynamic algebra $\Lambda^*_{\mathsf{pcf}}$. More specifically, we see the token as a global state of the computation where we will represent $\mathcal{M}$ and $\mathcal{D}$ as registers (R0 and R1) and $\mathcal{E}$ as a dynamic tree data structure.

The token will compute the execution path of a program of base type by traversing the structure of the original network, guided by rules governed by the Geometry of Interaction. At each node on the graph the token will be transformed, and directed to the next node. This process will continue until the token returns to the root of the term (if the term normalises), otherwise the token will travel an infinite path in the term.

We begin by stating our object language which we take to be a very simple assembly language. We will assume a simple register machine with, say, 32 bit (unsigned) registers $R0$, $R1$ etc. The operations over these registers will be simple logical operations (such as shifts), comparisons and branching instructions. We propose a basic set of instructions which have trivial implementations on any architecture. The following table gives the intended meaning of the instructions that we require to code the multiplicative and constant information. We will introduce the additional structure for the exponentials on demand. Additionally, we will define several *macros* to give greater abstraction for the compilation.

| Macro | Instruction | Meaning |
|---|---|---|
| p | lsl R0 | logical shift left R0 |
| q | lsl R0 | logical shift left R0 |
| | inc R0 | increment R0 |
| | lsr Rn | logical shift right Rn |
| | mov n R1 | move value n into R1 |
| | br l | branch always to l |
| | be l | branch to l if carry equal to zero |
| | cmp0 Rn | compare Rn with zero |
| | inc Rn | increment Rn |
| | dec Rn | decrement Rn |

### 4.1 Identities

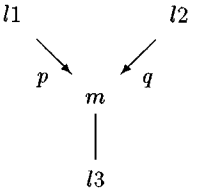
The compilation of the identities of Linear Logic proof structures produces no code, they just provide *linking information* for connecting nodes.

**Axiom:** The axiom is given by the identity on contexts, hence no context transformation is required. For our implementation, the axiom provides a piece of linking information to state which nodes are to be connected.

**Cut:** Again, no context transformation is required. The cut link specifies which nodes should be cut against each other, hence again we just provide linking information here.

### 4.2 The Multiplicatives

The multiplicatives ($m = \otimes, \parr$) are straightforward to implement, with a small run-time system. In fact a *single* register will suffice.



If we arrive from the left hypothesis (l1), with context $M$, we pass control to the conclusion and record in the multiplicative stack the information that we came from the left: $l : M$.

This indicates a very simple compilation directly into assembly language, where we can use a single, "potentially infinite" register R0 as the multiplicative register. Using 0 as the coding of "left" we can simply compile this as:

```
l1 :  p
      br l3_out
```

Where l3_out is the *other end* of the edge $l3$, which will be an input to another node; this label is linked to the other node by the linking information provided by the identities. Recall that the p macro is just a shift on a register.

The token $M$ arriving from the right hypothesis $l2$ requires that we record the information that we came from the right: $r : M$. Similarly to above, using 1 as the coding of "right", we can compile this as:

```
l2 :  q
      br l3_out
```

Finally, arriving from the conclusion, we examine the top element of the multiplicative stack and branch to either the left or right premise, and drop the top element of the stack $M$. This gets a straightforward compilation as:

```
l3 :  isr R0
      be l1_out
      br l2_out
```

Hence, for each multiplicative node in a proof structure we have the following encoding. Each node is translated in to a piece of assembly language code with three entry points. The compiler connects all labels which is the linking phase specified by the identities and cuts of the proof.

| | |
|---|---|
| l1 : p<br>br l3_out | l2 : q<br>br l3_out |
| l3 : isr R0<br>be l1_out<br>br l2_out | |

With these very simple rules we have the power to implement the linear $\lambda$-calculus, with *just one register run time system*. The size of the object code (number of lines of assembly language) is just eight times the number of constructors in the original program.

We suggested that R0 is a *potentially* infinite register in the sense that in general we have no idea just how large this stack will grow. However, in a typed setting, we have the following result:

**Proposition 4.1** *The size of the multiplicative stack $\mathcal{M}$ is bounded by the maximum size of the type of any subterm.*

Here we will assume that no program ever goes beyond a 32 bit register, which will allow the coding of programs at *very high* types. Of course, if we needed higher types, then we could extend the above encoding to handle larger stacks in a very simple way. Our aim here is to try to show just how simple we can code a PCF program.

### 4.3 The Exponentials

For the exponentials we are not so lucky in that we require a data structure a little more complicated than a simple register. We will assume a *double linked* tree data type, (so there are pointers to the parent node), corresponding to the exponential context tree $\mathcal{E}$. We define operations on this data structure corresponding to the exponential context semantics. Let *ep* (environment pointer) be a pointer to a node in the tree which indicates which part of the context we are currently transforming. There are some natural implementation techniques that we use for these which we will take for granted, however we remark that there is some interesting theory behind this that has recently been studied by Danos and Regnier [personal communication].

#### Dereliction

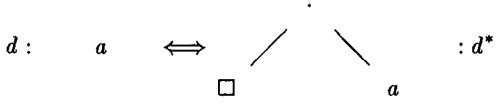
This is the only connective that generates or destroys part of a context tree; it creates/removes levels of the environment.




If we arrive from $l1$ with exponential context tree $a$ then we add a new level to the environment, hence create a new empty tree. We then continue downwards to the node connected to $l2$. Arriving from $l2$ we do the reverse operation, and destroy part of the environment. The compilation rule for this node is given by:

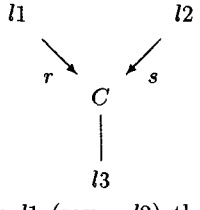
| | |
|---|---|
| l1: d<br>br l2_out | l2: d*<br>br l1_out |

where we can code $d$ and $d^*$ by a sequence of instructions which can be seen diagrammatically as the following context tree transformation:




#### Contraction

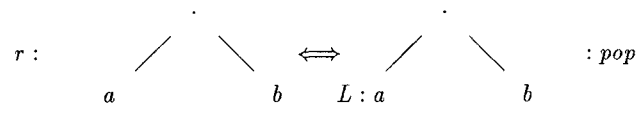
A Contraction is entirely similar to the multiplicative node, except we must place the information on the current part of the exponential stack.




If we arrive from $l1$ (resp. $l2$) then we should record the information that we came from the left (resp. right) on the context tree, and continue towards the node connected to $l3$. Arriving from $l3$ will pop the current exponential information from the context tree and branch to either $l1$ or $l2$ depending on the status of the context tree. Hence the compilation scheme for this node is given by:

| | |
|---|---|
| l1: r<br>br l3_out | l2: s<br>br l3_out |
| l3: pop<br>be l1_out<br>br l2_out | |

Where the following explains the action of the code on the context tree:




And similarly for $s$, where pop will set the carry flag accordingly.

### Weakening

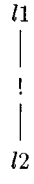
Since we are working on programs of ground type, and computing a root-to-root path, no path will ever arrive to a Weakening node, hence there is no need to produce any code.

### Promotion

As the token traverses the structure of the proof, we will enter and leave ! boxes. This is the level of the environment that we are working in. There are two different cases to consider, depending on *where* we enter and leave an exponential box. We will look at each one in turn.

### Principal door

If we arrive at the principal door of a box ($l1$), then we increase the level of the environment we are working in. Dually, exiting a box at the principal door at ($l2$) requires that we decrement the level of the environment.



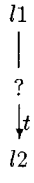

The compilation of this idea is given by the following code:

| | |
|---|---|
| l1: ep = ep→right<br>br l2_out | l2: ep = ep→prev<br>br l1_out |

The effect of ep=ep→right and ep=ep→prev can be seen diagrammatically as the following *walk* on the context tree, as shown in Figure 2.

### Auxiliary Doors

There are two steps involved in the token passing through an auxiliary door of a box.




- Analogously with passing through the main door, we must increase (if entering the box from $l2$) or decrease (if leaving the box from $l1$) the level of the environment that the token is operating at.
- If the token is leaving the box then the exponential context tree is transformed in such a way that the context of the box is saved somewhere safe so that it can be restored when the token re-enters the box. This is achieved using the $t$ operator (associativity) which can be thought of as pushing the environment onto a stack for later use.

  The dual of the above is performed if the token is entering the box from an auxiliary door.

Hence the following code is compiled for each auxiliary door of a box:

| | |
|---|---|
| l1: ep=ep—prev<br>t<br>br l2_out | l2: t*<br>ep=ep→right<br>br l1_out |

We can encode $t$ and $t^*$ by a sequence of instructions which we can see diagrammatically as the following context tree transformation given in Figure 3.

## 4.4 PCF constants

Here we give the compilation rules for each of the PCF constants. We remark that for the translation that we are using into Linear Logic proof nets we require the use of exponentials with these. Here we just give the codings of the constants and leave the reader to compose the full compilation using the (previously defined) code fragments for the exponentials.

### Constants at base type

Both natural numbers and boolean values are straightforward to code—we just return along the same path having pushed the value onto the data part of the context, which again we can see as being a single register, $R1$. We will code **tt** as 0 and **ff** as 1, and natural numbers are compiled trivially. We write $c$ for a general data value.

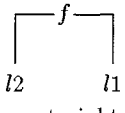



The compilation is straightforward, yielding the following code module for a constant:

| |
|---|
| l: mov c R1<br>br l_out |

### Arithmetic functions

The unary functions **succ**, **pred** and **iszero** are given by the following:




From $l1$ we simply pass straight through without change to the context. On the return up through $l2$ we apply the function to the data item. The compilation of this module can be given by:

| | |
|---|---|
| l1: br l2_out | l2: f<br>br l1_out |

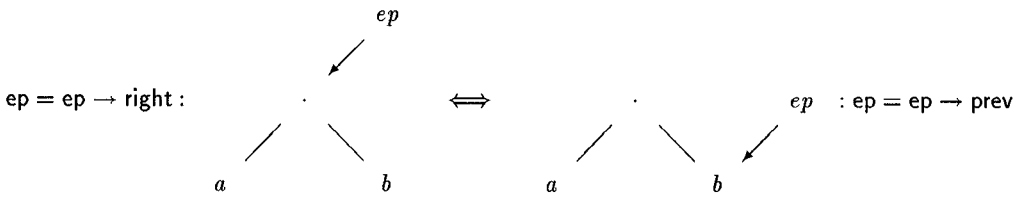



Figure 2: Entering and leaving a box structure

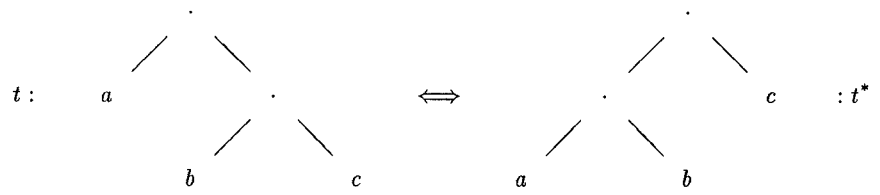


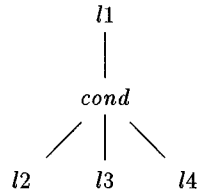

Figure 3: Context tree transformations $t$ and $t^*$

Where the different functions (f) are coded as the following simple register operations:

| | | |
|---|---|---|
| **succ** | = | inc R1 |
| **pred** | = | dec R1 |
| **iszero** | = | cmp 0 R1 |

### Conditional

We have the following module with four inputs/outputs.




When we ask for the result of a conditional (arriving along $l1$), we ask for the boolean value, so we travel along the path out of $l2$. When we return, we either pass control to the true of false branch, depending on the answer in the register $R1$. Note that this data value is consumed by the test. The result of either branch is then passed back as the result of the conditional. This suggests the following code fragment:

| | |
|---|---|
| l1: br l2_out | l2: isr R1<br>be l3_out<br>br l4_out |
| l3: br l1_out | l4: br l1_out |

### Recursion

The compilation rules for recursion are given in terms of the codings of Contraction ($C$), Dereliction ($D$) and Tensor ($\otimes$).

## 4.5 Execution

The execution of a program of base type is simply given by starting at the root of the term with the empty context $(\Box, < \Box, \Box >, \Box)$ (so clearing R0, R1 and initialising the tree structure to be the empty tree) and following the path in the code generated. When the path reaches the root again we will have a token structure $(\Box, < \Box, \Box >, d)$ where $d$ is the result of the computation (hence the result is in R1). Note that we set the exponential context tree to be $< \Box, \Box >$. This is required since the PCF constants under the chosen translation are represented by boxed values.

The correctness of this implementation comes directly from the correctness of the geometry of interaction. We recall a result from [Mac94] which states that the path computation is as good as reduction in PCF.

**Proposition 4.2** *Let $M$ be a PCF program at base type. If $M$ reduces to a constant $c$ then, for the execution path $\phi$, we have $\Lambda^*_{\mathsf{pcf}} \vdash \phi = c$.*

**Example 4.3** *We give a small example to show how things work. Space prohibits giving a substantial example, but the following at least gives some indication of how execution proceeds.*

*The example that we take is simply $(\lambda x.x)3$. The execution path for this term is given by:*

$$q^*d^*!(qp^*)dp!(\overline{3})p^*d^*!(pq^*)dq$$

*which, using the equations in Section 2, reduces to $!(3)$ as required.*

*The output of our compilation algorithm is given in Figure 4. When run on our implementation, with tracing set, produced the following output:*

```
Finished after 19 stack operations
Result: 3
```

*The implementation also produced the trace of context transformations given in Figure 5, which shows all 19 operations. The trace shows the sequence of instructions (context transformers) which we name using the operations from $\Lambda^*_{\mathsf{pcf}}$; with the addition of using the notation $!$ and $!^*$ to indicate that we enter and leave a box respectively.*

## 5 Optimisations

The Geometry of Interaction can be regarded as being *locally* very efficient — it has provided a decomposition of the $\lambda$-calculus to a level where we can directly implement it on

```
l011:   q                       l322:   q
        d                               ep=ep→prev
        br l221                         br l213
l112:   p                       l323:   p
        d                               ep=ep→prev
        br l221                         br l213
l213:   d*                      l131:   ep=ep→right
        lsr R0                          mov 3 R1
        be l131                         ep=ep→prev
        br l000                         br l112
l221:   ep=ep→right             l000:   end
        lsr R0
        be l322
        br l323
```

Figure 4: Object code generated

| *Instruction* | $\mathcal{M}$ | $\mathcal{E}$ | *ep* | $\mathcal{D}$ |
|---|---|---|---|---|
| | $\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | $\Box$ |
| $q$ | $r:\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | $\Box$ |
| $d$ | $r:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,<\Box,\Box>>$ | $\Box$ |
| $!$ | $r:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,\Box>$ | $\Box$ |
| $q^*$ | $\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,\Box>$ | $\Box$ |
| $p$ | $l:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,\Box>$ | $\Box$ |
| $!^*$ | $l:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,<\Box,\Box>>$ | $\Box$ |
| $d^*$ | $l:\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | $\Box$ |
| $p^*$ | $\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | $\Box$ |
| $!$ | $\Box$ | $<\Box,\Box>$ | $\Box$ | $\Box$ |
| $\overline{3}$ | $\Box$ | $<\Box,\Box>$ | $\Box$ | 3 |
| $!^*$ | $\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | 3 |
| $p$ | $l:\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | 3 |
| $d$ | $l:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,<\Box,\Box>>$ | 3 |
| $!$ | $l:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,\Box>$ | 3 |
| $p^*$ | $\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,\Box>$ | 3 |
| $q$ | $r:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,\Box>$ | 3 |
| $!^*$ | $r:\Box$ | $<\Box,<\Box,\Box>>$ | $<\Box,<\Box,\Box>>$ | 3 |
| $d^*$ | $r:\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | 3 |
| $q^*$ | $\Box$ | $<\Box,\Box>$ | $<\Box,\Box>$ | 3 |

Figure 5: Example trace of the Geometry of Interaction Machine

a concrete machine at the lowest level of detail. However, this is offset by the *global* performance. We have that the length of a path *may* grow exponentially in the number of cut elimination steps [Dan90].

By looking at properties arising naturally from the underlying semantics, we can generate some useful optimisations of this implementation. Here we present one such optimisation, and we refer the reader to [Mac94] for a more detailed treatment of this material.

### Questions and answers

Here we will glean some insight into implementing paths by recalling some properties about paths from [Mac94].

The property that we are going to make significant use of is the notion that paths return back to the same place to answer the question asked, and moreover, this preserves the so-called well bracketed condition. What we shall outline here is a technique to avoid returning back along the execution path with each answer; we will jump back with the result to the previous question asked, and restore the token structure. We begin with an overview, then give an encoding of this idea as a modification of the Geometry of Interaction Machine.

A sequence of contexts (the trace of a path computation) gives rise to a well-bracketed sequence of questions and answers. In particular, there are critical points in the trace that ask questions of *base* type, for example the boolean test for a conditional, and the argument to a function of base type (e.g. succ). At each question $q_i$ asked, we will store the context, and a return address on a stack of *questions to be answered*. The computation continues (maybe asking more questions, answering questions, etc.) until we come to the point where we are answering the question $q_i$. At this point, rather than traversing the path backwards to where the question was asked, we pop the question stack, restore the token to be the state it was when it asked the question together with the answer, then return to the place where the question was asked. This then gives a optimisation on the path computation by reducing the length of the path being traversed by half.

We henceforth need to modify the general structure of the Geometry of Interaction Machine to handle this feature. The only change that we need make in fact is the encodings of the constants, arithmetic functions and the conditional for PCF.

We introduce a *question stack* (qs) which we define as: $qs ::= (\mathcal{M}, \mathcal{E}, l) : qs \mid \Box$, where $\mathcal{M}$ and $\mathcal{E}$ are the multiplicative and exponential contexts respectively, and $l$ is a label—the return address. To avoid repeating the diagrams for the data flow, we refer the reader to the diagrams given for the coding of the PCF constants given in Section 4; in particular, we will use the same names for the labels.

**Constants of base type:** A path entering a constant is answering a unique question. We must examine the question stack to restore the context, update the context with the answer, and branch to the place that the question was asked. Hence we have the following code, which we will write in an informal notation:

```
l:  let (M,E,l') = top qs
    let qs = pop qs
    let context = (M,E,c)
    br l'
```

**Constant functions:** Depending on which direction we are arriving from, we are either asking a new question, or answering a previous question. Arriving on $l1$ is asking a new question, so we push the current context onto the question stack together with the return address, and continue to traverse the path. Arriving on $l2$ with a context $(M', E', c)$ is answering a question, so we update the token and the question stack and branch to place that the question was asked.

```
l1:  let qs = push (M,E,l2) qs
     br l2_out
```

```
l2:  let (M,E,l) = top qs
     let qs = pop qs
     let token = (M,E,f c)
     br l
```

**Conditional:** Arriving at the top of a conditional requires that we ask a question for the boolean argument:

```
l1:  let qs = push (M,E,l2) qs
     br l2_out
```

Arriving on each of $l2$, $l3$ or $l4$ is exactly the same as the unoptimised version.

This optimisation to the Geometry of Interaction Machine of course is not obtained for free; we have had to trade off space for time. Essentially, we are building in features from environment machines to avoid recomputation. The novelty of this is that the ideas arise very naturally from the underlying semantics.

## 6 Conclusions

In this paper we have considered a very novel idea of an implementation of a simple functional programming language based on the notion of path computations from the Geometry of Interaction. This generated a very compact coding of a simple functional language which has an incredibly simple (and small) runtime system. We propose that the basic machine could have applications where space is of this highest concern, and efficiency is not so important.

There are a substantial number of optimisations that can be incorporated into this framework, and this is our current direction of research. Preliminary investigations have resulted in simple program transformations that can reduce the length of the path computation; local (peephole) optimisations that can substantially reduce the size of the object code, and increase performance; and finally, what we feel to be the most interesting direction, there are clear ways to incorporate not only sharing, but partial sharing as used in the study of optimal reduction [Lev80] and its implementations [GAL92, AL91, AL93] as interaction nets [Laf90].

The extended system with the optimisations starts to look like a promising alternative to extant implementation techniques. The implementation comes directly from the underlying semantics, and all optimisations come from the underlying theory of $\Lambda^*_{pcf}$. Experience with a prototype implementation has left us with very positive feelings about this paradigm of compiler technology, and we hope that it may ultimately provide a new way of implementing ideas from optimal reduction.

The compilation technique is very simple, and the ideas seem to be general enough to extend to something like Standard ML (see [Mac94] for hints on how to extend these ideas to general languages). There is still a great deal of work that

we need to do to achieve this goal, but the way forward seems clear; we hope to be able to report on this development in the not too distant future. Only then will we really be in a position to talk about performance issues.

## References


[ADLR94] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the $\lambda$-calculus: Three years of communication without understanding. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 426–436. IEEE Computer Society Press, 1994.

[AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In Masami Hagiya and John C. Mitchel, editors, *Theoretical Aspects of Computer Software. International Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.

[AL91] Andrea Asperti and Cosimo Laneve. Interaction systems I: The theory of optimal reductions. Technical Report 1748, INRIA Rocquencourt, 1991.

[AL93] Andrea Asperti and Cosimo Laneve. Interaction systems II: The practice of optimal reductions. Technical Report UBLCS-93-12, Laboratory for Computer Science, University of Bologna, May 1993.

[Dan90] Vincent Danos. *Une Application de la Logique Linéaire à l'Étude des Processus de Normalisation (principalement du $\lambda$-calcul)*. PhD thesis, Université Paris VII, June 1990.

[DR93] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 296–306. IEEE Computer Society Press, 1993.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of ACM Symposium Principles of Programming Languages*, pages 15–26, January 1992.

[Gir87] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[Gir89a] Jean-Yves Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro et al., editor, *Logic Colloquium 88*. North Holland, 1989.

[Gir89b] Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and Andre Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.

[Laf90] Yves Lafont. Interaction nets. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 95–108. ACM, ACM Press, January 1990.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lev80] Jean-Jacques Levy. Optimal reductions in the lambda calculus. In J.P. Hindley and J.R Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[Mac94] Ian Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science Technology and Medicine, 1994.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.