The geometry of optimal lambda reduction

Georges Gonthier* Martín Abadi† Jean-Jacques Lévy*

Abstract

Lamping discovered an optimal graph-reduction implementation of the λ-calculus. Simultaneously, Girard invented the geometry of interaction, a mathematical foundation for operational semantics. In this paper, we connect and explain the geometry of interaction and Lamping's graphs. The geometry of interaction provides a suitable semantic basis for explaining and improving Lamping's system. On the other hand, graphs similar to Lamping's provide a concrete representation of the geometry of interaction. Together, they offer a new understanding of computation, as well as ideas for efficient and correct implementations.

Acknowledgements

We have enjoyed discussions with Pierre-Louis Curien, Jean-Yves Girard, Yves Lafont, and John Lamping. Gordon Plotkin made useful suggestions on the presentation. Many phrases and attitudes in this paper are borrowed from Girard; no criticism is implied.

1 Patter

This paper develops for the third time a semantics of computation free from the twin drawbacks of *reductionism* (which leads to static modelisation) and *subjectivism* (which leads to syntactical abuses, in other terms bureaucracy). Such a semantics was developed previously by Jean-Yves Girard [Gir89, Gira] and by John Lamping [Lam90]. Girard is a logician and Lamping is an autodidactic engineer. It is no surprise that they never read one another—although they were working on the same problem from different perspectives.

Girard proposed the program of the geometry of interaction. The geometry of interaction provides an abstract semantics for algorithms, based on the judicious use of C*-algebras. Girard's program has three parts. The first two parts are concerned with defining a model and showing that it is suitably rich; the success here has been clearly remarkable. The third part is concerned with the possibility of implementing the geometry of interaction with an ad hoc machine, and it has not yet been developed.

In this paper we pursue the implementation part of Girard's program. We implement the geometry of interaction with mere graph reduction; the graphs used are a variant of Lamping's, and they are interaction nets in the sense of Lafont [Laf90]. We feel that our incredibly concrete formalism sheds some light on the geometry of interaction. We undertake to explain the geometry of interaction without using any relativity theory, any quantum theory, or for that matter, any mathematics.

Lamping described a graph-reduction implementation of the λ-calculus. The implementation provides a new, fine analysis of computation in the λ-calculus, to the point of being optimal in the sense defined in [Lév80]. After trying to read Girard's papers on the geometry of interaction, Lamping's "An Algorithm for Optimal Lambda Calculus Reduction" sounds like "TV Digest." Nevertheless, it seems fair to say that Lamping's algorithm is rather complicated and obscure. (Recently, Kathail proposed another optimal algorithm [Kat90]; we consider it in the full paper.)

It is our thesis that the geometry of interaction gives the proper understanding of Lamping's system. This view leads to some considerable simplifications, to a semantic basis, and to principled tech-

*INRIA Rocquencourt.
†Digital Equipment Corporation, Systems Research Center.

0

niques for correctness proofs. It also helps us in generalizing from the λ-calculus to the proof nets of linear logic [Gir87]. (We have dealt only with multiplicative-exponential linear logic so far.)

We believe that research on optimal reductions is of some practical importance. Lamping's work suggests some useful techniques for partial sharing, which potentially apply to a wide range of systems, from compilers to theorem provers. We hope that our study will further the impact of these techniques and contribute some more.

The next section introduces some of the main themes of the paper, informally. Section 3 describes our graphs and the reduction rules that apply to them; section 4 then presents the implementation of the λ-calculus. The rest of the paper is devoted to discussing semantics and the correctness of these implementations (section 5), optimality (section 6), and directions for future work (section 7).

For the sake of simplicity, we concentrate on the pure λ-calculus, only hinting at the more general treatment of linear logic.

2 Overview

This section introduces some of the ideas of a graph representation for λ-terms and corresponding graph-reduction rules. The system described is based on Lamping's, with several improvements. Then we discuss a semantics in the geometry of interaction and the delicate problem of correctness.

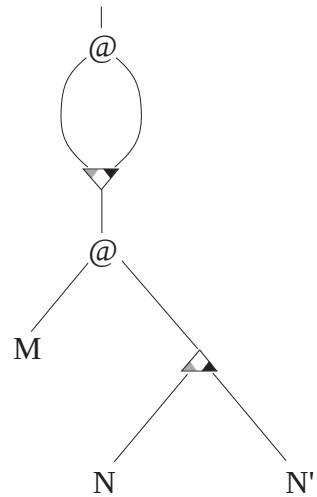
2.1 Combinators for sharing

The sharing of common subexpressions is an important optimization in the implementation of a variety of formal systems. It is typically associated with various graph representations and graph-reduction mechanisms. In a graph, sharing is represented by a fan-in.

Some time ago, an optimality criterion for λ-calculus reductions was defined. It was soon recognized that sharing of common subexpressions is not sufficient for optimality [Wad71, Lév80, Fie90]. Recently, a generalization of sharing was introduced that does support optimal reductions. The idea is to allow not only fan-in but also fan-out. Fan-in nodes and fan-out nodes are drawn

Fan-in nodes and fan-out nodes have symmetrical syntactic and semantic descriptions. As the graphs of interest to us are undirected, they are identical, formally; they are all called fan nodes.

Fan-out nodes allow partial sharing: terms can share a common subterm with a hole which may be filled in different ways in different versions. For example, the term $(MN)(MN')$ can be represented by the graph



Here $(M\ \_)$ is shared by the function and the argument of the top application; the hole is filled with N for the function and with N' for the argument. We write @ for application. (Application nodes appear in Lamping's graphs, but, as we shall see shortly, not in ours.)

The λ-term represented can be recovered by traversing the graph. For this read-back to be correct, it is essential to take matching branches of fan-in and fan-out nodes during the traversal. This is why two distinct marks, one grey and one black, label two of the ports of these nodes. The graph is traversed so that a path that goes through a grey mark at a fan-in node also goes through a grey mark at the corresponding fan-out node, and similarly for black marks.

To make precise this constraint on marks, Lamping introduced the notion of a context. A context records how fan-in nodes and fan-out nodes are traversed. In some simple cases a context can be represented by a stack; then going through a fan-in node pushes the mark traversed, and it is popped at the matching fan-out node, to select the port of exit.

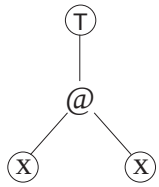
Fan nodes do not suffice in implementing partial sharing: one must guarantee that fan-in nodes and fan-out nodes are matched properly. For this purpose, Lamping introduced three bracketing constructs, of which we keep two (called bracket and croissant). It follows that contexts become more complex and structured; fortunately, it will turn out that very sim-

ple trees suffice for representing contexts in our system.

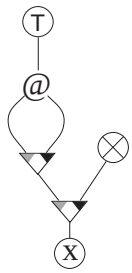
In addition, there are two kinds of unary nodes, root and void.

Informally, a root node terminates every important dangling edge; for a λ-term, these are the edges that correspond to free variables and the edge for the result (the value of the term). Although variable names are not a formal part of our graph representation, they sometimes appear in root nodes, for the sake of clarity. Void nodes are mere plugs. Thus, for example, the graph

represents x,



represents xx, and



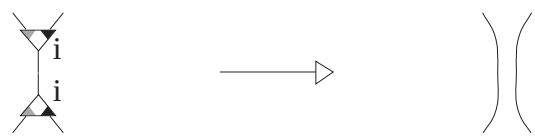
represents xx in a more contrived way.

Finally, one might expect nodes that correspond to abstraction and to application, as in the examples above. Certainly these nodes appear in Lamping's graphs. Perhaps surprisingly, these nodes are not needed. Fan nodes can be used instead of abstractions and applications. This is a simplification in the syntax of the graphs and in the corresponding reduction rules.

It is also a significant simplification in their semantics. All we have left are nodes that transform contexts. Therefore, the semantics of a graph can be given in terms of context transformations. This is directly in the spirit of the geometry of interaction. The geometry of interaction is concerned with operations on C*-algebras; our contexts provide a particular, concrete C*-algebra.

Some of the nodes we inherit from Lamping have indices, as in Lamping's system. Indices are natural numbers; intuitively, an index says at what depth in the context the node operates. In section 3.3 we describe an alternative to the use of indices; edges ("wires") are replaced with bundles of wires ("buses"), and then indices become superfluous, as one can explicitly draw on what wires in the bus an operation should act.

The evaluation of λ-expressions is based on graph reduction. The notion of graph reduction at play is a particularly benign one. Each sort of node has an interaction port. When two nodes are adjacent and their interaction ports are on the same edge, then a local transformation on the graph happens. For example, fan nodes interact through their unmarked ports; the interaction rule for two fan nodes with the same index is the expected one:



Lamping's rules were often of this form, and ours all are. This means that our graphs are interaction nets.

2.2 The geometry of interaction

There is no hope of explaining what is a C*-algebra within this restricted space. There is no need for such an explanation either.

From our perspective, the essence of the geometry of interaction is representing computing devices as context transformers. This representation appears in Lamping's work, of course, and we develop it further.

The use of C*-algebras comes from a justifiable desire to provide the most abstract possible formulation of the geometry of interaction. But we have no immediate need for C*-algebras, and hence omit dealing with these new bêtes noires of semantics.

2.3 On correctness

Computer science is a young discipline, but it already shows signs of senility. There is an obsession with correctness which often prevents the practitioners from understanding the finitary dynamics of computation.

In this paper, we yield to this obsession. Our implementations are all correct.

Several notions of correctness are possible. The simplest one is based on contexts. Graph reduction is sound in that it preserves the context semantics of graphs. It remains to see that the context semantics corresponds to the usual notions of the λ-calculus, and in particular that read-back yields appropriate λ-terms.

More precisely, one would like a read-back procedure R (a partial function) that maps graphs to λ-terms with two properties:

- if a graph G reduces to a graph G' then $R(G)$ reduces to $R(G')$;
- if a graph G' is in normal form then $R(G')$ is in normal form as well.

The geometry of interaction provides an easy proof that the graph reductions are correct in the case in which G' is an "observable," for example if it represents a boolean. Girard has obtained a result of this sort, and his proof applies in our setting. He has argued that this theorem should be considered satisfactory, explaining the reason that it might be the best possible:

> In fact from a purely syntactical viewpoint, the execution makes "mistakes", but it is precisely because of these "mistakes" that we can free ourselves from the need of a universal time!

Thus, it would almost seem that one cannot expect correct, parallel, higher-order computations.

Unfortunately, correctness for observables does not suffice. First, it is very dissatisfying. It is also insufficient for optimality arguments, as these require consideration of intermediate results, and not just of certain normal forms.

The solution is found by pursuing Girard's ideology of communication without understanding. Communication without understanding is communication where certain parts of messages are treated only in a generic way, and any "isomorphic" transmission would do just as well, without confusion. Typing guarantees the possibility of communication without understanding. Roughly, the type of a port induces a class of isomorphisms that can be applied to contexts communicated on this port without confusion.

Communication without understanding also implies that generic parts of messages cannot be created spontaneously. They must be copies of parts of previously received messages. Given a graph, a context is called accessible if either it contains no generic parts or it can be obtained from an accessible context by applying the context transformation that the graph defines and "cut-and-paste" of generic parts.

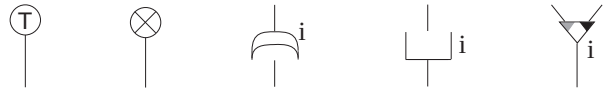
The graphs that one obtains by encoding the λ-calculus, or for that matter linear logic, admit a typing. (For the pure λ-calculus, the typing requires recursive types, but this hardly matters.) The type system for these formalisms then determines accessible domains and a suitable class of isomorphisms; β-reduction (or cut-elimination) is sound with respect to the accessible part of the semantics, up to these isomorphisms.

3 The Ad Hoc Machine

In this section we define two graph-reduction formalisms. The first one is a simplification of Lamping's, and relies on indices. The second one seems more primitive; the use of buses frees us from certain unnecessary, bureaucrating reductions.

3.1 Nodes

We start by considering undirected graphs built à la Lamping. We need only some of his nodes. The nodes are those presented in the introduction:



root void croissant bracket fan

3.2 Reduction

The rules for reduction are particularly simple, since we want not to be ridiculous. (To our knowledge, Girard was the first to write on the desire not to be ridiculous [Girb]; we continue his work in this respect too.)

In the rules, given in Figure 1, it is assumed that $0 \leq i < j$.

Because of the form of these rules, we are dealing with interaction nets. An immediate consequence is that the system is Church-Rosser, as there are no critical pairs. Interaction nets have trivial parallel implementations.

Some pairs of operators do not have a rule for interacting, for example brackets and fans with the same index. When these operators meet face to face, we have a deadlock. The context semantics of operators will make clear that a deadlock arises exactly when

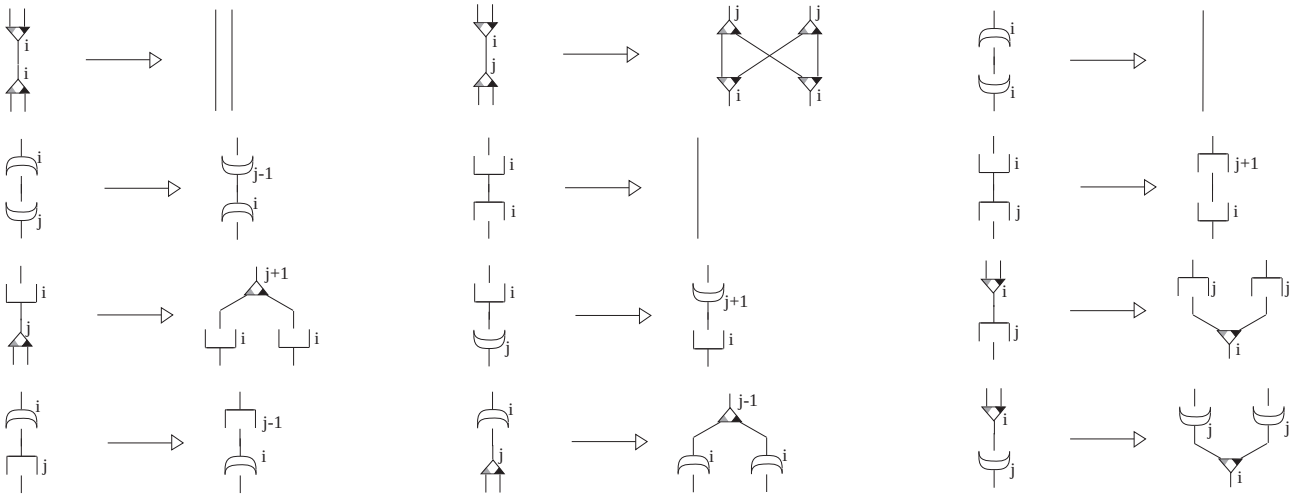


Figure 1: Reduction rules with indices

incompatible constraints are put on the context in a wire.

Garbage-collection rules are natural, and seem appealing for the sake of efficiency. Their function is to eliminate useless parts of graphs. For example, we may add:

Lamping has included a number of similar rules in his system. The rules are not essential for correctness or for optimality. They are not complete, in that they do not collect all garbage. In particular, they fail to collect certain kinds of cyclic garbage. For this reason we prefer not to include garbage-collection rules in our basic system.

In order to achieve optimality, a strategy needs to be followed. Simple strategies (e.g., leftmost-outermost) will do. We come back to this point below.

3.3 Decomposing the operators

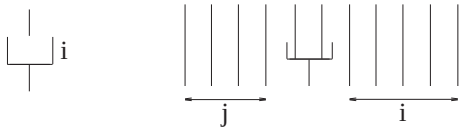
There are some hints that the operators presented above are not as primitive as possible. Consider, for example, a row of brackets and croissants. Bringing this row to normal form may take quadratic time, because of uninteresting commutations. It is this sort of bureaucratic overhead that we should try to avoid.

In this subsection we describe a more explicit graph formalism. This system decomposes the operators into more primitive ones, eliminating the need for indices. The main idea is that we are going to view a wire in the old system as a bus, a bundle of wires running in parallel; indices are not necessary, because operators can act on only a part of the bus.

The translation is given below. (The translation is presented only informally; a precise definition would require, in particular, the notions of "left" and "right" in buses, which are introduced easily for roots and then extended by contiguity.)

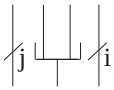
Root nodes and void nodes are as usual, but their arity increases. (Alternatively, we could simply have unary nodes and a metalinguistic way to group them.)

Brackets become ternary nodes. Their role is to combine two wires:



old bracket new bracket

Using hardware notation, this new bracket can be represented as:



Croissants become unary nodes. Their role is to creat a wire ex-nihilo:

old croissant new croissant

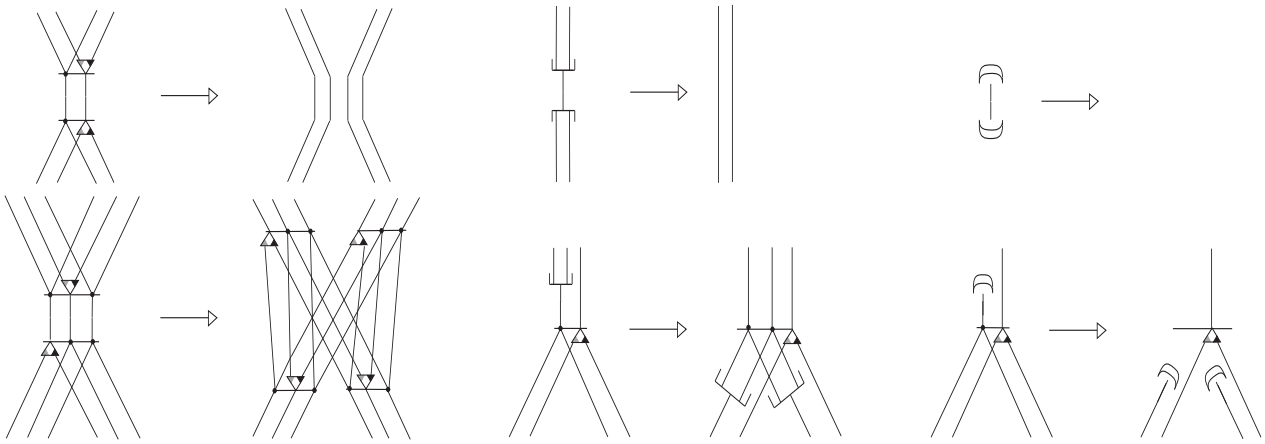
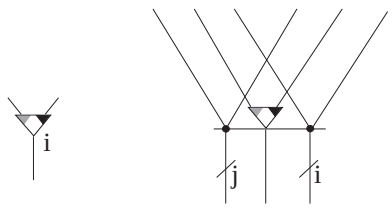


Figure 2: Reduction rules with buses

The most complicated change is for fan nodes. For a bus of width n ($n = i + j + 1$), we are going to use a fan node of arity $3n$.



old fan new fan

As in this picture, one of the levels is marked. This is the level at the depth given by the index. We call this the main level of the node. It is at the main level that a context mark is added. The other levels do not modify contexts.

We have considered going further, decomposing fan nodes into slices, one for each level. Each slice of a fan node would propagate on its own, but still depend on information that should come from the slice for the main level. This is a promising direction; we postpone further discussion to the full paper.

3.4 More on reduction

With these more primitive operators, the rules of interaction become clearer. They appear in Figure 2, where we draw only special cases, from which the general cases can be deduced by varying the width of buses.

With the old rules, a bracket and a croissant with different indices could meet and interact. With the new rules, this is no longer the case, as they simply cross on different wires on a bus. The commutation is free.

We tend to use the two systems of nodes interchangeably. It is generally more enlightening to use the system with buses. We use the other system only for compactness of notation and in order to relate our work to Lamping's and Girard's.

4 Implementations

The graphs described above can be used to encode a variety of computing formalisms. Here we just demonstrate this for the pure λ-calculus. We postpone correctness and optimality considerations to later sections, and only hint at the systematic treatment of linear logic.

4.1 Implementing the λ-calculus

The translation of the λ-calculus into graphs has two stages. In the first stage, we define incomplete graphs, where not all edges are terminated by nodes. Edges are directed, and those that correspond to variables labelled with variable names. (We will not draw directions explicitly, but will put result edges upwards and free-variable edges downwards.) The second stage simply closes the graph, adding root nodes as appropriate. Also, the direction of edges and the variable names on the edges are removed; variable names are put on roots.

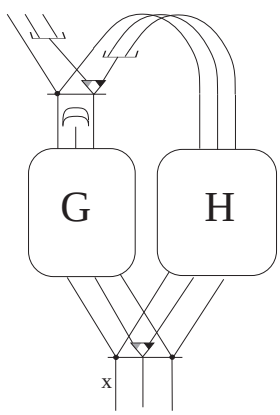
The first stage of translation is defined inductively. A variable is represented with a bus of width 3:

Intuitively, this bus carries commands between the occurrence and the value of the variable. In the λ-calculus, these commands can be construed as "call a function," "return to a continuation," "access an argument," and "report an argument value." Commands can contain subcommands, allowing for arbitrary complex meanings.

Each command pertains to a function call. Because of sharing, commands generated at a single location in the graph can refer to several actual calls, so each must carry an "address" identifying the relevant call. This address splits into a base address, shared by all commands in an instance of a lexical scope, and an offset. Subcommands need only an offset relative to their base commands.

Thus, commands are composed of three parts, which are carried by the three wires of the bus drawn above. The left wire carries the base address, the middle one the offset, and the right one the actual command, piled on top of its subcommands and their offsets.

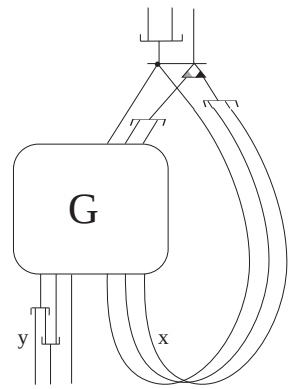
If G and H represent M and N, respectively, then



represents GH. Here we have drawn only the case where M and N have the variable x in common; fan-ins are used to combine all references to common variables.

Intuitively, a command coming from the top is nested as a subcommand of a "call" command and sent to G: the top fan piles a grey mark (the "call" token) on the pair $\langle \text{offset}, \text{command} \rangle$ created by the top bracket. The croissant generates a null offset for the call. When the argument H reports back a command to G, the fan packs it under a "report" command (black mark). Dually, the fan directs to H a subcommand packed under an "access" command (black mark) from G, and directs to the top edge a subcommand packed under a "return" command (grey mark). Finally, commands directed to the common variable x are sent out on the same edge, but are given different offsets by the fan so that responses from x can be sorted out.

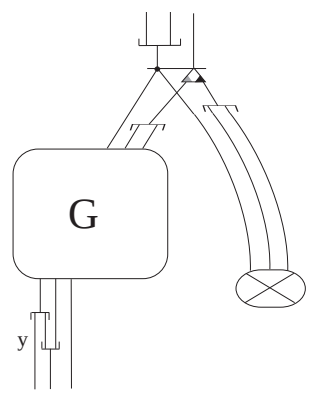
If G represents M and x occurs in M then



represents $\lambda x.M$. We have drawn only the case where $\lambda x.M$ has one free variable y; brackets are added to all edges for free variables.

The command handling for the abstraction fan is exactly dual to that for the application fan; only the address management differs. A lexical scope is opened for the λx by pushing the call offset onto the base address for the entire body G. This ensures that all commands in G will be relative to the call instance. Dually, commands generated by the fan simply pop the call offset off the base address when they cross the top bracket in the reverse direction. Upon leaving the scope through a free variable y, the base address is also restored and the popped offset is added to the command offset so no information is lost and responses from y reach their intended recipient.

If G represents M and x does not occur in M then



represents $\lambda x.M$.

Since the argument is never accessed, the black branch of the fan is effectively dead, and we can simply terminate it by a plug.

Note that there is no attempt to recognize common subexpressions in the λ-term given for reduction. Note also that this representation uses fan-in nodes,

but not fan-out nodes. Sharing appears in the course of reduction.

4.2 Implementing linear-logic proofs

The proof nets of linear logic can also be translated into our graphs, at least for the fragment of linear logic with multiplicative and exponential connectives. No new nodes are needed. The main contribution of this translation is the elimination of the fastidious boxes of linear logic. We break these boxes in our implementation, allowing their partial sharing.

Fan nodes represent the multiplicative connectives and contractions; croissants represent derelictions; void nodes represent weakenings; the edge of boxes is indicated by brackets, which can propagate independently. Our implementation of the λ-calculus puts a box around each abstraction: this box is implemented by the brackets above fan nodes and on free variable edges.

4.3 Types

As discussed earlier, types can be attached to the ports of the graphs obtained by translation from the λ-calculus or linear-logic proof nets. For the pure λ-calculus, this is simple, as it consists in distinguishing free variables from results.

This is not to say that types can be attached to all edges in the course of reduction. In this sense, the various logics that serve as type systems for the λ-calculus and for proof nets are not suitable logics for our graphs. The problem of finding such a logic remains open.

5 Semantics

In this section we discuss the semantics of our graphs. First, we describe the graphs as context transformers. This semantics is of help in understanding the rules of reduction, but it does not suffice in proving the soundness of our implementation of the λ-calculus, for example. A more sophisticated semantics appears later.

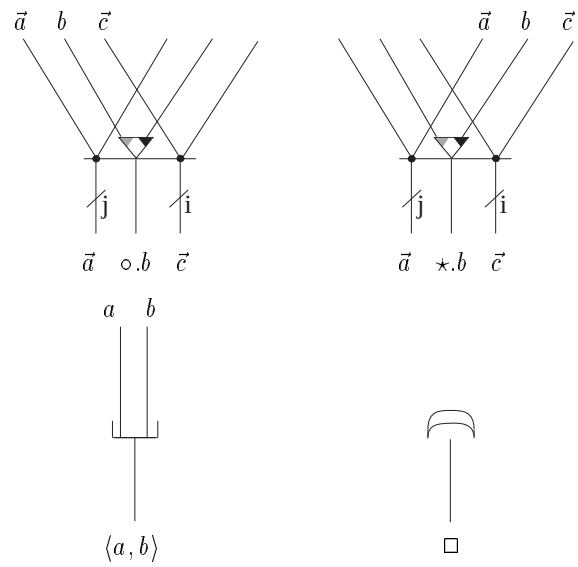
5.1 Rudimentary correctness

The first semantics is based on contexts. Contexts are trees, defined by:

- $\square$ is a context. It represents a node with no descendants.
- If a is a context then so are $\circ.a$ and $\star.a$. These terms represent the trees obtained by taking a node and putting a under it, either to the left or to the right. (The notations $\circ$ and $\star$ come from Lamping's work, where they are used instead of our grey and black marks, respectively.)
- If a and b are contexts then so is $\langle a, b\rangle$. This term represents the cons of a and b.

We denote $a.b$ the context a where the rightmost $\square$ is replaced by the context b. This is consistent with the $\circ.a$ and $\star.a$ notations above if we let $\circ$ denote the tree with only one left node ($\circ = \circ.\square$), and similarly for $\star$. The concatenation operator "." is clearly associative and $\square$ is its neutral element. Finally we abbreviate $\langle a, \square\rangle$ by $\langle a\rangle$, so we have $\langle a\rangle.b = \langle a, b\rangle$.

Let A be the set of contexts. The semantics of the nodes is given by relations on A. All variables represent contexts. Nothing (the absence of a context) is denoted by a blank space.



The semantics is extended to graphs, by composition. When the individual wires of a bus section of width k are labeled $a_1, \ldots, a_k$ we say that the bus section is labeled with the context $\langle a_1\rangle \ldots \langle a_{k-1}\rangle.a_k$. Assuming that the roots of a graph are numbered, from 1 to n, the context semantics $C(G)$ of a graph G is a relation $R_{i,j}$ between contexts for each pair of conclusions (i, j). The relation $R_{i,j}$ relates d to d' if there is a path from conclusion i to conclusion j with a consistent labelling and with d labelling the bus at conclusion i and d' the bus at conclusion j.

It is simple to prove that the reduction rules of section 3.4 preserve the context semantics. Good start! (This property fails for the system that uses indices instead of buses, for rather trivial reasons, not worth discussing here.)

Theorem 1 *If $G \rightarrowtriangle G'$ then $C(G) = C(G')$.*

One may remark that contexts of left, middle, and right wires are of rather different kinds, as already mentioned in section 4.1. The left wire is a stack of offsets, the middle wire is an offset, the right wire is a list of pairs $\langle \text{offset}, \text{command} \rangle$. Specifically, the left wire always contains a context

$$\langle \ldots \langle \langle a_1, a_2 \rangle, a_3 \rangle \ldots a_n \rangle$$

and the right wire a list

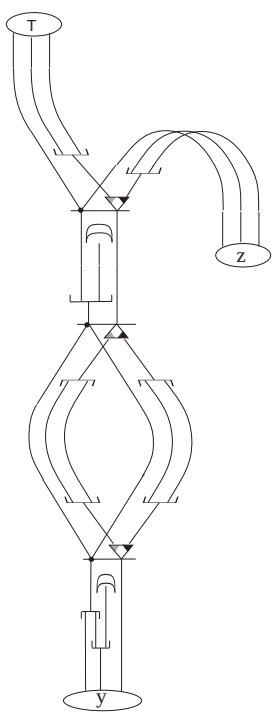
$$\otimes_1 . \langle a_2, \otimes_2 \rangle . \langle a_3, \otimes_3 \rangle \ldots \langle a_{n-1}, \otimes_{n-1} \rangle . a_n$$

where a_i are offsets (arbitrary contexts), $\otimes_i$ are commands $\circ$ or $\star$, and $n \geq 0$. When $n = 0$, the context is $\Box$. The middle wire contains always an a, the offset corresponding to $\otimes_1$.

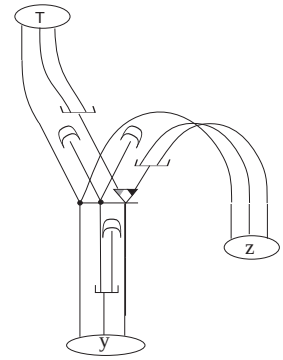
5.2 The read-back problem

Intuitively, we would expect to read back a λ-term by removing all sharing from the graph that represents it. In fact, this is how Lamping proves the correctness of his system. The proof is laborious, and somewhat ad hoc.

A proof based on the context semantics seems more appealing. Unfortunately, the context semantics does not match the usual semantics exactly. For example, the λ-term $(\lambda x.yx)z$ is translated into



and then this graph reduces to



When we try to recover a λ-term from this graph, we notice that some spurious brackets are in the way, right above the fan node that represents the application. The context semantics does not justify removing them.

5.3 Isomorphisms and accessibility, or communication without understanding

We would like to have a principled way to understand and to remedy the errors of graph reduction. This would lead to a correct interpretation of graphs, and connect the context semantics with the λ-calculus.

As discussed in the introduction, we propose to take a semantics coarser than the context semantics, by taking types into account. From the type of a root we can infer properties of the context semantics at that root, and especially of its effect on the right wire. In the case of our translation for the λ-calculus we have only two (recursive) types, result $D = !(D \multimap D)$ and variable D^-, from which we can infer the alternating structure of the right wire: each multiplicative corresponds to a fan mark ($\star$ or $\circ$), and each modality to a pair with an offset ($\langle a \rangle$). Moreover the offset for a ! modality can be chosen arbitrarily without changing the path.

More formally, say a context is *even* when it has an even number of $\star$ branches on its right spine, and *odd* otherwise (so $\Box$ is even, $\langle a \rangle . c$ and $\circ . c$ are even when c is, and $\star . c$ is even when c is odd). Say a context is *input* at the top root when it is even, and *output* when it is odd; dually, a context is input at a variable root when it is odd, and output when it is even. Finally, say that an offset a is input in the context $c.\langle a \rangle.d$ at root X when c is input at X, and dually for outputs. Then it can be shown that:

Proposition 1 *A consistent path σ between two roots X_1 and X_2 of the translation of a λ-term does not depend on input offsets; specifically, if there is a consistent labelling of σ with the bus at X_i labelled*

$\langle a_i \rangle.c_i$, then given any assignment of values for the input offsets in c_i there is a corresponding assignment of the output offsets that yields a consistent labelling for σ. This is also true when $X_1 = X_2$ if the assignment gives identical values to input offsets a' that occur after the same prefix p input at X_i ($c_i = p.\langle a' \rangle.c'_i$ for $i = 1, 2$).

An easy induction also shows that the left wire must be invariant ($a_1 = a_2$).

The environment should handle output offsets just as a term treats input offsets, that is, opaquely, as tokens. In particular, it should not be able to generate those offsets spontaneously to prod the internal structure of the graph. So not all contexts should be used at the start of a path. Specifically, noting that every consistent labelling of a root-to-root path has an input context at one end and an output context at the other, no prefix of an input context at a root should contain an output offset unless it is also a prefix of an output context at that root.

This can be formalized as follows: define $E = \{\Box\} \cup \langle A \rangle.E \cup \circ.E$, the set of *star-free* contexts. Now for $e, e' \in E$, define the operation of *shunting* e to e' by

$$c.\circ.e \xrightarrow[e']{e} c.\star.e'$$

Then an *access path* T in a λ-term graph G is a directed bus path labeled with contexts, starting at the top root with a star-free label, such that any two successive bus edges t_1 and t_2 of T are either consistent with the context semantics or are linked by a *shunt loop*: they respectively lead to and from a root of G, are labeled c_1, c_2 with $c_1 \xrightarrow[e']{e} c_2$ for some star-free contexts e, e'.

Thus, an access path can be pictured as

$$E \ni d_0 \xrightarrow{\sigma_0} d'_0 \xrightarrow[e'_1]{e_1} d_1 \xrightarrow{\sigma_1} d'_1 \xrightarrow[e'_2]{e_2} \ldots \xrightarrow{\sigma_n} d'_n \qquad (1)$$

where the σ_i are (simple) paths in G from root X_i to root X_{i+1} that can be consistently labelled with d_i at X_i and d'_i at X_{i+1}, with the obvious provisos for d'_n if σ_n does not end at a conclusion.

Note that all the d_i are input contexts, and all the d'_i are output contexts. Note also that any edge in an access path in the initial translation is traversed upwards iff the context on the rightmost wire is odd. This means that the edge direction which was forgotten in the last translation step can be recovered from the context semantics. Since the context semantics is preserved by graph reduction, the orientation can also be recovered in all residuals.

Now say a path is *accessible* iff it is the suffix of an access path. Special cases are: a labelled edge or a node is accessible iff it is part of an access path, a context c is accessible at a conclusion X iff a bus edge leading to or from X labelled c is accessible. The accessible semantics $C_a(M)$ of a term M is the subset of $C(G_M)$ generated by the accessible paths, where G_M is the graph representation of M. Furthermore, we show:

Proposition 2 *C_a is compositional: an access path in the graph $G_{\lambda x.M}$ representing $\lambda x.M$ only traverses its subgraph G_M representing M through accessible paths, and similarly for G_{MN} and G_M, G_N.*

This is proved by structural induction, in conjunction with another important property of access paths:

Proposition 3 *(access-path shunting) Given an access path as in (1), ending at a conclusion X_{n+1}, either $d'_n \in E$ or there is a unique i such that $X_i = X_{n+1}$ and $d_i \xrightarrow[e']{e} d'_n$ for some $e, e' \in E$.*

Now if X_{n+1} is the top root, then d'_n must be odd so the access path must end with a shunt. Since shunting cannot change the address on the left wire or the offset on the middle wire, this immediately shows that an access path cannot traverse an edge upwards unless it has already traversed it downwards with the same context on the left and middle wires. This is important for garbage collection and the read-back procedure.

Proposition 3 actually gives an algorithm for reading out the Böhm tree of the λ-term represented by a graph. For $e \in E$, let $|e|$ denote the number of $\circ$'s on the right spine of e. By Proposition 1 an access path does not depend on the choice of the offsets in d_0 or any of the e'_i, so the only real choice initially is $|d_0|$. Let n_0 be the minimal value of $|d_0|$ such that there is an access path starting with d_0 and leading to a root X_1. If there is no such d_0 then the term diverges, and the Böhm tree is Ω. Otherwise n_0 is the number of λ's in the head normal form of the term. Now by Proposition 3, either X_1 is the root for some variable x and $d'_0 \in E$, or $d_0 \xrightarrow[e']{e}$ for some $e, e' \in E$. In the first case x is the head variable, above which $m'_0 = |d'_0|$ applications are nested. In the second case, the head variable is a bound variable with de Bruijn index $n'_0 = |e|$, above which $m'_0 = |e'|$ applications are nested. To continue reading out the arguments for the applications, simply choose $|e_1| = m_1$ to select the argument of the $(m_1 + 1)$st application (counted from the outside in). In the general case, the de Bruijn index of a bound head variable is $n'_n + \sum_{j=i+1}^{n} n_j$.

Now this algorithm does not depend on the value of input offsets, and only depends on output offsets

through its use of Proposition 3. Thus it is insensitive to a remapping of offsets that leaves the shunt suffixes invariant, so we will read back the same tree for two terms whose accessible context semantics differ only by such an *isomorphism*. Now it is easy to see that contracting the head redex of a term leaves its semantics invariant up to such an isomorphism, and that the read-back of a redex-free tree indeed yields back the tree. Conversely, if the procedure computes the same tree for two terms, simply matching the steps defines the offset mapping.

Theorem 2 *The geometry of interaction semantics, restricted to accessible contexts and quotiented by isomorphisms, is equivalent to the Böhm tree semantics.*

It is tempting to consider a much simpler definition of isomorphism: a context mapping that preserves the right spine and commutes with the accessible semantics. With our translation it fails to yield the Böhm tree because the quotient semantics still conveys scoping and sharing information for the applications in the tree. Interestingly, the standard translation of the λ-calculus in linear logic, based on $D = (!D) \multimap D$, does not have this problem because it enforces less sharing.

We are now ready to consider directly the read-back problem. It is quite easy to read back a λ-term from its translation: we know that fans on the rightmost wires correspond to nodes in the syntax tree (application, abstraction, variable), and that edges in the graph are oriented as in the syntax tree, so we get the skeleton of the expression simply by traversing the graph downwards; furthermore when we do so we must have the same context on the left wire when we enter an abstraction and when we reach its bound variable, and nested λ's must have left contexts with different left spine length, so this property is enough to identify binders.

Now we have shown that edge orientation is defined solely from the context semantics. We show that we can use the same procedure to read back a λ-term from any residual of the initial graph. Now from Proposition 1 the consistency of a downward leftmost path in the graph does not depend on the contents of the rightmost wire, except perhaps for the number of $\circ$'s. Hence we have a very simple read-back procedure R: simply follow all downward paths that are consistent on all but the rightmost wire, and produce syntax nodes for each rightmost fan encountered, according to the orientation. A variable is bound by the λ with the same left context.

Theorem 3 *For any λ-term M, $R(G_M) = M$. Moreover, if $G_M \rightarrowtriangle_* G$, then $G \rightarrowtriangle_* G'$ implies $R(G) \rightarrowtriangle_* R(G')$, and G in normal form implies $R(G)$ in normal form.*

Proof sketch: Obviously reductions other than rightmost fan elimination don't change the read-back. For β redexes, note that the two downward paths to the variable and from the top of the application will be consistent iff they are part of the same access path in the application subterm, that is, iff they have the same left context, that is, iff the variable was bound by the redex λ.

6 On Optimality

Girard wrote that the third part of his geometry of interaction program should be concerned with the study of efficiency, as long as this remains a mathematical problem. The following discussion of optimality follows his suggestion; clearly much more can be done in this area.

6.1 Read-back and labelling

Because we have been able to solve the read-back problem, we are now in a position to consider optimality issues. It is straightforward to establish correspondences between a λ-term with a labelling [Lév80] and a suitably labelled graph that represents it. (The graph has labels on the rightmost wires.) We can then argue about how labels evolve through reduction.

The reduction rules are easily extended so that labels commute with all nodes except rightmost fans, and labels are concatenated when edges are, the direction of concatenation matching the edge direction. A label caught between two facing rightmost fans is copied (with overlining and underlining to match the λ-term labelling) when the redex is contracted; it is called the redex label.

Theorem 4 *During reduction, no two redexes of rightmost fans have the same label. Therefore, graph reductions are optimal.*

This means that the rules will never duplicate work, but leaves open the possibility of work on subterms that will turn out to be useless. Adopting a normal-order strategy suffices to avoid this possibility.

Proof sketch: Define the flat labelling of a path to be the concatenation of all labels traversed by the path with the following provisos: the order of symbols in a composite label on an edge that is traversed backwards is reversed, and overlining and underlining are removed, reversing the order of symbols under each removed underline. It is easy to see that any consistent path between rightmost fans has the same flattened labelling as any of its ancestors in a

graph reduction. Now suppose that during a graph reduction we have two redexes of rightmost fans with the same redex label. Consider the two paths consisting of the forward edge reduced in each redex. They have identical flat labelling, so they must have ancestors in the translation of the initial λ-term with that same flat labelling; but in this graph all paths between rightmost fans bear a different label, so the two ancestor paths must be equal. Now a path between facing fans has a unique residual until either end fan is deleted, so the two redex paths must be equal, hence the redexes must be equal.

6.2 Garbage-collection issues

The optimality criteria considered do not say anything about garbage collection, which we have deliberately neglected. In fact, garbage collection with local rules (as in Lamping's paper) destroys optimality, because it cannot collect the cyclic graphs the algorithm creates without first executing these graphs to unravel their structure.

However, Proposition 3 allows us to reclaim all nodes that are unreachable from the top root in the *oriented* graph, much as in Kathail's algorithm. This standard collection procedure is complete, in the sense that in a graph with no accessible redexes, nodes are reachable from the top root in the oriented graph exactly when there is an access path to them. Thus, special garbage-collection rules are not necessary.

Finer schemes could work by running the read-back algorithm "intelligently" so that a common subterm is not traced twice, and then collecting all untraced nodes. Obviously, determining the exact extent of garbage (nodes with no accessible path to them) requires execution, and is thus undecidable.

7 Extensions

This work leaves open a number of interesting questions. Foremost is the extension of this formalism to the whole of linear logic, including additives and quantifiers, and from there to classical logic [Gir91]. We should establish a clear relationship between the coherence semantics of linear logic [Gir87] and the geometry of interaction.

This should give us some guidance for the remaining two problems: finding a type system that abstracts the behavior on the left wire well enough to extend to arbitrary graphs (with "sharing" types), and decomposing fan nodes so that the synchronization they impose on the bus wires is relaxed.

References

[Fie90] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15. ACM, January 1990.

[Gira] Jean-Yves Girard. Geometry of interaction II: Deadlock-free algorithms.

[Girb] Jean-Yves Girard. Linear logic and parallelism.

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

[Gir89] Jean-Yves Girard. Geometry of interaction I: Interpretation of system F. In Ferro, Bonotto, Valentini, and Zanardo, editors, *Logic Colloquium '88*, pages 221–260. Elsevier Science Publishers B.V. (North Holland), 1989.

[Gir91] Jean-Yves Girard. A new constructive logic: Classical logic. Technical report, June 1991. INRIA Report 1443.

[Kat90] Vinod Kathail. *Optimal interpreters for lambda-calculus based functional languages.* PhD thesis, MIT, May 1990.

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

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM, January 1990.

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[Wad71] Christopher P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* PhD thesis, Oxford, 1971.