Flow Analysis in the Geometry of Interaction*

Thomas P. Jensen and Ian Mackie

Laboratoire d'Informatique (CNRS)
Ecole Polytechnique
91128 Palaiseau Cedex, France
{jensen,mackie}@lix.polytechnique.fr

Abstract. This paper describes a framework for flow analysis of programs with higher-order functions with normal-order reduction. The framework is based on an abstract machine derived from the Geometry of Interaction semantics for reduction in linear logic proof nets. By standard methods from abstract interpretation the transition system defined by the machine induces a set of equations defining the flow between the program points. This set of equations defines a collecting semantics for the program and is amenable to further analysis by abstraction-based approximation. As examples of its application we show how to obtain information about strictness, control-flow and usage of data.

1 Introduction

Higher-order functions form an important part of declarative programming but are in general difficult to implement efficiently. To overcome this difficulty, a number of analyses and optimisations have been proposed. Some of these optimisations are based on information about *what* a function computes *i.e.*, on its input-output behaviour. Others need to know *how* the computation is done. Analyses for finding the former kind of *extensional* properties can be argued correct with respect to a standard denotational semantics for the language; an example is the correctness proofs for strictness analysis. Analyses for the latter kind of *intensional* properties must be argued correct with respect to an operational model and, although much effort has been invested, no "canonical" framework similar to denotational semantics has appeared. This paper proposes a framework for deriving analyses from a particular operational model of normal-order computation: the Geometry of Interaction.

The Geometry of Interaction [8] operates on a graph representation of programs. The semantics of a program is a *path* through its graph; this path is a trace of the normal-order evaluation of the program. It describes exactly how and in which order the nodes of the graph are traversed during evaluation. An *evaluator* is then any mechanism that traces out the path in the graph. We present such a mechanism (the Geometry of Interaction machine [10]) in the guise of a transition system over the graph. The upshot of this is that the transition system can be analysed by standard methods from abstract interpretation [5]. In this

* To be presented at European Symposium on Programming 1996

way we obtain a framework for analysing a fine-grained operational semantics for normal-order evaluation of higher-order programs.

The paper is organised as follows. Section 2 reviews the graph representation of programs and the path algebra defining the Geometry of Interaction. Section 3 defines the Geometry of Interaction machine: a transition system semantics which will be the semantics underlying the analyses to follow. In Section 4 the transition system semantics is lifted to a transition system on sets of states, defining the collecting semantics of a program. From the collecting semantics a number of different kinds of information can be extracted; we consider strictness, closure and usage analysis in Section 5. In Section 6 we conclude our ideas and suggest further work.

Acknowledgements The work of Ian Mackie was funded by a British Royal Society Research Fellowship. The authors would like to express their gratitude to the anonymous referees, to Patrick Cousot and to Carolyn Talcott for suggesting valuable improvements to this paper.

2 A functional language

We begin by stating our language of study, for which we take a simple functional language akin to PCF [13]. The syntax is given by the following grammar:

$$\begin{array}{lcl} exp & ::= & variable \mid integer \mid \mathtt{t} \mid \mathtt{f} \\ & \mid & \lambda x.exp \mid exp\ exp \\ & \mid & unop\ exp \mid \mathsf{if}\ exp\ \mathsf{then}\ exp\ \mathsf{else}\ exp \end{array}$$

$$unop ::= \mathbf{rec} \mid \mathbf{succ} \mid \mathbf{pred} \mid \mathbf{iszero}$$

We refer the reader to the literature on PCF to complete the description of the language, for example the rewrite rules, and here we just remark that we are assuming a call-by-name evaluation order for the reduction process.

2.1 Graphs

Recall that a labelled directed graph (di-graph) $G = (N, E, w)$ is specified by a collection of nodes (N); a collection of edges (E); and a labelling (weight) function $w : E \to L$, for some label set L.

It is well-known how to represent the λ-calculus (and functional languages) as graphs [12]. Here we present a slight variant, in that:

- We avoid the use of variable nodes in the graph by simply connecting the occurrence of a variable in a term to its binding λ. (The reader should observe that this is nothing more than a graphical representation of de Bruijn notation for the λ-calculus.) For the purpose of this paper we only consider *programs* (closed terms), so that all variables will be bound to some λ.

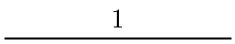
  There are two consequences of this "wiring" representation of the λ-calculus:

1. If a term contains multiple occurrences of a variable then we need a way of explicitly joining (merging) edges. For this we use a binary *fan* node in the graph that is sufficient to join multiple occurrences of a variable to a λ node. The other problematic case is when a variable does not occur in the body of a term being abstracted. For this we simply add a *plug* node to mark the end of an edge explicitly.
2. We require a way of representing the scope of a binding λ. Graphically, this corresponds to placing a box around each λ, and marking the point where an edge exits from this box. This is the purpose of the ?-node. (Again, it is fruitful to think of de Bruijn λ-calculus.)

– We use labelled graphs because we want to talk about paths in these graphs. The labels that we use will become an *algebra* for paths which allows us to select particular ways of navigating through a program. To be precise, the algebra will allow us to select only those paths in a graph representation of a program that *survive reduction*. That is to say that we can navigate through a program as though it was in its normal form, since only the paths that are in the normal form can be picked out in the original term.

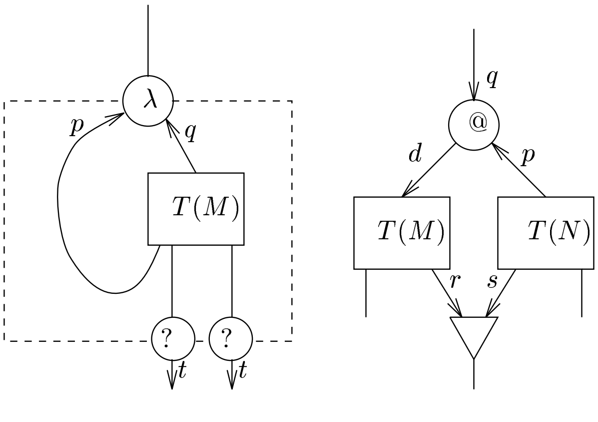
To make the above concepts explicit, we give the translation of our language into labelled di-graphs, introducing the nodes and labels on demand.

Variables The variable x is simply translated into a connecting edge in the graph. We label this with 1, which will become the identity for the algebra. Paths are invariant over variable edges, that is to say that a path is not changed over these edges. We draw these simply as:



Note that for this edge we did not put a direction. We will see below that the algebra of labels makes both directions equal.

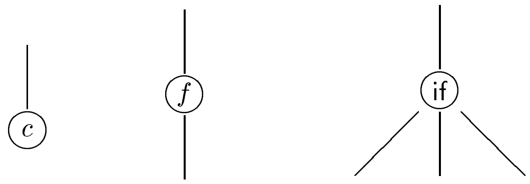
Abstraction and application An abstraction $\lambda x.M$ and an application MN are translated in a very similar way to standard graph representations of the λ-calculus. The only differences being the points mentioned above.



For the abstraction, to emphasise the notion of scope we have drawn a dashed box (which is not part of the graph, but there to help the reader). The box is actually represented by the λ node at the top, and the ? nodes (labelled with t) at the free variables of the abstraction. We use the label t to identify the free variables of the abstraction, p is used for the variable edge, and q for edge leading to the body of the abstraction. If the variable x doesn't occur in M, then we label the variable edge with 0.

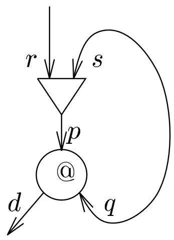
For the application, we can now see more clearly how we merge multiple occurrences of a variable with the *fan* node. We label this fan node with r for the left and s for the right edge. The application is labelled with q for the top of the application, p for the argument, and d for the function.

Constants, unary operation and conditional The constants of the language $c \in \{n, \mathtt{t}, \mathtt{f}\}$ are simply terminal nodes in the graph. The unary functions $f \in \{\mathbf{succ}, \mathbf{pred}, \mathbf{iszero}\}$ are binary nodes — one edge for the result and one edge for the argument. The conditional is a node with 4 edges for the result, boolean test and true and false branches. We draw the above as follows:



The edges of these operations are labelled with the identity 1. However, as we will see below when we give the algebra of paths, these constants of the language cause, and work by, *side effects*. In particular, for the conditional we use the value of the boolean test to dictate which branch to follow.

Recursion We code recursion without the need to introduce any new nodes by generating a cyclic structure that explicitly "ties the knot". The following diagram shows the graph representation of the **rec** combinator.



Note that there is a *path* rather than an edge making the cycle. The reader should also observe that this corresponds exactly to the coding of recursion in graph reduction, see [12] for example.

This completes the translation of our language into labelled di-graphs, so we are now in a position to talk about *paths* in such a graph. The translation shows all the edges (paths of length 1). The following gives us a way of generating all the paths in a graph. First, if $l \in \{p, q, r, s, t, d\}$ is a directed edge in the graph between two points, say u and v, then l^* is an edge from v to u. Hence, if we traverse an edge in the opposite direction from the arrow, then we simply reverse the edge. Any edge, or reversed edge, is a path in the graph.

Concatenation If ϕ_1 from x to y and ϕ_2 from y to z are two paths, then the composition (concatenation) $\phi_2\phi_1$ is a path from x to z.

Reversing If ϕ is a path from x to z, then ϕ^* is a path from z to x.

Lifting $!(\phi)$ is a path that is contained within a box. We also use the notation $b^*\phi b$ later in the paper so that we can identify entering and leaving a box in a local way.

Side effects As we traverse a path in the graph, we can apply constant functions to some notion of state. For example, a path entering the constant `t` (which is a terminal node in the graph) will bounce back, but we store the value `t` in a state. This value in the state will be used later during the path traversal, for example to select the appropriate branch of a conditional.

The above gives a general way of generating paths in our labelled di-graphs. The next step is to introduce an algebra on the labels that will pick out certain paths that satisfy an interesting condition: if term t reduces to t', then we want to look at the relationship between the corresponding graphs G and G'. If there is a path between two nodes, say x and y in G, then if there is still a path between x and y in G' then we say that the path is *non-null*; intuitively, it has *survived the action of reduction*. The Geometry of Interaction interpretation gives us an algebra Λ^* which is a mechanism for identifying non-null paths in the graph G. That is to say that we can pick out a path in a graph representation of a term that will survive *all the way* through to the graph representation of the normal form of t. Hence if we can compute these paths, then we have an alternative evaluation mechanism for functional programs.

We now present the algebra that will do this, then go on to show how to compute paths in this algebra.

2.2 Algebra

Here we give the equations for the algebra of paths Λ^*.

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

There are six constants to consider: p and q (for application and abstraction), r and s (for the fan), d (for the application), and finally t for the free variables of a box.

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

We remark that $pp^* = 0$ does not necessarily hold. Next a set of equations that shows how the constants behave with respect to the ! lifting.

$$!(x)r = r!(x) \quad !(x)s = s!(x) \quad !(x)t = t!!(x) \quad !(x)d = dx$$

Finally, the equations for the constants of the language, which give us an algebra of the side effects mentioned previously.

$$\begin{array}{lll} \mathtt{t}^*\mathtt{t} = \mathtt{f}^*\mathtt{f} = 1 & \textbf{iszero}\ \overline{0} = \mathtt{t} & \textbf{pred}\ \overline{0} = \overline{0} \\ \mathtt{f}^*\mathtt{t} = \mathtt{t}^*\mathtt{f} = 0 & \textbf{iszero}\ \overline{n+1} = \mathtt{f} & \textbf{pred}\ \overline{n+1} = \overline{n} \\ & \textbf{succ}\ \overline{n} = \overline{n+1} & \end{array}$$

Note that there are no equations for the conditional, since it is coded by the equations $\mathtt{t}^*\mathtt{t} = \mathtt{f}^*\mathtt{f} = 1$ and $\mathtt{f}^*\mathtt{t} = \mathtt{t}^*\mathtt{f} = 0$. We are now in a position to define precisely the notion of a non-null path in a program.

Definition 1. A non-null path in a graph is any path in the graph ϕ such that $\Lambda^* \vdash \phi \neq 0$.

3 An Abstract Machine

Having a specification of what the paths are is one thing, but having an effective way of computing them is another. Here we shall see that this is easily achieved by looking at a model of the algebra. There are many possibilities for a model, but there is an important result (see [1]) that states that *any* non-trivial model is "good enough". Here we will present one of the simplest models [10], based on a state consisting of three stacks: Multiplicative, Exponential and Data.

$$\begin{array}{ll} \mathit{State} & = \mathcal{M} \times \mathcal{E} \times \mathcal{D} \\ \mathcal{M} & = p : \mathcal{M} \mid q : \mathcal{M} \mid \Box \\ \mathcal{E} & = r : \mathcal{E} \mid s : \mathcal{E} \mid \mathcal{E} : \mathcal{E} \mid \Box \\ \mathcal{D} & = n \mid \mathtt{t} \mid \mathtt{f} \mid \Box \end{array}$$

Where : is a cons operation and $\Box$ is an empty element of the state. We remark that it is possible to model the algebra with a single stack, but the present one splits three components of the stack into separate stacks which will make the analysis clearer later in the paper.

The labels that we have defined now become state transformers. 1 is the identity, and 0 is the no-where defined state transformer. Composition in the algebra becomes composition in the model, ϕ^* is the partial inverse function ϕ^{-1} on State, and $!(f)(m, c_1 : c_2, v) = (m', c_1 : c_2', d)$ where $(m', c_2', d) = f(m, c_2, v)$.

$$\begin{array}{ll} p(m,c,v) = (p : m, c, v) & q(m,c,v) \qquad\quad\;\; = (q : m, c, v) \\ d(m,c,v) = (m, \Box : c, v) & t(m, x : y : c, v) = (m, (x : y) : c, v) \\ r(m,c,v) = (m, r : c, v) & s(m,c,v) \qquad\quad\;\; = (m, s : c, v) \end{array}$$

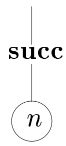
The following operations show how we model the *side effects* of the constants and constant functions. That is, when we pass constant nodes as described previously, we side effect the data component of the state as follows:

$$
\begin{array}{llll}
n(m,c,\Box) & = (m,c,n) & \mathbf{pred}(m,c,0) & = (m,c,0) \\
\mathtt{t}(m,c,\Box) & = (m,c,\mathtt{t}) & \mathbf{pred}(m,c,n+1) & = (m,c,n) \\
\mathtt{f}(m,c,\Box) & = (m,c,\mathtt{f}) & \mathbf{iszero}(m,c,0) & = (m,c,\mathtt{t}) \\
\mathbf{succ}(m,c,v) & = (m,c,v+1) & \mathbf{iszero}(m,c,n+1) & = (m,c,\mathtt{f})
\end{array}
$$

The above model gives a model of the algebra in the sense that $\phi \neq 0$ in the algebra iff $\phi(\mathcal{M}, \mathcal{E}, \mathcal{D})$ is defined for some $(\mathcal{M}, \mathcal{E}, \mathcal{D})$. It is now possible to compute the result of a program with this abstract machine. We start at the root of the graph with the empty State, and follow the path. The key result that connects path computation with reduction is given by:

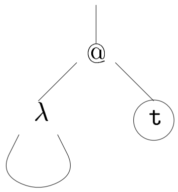
Proposition 2. *For a program of base type, there is a unique path starting from the root of the graph. Moreover, if the program has a normal form v, then the State at the end of the graph traversal gives:* $(\Box, \Box, v)$.

Examples We give two simple examples to enlighten the reader to how this semantic paradigm works. First, consider the term **succ** n which we draw as the following graph:



Since this example only requires the use of the Data part of the stack, we will consider the state as consisting of only this component. Starting at the root of the graph with the empty state gives a sequence of state transformers. Starting with $\Box$, we proceed along the path towards the **succ** node. Since the label of the graph is 1 there is no change to the state. We thus continue towards the constant n, at which point we side effect and change the state to n, and return back along the path. At the **succ** node we again side effect the state yielding the state $n+1$. We now arrive back at the root of the term with the answer to the computation, *without* rewriting the graph.

Next, consider the term $(\lambda x.x)\mathtt{t}$. We draw this as the following graph:



This example follows the same scenario as the previous one, except that we now need to use all the state. We begin at the root with $(\Box, \Box, \Box)$ and proceed towards the application node. The edge is labelled with q, so we transform the token to

$(q : \Box, \Box, \Box)$. Now we have a choice: either towards the function, or towards the argument. However, the operation p^* does not apply (since $p^* q = 0$), so we are forced towards the function with the new state $(q : \Box, \Box : \Box, \Box)$. We now enter the box with state $(q : \Box, \Box, \Box)$ and continue towards the λ node. Again we are faced with a choice, and again, the history in the state tells us which way to go. This time we go along q^* into the body of the term with state $(\Box, \Box, \Box)$, and since this is just a variable, we immediately arrive back to the variable edge of the λ node. We now apply the p operation, exit the box and traverse the d^* edge: $(p : \Box, \Box, \Box)$. Into the argument, and then side effect when we arrive at the $\mathtt{t}$ node, yielding the state $(\Box, \Box, \mathtt{t})$. Follow the return path, back through the λ the way we came, and back to the root giving the final state $(\Box, \Box, \mathtt{t})$.

From now on we take the state to be just a pair $(\mathcal{M}, \mathcal{E})$ since the constants play no rôle in the analyses that follow. The definition of the labels as state transformers extends to a definition of the state changes that take place when traversing a node in a graph. For example, entering an application node from the edge labelled d (called the d-port of the application) with a state of the form $\langle p : m, e\rangle$ will cause an exit on the p-port with state $\langle m, d^*(e)\rangle$ The p on top of the stack prevents us from exiting via the q-edge since the operation q^* is not defined on State of form $\langle p : m, e\rangle$.

In the precise definition of the abstract machine as a transition system we have to distinguish between data flowing into a node and out of a node along a given edge. To each contact point between a node and an edge we associate two *ports*: an input port and an output port. These ports specify exactly where a data item is and in which direction it is moving in the graph. The transition through the application node described above can now be expressed formally as

$$(I_d, \langle p : m, e\rangle) \to (O_p, \langle m, d^*(e)\rangle)$$

where I_d and O_p are the input port of the d-edge and the output port of the p-edge, respectively. More formally, each node gives rise to a set of transition rules; one for each possible path through the node. A non-null path $\phi = x_n \ldots x_1$ from port l_1 to l_2 defines a transition relation

$$(I_{l_1}, s) \to (O_{l_2}, x_n(\ldots(x_1(s))\ldots))$$

The edges in a graph (N, E) transfer the states from port to port: an edge between ports l_1 and l_2 means that output from l_1 becomes input to l_2 and *vice versa*. In formal terms it means that an edge introduces two transition rules:

$$(O_{l_1}, s) \to (I_{l_2}, s) \text{ and } (\mathrm{O}_{\mathrm{l}_2}, \mathrm{s}) \to (\mathrm{I}_{\mathrm{l}_1}, \mathrm{s}).$$

We can now define a transition system semantics of a graph. Let $\mathcal{P}_{I/O}$ be the set of ports in graph G.

Definition 3. Graph $G = (N, E)$ induces a transition system $(\mathcal{P}_{I/O} \times \mathit{State}, \to)$ where the transition relation $\to$ is the union of the transition relations defined by each node in N and edge in E.

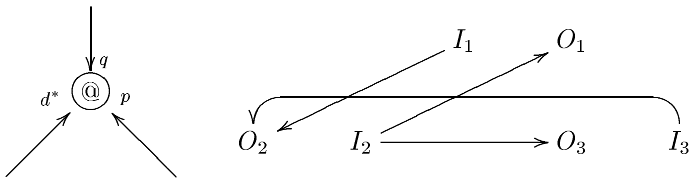
Finally we remark on an important result that relates the size of the multiplicative stack m and the type of the term.

Proposition 4. *The size of the multiplicative stack is bounded by the size of the types of the subterms (measured as depth of the tree representation of the type).*

From an analysis point of view it means that we do not have to abstract this component in order to obtain a decidable analysis. This is not the case with the context component which can grow infinitely large.

4 Collecting semantics

The purpose of this section is to explain how the techniques of abstract interpretation [4] can be applied to the semantics defined in the previous section. During evaluation we can reach the same point in the λ-graph several times with different states. The purpose of a *collecting* semantics is to assign to each point the set of states that can occur at that point. A node in the graph thus becomes a transformer on sets of states. Given a set of input states for one of the ports we define what are the possible sets of output states at the other ports. To obtain such an interpretation of the graph we re-interpret the labels as operating on sets of states rather than single states. The way nodes are linked in the graph determines the way sets of states flow form one node to another. We treat the application node in some detail here and list the equations arising from the other types of nodes. To each edge meeting a node we have associated two ports and the two sets I_i and O_i of input states and output states. Assuming a numbering of port going counter-clockwise beginning at the top port, the node and the input-output dependencies look as follows:



The labels can be interpreted as operations on sets of states by stipulating that

$$x(S) = \{x(\sigma) : \sigma \in S\} \qquad S \subseteq \mathit{States}$$

where x ranges over the set of labels. When traversing the node as prescribed by the flow diagram the states will be changed by the labels on the edges traversed. This change is described by a set of flow equations, one for each output port. For the application node this set is:

$$\begin{aligned} O_1 &= q^* d^*(I_2) \\ O_2 &= dq(I_1) \cup dp(I_3) \\ O_3 &= p^* d^*(I_2) \end{aligned}$$

Note how the fact that we can arrive at O_2 from both I_1 and I_3 translates to a union of sets of states. In general, we have:

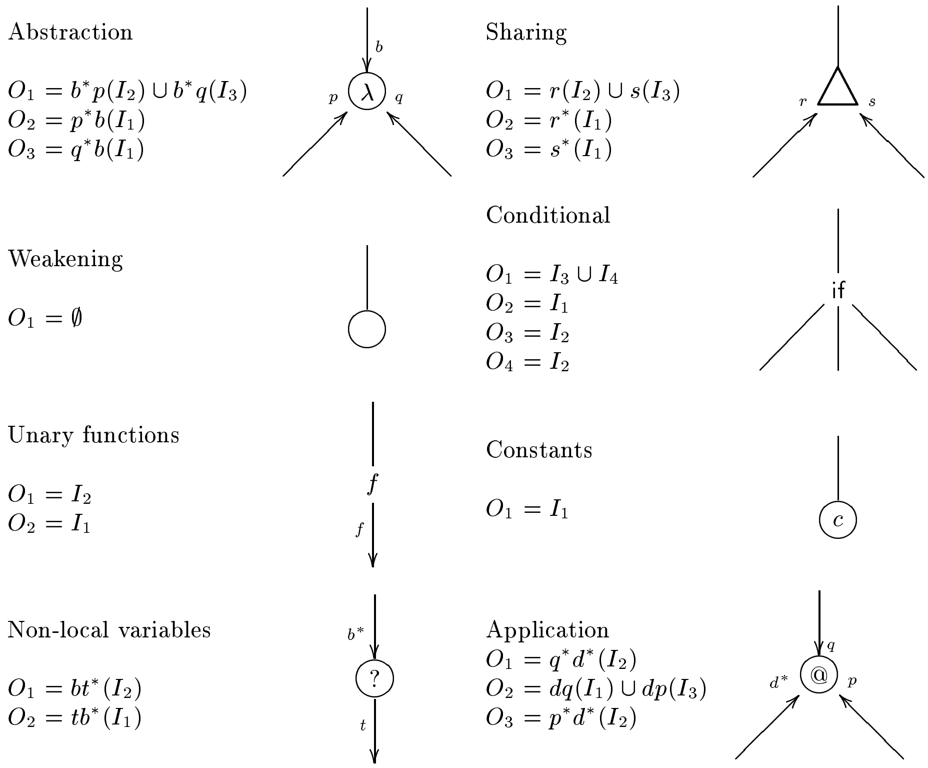


Fig. 1. Flow equations for nodes. Ports numbered counter-clockwise from top.

Definition 5. The flow equations $\mathcal{E}_n$ for a node n is the set

$$\{O_x = \phi_1(I_{y_1}) \cup \ldots \cup \phi_n(I_{y_n})\}$$

where $x, y_1, \ldots, y_n$ are ports of n and ϕ_i is a non-null path from y_i to x in n.

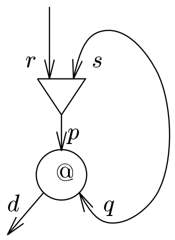
The flow equations for the other kind of nodes are given in Figure 1.

An edge between two nodes in a graph gives rise to two equations representing the flow of information from one node to the other. The principle is simply that output from one node becomes input to the other node.

Definition 6. Edge e between ports x and y gives rise to the equations

$$I_x = O_y$$
$$I_y = O_x$$

This set is denoted by $\mathcal{E}_e$.



$$
\begin{aligned}
O_1 &= r(I_2) \cup s(I_3)\\
O_2 &= r^*(I_1)\\
O_3 &= s^*(I_1)\\
O_4 &= q^* d^*(I_5)\\
O_5 &= dq(I_4) \cup dp(I_6)\\
O_6 &= p^* d^*(I_5)\\
I_1 &= O_4\\
I_4 &= O_1\\
I_6 &= O_3\\
I_3 &= O_6
\end{aligned}
$$

Fig. 2. Flow equations for recursion.

Together with the equations for each node this defines the complete set of equations representing the flow of information of the program. We note that since a node has at most four ports and that one port can be linked to at most one other port, the number of equations arising from a graph is linear in the number of nodes in the graph.

Definition 7. The flow equations $\mathcal{E}_G$ of a λ-graph $G = (N, E)$ is the set

$$\mathcal{E}_G = \bigcup_{n \in N} \mathcal{E}_n \cup \bigcup_{e \in E} \mathcal{E}_e.$$

For a λ-graph of a closed expression there will be exactly one port in the graph that is not linked to other ports; we call this the principal port and number it 0. This port is the input port for the program where the initial state of the computation is specified. A specific equation must define the possible set of input states, I_{init}.

Definition 8. The *collecting semantics* [5], $[\![G]\!]_{coll}$ of a λ-graph G is the set of descendants of the initial state at each port P:

$$[\![G]\!]_{coll}(P) = \{s \mid \exists s_0 \in I_{init} : (I_0, s_0) \to^* (P, s)\}$$

The following theorem gives a least fixed point characterisation of the collecting semantics. Its proof follows from [3].

Theorem 9. *The collecting semantics $[\![G]\!]_{coll}$ is the smallest solution to the equation set $\mathcal{E}_G \cup \{I_0 = I_{init}\}$ in the complete lattice $(\mathcal{P}(States), \subseteq)$.*

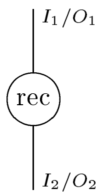
4.1 Example: equations for recursion

Recall that a recursion node was obtained by linking a sharing node and an application node by combining the unlabelled port of the sharing node with the q-labelled port of the application node. The resulting graph has two unlinked ports, the other four being linked together. Figure 2 shows the graph with its set of flow equations where the latter four equations arise from the way the two nodes are linked together. We would like to reduce this to a set of equations only in the variables of the ports that are not linked together, *i.e.*, to a set of equations in the variables O_2, I_2, O_5, I_5. After a series of calculations using the path algebra from Section 3 we obtain the following two flow equation for the compound recursion node:

$$O_2 = r^*q^*d^*(I_5)$$
$$O_5 = dqr(I_2) \cup dqsp^*d^*(I_5) \cup dps^*q^*d^*(I_5)$$

Using these equations we can now treat recursion as an atomic node of the graph.

Definition 10. The recursion operator is translated into a two-port recursion node with flow equations as follows:



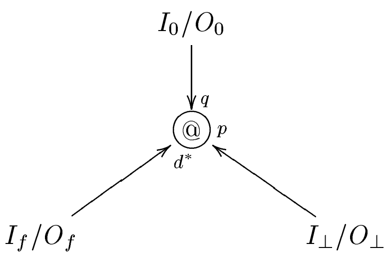
$$O_1 = r^*q^*d^*(I_2)$$
$$O_2 = dqr(I_1) \cup dqsp^*d^*(I_2) \cup dps^*q^*d^*(I_1)$$

5 Analyses

In this section we show how various kinds of information can be extracted from the collecting semantics. We consider strictness properties, the flow of functional data objects (closures) and the number of use of data.

5.1 Strictness

We recall that a function is called strict if the result of applying the function to an undefined argument is undefined. In our framework, a result of a computation is undefined if the set of output states of the graph is empty. The flow equation for an undefined argument, $\perp$ is therefore simply $O_\perp = \emptyset$. Denoting by O_f and I_f the principal port of the function we are analysing, we thus have to decide under which conditions the set of output state, O_0 for the graph



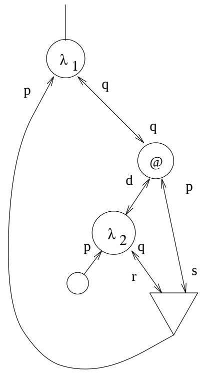


Fig. 3. $\lambda x.(\lambda y.x)x$

is empty. The graph has flow equations

$$\begin{cases} O_0 = q^* d^*(O_f) \\ I_f \ = dq(I_0) \cup dp(O_\perp) = dq(I_0) \\ \quad + \\ \text{equations for f} \end{cases}$$

From this we see that a *sufficient* condition for O_0 to be empty is that every state in O_f has a p on top of the stack. Furthermore, we are always certain to have a q on top of the stack in the set I_f. Writing $\langle x : M, E\rangle$ for the set of states where the $\{p, q\}$-stack has an x on top of the stack, the above observations justify the following strictness criterion:

Proposition 11. *A function represented by graph G with principal port f is strict if $I_f \subseteq \langle q : M, E\rangle \Rightarrow O_f \subseteq \langle p : M, E\rangle$.*

This strictness criterion allows to check for strictness when the argument to the higher-order function is of base type. For higher-order arguments we can represent the situation where the argument is completely undefined and prove properties based on this fact. However, in order to prove properties such as "$Twice(f) = \lambda x.f(f(x))$ is strict if f is strict" we need to replace f with a graph that represents all strict functions, *i.e.*, a graph with the input-output behaviour described by the strictness criterion from above, and then solve the equations with this as input. Thus the analysis does not function as the strictness analysis of Burn *et. al.* [2] which will tabulate the entire abstraction of a higher-order function. The information must be obtained by posing specific questions to the analyser.

5.2 Usage analysis

The collecting semantics contains information about the use of values during evaluation. The information of interest is whether the value of a given expression is never used or is used at most once. With this style of information it is possible to avoid storing data that is never used in the subsequent computation. We give a small example to show the kind of information we can find. In the λ-graph in Fig. 3 the function λ_2 does not use its argument. This means that the set of states entering on the p-branch of λ_2 is empty, therefore no states arriving at the d-branch of the application node will have a p on top of the stack hence the set of states travelling down the p-branch will be empty. This means that the r-branch will receive an empty set of states hence the states leaving the sharing nodes can only have an s on top of their $\{r, s\}$-stack. Thus the analysis has shown that the argument to the application node will not be used and that the argument to λ_1 will be used at most once. More generally we can identify functions that have multiple occurrences of bound variables, but only use a subset of them to access their arguments.

5.3 Closure analysis

Let $e_1@e_2$ stand for the application of the function obtained by evaluating e_1 to the argument e_2. Which function is obtained will usually depend on the run-time arguments to the program as in $\lambda v. \ldots (\lambda x.x + v)@e_2 \ldots$ and a given expression can evaluate to different functions if evaluated in different contexts as is the case if it is part of a functions that is called several times. A control-flow analysis (also called *closure analysis* by Sestoft [15, 16]) will determine an approximate description of the set of functions that an expression can evaluate to during execution of a program. We shall follow the approach taken in Sestoft's analysis and Shivers' 0-CFA analysis [17] and abstract a function to the λ-expression it was obtained from, *i.e.*, we ignore the environment of a closure.

We assume that the nodes in the graph G are given unique names. This naming is extended to the labels by subscripting all labels of an edge of the node named ℓ with the name ℓ. This means that for a state $\langle p_\ell : m, e\rangle$ we can tell which node pushed the p on the stack. We recall that when entering an application node labelled ℓ, a q_ℓ is pushed onto the stack. When the corresponding λ node is entered, this q_ℓ is removed from the stack. Let n be the !-port of a λ node. The set of application nodes that this λ can be passed to as a function is therefore included in the set

$$\{\ell : \langle q_\ell : m, e\rangle \in [\![G]\!]_{coll}(I_n)\}$$

i.e., the set of labels ℓ such that $\langle q_\ell : m, e\rangle$ is an input state at the principal port of the λ node. Dually, the set of λ nodes that can appear as functions at an application node with d-port n is approximated by the set

$$\{\ell : \langle q_\ell : m, e\rangle \in [\![G]\!]_{coll}(I_n)\}$$

of labels of qs pushed onto the stack when control returns from evaluation of the body of a λ-expression.

An important difference between this analysis and Sestoft's closure analysis is that this analysis takes the call-by-name evaluation order into account because it combines information about closures with the usage information described above.

If the analysis can detect of an expression that it is never evaluated (*i.e.*, the set of input states on the principal port of the corresponding graph is empty) then no bindings from this expression will appear in the final result. In contrast to this, Palsberg [11] has showed that closure analysis is valid under all evaluation strategies and must therefore take call-by-value evaluation into account. This would include the closures in the unused argument (Shivers' analysis is for a call-by-value language so this remark is irrelevant for his analysis).

6 Conclusion

We have presented a new framework to reason about properties of higher-order call-by-name functional programs coming from an *intensional* semantics: the Geometry of Interaction. The goal was to analyse properties that cannot be characterised by extensional means, and to treat different analyses in one framework. Choosing the Geometry of Interaction as semantic foundation carries perhaps an initial cost since it is not widely known. In our opinion this is out-weighed by the ease with which one passes by from semantics to collecting semantics and to a computable approximation. Our work builds upon [4, 5, 3], and continues and concretises a line of work suggested by Jones [9] who proposed to apply abstract interpretation to an environment-based evaluator for the λ-calculus. The operational semantics chosen here is in fact a refinement of Jones' evaluator; this refinement leads to simpler abstract operations and allows to extract the flow equations in a straightforward manner. A similar approach, but for a call-by-value evaluator, was considered by Deutsch [7] for aliasing and life-time analysis of higher-order functions.

Other approaches in the field of closure analysis include Shivers' [17] where he develops a denotational semantics for Scheme with enough operational content to obtain his control-flow analyses by abstraction. Although an impressive piece of semantics, it is probably too costly to develop the denotational semantics for every new analysis, compared to what is gained from the the generic principles for abstracting such a semantics. Sestoft [16] proves the correctness of his closure analysis with respect to Krivine's abstract machine. This is closer to our approach but he does not consider how the analysis could be derived from the machine. The "balanced traces" used by Sestoft to identify matching application and λ nodes seems to correspond closely to the notion of "well-balanced path" in the Geometry of Interaction. We have not yet established an exact relationship between these analyses and ours, nor can we state an exact comparison between our strictness analysis and the one cited.

The Cousot and Cousot framework [6] has broader scope than the present work since it deals with abstracting higher-order functions in general. As an application they define a comportment analysis that generalises classical strictness and termination analysis. Our approach circumvent the use of lattices of functions and relations by passing to the abstract machine level. Evidently this narrows the scope of application to normal order evaluation of higher-order functions but allows simple correctness proofs and implementation scheme.

The analysis framework so obtained is flexible in that several kinds of properties can be expressed with the collecting semantics and a range of approximation techniques for stacks, lists and multisets can and must be employed to abstract the states of the Geometry of Interaction machine. Given the abstractions, the implementation of the analyses is simple since it only involves solving a set of flow equations. The calculations themselves are straightforward but the results tend to be unwieldy; this together with space considerations have prevented us from including examples of analyses.

Further work also involves using the path semantics to argue the correctness of various evaluation-order analyses. These cannot be formalised by considering individual sets of states; traces are needed. It is also hoped that complexity issues such as those studied by Sands [14] can be dealt with here. The semantics should be sufficiently fine-grained to express a faithful complexity measure for higher-order programs.

References

1. A Asperti, V Danos, C Laneve, and L Regnier. Paths in the λ-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.
2. G Burn, C Hankin, and S Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.
3. P Cousot. Semantic foundations of program analysis. In N Jones and S Muchnick, editors, *Program flow analysis : theory and application*, pages 303–342. Prentice-Hall, 1981.
4. P Cousot and R Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.
5. P Cousot and R Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York.
6. P Cousot and R Cousot. Higher-order abstract interpretation (and application to comportment analysis generalising strictness, termination, projection and PER analysis of functional languages. In *Proceedings of 1994 International Conference on Computer Languages*, pages 95–112. IEEE Computer Society Press, 1994.
7. A Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proc. of 17. ACM Symposium on Principles of Programming Languages*, pages 157–168, San Francisco, 1990. ACM Press.
8. J.-Y Girard. Towards a Geometry of Interaction. In J. W Gray and A Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.
9. N Jones. Flow analysis of lambda expressions. In S Even and O Kariv, editors, *Proc. of 8th International Colloquium on Automata, Languages and Programming*, pages 114–128. Springer LNCS vol. 115, 1981.
10. I Mackie. The geometry of interaction machine. In *Proceedings, 22nd ACM Symposium on Principles of Programming Languages*, pages 198–208, 1995.
11. J Palsberg. Closure analysis in constraint form. *ACM Trans. on Programming Languages and Systems*, 17(1):47–62, 1995.

12. S. L Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.
13. G Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.
14. D Sands. A naive time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.
15. P Sestoft. Replacing function parameters by global variables. Master's thesis, Univ. of Copenhagen, 1988.
16. P Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, 1991. Available as DIKU Report 92/6.
17. O Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnegie-Mellon University, 1991.

This article was processed using the LaTeX macro package with LLNCS style