A fully abstract semantics for concurrent graph reduction: extended abstract

Alan Jeffrey

Abstract. This paper presents a formal model of the *concurrent graph reduction* implementation of non-strict functional programming. This model differs from other models in that:

- It represents concurrent rather than sequential graph reduction.
- It represents low-level considerations such as garbage collection.
- It uses techniques from concurrency theory to simplify the presentation.

There are three presentations of this model:

- An operational semantics based on graph reduction.
- A denotational semantics in the domain $D \simeq (D \to D)_\perp$.
- A program logic and proof system based on COPPO types.

We can then use ABRAMSKY and ONG's techniques from the *lazy λ-calculus* to show that the denotational semantics is *fully abstract* for the operational semantics. This proof requires some results about the operational semantics:

- Since the operational semantics includes garbage collection, reduction is not *confluent*. We find a confluent reduction strategy which has the same convergence properties as graph reduction.
- We use a *sequential* reduction strategy to show that concurrent and sequential graph reduction have the same convergence properties.
- We use *simulation* between nodes in a graph to show *referential transparency*.

These properties are important in implementations as well as in showing full abstraction.

Address: COGS, University of Sussex, Brighton BN1 9QH, UK
Email: alanje@cogs.susx.ac.uk

This work has been funded by SERC project GR/H 16537.

1 Introduction

This paper is about the relationship between two fields of computer science: *full abstraction*, and *concurrent graph reduction*. Full abstraction is the study of relating denotational and operational semantics. Concurrent graph reduction is an efficient parallel implementation technique for non-strict functional programming languages.

FULL ABSTRACTION. Full abstraction, originally defined by MILNER (1977), explores the relationship between an operational semantics and its models. The operational view of a programming language is given by:

- A set of syntactic *terms* T, and a subset of terms called *programs*. The programs are then given an operational semantics as a reduction relation.
- A set of *tests* together with an operational definition of when a term *passes* a test. This induces the *testing preorder* on terms $t \sqsubseteq_O u$ iff every test t passes is passed by u.

A *model* of such an operational view is:

- A partially ordered set $(D, \leq)$.
- A function $[\![\cdot]\!] : T \to D$. This induces the *denotational preorder* on terms $t \sqsubseteq_D u$ iff $[\![t]\!] \leq [\![u]\!]$.

We characterize such models:

- D is *correct* iff $t \sqsubseteq_O u$ implies $t \sqsubseteq_D u$.
- D is *complete* iff $t \sqsubseteq_D u$ implies $t \sqsubseteq_O u$.
- D is *fully abstract* iff it is correct and complete.

For example, in ABRAMSKY (1989) and ONG's (1988) analysis of the untyped λ-calculus:

- A term is an untyped λ-calculus term, and a program is a closed term. The operational semantics is given as *leftmost–outermost* reduction between programs $M \to N$.
- A test is a closing context $C[\cdot]$. A term M passes $C[\cdot]$ iff $C[M]$ evaluates to *weak head normal form*, that is a λ-term $\lambda w \,.\, N$.

This is then given a denotational semantics in the domain $D \simeq (D \to D)_\perp$. ABRAMSKY and ONG showed that this denotational semantics is correct but not complete, and that the completeness problem can be reduced to definability, in that there is no untyped λ-calculus 'parallel convergence test' term P with the semantics:

$$[\![\mathsf{P}\, xyz]\!]\sigma = \begin{cases} \perp & \text{if } [\![x]\!]\sigma = [\![y]\!]\sigma = \perp \\ [\![z]\!]\sigma & \text{otherwise} \end{cases}$$

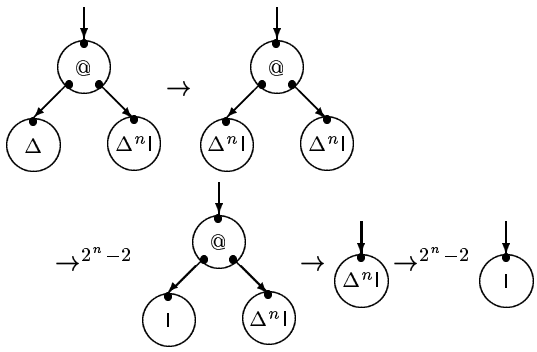
and that if such a term is added (and given an appropriate operational semantics) then the semantics is complete.

CONCURRENT GRAPH REDUCTION. Graph reduction is an efficient implementation technique for non-strict functional programming languages, such as AUGUSTSSON's (1984) Lazy ML, FAIRBURN's (1982) Ponder, JONES's (1992) Gofer, TURNER's (1985) Miranda[1], and Haskell (HUDAK *et al.*, 1992).

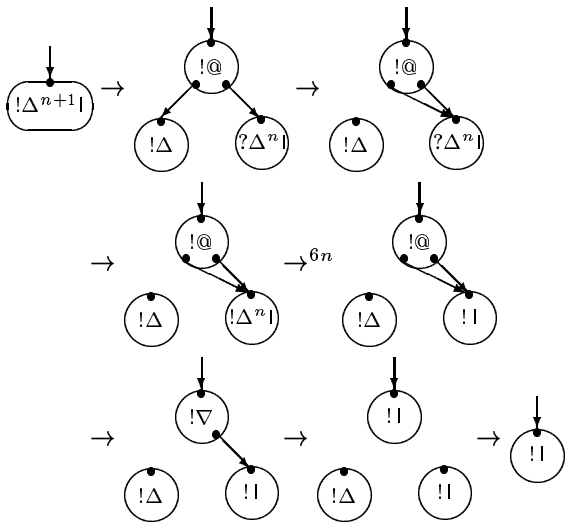
It was developed by WADSWORTH (1971) as an implementation of leftmost–outermost reduction. He observed that leftmost–outermost reduction can take exponential time to evaluate an expression, due to loss of *sharing* information. For example, if we define:

$$\mathsf{I} = \lambda x \,.\, x \qquad \Delta = \lambda x \,.\, xx$$
$$M^0 N = N \qquad M^{n+1} N = M(M^n N)$$

Then the evaluation of $\Delta^{n+1}\,\mathsf{I} \to^* \mathsf{I}$ is, where '@' denotes function application:



Thus, $\Delta^n\,\mathsf{I}$ takes $2^n - 2$ reductions to terminate. WADSWORTH observed that this inefficiency can be removed by reducing syntax *graphs* rather than trees. The graph evaluation of $\Delta^{n+1}\,\mathsf{I} \to^* \mathsf{I}$ is, where '∇' denotes an indirection, ! denotes a node which has been *tagged* for evaluation, and ? denotes a note which is *untagged* and is not currently being evaluated:



Thus, $\Delta^n\,\mathsf{I}$ takes $6n$ reductions to terminate. Note that

[1] When used as the name of a programming language, Miranda is a trademark of Research Software Limited

a graph may contain a number of tagged nodes, which allows for *concurrent* execution. The tagged nodes correspond to PEYTON JONES's (1987) *program annotations*, and also record the *blocking information* of the graph.

FULL ABSTRACTION AND GRAPH REDUCTION. There has been a number of papers showing full abstraction for tree reduction, notably PLOTKIN's (1977) full abstraction for PCF with parallel conditionals, ABRAMSKY (1989) and ONG's (1988) full abstraction for the lazy λ-calculus, and BOUDOL's (1992) full abstraction for a λ-calculus with call-by-value abstraction and parallel evaluation.

There has also been a number of papers showing the correctness of graph reduction, notably by WADSWORTH (1971), BARENDREGT *et al.* (1987), KENNAWAY *et al.* (1993), LESTER (1989), LAUNCHBURY (1993), PURUSHOTHAMAN and SEAMAN (1992), and the author (1993).

However, there have been no proofs of full abstraction for concurrent graph reduction. In this paper, we will follow ABRAMSKY (1989) when he said:

> Since current practice is well-motivated by efficiency considerations and is unlikely to be abandoned readily, it makes sense to see if a good modified theory can be developed for it.

OVERVIEW. In this paper we present a formal treatment of concurrent graph reduction, based on BERRY and BOUDOL's (1990) *Chemical Abstract Machine* (CHAM). This operational semantics includes:

- Tagged and untagged nodes.
- Garbage collection.
- Deadlocked graphs.

We also present a denotational semantics in the domain $D \simeq (D \to D)_\perp$ in which:

- Whether a node is tagged or not is irrelevant.
- Garbage collection is semantically unimportant.
- Deadlock and divergence are identified.

We then apply ABRAMSKY (1989) and ONG's (1988) techniques to show that this semantics is correct, and that by including *parallel convergence* nodes in the syntax, the semantics is complete.

2 The λ-calculus with recursive declarations

Terms from the λ-calculus with rec are:

- ∇x is an *indirection* pointing to x.
- $x@y$ is an *application* applying the function pointed to by x to the argument pointed to by y.
- $x \vee y$ is a *fork* which evaluates the terms pointed to by x and y and returns the identity function if one of them reaches weak head normal form. This is the *parallel convergence* operator $\mathsf{P}\,xy$ of the lazy λ-calculus.
- $\lambda x\,.\,M$ is an *abstraction.*
- $\mathsf{rec}\,D\ \mathsf{in}\ M$ is a local *recursive declaration* of D in M.

Declarations are:

- $x := !M$ is an *tagged node* declaring x to be M, and that M should be evaluated immediately.
- $x := ?M$ is an *untagged node* declaring x to be M, and that M should not be evaluated until it is needed.
- ϵ is the *empty* declaration.
- D, E is the *concatenated* declaration of D and E.
- $\nu x\,.\,D$ is the declaration D with a *local variable* x.

An *expression* is a term or a declaration. For example, the term:

$$\mathsf{rec}\,x := ?M, y := ?N\ \mathsf{in}\ x@y$$

declares x to be M and y to be N, then applies x to y. This can be contrasted with the term:

$$\mathsf{rec}\,x := !M, y := !N\ \mathsf{in}\ x@y$$

which is semantically equivalent, but allows evaluation of M and N to be performed concurrently. In the declaration:

$$x_1 := !M_1, \ldots, x_m := !M_m,$$
$$y_1 := ?N_1, \ldots, y_n := ?N_n$$

the terms M_i are tagged, and so they can all be evaluated concurrently, whereas the terms N_i are untagged, and so are evaluated when they are needed. All declarations are considered to be recursive, for example the fixed point of M is:

$$\mathsf{rec}\,x := !M, y := !x@y\ \mathsf{in}\ y$$

We have only allowed local declarations in terms, not in declarations. However, since we have allowed local variables $\nu x\,.\,D$, we can define the local declaration $\mathsf{rec}\,D\ \mathsf{in}\ E$. For example, $\mathsf{rec}(x = ?M)\ \mathsf{in}\ (y = ?N)$ is:

$$\nu x\,.\,(x := ?M, y := ?N)$$

The handling of local variables here is similar to *scope* in MILNER's (1991) polyadic π-calculus, and indeed has a very similar operational semantics.

DEFINITION. Lam and Dec are defined:

$$M ::= \nabla x \mid x@y \mid x \vee y \mid \lambda x\,.\,M \mid \mathsf{rec}\,D\ \mathsf{in}\ M$$
$$D ::= x := !M \mid x := ?M \mid \epsilon \mid D, D \mid \nu x\,.\,D$$

Let $D = E$ mean D and E are syntactically identical. □

EXAMPLES. Given a vector $\vec{x} = x_1 \ldots x_n$, we define:

$$\nu\vec{x}\,.\,D = \nu x_1\,.\,\ldots\,.\,\nu x_n\,.\,D$$

We implement the *black hole* term:

$$\mho = \mathsf{rec}\,x := !\nabla x\ \mathsf{in}\ x$$

We implement ABRAMSKY and ONG's *lazy λ-calculus* as (when x and y do not occur in M or N):

$$\begin{aligned} x &= \nabla x \\ MN &= \mathsf{rec}\, x := !M, y := ?N \;\mathsf{in}\; x@y \\ \mathsf{P}\, MN &= \mathsf{rec}\, x := !M, y := !N \;\mathsf{in}\; x \vee y \end{aligned}$$

For example, we can define the *diagonal* and *unsolvable* terms:

$$\Delta = \lambda x \,.\, xx \qquad \Omega = \Delta\Delta$$

We shall see that $\mho$ is *deadlocked* whereas Ω is *divergent*. □

Unfortunately, at the moment, there is nothing to prevent inconsistent declarations such as:

$$x := !M, x := !N$$

or declarations with dangling pointers such as:

$$\nu y \,.\, (x := !\nabla y)$$

We would like to avoid such terms, since their semantics is by no means obvious. We achieve this by restricting our attention to *well-formed* expressions, with no inconsistency or dangling pointers.

DEFINITION. The *written* variables of a declaration are:

$$\mathsf{wv}(x := !M) = \{x\} \quad \mathsf{wv}(x := ?M) = \{x\} \quad \mathsf{wv}\,\epsilon = \emptyset$$
$$\mathsf{wv}(D, E) = \mathsf{wv}\, D \cup \mathsf{wv}\, E \quad \mathsf{wv}(\nu x \,.\, D) = \mathsf{wv}\, D \setminus \{x\}$$

An expression is *well-formed* iff:

- every subexpression D, E has $\mathsf{wv}\, D \cap \mathsf{wv}\, E = \emptyset$.
- every subexpression $\nu x \,.\, D$ has $x \in \mathsf{wv}\, D$.

From now on, we shall only consider well-formed expressions. □

Similarly, we define the read variables and free variables of an expression.

DEFINITION. The *read* variables of an expression are:

$$\mathsf{rv}(\nabla x) = \{x\} \quad \mathsf{rv}(x@y) = \{x, y\}$$
$$\mathsf{rv}(x \vee y) = \{x, y\} \quad \mathsf{rv}(\lambda x \,.\, M) = \mathsf{rv}\, M \setminus \{x\}$$
$$\mathsf{rv}(\mathsf{rec}\, D \;\mathsf{in}\; M) = (\mathsf{rv}\, M \cup \mathsf{rv}\, D) \setminus \mathsf{wv}\, D$$
$$\mathsf{rv}(x := !M) = \mathsf{rv}\, M \quad \mathsf{rv}(x := ?M) = \mathsf{rv}\, M \quad \mathsf{rv}\,\epsilon = \emptyset$$
$$\mathsf{rv}(D, E) = \mathsf{rv}\, D \cup \mathsf{rv}\, E \quad \mathsf{rv}(\nu x \,.\, D) = \mathsf{rv}\, D \setminus \{x\}$$

The *free* variables of an expression are:

$$\mathsf{fv}\, M = \mathsf{rv}\, M \quad \mathsf{fv}\, D = \mathsf{rv}\, D \cup \mathsf{wv}\, D$$

A declaration is *closed* iff $\mathsf{rv}\, D \subseteq \mathsf{wv}\, D$. □

In implementation terms, the read variables of a declaration are the pointers leading out of it, and the written variables are pointers leading into it. For example, x is a pointer into $x := !\nabla y$ and y is a pointer out of it.

DEFINITION. A *renaming* is a function $\rho : V \to V$ which is almost everywhere the identity. Define:

- $M[\rho]$ is M with any read variable x replaced by ρx.
- $D[\rho]$ is D with any read variable x replaced by ρx.
- $[\rho]D$ is D with any written variable x replaced by ρx.

In each case we apply appropriate α-conversion to avoid capture of free variables. □

EXAMPLES. Some example renamings are:

$$\begin{aligned} (x := !\nabla x)[y/x] &= (x := !\nabla y) \\ [y/x](x := !\nabla x) &= (y := !\nabla x) \\ [y/x](x := !\nabla x)[y/x] &= (y := !\nabla y) \\ (\nu y \,.\, (y := !\nabla x))[y/x] &= \nu z \,.\, (z := !\nabla y) \end{aligned}$$

If $\mathsf{wv}\, D$ and $\mathsf{wv}\, E$ are disjoint then we define a localized declaration as:

$$\mathsf{rec}\, D \;\mathsf{in}\; E = \nu(\mathsf{wv}\, D) \,.\, (D, E)$$

this can be generalized to any D and E by α-converting D first. If $\mathsf{wv}\, D = \{x_1, \ldots, x_n\}$ and $y_1, \ldots, y_n$ are fresh then:

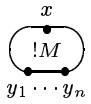
$$\mathsf{rec}\, D \;\mathsf{in}\; E = \nu \vec{y} \,.\, ([\vec{y}/\vec{x}]D[\vec{y}/\vec{x}], E[\vec{y}/\vec{x}])$$

for example:

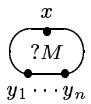
$$\begin{aligned} &\mathsf{rec}(x := ?\nabla x) \;\mathsf{in}\; (x := !\lambda w \,.\, x) \\ &\quad = \nu y \,.\, (y := ?\nabla y, x := !\lambda w \,.\, y) \end{aligned}$$

We shall see below that $x := !(\mathsf{rec}\, D \;\mathsf{in}\; M)$ is semantically equivalent to $\mathsf{rec}\, D \;\mathsf{in}\; (x := !M)$. □

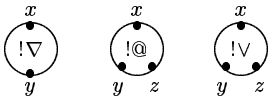
DEFINITION. We can draw a declaration as a graph, in the fashion of MILNER's (1989) flow graphs for CCS. A declaration $x := !M$ with read variables $y_1, \ldots, y_n$ is drawn:



Similarly, a declaration $x := ?M$ is drawn:



When M is ∇y, $y@z$ or $y \vee z$ we usually elide the read variables, drawing $x := !\nabla y$, $x := !y@z$ and $x := !y \vee z$ as:



A declaration ϵ is drawn as the empty graph.

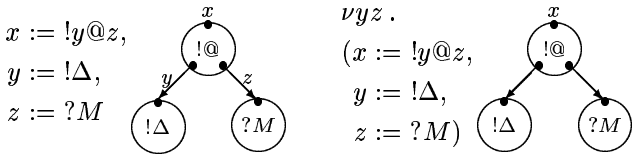
A declaration D, E is drawn by superimposing D on E.

A declaration $\nu x \,.\, D$ is drawn by drawing D and erasing any occurrence of x.

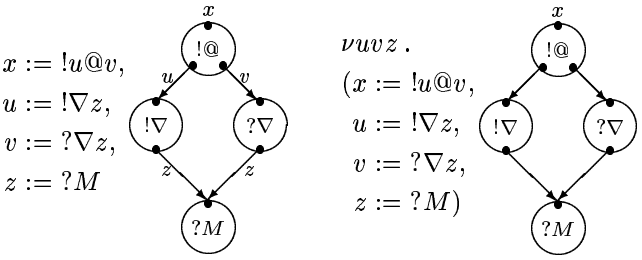
Whenever we have the same variable being read and written in a graph, we draw an arrow from the read

variable to the written variable. □

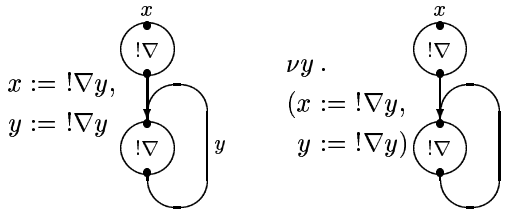
EXAMPLES. The application of Δ to M is drawn:



The application of M to itself, with sharing is drawn:



A cyclic graph is drawn:



We shall see that such tight cyclic graphs give rise to deadlock. □

3 Operational semantics

In this section, we give a formal presentation of the concurrent graph reduction algorithm described by PEYTON JONES (1987). We shall use declarations to represent graphs, and give the operational semantics as a reduction relation $D \to E$ between declarations.

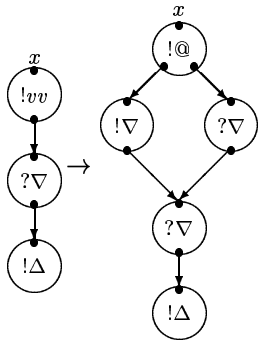
We give our operational semantics in two parts, based on BERRY and BOUDOL's (1990) *Chemical Abstract Machine*. We shall first define a syntactic equivalence $\equiv$ on declarations, and then define an operational semantics on declarations up to $\equiv$. This allows us to abstract away from syntactic details such as associativity of concatenation, and present the 'bare bones' of the operational semantics.

A similar approach has been taken by MILNER (1991) in presenting the π-calculus, and we shall follow his example more closely than that of BERRY and BOUDOL.

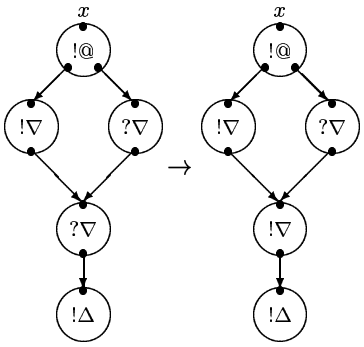
DEFINITION. $\equiv$ is given in Table 1. □

We use the equivalence $\equiv$ to simplify the operational semantics for graph reduction. This is given as eight axioms and three structural rules. The axioms are broken down into four phases:

- *Graph building*, in which a recursive declaration is expanded into a graph, for example:

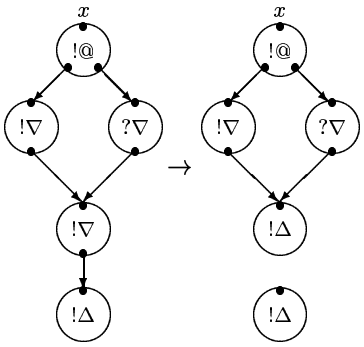


- *Spine traversal*, in which an untagged node pointed to by a tagged node becomes tagged, for example:



  There are three axioms, depending on whether the tagged node is an indirection, an application, or a fork.

- *Updating*, in which a node pointing to an abstraction is updated, for example:



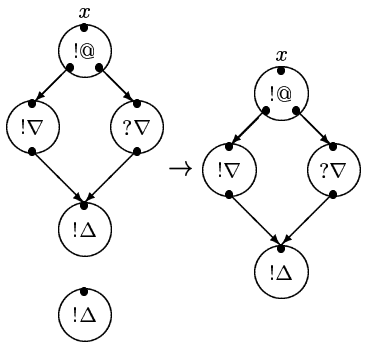
  There are three axioms, depending on whether the node is an indirection, an application, or a fork.

- *Garbage collection*, in which a sub-graph with no in-

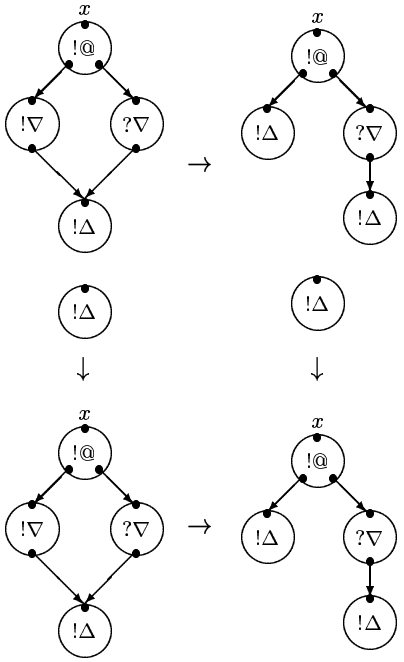
(ASSOC)	$D, (E, F) \equiv (D, E), F$	(SYMM)	$\dfrac{D \equiv E}{E \equiv D}$
(COMM)	$D, E \equiv E, D$	(TRANS)	$\dfrac{D \equiv E \equiv F}{D \equiv F}$
(UNIT)	$D, \epsilon \equiv D$	(L)	$\dfrac{D \equiv E}{D, F \equiv E, F}$
(α)	$\nu x \,.\, D \equiv \nu z \,.\, ([z/x]D[z/x])$	(R)	$\dfrac{D \equiv E}{F, D \equiv F, E}$
(νSWAP)	$\nu x \,.\, \nu y \,.\, D \equiv \nu y \,.\, \nu x \,.\, D$	(ν)	$\dfrac{D \equiv E}{\nu x \,.\, D \equiv \nu x \,.\, E}$
(νMIG)	$D, \nu z \,.\, E \equiv \nu z \,.\, (D, E)$		
($\vee$COMM)	$x := !(y \vee z) \equiv x := !(z \vee y)$		
(REFL)	$D \equiv D$		

TABLE 1. The definition of $\equiv$ (when $z \notin \mathsf{fv}\, D$)

coming pointers is removed, for example:



These phases are not sequential, and there may be more than one axiom which can be applied at any one point. Since each axiom (apart from garbage collection) uses a small number of nodes, there is much scope for concurrency, for example:



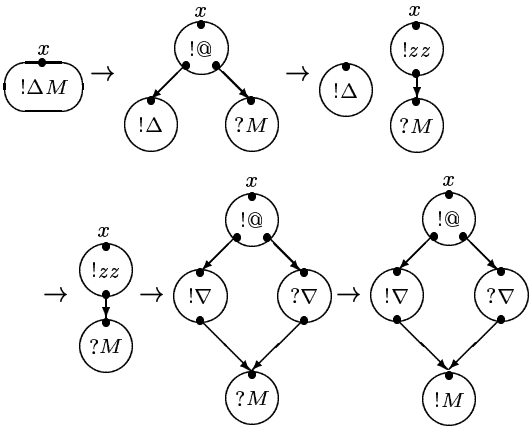
DEFINITION. $\mapsto$ is given in Table 2, and $D \to E$ iff $D \equiv \mapsto \equiv E$. □

EXAMPLES. For any M, the reduction:

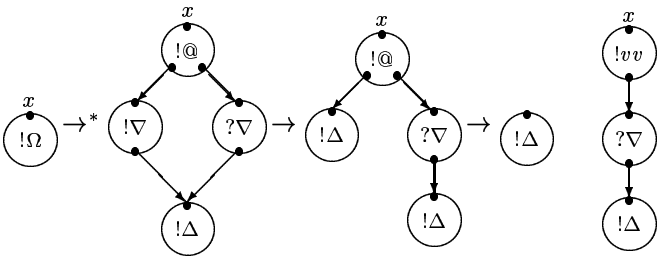
$x := !\Delta M$

$\quad \to^5 \nu uvz \,.\, (x := !u@v, u := !\nabla z, v := ?\nabla z, z := !M)$

is drawn:



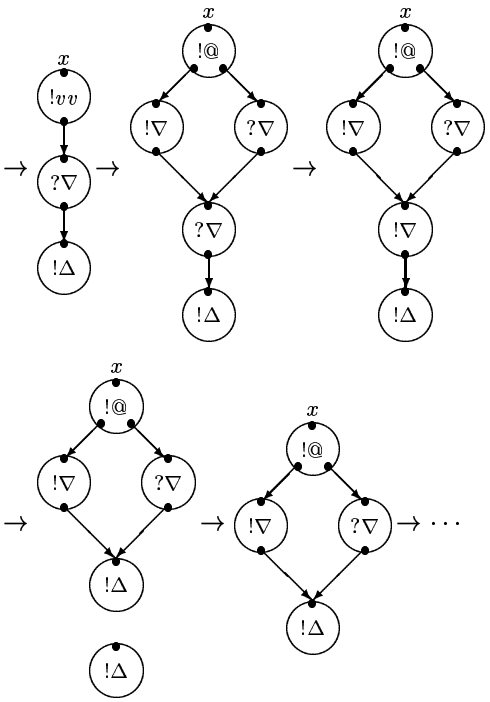
The reduction: $(x := !\Omega) \to^\infty$ is drawn:



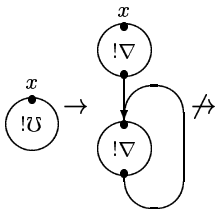
$$
\begin{array}{ll}
(\textsc{build}) & x := !(\mathsf{rec}\, D \;\mathsf{in}\; M) \mapsto \mathsf{rec}\, D \;\mathsf{in}\; (x := !M) \\
(\nabla\textsc{trav}) & x := !\nabla y, y := ?M \mapsto x := !\nabla y, y := !M \\
(@\textsc{trav}) & x := !y@z, y := ?M \mapsto x := !y@z, y := !M \\
(\vee\textsc{trav}) & x := !y \vee z, y := ?M \mapsto x := !y \vee z, y := !M \\
(\nabla\textsc{upd}) & x := !\nabla y, y := !\lambda w\,.\,M \mapsto x := !\lambda w\,.\,M, y := !\lambda w\,.\,M \\
(@\textsc{upd}) & x := !y@z, y := !\lambda w\,.\,M \mapsto x := !M[z/w], y := !\lambda w\,.\,M \\
(\vee\textsc{upd}) & x := !y \vee z, y := !\lambda w\,.\,M \mapsto x := !\mathsf{I}, y := !\lambda w\,.\,M \\
(\gamma) & \nu(\mathsf{wv}\, D)\,.\,D \mapsto \epsilon
\end{array}
$$

$$
(\textsc{l})\quad \frac{D \mapsto E}{D, F \mapsto E, F}
\qquad
(\textsc{r})\quad \frac{D \mapsto E}{F, D \mapsto F, E}
\qquad
(\nu)\quad \frac{D \mapsto E}{\nu x\,.\,D \mapsto \nu x\,.\,E}
$$

TABLE 2. The definition of $\mapsto$



This can be contrasted with the reduction:



Thus the declaration $x := !\mho$ is *deadlocked* rather than *divergent*. Denotationally, we shall identify the terms $\mho$ and Ω, since neither of them can reach weak head normal form, although operationally they are very different. □

We define x to be in *weak head normal form* (whnf) in D iff D contains $x := !\lambda w\,.\,M$.

DEFINITION.

- x is in whnf in $(x := !\lambda w\,.\,M)$.
- x is in whnf in (D, E) if x is in whnf in D or E.
- x is in whnf in $\nu y\,.\,D$ if x is in whnf in D and $x \neq y$.□

A variable x *converges* in D iff, once it has been tagged for evaluation, it can reach whnf.

DEFINITION. tag_x is defined (when $x \neq y$) as:

$$
\begin{aligned}
\mathsf{tag}_x(x := !M) &= (x := !M) \\
\mathsf{tag}_x(x := ?M) &= (x := !M) \\
\mathsf{tag}_x(y := !M) &= (y := !M) \\
\mathsf{tag}_x(y := ?M) &= (y := ?M) \\
\mathsf{tag}_x\, \epsilon &= \epsilon \\
\mathsf{tag}_x(D, E) &= (\mathsf{tag}_x\, D), (\mathsf{tag}_x\, E) \\
\mathsf{tag}_x(\nu x\,.\,D) &= \nu x\,.\,D \\
\mathsf{tag}_x(\nu y\,.\,D) &= \nu y\,.\,(\mathsf{tag}_x\, D)
\end{aligned}
$$

For closed D, $D \Downarrow_x E$ iff $\mathsf{tag}_x\, D \to^* E$ and x is in whnf in E, and $D \Downarrow_x$ iff $\exists E\,.\,D \Downarrow_x E$ □

This provides us with the *may-testing preorder*:

DEFINITION. $M \sqsubseteq_O N$ iff $C[M]\Downarrow_x \Rightarrow C[N]\Downarrow_x$ for any x and closing context C. □

4 Denotational semantics

The denotational semantics for Lam is given in the same domain $D \simeq (D \to D)_\perp$ as ABRAMSKY and ONG's lazy λ-calculus.

DEFINITION. D is the initial solution of:

$$D \simeq (D \to D)_\perp$$

Let the continuous functions $\mathsf{unfold} : D \to (D \to D)_\perp$ and $\mathsf{fold} : (D \to D)_\perp \to D$ form the above isomorphism.

An *environment* is a function $\sigma : V \to D$. Let Σ be the domain of environments, ordered pointwise.

This definition is made precise in the full paper. □

The semantics of a term M is given as an element $[\![M]\!]\sigma$ of D. The semantics of a declaration D is given as an element $[\![D]\!]\sigma$ of Σ. The main difference between the semantics of Lam and that of the lazy λ-calculus is that the former makes explicit use of recursion in the semantics of declarations.

DEFINITION. Define $[\![M]\!] : \Sigma \to D$ as:

$$\begin{aligned}
[\![\nabla x]\!]\sigma &= \sigma x \\
[\![x@y]\!]\sigma &= \mathsf{apply}(\sigma x)(\sigma y) \\
[\![x \vee y]\!]\sigma &= \mathsf{fork}(\sigma x)(\sigma y) \\
[\![\lambda x \,.\, M]\!]\sigma &= \mathsf{fold}(\mathsf{lift}([\![M]\!] \circ \mathsf{update}\,\sigma x)) \\
[\![\mathsf{rec}\, D \;\mathsf{in}\; M]\!]\sigma &= [\![M]\!]([\![D]\!]\sigma)
\end{aligned}$$

Define $[\![D]\!] : \Sigma \to \Sigma$ as:

$$\begin{aligned}
[\![x := !M]\!]\sigma &= \mathsf{fix}(\mathsf{set}\{x\}(x := [\![M]\!]))\sigma \\
[\![x := ?M]\!]\sigma &= \mathsf{fix}(\mathsf{set}\{x\}(x := [\![M]\!]))\sigma \\
[\![\epsilon]\!]\sigma &= \sigma \\
[\![D, E]\!]\sigma &= \mathsf{fix}(\mathsf{set}(\mathsf{wv}(D, E))([\![D]\!] \circ [\![E]\!]))\sigma \\
[\![\nu x \,.\, D]\!]\sigma &= \mathsf{new}\, x [\![D]\!]\sigma
\end{aligned}$$

where:

$$\mathsf{fork}\, ab = \begin{cases} \bot & \text{if } a = b = \bot \\ \mathsf{fold}(\mathsf{lift}\,\mathsf{id}) & \text{otherwise} \end{cases}$$

$$\mathsf{apply}\, ab = \begin{cases} fb \text{ if } \mathsf{unfold}\, a = \mathsf{lift}\, f \\ \bot \text{ otherwise} \end{cases}$$

$$\mathsf{update}\, \sigma x a y = \begin{cases} a & \text{if } x = y \\ \sigma y & \text{otherwise} \end{cases}$$

$$\mathsf{new}\, x f \sigma y = \begin{cases} \sigma x & \text{if } x = y \\ f \sigma y & \text{otherwise} \end{cases}$$

$$(x := f)\sigma y = \begin{cases} f\sigma \text{ if } x = y \\ \sigma y \text{ otherwise} \end{cases}$$

$$\mathsf{set}\, X f g \sigma x = \begin{cases} f(g\sigma)x \text{ if } x \in X \\ \sigma x \qquad \text{otherwise} \end{cases}$$

$$\mathsf{fix}\, f = \bigvee \{f^n \bot \mid n \text{ in } \omega\}$$

$M \sqsubseteq_D N$ iff $[\![M]\!] \leq [\![N]\!]$. □

The semantics agrees with ABRAMSKY and ONG's lazy λ-calculus:

$$\begin{aligned}
[\![x]\!]\sigma &= \sigma x \\
[\![MN]\!]\sigma &= \mathsf{apply}([\![M]\!]\sigma)([\![N]\!]\sigma) \\
[\![\mathsf{P}\, MN]\!]\sigma &= \mathsf{fork}([\![M]\!]\sigma)([\![N]\!]\sigma)
\end{aligned}$$

5 Program logic

The proof that D is fully abstract for Lam proceeds in much the same way as for ABRAMSKY and ONG's lazy λ-calculus. We present the program logic of COPPO types (BARANDREGT *et al.*, 1983) and use it as a link between the denotational and operational semantics. The program logic Φ has propositions:

- ω, satisfied by any closed term.
- $\phi \wedge \psi$, satisfied by any term that satisfies ϕ and ψ.
- $\phi \to \psi$, satisfied by any term that converges, and that when applied to any term satisfying ϕ the result satisfies ψ.

For example, a closed term satisfies $\gamma = \omega \to \omega$ iff it converges.

DEFINITION. Φ is defined as:

$$\phi ::= \omega \mid \phi \wedge \phi \mid \phi \to \phi$$

A *context* Γ is a list $x_1 : \phi_1, \ldots, x_n : \phi_n$ with distinct x_i.

- Let $\mathsf{wv}(x_1 : \phi_1, \ldots, x_n : \phi_n) = \{x_1, \ldots, x_n\}$.
- Let $(\Gamma, x : \phi, \Delta)(x) = \phi$, and $\Gamma(x) = \omega$ when $x \notin \mathsf{wv}\,\Gamma$.
- Let $\Gamma \wedge \Delta$ be the context s.t. $(\Gamma \wedge \Delta)(x) = \Gamma(x) \wedge \Delta(x)$.
- Let $\nu x \,.\, (\Gamma, x : \phi, \Delta) = \Gamma, \Delta$ and $\nu x \,.\, \Gamma = \Gamma$ when $x \notin \mathsf{wv}\,\Gamma$. □

Φ is given a proof system for judgements of the form $\Gamma \vdash M : \phi$ and $\Gamma \vdash D : \Delta$. This is first given as a preorder $\vdash \phi \leq \psi$, which characterizes when ψ is a refinement of ϕ.

DEFINITION. The preorder $\leq$ is given by axioms:

$$\begin{array}{ll}
(\text{ID}) & \vdash \phi \leq \phi \\
(\omega\text{I}) & \vdash \phi \leq \omega \\
(\wedge\text{E}a) & \vdash \phi \wedge \psi \leq \phi \\
(\wedge\text{E}b) & \vdash \phi \wedge \psi \leq \psi \\
(\to\omega) & \vdash \phi \to \omega \leq \omega \to \omega \\
(\to\wedge) & \vdash (\phi \to \psi) \wedge (\phi \to \chi) \leq \phi \to (\psi \wedge \chi)
\end{array}$$

and structural rules:

$$(\text{TRANS})\ \frac{\vdash \phi \leq \psi \leq \chi}{\vdash \phi \leq \chi} \qquad (\wedge\text{I})\ \frac{\vdash \phi \leq \psi \quad \vdash \phi \leq \chi}{\vdash \phi \leq (\psi \wedge \chi)}$$

$$(\to\leq)\ \frac{\vdash \phi' \leq \phi \quad \vdash \psi \leq \psi'}{\vdash (\phi \to \psi) \leq (\phi' \to \psi')}$$

Let $\vdash \phi = \psi$ iff $\vdash \phi \leq \psi \leq \phi$ and $\vdash \Gamma \leq \Delta$ iff $\forall x \,.\, \vdash \Gamma(x) \leq \Delta(x)$. □

For example, we can show that $\wedge$ is commutative, associative, idempotent and has unit ω in the equivalence $\vdash \phi = \psi$. The partial order $\vdash \phi \leq \psi$ is used in defining the proof system $\Gamma \vdash M : \phi$, since all of the structural rules (such as CUT, WEAKENING and CONTRACTION) can be given by one rule ($\leq$). The proof system induces a preorder on terms given by $M \sqsubseteq_S N$ iff N satisfies any property that M satisfies.

DEFINITION. The proof system $\Gamma \vdash M : \phi$ is given by axioms:

$$\begin{array}{ll}
(\omega\text{I}) & \vdash M : \omega \\
(\text{ID}) & x : \phi \vdash \nabla x : \phi \\
(\to\text{E}) & (x : \phi \to \psi) \wedge (y : \phi) \vdash x@y : \psi \\
(\vee a) & x : \gamma \vdash x \vee y : \phi \to \phi \\
(\vee b) & y : \gamma \vdash x \vee y : \phi \to \phi
\end{array}$$

and structural rules:

$$(\wedge\text{I})\ \frac{\Gamma \vdash M : \phi \quad \Gamma \vdash M : \psi}{\Gamma \vdash M : (\phi \wedge \psi)}$$

$$(\leq)\ \frac{\vdash \Gamma \leq \Delta \quad \Delta \vdash M : \phi \quad \vdash \phi \leq \psi}{\Gamma \vdash M : \psi}$$

$$(\rightarrow\text{I})\ \frac{\Gamma, x : \phi \vdash M : \psi}{\Gamma \vdash \lambda x \,.\, M : \phi \rightarrow \psi}$$

$$(\mathsf{rec})\ \frac{\Gamma \vdash D : \Delta \quad \Delta \vdash M : \phi}{\Gamma \vdash \mathsf{rec}\, D \,\mathsf{in}\, M : \phi}$$

The proof system $\Gamma \vdash D : \Delta$ is given by axiom:

$$(\bot)\quad \Gamma \vdash D : \nu(\mathsf{wv}\, D) \,.\, \Gamma$$

and structural rules:

$$(\wedge\text{I})\ \frac{\Gamma \vdash D : \Delta \quad \Gamma \vdash D : \Theta}{\Gamma \vdash D : (\Delta \wedge \Theta)}$$

$$(\leq)\ \frac{\vdash \Gamma \leq \Gamma' \quad \Gamma' \vdash D : \Delta' \quad \vdash \Delta' \leq \Delta}{\Gamma \vdash D : \Delta}$$

$$(!)\ \frac{\Gamma \vdash (x := !M) : \Delta \quad \Delta \vdash M : \phi}{\Gamma \vdash (x := !M) : (x : \phi)}$$

$$(?)\ \frac{\Gamma \vdash (x := ?M) : \Delta \quad \Delta \vdash M : \phi}{\Gamma \vdash (x := ?M) : (x : \phi)}$$

$$(\text{L})\ \frac{\Gamma \vdash D, E : \Delta \quad \Delta \vdash D : \Theta}{\Gamma \vdash D, E : \Theta}$$

$$(\text{R})\ \frac{\Gamma \vdash D, E : \Delta \quad \Delta \vdash E : \Theta}{\Gamma \vdash D, E : \Theta}$$

$$(\nu)\ \frac{\nu x \,.\, \Gamma \vdash D : \Delta}{\Gamma \vdash \nu x \,.\, D : \nu x \,.\, \Delta}$$

Then $M \sqsubseteq_S N$ iff $\forall \Gamma, \phi \,.\, \Gamma \vdash M : \phi \Rightarrow \Gamma \vdash N : \psi$. □

6 Confluence

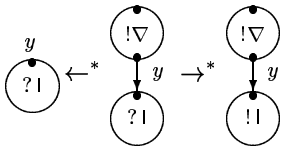
In the following three sections we consider three properties of the operational semantics for graph reduction:

- This section looks at *confluence*.
- Section 7 looks at *tagging*.
- Section 8 looks at *referential transparency*.

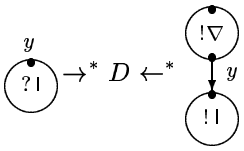
These three properties are used in the proof that D is fully abstract for Lam.

DEFINITION. A relation $\mathcal{R}$ is *confluent* iff $x\ \mathcal{R}^{-1}\mathcal{R}\ y$ implies $x\ \mathcal{R}\mathcal{R}^{-1}\ y$. □

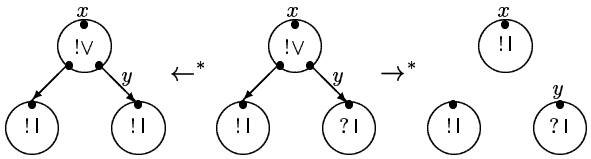
Confluence is very useful in proving results about an operational semantics. There are two reasons why $\rightarrow^*$ is not confluent. The first is due to garbage collection, since:



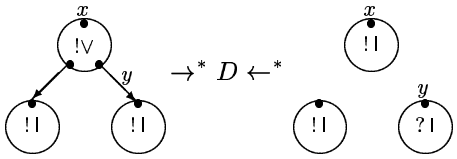
but there is no declaration D such that:



The second is due to fork updating, since:



but there is no declaration D such that:



In the full paper, we present a reduction strategy $\rightarrow_c$, which:

- Does not use the garbage collection axiom (γ).
- Replaces the fork updating axiom ($\vee$UPD) with axioms which only allow $x := !y \vee z$ to be updated when y and z have been tagged.

We then show that $\rightarrow_c$ is confluent, and that, for any closed D:

$$D{\Downarrow_x} \text{ iff } \mathsf{tag}_x\, D \rightarrow_c^* E \text{ and } x \text{ is in whnf in } E$$

One corollary of this is that garbage collection is semantically unimportant, since a term converges iff it converges without garbage collecting. This is unsurprising, since garbage collection is used to overcome memory limitations.

7 Independence from tagging

The denotational semantics for tagged $(x := !M)$ and untagged declarations $(x := ?M)$ is the same, despite the fact that tagged and untagged declarations have very different operational behaviour. For example the declaration:

$$\nu y \,.\, (x := !\,|, y := !\Omega)$$

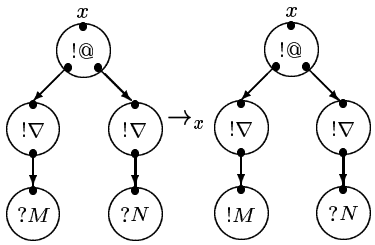
diverges, whereas the declaration:

$$\nu y \,.\, (x := !\,|, y := ?\Omega)$$

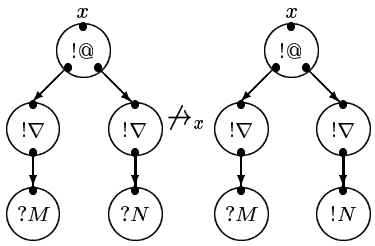
does not. However, both of them can reach whnf at x, and since the testing equivalence is based on reaching whnf, they are testing equivalent. In the full paper, we show that *convergence is independent of tagging*, that is:

$$D{\Downarrow_x} \text{ iff } \mathsf{tag}_y\, D{\Downarrow_x}$$

In order to show this, we present a reduction strategy $\rightarrow_x$, where a reduction $D \rightarrow_x E$ takes place only when the reduction is needed in order to evaluate x. For example, we allow:



since we need to evaluate M in order to evaluate x, but:



since we may not need to evaluate N in order to evaluate x. This reduction strategy corresponds to the *leftmost–outermost* reduction strategy used in sequential graph reduction. We then show that $\rightarrow_x$ ignores tagging information, in that:

$$\begin{array}{l}\text{if } \mathsf{tag}_x\,\mathsf{tag}_y\,D \rightarrow^* E \\ \text{then } \mathsf{tag}_x\,D \rightarrow^* F \text{ and } E \equiv \mathsf{tag}_y\,F\end{array}$$

and that:

$$D{\Downarrow_x} \text{ iff } \mathsf{tag}_x\,D \rightarrow_x^* E \text{ and } x \text{ is in whnf in } E$$

From this, it is simple to show that:

$$D{\Downarrow_x} \text{ iff } \mathsf{tag}_y\,D{\Downarrow_x}$$

One corollary of this is that tagging is semantically unimportant, since a term converges irrespective of which subterms have been tagged. This is unsurprising, since tagging is used for efficiency reasons.

8 Referential transparency

Referential transparency, introduced by EVANS (1968), means that the semantics of a term should be the same as the semantics of a pointer to a term. In our semantics this is the same as saying:

$$[\![x := !\nabla y, y := !M]\!] = [\![x := !M, y := !M]\!]$$

Denotationally, this is quite simple to prove (although it does require some non-trivial reasoning about fixed points). But to prove this operationally is much harder. We need to show that copying a section of graph is equivalent to making a pointer into a section of graph. Much of the work in showing this turns out to be in showing that if two variables point to the same term, then we can substitute one for the other, that is:

$$\begin{array}{c}[\![(D, x := !M, y := !M)[x/z]]\!] \\ = [\![(D, x := !M, y := !M)[y/z]]\!]\end{array}$$

In order to prove this operationally, we need to find some property of a declaration $(D, x := !M, y := M)$ which we can use as an operational invariant, so:

- If D satisfies the invariant and $D[x/z] \rightarrow_c E$ then $E \rightarrow_c^* F[x/z]$, $D[y/z] \rightarrow_c^* F[y/z]$, and F satisfies the invariant.

We use this to show that if $D[x/z]{\Downarrow_w}$ then $D[y/z]{\Downarrow_w}$. Unfortunately, we cannot use 'x and y point to syntactically identical terms' as the invariant, since:

$$\begin{array}{l}x := !(\mathsf{rec}\,w := !M \text{ in } \nabla w), y := !(\mathsf{rec}\,w := !M \text{ in } \nabla w) \\ \rightarrow^2 \nu v w\,.\,(v := !M[v/w], w := !M, x := !\nabla v, y := !\nabla w)\end{array}$$

and although x and y are syntactically identical in the LHS, they are not syntactically identical in the RHS. However, they *are* identical up to α-conversion, and we use this as the basis of an invariant: *simulation*, based on MILNER's (1989) definition of bisimulation between processes. Informally, two variables x and y are similar iff x points to M, y points to N, and M and N are identical, up to substitution of similar variables. More formally, we define a simulation for ν-less declarations as:

DEFINITION. The ν-less declarations are:

- ϵ, $x := !M$ and $x := ?M$.
- D, E when D and E are ν-less.

$\mathcal{R} \subseteq \mathsf{wv}\,D \times \mathsf{wv}\,D$ is a ν-less D-simulation iff D is ν-less, and for any $x\,\mathcal{R}\,y$:

- If $D \equiv (x := !M, E)$
  then $D \equiv (y := !N[\vec{y}/\vec{z}], F)$, $M = N[\vec{x}/\vec{z}]$, and $\vec{x}\,\mathcal{R}\,\vec{y}$.
- If $D \equiv (x := ?M, E)$
  then $D \equiv (y := ?N[\vec{y}/\vec{z}], F)$, $M = N[\vec{x}/\vec{z}]$, and $\vec{x}\,\mathcal{R}\,\vec{y}$.

where $\vec{x}\,\mathcal{R}\,\vec{y}$ iff $\forall i\,.\,x_i\,\mathcal{R}\,y_i$. □

For example, if E is ν-less, and D is the declaration:

$$x := !M, y := !M, E$$

then one ν-less D-simulation is: $\{(x, y)\}$ and so x is D-similar to y. We generalize simulation to any declaration D by converting it into the form $\nu\vec{x}\,.\,E$, and finding a ν-less E-simulation:

DEFINITION.

- $\nu x\,.\,\mathcal{R} = \{(y, z) \mid x \neq y\,\mathcal{R}\,z \neq x\}$.
- $\mathcal{R}$ is a D-simulation iff $D \equiv \nu\vec{x}\,.\,E$, $\mathcal{R}'$ is a ν-less E-simulation, and $\mathcal{R} = \nu\vec{x}\,.\,\mathcal{R}'$.
- $D \vdash x \sim y$ iff there is a D-simulation $\mathcal{R}$ with $x\,\mathcal{R}\,y$. □

For example:

$$(x := !M, y := !M, E) \vdash x \sim y$$

In the full paper, we show that similar variables have the same convergence, that is:

$$\text{if } D \vdash x \sim y \text{ then } D{\Downarrow_x} \Leftrightarrow D{\Downarrow_y}$$

We then use this to show referential transparency, in that:

$$(D, x := !M, y := !M){\Downarrow_z} \text{ iff } (D, x := !\nabla y, y := !M){\Downarrow_z}$$

9 Full abstraction

In the full paper, we show that D is fully abstract for Lam in three stages. Firstly, we first define two alternative interpretations of the program logic:

- A denotational interpretation $[\![\phi]\!] : D$ and $[\![\Gamma]\!] : \Sigma$.
- An operational interpretation $\Gamma \models M : \phi$, based on the operational semantics.

We then use the operational properties in Sections 6–8 to show that the three interpretations of the logic are equivalent:

$$\Gamma \vdash M : \phi \text{ iff } [\![\phi]\!] \leq [\![M]\!][\![\Gamma]\!] \text{ iff } \Gamma \models M : \phi$$

From this result, it is not difficult to show full abstraction:

$$M \sqsubseteq_O N \text{ iff } M \sqsubseteq_S N \text{ iff } M \sqsubseteq_D N$$

Thus, the techniques of ABRAMSKY and ONG can be adapted to provide a fully abstract semantics for a practical implementation of the untyped λ-calculus.

10 Conclusions

In this paper, we have investigated the relationship between the semantic notion of *full abstraction* and the implementation technique of *concurrent graph reduction*. We have shown that:

- Concurrent graph reduction can be given a simple operational presentation in the style of BERRY and BOUDOL's (1990) *chemical abstract machine*.
- The methods of ABRAMSKY (1989) and ONG's (1988) *lazy λ-calculus* can be used to show that the fully abstract model for leftmost-outermost reduction is also fully abstract for concurrent graph reduction.
- To show full abstraction, we discussed a *confluent* reduction strategy, the relationship between *concurrent* and *sequential* reduction, and *referential transparency*. These properties are also important in implementations, and it is reassuring that showing full abstraction and writing compilers have so many issues in common.

The full paper discusses related work in the semantics of graph reduction, and possible future work.

References

ABRAMSKY, S. (1989). The lazy lambda calculus. In TURNER, D., editor, *Declarative Programming*. Addison-Wesley.

AUGUSTSSON, L. (1984). A compiler for lazy ML. In *Proc. ACM Symp. Lisp and Functional Programming*, pages 218–227.

BARANDREGT, H. P., COPPO, M., and DEZANI-CIANCAGLINI, M. (1983). A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940.

BARENDREGT, H. P., VAN EEKELEN, M. C. J. D., GLAUERT, J. R. W., KENNAWAY, J. R., PLASMEIJER, M. J., and SLEEP, M. R. (1987). Term graph rewriting. In *Proc. PARLE 87*, volume 2, pages 141–158. Springer-Verlag. LNCS 259.

BERRY, G. and BOUDOL, G. (1990). The chemical abstract machine. In *Proc. 17th Ann. Symp. Principles of Programming Languages*.

BOUDOL, G. (1992). Lambda-calculi for (strict) parallel functions. Technical report 1387, INRIA Sophia-Antipolis.

EVANS JR, A. (1968). PAL—a language for teaching programming linguistics. In *Proc. ACM 23rd Natl. Conf.* Brandon/Systems Press.

FAIRBURN, J. (1982). Ponder and its type system. Technical report 31, Cambridge University Computer Lab.

HUDAK, P., PEYTON JONES, S. L., WADLER, P., *et al.*(1992). A report on the functional language Haskell. *SIGPLAN Notices*.

JEFFREY, A. (1993). A chemical abstract machine for graph reduction. In *Proc. MFPS 93*. Springer-Verlag. LNCS.

JONES, M. (1992). The Gofer technical manual. Part of the Gofer distribution.

KENNAWAY, J. R., KLOP, J. W., SLEEP, M. R., and DE VRIES, F. J. (1993). The adequacy of term graph rewriting for simulating term rewriting. In SLEEP, M. R., PLASMEIJER, M. J., and VAN EEKELEN, M. C. J. D., editors, *Term Graph Rewriting: Theory and Practice*, chapter 12. John Wiley and Sons.

LAUNCHBURY, J. (1993). A natural semantics for lazy evaluation. In *Proc. ACM Sigplan–Sigact POPL*.

LESTER, D. (1989). *Combinator Graph Reduction: A Congruence and its Applications*. D.Phil thesis, Oxford University.

MILNER, R. (1977). Fully abstract semantics of typed λ-calculi. *Theoret. Comput. Sci.*, 4:1–22.

MILNER, R. (1989). *Communication and Concurrency*. Prentice-Hall.

MILNER, R. (1991). The polyadic π-calculus: a tutorial. In *Proc. International Summer School on Logic and Algebra of Specification*, Marktoberdorf.

ONG, C.-H. L. (1988). *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. PhD thesis, Imperial College, London University.

PEYTON JONES, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

PLOTKIN, G. (1977). LCF considered as a programming language. *Theoret. Comput. Sci.*, 5:223–256.

PURUSHOTHAMAN, S. and SEAMAN, J. (1992). An adequate operational semantics of sharing in lazy evaluation. In *Proc. ESOP 92*.

TURNER, D. (1985). Miranda: A non-strict functional language with polymorphic types. In *Proc. IFIP Conf. Functional Programming Languages and Computer Architecture*. Springer-Verlag. LNCS 201.

WADSWORTH, C. P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. D.Phil thesis, Oxford University.