A Natural Semantics for Lazy Evaluation

John Launchbury
Computing Science Department
Glasgow University

jl@dcs.glasgow.ac.uk

Abstract

We define an operational semantics for lazy evaluation which provides an accurate model for sharing. The only computational structure we introduce is a set of bindings which corresponds closely to a heap. The semantics is set at a considerably higher level of abstraction than operational semantics for particular abstract machines, so is more suitable for a variety of proofs. Furthermore, because a heap is explicitly modelled, the semantics provides a suitable framework for studies about space behaviour of terms under lazy evaluation.

1 Introduction

In this paper we provide an operational semantics for lazy evaluation of an extended λ-calculus. *Laziness* implies a number of things: first that the language is non-strict, second that certain reductions are shared, and lastly that evaluation ceases once an outer lambda is encountered. The semantics captures each of these aspects.

Why bother with an explicit semantics for laziness at all? The reason is that it is often quite hard to know how a particular term will behave under lazy evaluation. Will a certain subcomputation be repeated or not? How much heap will be required? Will a particular closure be accessed once or many times?

The following example shows the effect of sharing. When evaluating the term,

$$let\ u = 3 + 2, v = u + 1\, in\ v + v$$

v is demanded twice but, because the closure which v represents is overwritten with its value (6) after the first demand, the computation $u + 1$ is only performed once (as a consequence, u is only accessed once). The *let* construct may be thought of as naming closures (or subcomputations) that are only evaluated when required.

For a more taxing example, consider the difference in evaluation of the following two terms.

$$\begin{array}{l} let\ u = 3 + 2, f = \lambda x.(let\ v = u + 1\ in\ v + x) \\ in\ f\ 2 + f\ 3 \end{array}$$

$$\begin{array}{l} let\ u = 3 + 2, f = (let\ v = u + 1\ in\ \lambda x.v + x) \\ in\ f\ 2 + f\ 3 \end{array}$$

In the first of these the computation $u + 1$ is repeated, once for each use of f (and so u is accessed twice). In the second, the computation $u + 1$ is performed once only (and u is accessed just once). Showing why this happens is exactly what the semantics is for. In particular, this work was motivated by a need to be precise about when closures where built, how often computations were performed, how often closures were accessed, and the operational implications of lambda lifting and full laziness [PL91]. Similarly, a semantics for laziness is vital precursor to being precise about the difference between laziness and optimal reduction strategies [Lév80].

The semantics for laziness presented in this paper are simpler than any previously published. The key reason is that we separate the semantics into two parts. The first stage is a static conversion of the λ-calculus into a form where the creation and sharing of closures is *explicit*. This leads directly to a very simple semantics at the level of closures.

The rest of the paper is organised as follows. After discussing related work, we define the explicit-closure version of the λ-calculus we use, and describe the static normalising transformation for converting arbitrary λ-expressions into that language. We then provide a natural semantics (a big-step operational semantics) for terms in the language, which both preserves and is computationally adequate with respect to an appropriate denotational semantics. The paper concludes with applications and extensions.

2 Related Work

There have already been a number of attempts to provide semantics for laziness. Perhaps the best known is due to Abramsky and Ong [Abr90, Ong88] where they explore the theoretical consequences of treating λ-terms in weak head normal form as values. Abramsky and Ong argue that since implementations of lazy languages cease reduction once an outer lambda is encountered (i.e. values are terms in weak head normal form—whnf), the semantic implications of this should be studied. This leads to a large and powerful theory, but one in which sharing is ignored completely. Thus they omits precisely the aspect of laziness we wish to study.

Abramsky and Ong's semantics are defined at a high level of abstraction. At the other end of the scale is the operational semantics given to define the behaviour of abstract machines. Examples of this are the G-machine [Jon84], the STG-machine [Pey92], the TIM [FW87] and TIGRE [KL89]. At the level of these machines we have to deal with code pointers, stacks, indirection nodes, and the like. These operational semantics capture laziness completely, but contain so much extra detail as to make reasoning or proofs nigh on impossible. Furthermore, being so specific makes it hard to translate results about one abstract machine to another. One of the goals of this work is to provide a common semantic base for a wide range of abstract machines.

To be able to study sharing, therefore, we need a semantics containing more detail than Abramsky and Ong's, but less than provided by particular abstract machines. The earliest intermediate level semantics seems to have been Josephs' [Jos89]. This denotational semantics is continuation-based, and manipulates both an environment and a store. Sharing is successfully modelled, including the sharing that occurs in implementing fixed points. However, using both a store and a continuation provides the semantics with all the usual mechanisms required for modelling imperative languages with *goto*s! so again it makes the prospect of performing proofs rather daunting. Furthermore, because the semantics was denotational, Josephs had to introduce a forcing function (corresponding to the *print*-demand) for controlling the extent of evaluation required at any point.

An operational alternative was adopted by Purushothaman and Seaman [PS92]. The authors present an operational semantics for Lazy PCF which they prove equivalent to a standard denotational semantics (observations at higher types are treated specially because of this). Their rules capture most sharing, but as many closures are built within terms, the application rule is greatly complicated, and also the semantics is unsuitable for studying space behaviour. However, the main weakness is the inability of the semantics to capture sharing in recursive computations. By their semantics, a recursive term $\mu t.e$ is equivalent to $e[(\mu t.e)/t]$. If e is of the form *if* e' *then Nil else Cons 1 t*, for example, then the computation of e' will be repeated for every element of the infinite list. Sharing has been lost.

Much work has been done on making substitution explicit, the most relevant for our purposes being that by Maranget [Mar91], where he develops a framework of *Labelled-Terms Rewriting Systems*. Using these he studies the weak λ-calculi and shows the lazy strategy to be optimal. The resulting semantics for laziness is significantly more complex than that presented here (having been developed with different goals in mind), and also omits recursive *let*s which are a vital part of modern lazy functional languages.

The semantics for Id also deserve a mention at this point [ANP89]. While not lazy, Id has a non-strict semantics with sharing defined by a small-step semantics of a core of the Id language. Many rules may apply at any one time, but as the system is confluent the result is deterministic. By defining a particular reduction order and extending their rules to discard unneeded redexes, laziness can be modelled. A further point of contact is that, when providing a semantics for the kernel of Id, Ariola and Arvind use a similar technique to ours for making closures explicit [AA91].

3 Modelling Laziness

The semantics we present is an intermediate-level operational semantics, lying midway between a straightforward denotational semantics (or, equivalently, the operational semantics of Abramsky and Ong) and a full operational semantics of an abstract machine. As such it accurately captures the sharing within lazy evaluation without requiring the extra machinery either of continuations or of stacks, code pointers, dumps, and the like. The heap is the only computational structure required.

We capture laziness in two stages. The first is a static transformation of the λ-expression to a normalised form in which sharing is easy to express, and the second is a dynamic semantics for normalised lambda expressions. Separating these phases means that the dynamic semantics is much simpler that would otherwise be the case.

3.1 Normalising Terms

We begin with a lambda calculus extended with (recursive) *let*s and normalise it to a restricted syntax. These normalised λ-expressions have two distinguishing features: all bound variables are distinct; and all applications are applications of an expression to a variable. Thus,

$$\begin{array}{lcll} x & \in & \mathit{Var} & \\ e & \in & \mathit{Exp} & ::= \lambda x.e \\ & & & \mid \; e\ x \\ & & & \mid \; x \\ & & & \mid \; \mathit{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathit{in}\ e \end{array}$$

Operationally *let*s may be viewed as the construct that builds closures in the heap, and the fact that *let*s are recursive allows a closure to contain a reference to itself. This can give rise to cyclic structures in the heap, exactly as arises in most implementations. Without *let*s it would be impossible to build cycles, so they are more than merely syntactic sugar.

Having distinct names means that scope becomes irrelevant. In particular, though *let*s permit recursion, they may be used to model a nonrecursive binding as no untoward name capture can occur.

The syntactic restriction on application means that arguments to functions are only ever explicitly-named closures. This is valuable in that it removes the issue of generating new closure sites from within the dynamic semantics.

The process of normalisation can be specified in two stages. The first, which we write as $\hat{e}$, is simply α-conversion: a renaming of all the bound variables in e using completely fresh variables. The second, which we write as e^*, ensures that function arguments are always variables. It is defined as follows.

$$\begin{array}{lcl} (\lambda x.e)^* & = & \lambda x.(e^*) \\ x^* & = & x \\ \multicolumn{3}{l}{(\mathit{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathit{in}\ e)^*} \\ & = & \mathit{let}\ x_1 = (e_1^*), \ldots, x_n = (e_n^*)\ \mathit{in}\ (e^*) \\ (e_1\ e_2)^* & = & (e_1^*)\ e_2 \qquad \textbf{if } e_2 \textbf{ is a variable} \\ & = & \mathit{let}\ y = (e_2^*)\ \mathit{in}\ (e_1^*)\ y \quad \textbf{otherwise} \\ & & [y \text{ is a fresh variable}] \end{array}$$

Thus, apart from α-conversion, normalisation consists purely of naming the argument terms in applications, and expressing that naming using *let*.

This process of normalisation borrows heavily from the STG language [Pey92], which has an even more restricted form of application. The value of the STG language is its direct operational reading (though far less abstract than appearing here).

3.2 Dynamic Semantics

The rules are presented in Figure 1. They obey the following naming conventions:

$$\begin{array}{rcccl} \Gamma, \Delta, \Theta & \in & \mathit{Heap} & = & \mathit{Var} \nrightarrow \mathit{Exp} \\ z & \in & \mathit{Val} & ::= & \lambda x.e \end{array}$$

A heap is a partial function from variables to expressions. It may be viewed as an (unordered) set of variable/expression pairs, binding distinct variable names to expressions. A value is a expression in whnf, i.e. whose outermost structure is a lambda. As we see later, it causes no problems to add constants and constructors and to treat these as values also.

Judgements of the form $\Gamma : e \Downarrow \Delta : z$ are to be read, "the term e in the context of the set of bindings Γ reduces to the value z together with the (modified) set of bindings Δ." In the course of evaluation, new bindings may be added to the heap, and old bindings which bound variables to unevaluated terms may be updated to bind those variables to their evaluated counterparts.

A proof of a judgement corresponds to a reduction sequence. A proof may fail in one of two ways: either there may be no *finite* proof that a reduction is valid, which corresponds to an infinite loop, or there may be no rule which applies in (a sub-part of) the proof. which corresponds to a so-called *black hole*. Denotationally, each of these failures corresponds to a value $\perp$.

3.2.1 Reduction Rules

Referring to Figure 1, the *Lambda* rule states that terms whose outermost component is a lambda rewrite to themselves without affecting the heap. Such terms are in whnf so are already values and have no need of further evaluation.

The *Application* rule reduces the term to the left of the application (the function), substitutes the argument in for the λ-variable, and continues reduction. Simple substitution is sufficient because we only substitute a variable, so no duplication of work is incurred. This is the point of the static e^* transformation.

The payoff of the renaming transformation $\hat{e}$ appears in the *Let* rule. The bindings may be added to the heap with no worries about name clash.

The most intriguing rule is the *Variable* rule. This is where sharing is captured. To evaluate a variable x in the context of some heap, the heap must contain a binding of the form $x \mapsto e$. Assuming this is the case, e is reduced in the context of the heap *omitting the reference to* x. If this reduction produces a value z, the new heap is augmented with the binding $x \mapsto z$, and a

$$\mathit{\Gamma} : \lambda x.e \ \Downarrow\ \mathit{\Gamma} : \lambda x.e \qquad \textit{Lambda}$$

$$\frac{? : e \ \Downarrow\ \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \ \Downarrow\ \Theta : z}{? : e\ x \ \Downarrow\ \Theta : z} \qquad \textit{Application}$$

$$\frac{? : e \ \Downarrow\ \Delta : z}{(?, x \mapsto e) : x \ \Downarrow\ (\Delta, x \mapsto z) : \hat{z}} \qquad \textit{Variable}$$

$$\frac{(?, x_1 \mapsto e_1 \ \cdots\ x_n \mapsto e_n) : e \ \Downarrow\ \Delta : z}{? : \mathit{let}\ x_1 = e_1 \ \cdots\ x_n = e_n\ \mathit{in}\ e \ \Downarrow\ \Delta : z} \qquad \textit{Let}$$

Figure 1: Reduction Rules

renamed version of z is returned as the result. This is a natural place for renaming to occur, as it is only here that terms may be duplicated. As we will show later, this one occurrence of renaming is sufficient to avoid all unwanted name capture.

What if x is recursive, and e has a (possibly indirect) reference back to x? It may seem that reducing e in the context of a heap which contains no reference to x could cause a problem. There are two possibilities: either e reduces to whnf without requiring the value of x, in which case we reintroduce a binding for x (binding it to its *value* now), or else e requires the value of x before reducing to whnf. This means that x depends directly on itself before any value can be returned, so should have denotation $\perp$. In this latter case we will attempt to reduce x in a heap containing no reference to x. As there is no rule which covers this situation the proof for the reduction sequence fails. Note that the variable rule is the only place where the proof may actually fail[1].

4 Examples

To examine the behaviour of the λ-expressions presented in the introduction, we will need to make use of the following additional rules. They are discussed in more detail in Section 6.1.

$$? : n \ \Downarrow\ ? : n$$

$$\frac{? : e_1 \ \Downarrow\ \Delta : n_1 \qquad \Delta : e_2 \ \Downarrow\ \Theta : n_2}{? : e_1 + e_2 \ \Downarrow\ \Theta : n_1 + n_2}$$

To stress the sequential nature of reduction we lay proofs out vertically: if $\mathit{\Gamma} : e \ \Downarrow\ \mathit{\Delta} : z$ we write

$$\begin{array}{l} \mathit{\Gamma} : e \\ \left[\ \textit{a sub} \perp \textit{proof} \right. \\ \left[\ \textit{another sub} \perp \textit{proof} \right. \\ \mathit{\Delta} : z \end{array}$$

with sub-derivations contained within the square brackets. To see this notation in action consider reducing $\mathit{let}\ u = \mathit{3} + \mathit{2}, v = u + \mathit{1}\ \mathit{in}\ v + v$ in the context of an

[1]Once we add constants the *Application* rule could cause failure on a type-incorrect term.

empty heap.

$$\begin{array}{l}
\{\} : \mathit{let}\ u = 3+2, v = u+1\ \mathit{in}\ v+v \\
\{u \mapsto 3+2, v \mapsto u+1\} : v+v \\
\left[\begin{array}{l}
\{u \mapsto 3+2, v \mapsto u+1\} : v \\
\left[\begin{array}{l}
\{u \mapsto 3+2\} : u+1 \\
\left[\begin{array}{l}
\{u \mapsto 3+2\} : u \\
\left[\begin{array}{l}
\{\} : 3+2 \\
\left[\ \vdots\right. \\
\{\} : 5
\end{array}\right. \\
\{u \mapsto 5\} : 5
\end{array}\right. \\
\left[\begin{array}{l}
\{u \mapsto 5\} : 1 \\
\{u \mapsto 5\} : 1
\end{array}\right. \\
\{u \mapsto 5\} : 6
\end{array}\right. \\
\{u \mapsto 5, v \mapsto 6\} : 6
\end{array}\right. \\
\left[\begin{array}{l}
\{u \mapsto 5, v \mapsto 6\} : v \\
\{u \mapsto 5, v \mapsto 6\} : 6
\end{array}\right. \\
\{u \mapsto 5, v \mapsto 6\} : 12
\end{array}$$

The result is the number 12, together with a heap in which u is bound to 5 and v to 6.

The next two examples exhibit the difference between defining a closure inside a lambda, and outside. First inside: (as a shorthand we will write $f \mapsto \cdots$ for $f \mapsto \lambda x.\mathit{let}\ v = u+1\ \mathit{in}\ v+x$)

$$\begin{array}{l}
\{\} : \mathit{let}\ u = 3+2, f = \lambda x.\mathit{let}\ v = u+1\ \mathit{in}\ v+x \\
\quad \mathit{in}\ f\ 2 + f\ 3 \\
\{u \mapsto 3+2, f \mapsto \cdots\} : f\ 2 + f\ 3 \\
\left[\begin{array}{l}
\{u \mapsto 3+2, f \mapsto \cdots\} : f\ 2 \\
\left[\begin{array}{l}
\{u \mapsto 3+2, f \mapsto \cdots\} : f \\
\{u \mapsto 3+2, f \mapsto \cdots\} \\
\qquad : \lambda x.\mathit{let}\ s = u+1\ \mathit{in}\ s+x
\end{array}\right. \\
\{u \mapsto 3+2, f \mapsto \cdots\} \\
\qquad : \mathit{let}\ s = u+1\ \mathit{in}\ s+2 \\
\{u \mapsto 3+2, f \mapsto \cdots, s \mapsto u+1\} : s+2 \\
\left[\ \vdots\right. \\
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6\} : 8
\end{array}\right. \\
\left[\begin{array}{l}
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6\} : f\ 3 \\
\left[\begin{array}{l}
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6\} : f \\
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6\} \\
\qquad : \lambda x.\mathit{let}\ t = u+1\ \mathit{in}\ t+x
\end{array}\right. \\
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6\} \\
\qquad : \mathit{let}\ t = u+1\ \mathit{in}\ t+3 \\
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6, t \mapsto u+1\} : t+3 \\
\left[\ \vdots\right. \\
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6, t \mapsto 6\} : 9
\end{array}\right. \\
\{u \mapsto 5, f \mapsto \cdots, s \mapsto 6, t \mapsto 6\} : 17
\end{array}$$

Notice each time f is called, its body is copied and renamed. After application and substitution f's body generates a new closure in the heap bound to the computation $u+1$, so the value of $u+1$ is *not* shared across separate applications of f (though the computation $3+2$ is shared). Contrast this with the case where the *let* occurs ouside the lambda of f:

$$\begin{array}{l}
\{\} : \mathit{let}\ u = 3+2, f = \mathit{let}\ v = u+1\ \mathit{in}\ \lambda x.v+x \\
\quad \mathit{in}\ f\ 2 + f\ 3 \\
\{u \mapsto 3+2, f \mapsto \mathit{let}\ v = u+1\ \mathit{in}\ \lambda x.v+x\} : \\
\quad f\ 2 + f\ 3 \\
\left[\begin{array}{l}
\{u \mapsto 3+2, f \mapsto \mathit{let}\ v = u+1\ \mathit{in}\ \lambda x.v+x\} : f\ 2 \\
\left[\begin{array}{l}
\{u \mapsto 3+2, f \mapsto \mathit{let}\ v = u+1\ \mathit{in}\ \lambda x.v+x\} : f \\
\left[\begin{array}{l}
\{u \mapsto 3+2\} : \mathit{let}\ v = u+1\ \mathit{in}\ \lambda x.v+x \\
\{u \mapsto 3+2, v \mapsto u+1\} : \lambda x.v+x
\end{array}\right. \\
\{u \mapsto 3+2, v \mapsto u+1, f \mapsto \lambda x.v+x\} : \\
\quad \lambda x.v+x
\end{array}\right. \\
\{u \mapsto 3+2, v \mapsto u+1, f \mapsto \lambda x.v+x\} : v+2 \\
\left[\begin{array}{l}
\{u \mapsto 3+2, v \mapsto u+1, f \mapsto \lambda x.v+x\} : v \\
\left[\ \vdots\right. \\
\{u \mapsto 5, v \mapsto 6, f \mapsto \lambda x.v+x\} : 6
\end{array}\right. \\
\{u \mapsto 5, v \mapsto 6, f \mapsto \lambda x.v+x\} : 8
\end{array}\right. \\
\left[\begin{array}{l}
\{u \mapsto 5, v \mapsto 6, f \mapsto \lambda x.v+x\} : f\ 3 \\
\{u \mapsto 5, v \mapsto 6, f \mapsto \lambda x.v+x\} : v+3 \\
\{u \mapsto 5, v \mapsto 6, f \mapsto \lambda x.v+x\} : 9
\end{array}\right. \\
\{u \mapsto 5, v \mapsto 6, f \mapsto \lambda x.v+x\} : 17
\end{array}$$

This time the closure for v is loaded into the heap once only, and the binding for f is updated to its whnf. Thus the computation of $u+1$ is performed just once, and the result is shared across all uses of f.

4.1 Recursion

The simplest case of recursion (using *let*) is

$$\mathit{let}\ x = x\ \mathit{in}\ x$$

An attempt to reduce this reaches a point where no rule applies, so no progress may be made toward finding a proof of reduction.

$$\begin{array}{l}
\{\} : \mathit{let}\ x = x\ \mathit{in}\ x \\
\{x \mapsto x\} : x \\
\left[\begin{array}{l}
\{\} : x \\
\mathit{failure}
\end{array}\right.
\end{array}$$

Many run-time systems would halt at this point and report a black-hole (a detectably self-dependent infinite loop). In contrast, the loop defined by

$$\mathit{let}\ f = \lambda x.f\ x\ \mathit{in}\ f\ 2$$

is not detected as a black hole. Its "evaluation" proceeds as follows.

$$\begin{array}{l}
\{\} : \mathit{let}\ f = \lambda x.f\ x\ \mathit{in}\ f\ 2 \\
\{f = \lambda x.f\ x\} : f\ 2 \\
\left[\begin{array}{l} \{f = \lambda x.f\ x\} : f \\ \{f = \lambda x.f\ x\} : \lambda x.f\ x \end{array}\right. \\
\{f = \lambda x.f\ x\} : f\ 2 \\
\left[\begin{array}{l} \{f = \lambda x.f\ x\} : f \\ \{f = \lambda x.f\ x\} : \lambda x.f\ x \end{array}\right. \\
\vdots
\end{array}$$

This time there is always an applicable rule, so an attempt to reduce this term leads to an infinite proof. More operationally, because there is no single closure which ends up pointing directly to itself, the loop is not discovered. This sort of example provides intuition as to the relative benefits of alternative definitions of *fix*. The most direct definition is:

$$\mathit{let}\ \mathit{fix} = \lambda f.f\ (\mathit{fix}\ f)\ \mathit{in}\ \mathit{fix}$$

but better sharing is given by the alternative

$$\mathit{let}\ \mathit{fix} = \lambda f.(\mathit{let}\ x = f\ x\ \mathit{in}\ x)\ \mathit{in}\ \mathit{fix}$$

because a new cycle in the heap is created each time the second definition is used. Using this second definition of *fix*, the reduction of *fix id* ceases with a black hole, but using the first definition of *fix* leads to an infinite loop.

The section on extensions includes more examples of recursion.

5 Semantic Properties

5.1 Retaining Normalisation

Having gone to the bother of normalising terms before beginning reduction, we should check that the properties of terms introduced by normalisation, are preserved throughout a reduction proof. Preservation of the property that functions are only ever applied to variables is immediately obvious: because we only ever substitute variables for variables it is impossible to create a term in which an expression is applied to a non-variable. The naming property is less obvious and requires a more general definition.

Definition 1
A heap/term pair $\Gamma : e$ is *distinctly named* if every binding occurring in Γ and in e binds a distinct variable (which is also distinct from any free variables of $\Gamma : e$). □

By "binding occurring in Γ" we mean either top level bindings, or *let* or lambda bindings occurring within bound expressions. Only the latter two can occur within expressions.

Theorem 1
If $\Gamma : e \Downarrow \Delta : z$ and $\Gamma : e$ is distinctly named, then every heap/term pair occurring in the proof of the reduction is also distinctly named.

Proof
The rules for *Lambda* and *Let* are trivial. *Application* follows as soon as we recognise that if $\Delta : \lambda y.e'$ is distinctly named, then so is $\Delta : e'[x/y]$. The only rule that could cause a problem is the *Variable* rule, but even here if $\Delta : z$ is distinctly named then so is $(\Delta, x \mapsto z) : \hat{z}$, because by definition of renaming with fresh variable, $\hat{z}$ will only bind completely fresh variables. Note that Δ cannot contain a binding for x else $(\Gamma, x \mapsto e) : x$ would not have been distinctly named. □

In the light of this result we restrict the definition of $\Downarrow$ to apply solely to distinctly named heap/term pairs. For the rest of the paper a statement like $\Gamma : e \Downarrow \Delta : z$ carries the assumption that $\Gamma : e$ is distinctly named.

5.2 Relating to Denotational Semantics

5.2.1 Semantics of Terms

Following Abramsky and Ong [Abr90], the denotational semantics models functions by a lifted function space, so it distinguishes between a term Ω (a non-terminating computation) and $\lambda x.\Omega$. This distinction in the model reflects the fact that reduction ceases at whnf rather than head normal form (hnf). We represent lifting using the injection *Fn*, and the projection using $\downarrow_{Fn}$ (written as a postfix operator).

An environment is a function mapping variables into values,

$$\rho \in \mathit{Env} = \mathit{Var} \to \mathit{Value}$$

where *Value* is some appropriate domain containing at least a lifted version of its own function space. The initial "undefined" environment ρ_0 maps all variables to $\perp$.

Meanings are given to expressions using the semantic function $[\![\perp]\!] : \mathit{Exp} \to \mathit{Env} \to \mathit{Value}$ which is defined as follows.

$$\begin{array}{lcl}
[\![\lambda x.e]\!]_\rho & = & \mathit{Fn}\ (\lambda \nu.[\![e]\!]_{\rho \sqcup \{x \mapsto \nu\}}) \\
[\![e\ x]\!]_\rho & = & ([\![e]\!]_\rho) \downarrow_{Fn}\ ([\![x]\!]_\rho) \\
[\![x]\!]_\rho & = & \rho(x) \\
[\![\mathit{let}\ x_1 = e_1\ \cdots\ x_n = e_n\ \mathit{in}\ e]\!]_\rho & & \\
 & = & [\![e]\!]_{\{\!\{ x_1 \mapsto e_1\ \cdots\ x_n \mapsto e_n \}\!\}\rho}
\end{array}$$

The recursion generated by a *let* is captured through a recursively defined environment, given by the semantic function $\{\!\{\perp\}\!\} : \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$ defined as follows.

$$\begin{array}{l} \{\!\{ x_1 \mapsto e_1 \ \cdots \ x_n \mapsto e_n \}\!\}\rho \\ \quad = \quad \mu\rho'.\rho \sqcup (x_1 \mapsto \llbracket e_1 \rrbracket_{\rho'} \ \cdots \ x_n \mapsto \llbracket e_n \rrbracket_{\rho'}) \end{array}$$

where μ stands for the least fixed point operator. Because the bindings in the heap are (mutually) recursive, we obtain a recursively defined environment. Note that the definition only makes sense on environments ρ which are consistent with Γ (i.e. if ρ and Γ bind the same variable, then they are bound to values for which an upper bound exists).

The function $\{\!\{\perp\}\!\} : \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$ should be thought of as an environment modifier—it extends its environment argument by the meanings of the bindings given in its heap argument. This flexibility is useful for giving a semantics to the heap itself in a heap/term pair to allow for free variables. Note that the rules nowhere require the terms to be closed. Indeed, because of the variable rule, some reductions are bound to be of open terms: a variable is only rebound once the expression to which it was bound reduces to whnf.

An equivalent definition of the semantics for heaps is

$$\begin{array}{lcl} \{\!\{ \emptyset \}\!\}\rho & = & \rho \\ \{\!\{ \Gamma, x \mapsto e \}\!\}\rho & = & \mu\rho'.\{\!\{ \Gamma \}\!\}\rho' \sqcup (x \mapsto \llbracket e \rrbracket_{\rho'}) \sqcup \rho \end{array}$$

It is an easy consequence of this definition that, assuming Γ and ρ are consistent, $\forall x.\rho(x) \sqsubseteq (\{\!\{ \Gamma \}\!\}\rho)(x)$. This is because the heap only adds new bindings, or refines old ones. Using this fact, and by unfolding the recursion, we can show that,

$$\begin{array}{l} \mu\rho'.\{\!\{ ? \}\!\}\rho' \sqcup (x \mapsto \llbracket e \rrbracket_{\rho'}) \sqcup \rho \\ \quad = \quad \mu\rho'.\{\!\{ \Gamma \}\!\}\rho' \sqcup (x \mapsto \llbracket e \rrbracket_{\{\!\{ \Gamma \}\!\}\rho'}) \sqcup \rho \end{array}$$

We will use this fact in the proof of the variable case of the Correctness Theorem (Theorem 2).

We also define an ordering $\leq$ on environments, which captures the concept of "added bindings". We define $\rho \leq \rho'$ to mean

$$\forall x \ . \ \rho(x) \neq \perp \ \Rightarrow \ \rho(x) = \rho'(x)$$

So if $\rho \leq \rho'$ then ρ' may bind more variables than ρ, but otherwise is equal to ρ.

5.2.2 Correctness

We are now in a position to state and prove the correctness of the operational rules with respect to the denotational semantics. The correctness theorem states that reductions preserve the meanings of terms and only alter the meaning of heaps by (possibly) adding new bindings.

Theorem 2
If $\Gamma : e \ \Downarrow \ \Delta : z$ then for all environments ρ,

$$\llbracket e \rrbracket_{\{\!\{ \Gamma \}\!\}\rho} = \llbracket z \rrbracket_{\{\!\{ \Delta \}\!\}\rho} \ \wedge \ \{\!\{ \Gamma \}\!\}\rho \leq \{\!\{ \Delta \}\!\}\rho$$

Proof
The proof is by induction on the structure of the derivation $\Gamma : e \ \Downarrow \ \Delta : z$. There are four cases depending on the form of e:
Case: $\lambda x.e$
This is immediate.
Case: $e \ x$
The first part is a direct calculation.

$$\begin{array}{ll} \multicolumn{2}{l}{\llbracket e \ x \rrbracket_{\{\!\{ \Gamma \}\!\}\rho}} \\ = & (\llbracket e \rrbracket_{\{\!\{ \Gamma \}\!\}\rho}) \downarrow_{Fn} \ (\llbracket x \rrbracket_{\{\!\{ \Gamma \}\!\}\rho}) \\ = & (\llbracket \lambda y.e' \rrbracket_{\{\!\{ \Delta \}\!\}\rho}) \downarrow_{Fn} \ (\llbracket x \rrbracket_{\{\!\{ \Delta \}\!\}\rho}) \\ & \quad \text{[Induction]} \\ = & (\lambda\nu.\llbracket e' \rrbracket_{\Delta\rho \sqcup \{y \mapsto \nu\}}) \ (\llbracket x \rrbracket_{\{\!\{ \Delta \}\!\}\rho}) \\ = & \llbracket e' \rrbracket_{\Delta\rho \sqcup \{y \mapsto \llbracket x \rrbracket_{\{\!\{ \Delta \}\!\}\rho}\}} \\ = & \llbracket e'[x/y] \rrbracket_{\Delta\rho} \\ = & \llbracket z \rrbracket_{\Theta\rho} \qquad \text{[Induction]} \end{array}$$

The second part follows from transitivity of $\leq$.
Case: x
The variable rule is only applicable if there is a binding for the variable in the heap, so the reduction is of the form: $(\Gamma, x \mapsto e) : x \ \Downarrow \ (\Delta, x \mapsto z) : \hat{z}$. By induction we may assume that $\llbracket e \rrbracket_{\{\!\{ \Gamma \}\!\}\rho} = \llbracket z \rrbracket_{\{\!\{ \Delta \}\!\}\rho}$ and that $\{\!\{ \Gamma \}\!\}\rho \leq \{\!\{ \Delta \}\!\}\rho$. We are required to prove that, $\llbracket x \rrbracket_{\{\!\{ \Gamma, x \mapsto e \}\!\}\rho} = \llbracket \hat{z} \rrbracket_{\{\!\{ \Delta, x \mapsto z \}\!\}\rho}$ and that $\{\!\{ \Gamma, x \mapsto e \}\!\}\rho \leq \{\!\{ \Delta, x \mapsto z \}\!\}\rho$. We shall do the second first.

$$\begin{array}{ll} \multicolumn{2}{l}{\{\!\{ ?, x \mapsto e \}\!\}\rho} \\ = & \mu\rho' \ . \ \{\!\{ \Gamma \}\!\}\rho' \ \sqcup \ (x \mapsto \llbracket e \rrbracket_{\rho'}) \ \sqcup \ \rho \\ = & \mu\rho' \ . \ \{\!\{ \Gamma \}\!\}\rho' \ \sqcup \ (x \mapsto \llbracket e \rrbracket_{\{\!\{ \Gamma \}\!\}\rho'}) \ \sqcup \ \rho \\ = & \mu\rho' \ . \ \{\!\{ \Gamma \}\!\}\rho' \ \sqcup \ (x \mapsto \llbracket z \rrbracket_{\{\!\{ \Delta \}\!\}\rho'}) \ \sqcup \ \rho \\ & \quad \text{[Induction]} \\ \leq & \mu\rho' \ . \ \{\!\{ \Delta \}\!\}\rho' \ \sqcup \ (x \mapsto \llbracket z \rrbracket_{\{\!\{ \Delta \}\!\}\rho'}) \ \sqcup \ \rho \\ \leq & \mu\rho' \ . \ \{\!\{ \Delta \}\!\}\rho' \ \sqcup \ (x \mapsto \llbracket z \rrbracket_{\rho'}) \ \sqcup \ \rho \\ = & \{\!\{ \Delta, x \mapsto z \}\!\}\rho \end{array}$$

Using this, we can now show the first part.

$$\begin{array}{lll} \multicolumn{3}{l}{\llbracket x \rrbracket_{\{\!\{ \Gamma, x \mapsto e \}\!\}\rho}} \\ = & \{\!\{ \Gamma, x \mapsto e \}\!\}\rho \ (x) & \\ = & \{\!\{ \Delta, x \mapsto z \}\!\}\rho \ (x) & \text{[defn of } \leq\text{]} \\ = & \llbracket z \rrbracket_{\{\!\{ \Delta, x \mapsto z \}\!\}\rho} & \text{[defn of } \{\!\{\perp\}\!\}\text{]} \\ = & \llbracket \hat{z} \rrbracket_{\{\!\{ \Delta, x \mapsto z \}\!\}\rho} & [\alpha\text{-conversion]} \end{array}$$

Case: $\mathit{let} \ x_1 = e_1, \ldots, x_n = e_n \ \mathit{in} \ e$
For the first part,

$$\begin{array}{ll} \multicolumn{2}{l}{\llbracket \mathit{let} \ x_1 = e_1, \ldots, x_n = e_n \ \mathit{in} \ e \rrbracket_{\{\!\{ \Gamma \}\!\}\rho}} \\ = & \llbracket e \rrbracket_{\mu\rho'.\{\!\{ \Gamma \}\!\}\rho \sqcup (x_1 \mapsto \llbracket e_1 \rrbracket_{\rho'} \cdots x_n \mapsto \llbracket e_n \rrbracket_{\rho'})} \\ = & \llbracket e \rrbracket_{\{\!\{ \Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \}\!\}\rho} \\ & \quad \text{[Variables } x_1, \ldots, x_n \text{ not bound in ?]} \\ = & \llbracket z \rrbracket_{\{\!\{ \Delta \}\!\}\rho} \qquad \text{[Induction]} \end{array}$$

as required. For the second part,

$$\begin{array}{lll} \{\!\{ \Gamma \}\!\} \rho & \leq & \{\!\{ \Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \}\!\} \rho \\ & & \quad [\text{Variables } x_1, \ldots, x_n \text{ not bound in } ?] \\ & \leq & \{\!\{ \Delta \}\!\} \rho \qquad [\text{Induction}] \end{array}$$

□

5.2.3 Computational Adequacy

Having proved that when reductions exist they preserve the denotational semantics, we must now characterise *when* reduction exist.

Theorem 3
$[\![e]\!]_{\{\!\{ \Gamma \}\!\} \rho_0} \neq \bot \ \Leftrightarrow \ (\exists \Delta, z \ . \ \Gamma : e \ \Downarrow \ \Delta : z)$

The theorem states that a heap/term pair reduces exactly when its denotation is non-bottom (in the initial environment ρ_0).

The one direction is easy to show as it arises as a corollary to Theorem 2.

Theorem 4
$\Gamma : e \ \Downarrow \ \Delta : z \ \Rightarrow \ [\![e]\!]_{\{\!\{ \Gamma \}\!\} \rho_0} \neq \bot$

Proof
The term value z can only be of the form $\lambda x.e'$. Then

$$\begin{array}{lll} [\![e]\!]_{\{\!\{ \Gamma \}\!\} \rho_0} & = & [\![\lambda x.e']\!]_{\{\!\{ \Delta \}\!\} \rho_0} \\ & = & Fn \ (\lambda \nu . [\![e']\!]_{\{\!\{ \Delta \}\!\} \rho_0 \sqcup \{x \mapsto \nu\}}) \\ & \neq & \bot \end{array}$$

□

The other direction (that is, if the denotational semantics is non-bottom then the natural semantics has a valid reduction) is harder to show and requires a number of intermediate steps. As they stand, the denotational and natural semantics formulations are too far apart to be related directly. To remedy this we introduce a new version of each which can then be directly related to each other.

We begin with the denotational semantics, and define a *resourced* denotational semantics. The semantic function takes as an extra argument an element of the countable chain domain C defined as the least solution to the domain equation $C = C_\bot$. We represent lifting in C by the injection function $S : C \to C$. Thus the elements of C are $\bot$, $S \ \bot$, $S(S \ \bot)$, and so on, with limit element $S(S(S \ \cdots))$ which we write as ω.

The type of the environment is given by $\sigma : Var \to (C \to Val)$. That is, variables are bound to functions which, when provided with a resource, yield a value.

$$\begin{array}{lll} \mathcal{N}[\![e]\!]_\sigma \ \bot & = & \bot \\ \mathcal{N}[\![\lambda x.e]\!]_\sigma \ (S \ k) & = & Fn \ (\lambda \tau . \mathcal{N}[\![e]\!]_{\sigma \sqcup \{x \mapsto \tau\}}) \\ \mathcal{N}[\![e \ x]\!]_\sigma \ (S \ k) & = & (\mathcal{N}[\![e]\!]_\sigma \ k) \downarrow_{Fn} \ (\mathcal{N}[\![x]\!]_\sigma) \ k \\ \mathcal{N}[\![x]\!]_\sigma \ (S \ k) & = & \sigma \ x \ k \\ \multicolumn{3}{l}{\mathcal{N}[\![let \ x_1 = e_1 \ \cdots \ x_n = e_n \ in \ e]\!]_\sigma \ (S \ k)} \\ \multicolumn{3}{l}{\quad = \ \mathcal{N}[\![e]\!]_{\mu \sigma' . (\sigma \sqcup x_1 \mapsto \mathcal{N}[\![e_1]\!]_{\sigma'} \sqcup \cdots \sqcup x_n \mapsto \mathcal{N}[\![e_n]\!]_{\sigma'})} \ k} \end{array}$$

This resourced semantics equals the original semantics if given infinite resources. That is, if $\forall x . \rho \ x = \sigma \ x \ \omega$ then

$$[\![e]\!]_\rho = \mathcal{N}[\![e]\!]_\sigma \ \omega$$

Put the other way around, the resourced semantics allows us to focus on approximations to the original denotational semantics. In particular, as the resourced semantics is a continuous function (being defined using continuous operations only), then if the original semantics assigns a non-bottom value to some term, so does some finite approximation. This provides the proof of the following lemma.

Lemma 5
If $[\![e]\!]_\rho \neq \bot$, then there exists a natural number m such that $\mathcal{N}[\![e]\!]_\sigma \ (S^m \bot) \neq \bot$ where $\forall x \ . \ \rho \ x = \sigma \ x \ \omega$.

Returning to our overall goal, we now have to work from the other direction. We define an alternative natural semantics in which *Application* and *Variable* rules are replaced with the following alternatives:

$$\frac{? : e \ \Downarrow \ \Delta : \lambda y.e' \quad (\Delta, y \mapsto x) : e' \ \Downarrow \ \Theta : z}{? : e \ x \ \Downarrow \ \Theta : z} \ App$$

$$\frac{(?, x \mapsto e) : \hat{e} \ \Downarrow \ \Delta : z}{(?, x \mapsto e) : x \ \Downarrow \ \Delta : z} \ Var$$

The only effect of the new application rule is to increase the number of closures by adding indirections whenever a lambda is reduced, rather than by substituting the new value for the bound variable. It mimics the operation on the environment in the denotational semantics more closely than the original version did.

The effect of the new variable rule is to remove updating from the semantics, and to remove the possibility of detecting black holes (the only way a reduction proof may fail in the revised system is by being infinite).

Apart from these changes the two versions of the natural semantics are equivalent: they both respect the denotational semantics and, furthermore, the same heap/term pairs reduce successfully. This may be shown by induction on the reduction proofs.

Finally, the last link needed is to relate the resourced semantics to the alternative version of the natural semantics. This is done by the next lemma.

Lemma 6
If $\mathcal{N}[\![e]\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma \sqcup \cdots \sqcup x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma)}\ (S^m \bot) \neq \bot$, then there exists a heap Δ and a value z such that $(x_1 \mapsto e_1 \ \cdots\ x_n \mapsto e_n) : e \Downarrow \Delta : z$ in the alternative version of the natural semantics.

Proof
By induction on m. □

At last we are in a position where we can put the pieces together and prove the second part of the computational adequacy theorem.

Theorem 7
$[\![e]\!]_{\{\!\{ \Gamma \}\!\}\rho_0} \neq \bot \ \Rightarrow\ (\exists \Delta, z\ .\ \Gamma : e \Downarrow \Delta : z)$

Proof
If the original semantics is non-bottom, then by Lemma 5 there exists an m such that

$$\mathcal{N}[\![e]\!]_{\mu\sigma.(x_1 \mapsto \mathcal{N}[\![e_1]\!]_\sigma\ \cdots\ x_n \mapsto \mathcal{N}[\![e_n]\!]_\sigma)}\ (S^m \bot) \neq \bot$$

However, by Lemma 6 this implies a reduction proof in the alternative version of the natural semantics, which in turn implies a reduction proof in the original version.
□

6 Extensions

Now that we have defined the semantics and shown it to be correct the obvious response is, "so what?" In the example section we have already seen the value of the semantics for demonstrating when closures are built, updated, and accessed, and when computations are repeated or shared. While this is an important use of the semantics it is not the only one.

One of our aims is to provide a high level base to which various gadgets may be added and studied in the context of lazy evaluation. In this section we look at a number of possibilities. It does not constitute anything like a detailed examination of these areas; rather its purpose is to demonstrate potential.

6.1 Constructors and Constants

Adding extra constructs to the language causes no real difficulty. For example, to extend the language with constructors we would add the following syntactic forms:

$$\begin{array}{lllll} c & \in & \mathit{Constructor} & & \\ e & \in & \mathit{Exp} & ::= & c\ x_1 \cdots x_m \\ & & & | & \mathit{case}\ e\ \mathit{of}\ \{c_i\ y_1 \cdots y_{m_i} \to e_i\}_{i=1}^{n} \\ & & & & \vdots \end{array}$$

As before, we expect arbitrary terms to be statically normalised in order not to clutter the reduction rules. Constructors are like functions, so they are only to be applied to variables and, furthermore, they should be *saturated*, that is, fully applied (by introducing new λ-bound variables if necessary).

Adding numbers likewise requires the following.

$$\begin{array}{lllll} n & \in & \mathit{Number} & & \\ \oplus & \in & \mathit{Primitive} & & \\ e & \in & \mathit{Exp} & ::= & n \\ & & & | & e_1 \oplus e_2 \\ & & & & \vdots \end{array}$$

Numbers may be viewed as nullary constructors, so really *Numbers* $\subseteq$ *Constructor*. We assume the operators $\oplus$ (e.g. $+$, $*$, $>$, etc.) are strict in both arguments, and will produce a nullary constructor such as a number, or a boolean as a result. If this is the case, then $\oplus$ operations require no special treatment during normalisation.

Figure 2 shows the rules for these new constructs. Constructors (including numbers) immediately evaluate to themselves. Primitive operations are evaluated by evaluating the left operand, then the right, and then by carrying out the appropriate operation on the resulting values.

The *Case* rule looks complicated purely because of the subscripts needed to express the flexibility of the construct which allows any number of cases, each constructor having its own number of arguments. The rule only succeeds if the constructor returned by the evaluation of e is contained in the case list. If so, then again simple substitution of the arguments of the constructor for the formal variables is sufficient: normalisation ensures that these are variables so no sharing is lost.

Constructors provide a very simple example of sharing in recursively defined structures. Taking the example from the discussion of related work:

$$\mathit{let}\ u = \mathit{False}, t = \mathit{if}\ u\ \mathit{then}\ \mathit{Nil}\ \mathit{else}\ \mathit{Cons}\ 1\ t\ \mathit{in}\ t$$

(where the *if* construct may be thought of as syntactic sugar for a *case* over booleans). This reduces as follows.

$$\begin{array}{l} \{\} : \mathit{let}\ u = \mathit{False}, t = \mathit{if}\ u\ \mathit{then}\ \mathit{Nil}\ \mathit{else}\ \mathit{Cons}\ 1\ t \\ \quad \mathit{in}\ t \\ \{u \mapsto \mathit{False}, t \mapsto \mathit{if}\ u\ \mathit{then}\ \mathit{Nil}\ \mathit{else}\ \mathit{Cons}\ 1\ t\} : t \\ \left[\begin{array}{l} \{u \mapsto \mathit{False}\} : \mathit{if}\ u\ \mathit{then}\ \mathit{Nil}\ \mathit{else}\ \mathit{Cons}\ 1\ t \\ \left[\begin{array}{l} \{u \mapsto \mathit{False}\} : u \\ \{u \mapsto \mathit{False}\} : \mathit{False} \end{array} \right. \\ \{u \mapsto \mathit{False}\} : \mathit{Cons}\ 1\ t \end{array} \right. \\ \{u \mapsto \mathit{False}, t \mapsto \mathit{Cons}\ 1\ t\} : \mathit{Cons}\ 1\ t \end{array}$$

The closure for t has been updated to whnf, and u will never be accessed again.

$$\Gamma : c\ x_1 \cdots x_n \ \Downarrow\ \Gamma : c\ x_1 \cdots x_n \qquad \textit{Constructors}$$

$$\frac{? : e \ \Downarrow\ \Delta : c_k\ x_1 \cdots x_{m_k} \qquad \Delta : e_k[x_i/y_i]_{i=1}^{m_k} \ \Downarrow\ \Theta : z}{? : \mathit{case}\ e\ \mathit{of}\ \{c_i\ y_1 \cdots y_{m_i} \to e_i\}_{i=1}^{n} \ \Downarrow\ \Theta : z} \qquad \textit{Case}$$

$$\frac{? : e_1 \ \Downarrow\ \Delta : n_1 \qquad \Delta : e_2 \ \Downarrow\ \Theta : n_2}{? : e_1 \oplus e_2 \ \Downarrow\ \Theta : n_1 \oplus n_2} \qquad \textit{Primitive}$$

Figure 2: Additional Reduction Rules

6.2 Garbage Collection

It is perhaps slightly surprising that a notion of garbage collection could find its way into the semantics. In retrospect, however, it is vital that it does so if we ever wish to study the space behaviour of lazy programs at a higher level of abstraction than provided by abstract machines. When a term is said to evaluate in constant space under lazy evaluation, it only does so if discarded cells are reclaimed. Thus we need a rule for garbage collection which may be applied at any time, and which will discard from the heap any unnecessary closures.

To do this we need to augment the $\Downarrow$ relation with a set of "active" names, that is, variables still possibly needed in the reduction but not necessarily pointed to from the expression currently under reduction. The only rule where this shows up non-trivially is the *Application* rule which becomes,

$$\frac{? : e \ \Downarrow_{N \cup \{x\}}\ \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \ \Downarrow_N\ \Theta : z}{? : e\ x \ \Downarrow_N\ \Theta : z}$$

Then one possible rule for garbage collection is,

$$\frac{? : e \ \Downarrow_N\ \Delta : z}{(?, x \mapsto e') : e \ \Downarrow_N\ \Delta : z} \quad \textbf{if } x \notin \mathcal{R}(\Gamma, e, N)$$

where $\mathcal{R}(\Gamma, e, N)$ is the set of variables reachable from e or N via Γ. There are clearly many ways in which this function may be defined, the obvious one corresponding to the usual method of marking variables, but reference counting is another (possibly giving an over approximation with cyclic structures).

Note that the garbage collection rule does not preserve the semantics of heaps, so a proof showing the equivalence of the system with and without garbage collection is required.

Once we can specify garbage collection like this, we have the opportunity to explore alternative methods (perhaps generational or parallel methods) of collecting and/or marking. The greater level of abstraction means that the results are not tied to one particular abstract machine.

6.3 Counting the Cost

In the same way that the reduction rules were augmented to compute extra information for garbage collection, a similar mechanism could be developed for recording the cost of computation. For example, the *Application* rule might become,

$$\frac{? : e \ \Downarrow_p\ \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \ \Downarrow_q\ \Theta : z}{? : e\ x \ \Downarrow_{p+q+1}\ \Theta : z}$$

Now the subscripts indicate how many reduction steps were performed. This may prove to be a useful route for formalising the notion of *cost centres* [SP92], and is the topic of current work.

6.4 Abstractions and Analyses

One motivation for this work was a desire to avoid unnecessary updates [Lau92]. Previously we had no good semantics against which to prove our analysis correct. Now that we have such a semantics we can not only hope to be able to verify our existing analysis, but can also expect that abstracting the semantics will lead to other (perhaps weaker and cheaper) forms of the analysis.

7 Conclusion

In this paper we have presented a natural semantics which models lazy evaluation as it is commonly implemented. The model works on the level of terms, using a heap to capture sharing. The result is remarkably simple, especially when compared with other attempts.

The semantics doesn't directly provide a specification for an abstract machine because there is no notion of explicit control. It seems as though this aspect is the essential cause of the wide diversity of the many abstract machines, so its omission from the semantics means that there is a reasonable hope for it to provide a basis for studying a broad spread of implementations.

Finally it is worth commenting on the presence of *let*s. It turns out that much of the work can be done without them. Indeed, by replacing the application rule with the one used in the proof of computational adequacy even the lifting out of arguments can be omitted as sharing is captured by the addition of the indirection. However, *let*s are irreplaceable in lazy functional languages in that they may create cyclic structures in the heap. This cannot be achieved without them (or without something essentially equivalent), and cyclic structures have an huge implication for the amount of computation that may be performed.

8 Acknowledgements

Having insightful colleagues is a great boon, and I would like to thank Cordelia Hall, John Hughes, Andy Moran, Simon Peyton Jones and Philip Wadler for excelling in that capacity.

References

[Abr90] S.Abramsky, *The Lazy Lambda Calculus*, in D.Turner ed., Declarative Programming, Addison-Wesley, 1990.

[AA91] Z.Ariola and Arvind, *A Syntactic Approach to Program Transformations*, in Proc. SIGPLAN PEPM 91, New Haven, pp 116-129, 1991.

[ANP89] Arvind, R.Nikhil and K.Pingali, *I-Structures: Data Structures for Parallel Computing*, in TOPLAS (11) 4 pp 598-632, Oct 1989.

[FW87] J.Fairbairn and S.Wray, *A Simple Lazy Abstract-Machine to Execute Supercombinators* , in Proc. FPCA, Portland, pp 34-45, S-V, 1987.

[Jon84] T.Johnsson, *Efficient Compilation of Lazy Evaluation*, in Proc. SIGPLAN Symp. on Compiler Construction, SIGPLAN Notices 19 pp 58-59, 1984.

[Jos89] M.Josephs, *The Semantics of Lazy Functional Languages*, in TCS 68, pp 105-111, 1989.

[KL89] P.Koopman and P.Lee, *A Fresh Look at Combinator Graph Reduction*, in SIGPLAN PLDI 89, Portland, pp 110-119, 1989.

[Lau92] J.Launchbury, A.Gill, J.Hughes, S.Marlow, S.Peyton Jones and P.Wadler, *Avoiding Unnecessary Updates*, Glasgow Functional Programming Workshop, Ayr, (draft proceedings), 1992.

[Lév80] J.-J.Lévy, *Optimal Reductions in the Lambda Calculus*, in Seldin and Hindley eds., *To H.B.Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pp 159-191, Academic Press, 1980.

[Mar91] L.Maranget, *Optimal Derivations in Weak Lambda-calculi and in Orthogonal Term Rewriting Systems*, in Proc SIGPLAN POPL 91, Orlando, pp 255-269, 1991.

[Ong88] C.-H.L.Ong, *The Lazy Lambda Calculus: An Investigation in the Foundations of Functional Programming*, PhD Thesis, Imperial College, London, 1988.

[Pey92] S.Peyton Jones, *Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine*, Journal of Functional Programming, CUP, 1992, to appear.

[PL91] S.Peyton Jones and D.Lester, *A Fully-Lazy Lambda-Lifter in Haskell*, Software Practice and Experience, 21 (5), pp 479-506, 1991.

[PS92] S.Purushothaman and J.Seaman, *An Adequate Operational Semantics of Sharing in Lazy Evaluation*, in Proc ESOP 92, Rennes, S-V, 1992.

[SP92] P.Sansom and S.Peyton Jones, *Profiling Lazy Functional Languages*, Glasgow Functional Programming Workshop, Ayr, (draft proceedings),1992.