The Lazy Lambda Calculus with Constants

Anthony H. Dekker

Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511
e-mail: `dekker@ACM.org`

July 22, 1994

Abstract

In this paper we define the Lazy Lambda Calculus with constants, which extends Abramsky's pure lazy Lambda Calculus. This calculus forms a model for modern lazy functional programming languages. Such languages usually provide a call-by-value facility which is able to distinguish between the values $\perp$ and $\lambda x.\perp$. We study the operational and denotational semantics of this calculus both with and without a superimposed type inference system. Finally we compare the operational and denotational semantics, and give a limited full-abstraction result based on that of Plotkin.

1 Introduction

In this paper we present the *Lazy Lambda Calculus with constants*, which we assert forms a genuine theoretical basis for modern lazy functional programming languages. It is derived from the Lazy Lambda Calculus [1, 14] which incorporates the idea that evaluation of a function terminates with an expression of the form $\lambda x.e$ (a *weak head normal form*). It is true that in pure lazy functional programming languages $\perp$ and $\lambda x.\perp$ have exactly the same behaviour, but most lazy functional programming languages also provide a call-by-value (or strict function) facility which distinguishes between these two values. This makes the Lazy Lambda Calculus the only possible model. In this work we extend the Lazy Lambda Calculus with natural numbers and arithmetic. Consequently we will be interested not only if evaluation of a term terminates, but with what *value* it terminates. A combinatory logic version of this calculus was presented in [6].

It is true that natural numbers can be modelled in the Lambda Calculus using Church Numerals [9, p 48], but this does not give an accurate representation of arithmetic in the Lazy Lambda Calculus. In particular, with:

$$\begin{aligned} \bar{n} &\equiv \lambda f.\lambda x.f^n x \\ + &\equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \\ Y &\equiv (\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy)) \end{aligned}$$

we obtain:

$$\begin{array}{rcl} Y(+\bar{1}) & \to^* & (+\bar{1})(Y(+\bar{1})) \\ & \to^* & \lambda f.\lambda x.\bar{1}f(Y(+\bar{1})fx) \end{array}$$

which is in weak head normal form, although in a lazy functional programming language we would expect to have $Y(+1)$ correspond to a nonterminating computation (i.e. $\perp$).

The difficulty is that arithmetic operators are naturally strict, while β-substitution is naturally lazy. This distinction is a common feature of modern lazy functional programming language implementations, which often involve two different mechanisms, e.g. a heap and a value stack to handle lazy β-substitution and strict arithmetic [8, 10, 15, 19]. However, the theoretical basis for these implementations has been neglected until now, because of the added complexity of the required rewrite rules, and most research has concentrated on typed or pure untyped lambda calculi. Abramsky and Ong [2] briefly discuss adding a single constant to the Lazy Lambda Calculus, but this lacks the complexity (and the usefulness) of adding numbers. The *denotational semantics* of the Lazy Lambda Calculus extended with some kind of primitive values is, however, well understood, since taking a separated sum in the domain equation ($\mathbf{E} \cong \mathbf{A} + [\mathbf{E} \to \mathbf{E}]$) naturally distinguishes $\perp$ and $\lambda x.\perp$. This is discussed in e.g. Wadsworth [23, p 518] and Stoy [20, pp 156–176].

2 Terms

The Lazy Lambda Calculus with constants consists of the following terms:

• variables	$x, y, z, \ldots$
• applications	$e_1 e_2$
• abstractions	$\lambda x.e$
• numbers	$i, j, k \in 0, 1, 2, \ldots$
• run-time error	W
• strict constants	$b \in C, P, E, P_i, E_i$ for $i \in 0, 1, 2, \ldots$

We refer to W and the strict constants together as *constants*, and constants and numbers together as *atoms*. We denote atoms using a, with possible subscripts and superscripts. The constants P (plus) and E (equals) are used for arithmetic, with the constants P_i and E_j representing the result of curried application of those operators. The result of equality is a boolean value represented by a lambda term in the usual way. The constants P and E are sufficient to define all partial recursive functions, but additional arithmetic operators could be defined in a similar way. The constants C, P, E, P_i, E_j are *strict* in that they only rewrite when their argument is an abstraction or atom.

The constant W is used as the result of run-time errors, which become possible as soon as numbers and functions are mixed in a reduction system. The constant C is essentially the same as that of Ong [14, p 134] or the *seq* operator of Miranda[1]

[1]Miranda is a trademark of Research Software Limited

[17], with the rewrite rule:

$$C e \to \lambda x.x \quad \text{for } e \text{ an abstraction, or } e \neq W \text{ an atom}$$

This operator is definable in any lazy functional programming language which provides a call-by-value facility (it is obtained by making $\lambda x.\lambda y.y$ strict on its first argument). Indeed, the work reported in this paper was motivated by the need for a formal semantics of the *seq* operator [21].

We use $\equiv$ for syntactic equality of terms (modulo a change of bound variables), and write $\vec{e}$ for a vector of terms $e_1 e_2 \ldots e_n$ with length $n \geq 0$. A term is *open* if it contains free variables, and *closed* otherwise. The set of free variables of a term e is denoted $FV(e)$. Following Klop [11, p 122] we consider $\lambda x.e$ as shorthand for $\lambda([x]e)$, where $[x]$ is a variable binding construct, and λ is a special constant. This makes it possible to use $\lambda x.e$ as a pattern in the left-hand side of a rewrite rule, by matching on the presence of the constant λ.

3 Rewrite Rules

The rewrite rules for the Lazy Lambda Calculus with constants are as follows:

- β (β-reduction) $\quad (\lambda x.e)\, e' \to e\,[e'/x]$
- W (error-propagation) $\quad W e \to W$
- ν (number-swap) $\quad i\, e \to e\, i$
- δ (strict-constant) $\quad b\, e \to e'$ as given in the table below

For each strict constant b, there are an infinite number of rewrite rules of the form $b\, e \to e'$, where e is an atom or abstraction (one rule for each combination of i and j). The contractum e' for each combination of b and e is given by the following table. Note that e' is an atom or closed abstraction, in every case:

b	**e**				
	i	$j \neq i$	b'	$\lambda x.e''$	W
P	P_i	P_j	W	W	W
E	E_i	E_j	W	W	W
P_i	$i+i$	$i+j$	W	W	W
E_i	$\lambda x.\lambda y.x$	$\lambda x.\lambda y.y$	W	W	W
C	$\lambda x.x$	$\lambda x.x$	$\lambda x.x$	$\lambda x.x$	W

Each of the four possible redexes $(\lambda x.e)\, e'$, $W e$, $i\, e$, or $b\, e$ begins with an atom (the *redex head symbol*) i.e. λ, W, i or b. As usual, we write $e \to e'$ (*single-step reduction*) if e_1 is a subterm of e, $e_1 \to e_2$ is an instance of a rewrite rule, and e' is the result of replacing e_1 in e by e_2. In the case that $e_1$is the leftmost redex in e, we write $e \to_{lf} e'$. We write $\to^*$ (*multi-step reduction*) for the transitive reflexive closure of $\to$, $=$ (*convertibility*) for the transitive, symmetric, and reflexive closure of $\to$, and $\to_{lf}^*$ for the transitive reflexive closure of $\to_{lf}$. A term which contains no redex is said to be in *normal form*. We also define:

$$\begin{aligned}
e \downarrow e' &\iff e \to^* e' \text{ and } e' \text{ is in normal form} \\
e \downarrow &\iff e \downarrow e' \text{ for some } e' \\
e \uparrow &\iff \sim e \downarrow
\end{aligned}$$

If $e \downarrow e'$, we say e' *is a normal form of* e.

The number-swap rule ν allows us to write addition $(e_1 + e_2)$ as $e_2(e_1 P)$, since if $e_1 \rightarrow^* i$ and $e_2 \rightarrow^* j$, then:

$$e_2(e_1 P) \rightarrow^* j(e_1 P) \rightarrow e_1 P j \rightarrow^* i P j \rightarrow P i j \rightarrow P_i j \rightarrow i + j$$

If P (and similarly E, P_i, E_i) are only introduced in terms of this form, it can be guaranteed that lazy evaluation will evaluate e before producing terms of the form Pe. This forms the basis for the handling of arithmetic in the TIM abstract machine [8]. However, there is no such mechanism for 'pre-evaluating' the argument to C. Another consequence of the ν rule is that:

$$00 \rightarrow 00$$

i.e. 00 (like $(\lambda x.xx)(\lambda x.xx)$ or $Y(\lambda x.x)$) has an infinite reduction sequence. We define Ω as shorthand notation for $Y(\lambda x.x)$.

The reduction system defined by these rules is a combinatory reduction system [11, p 120] and furthermore is left-linear (metavariables e, e', e'' do not occur multiple times in the left-hand side of any rule), non-ambiguous (the rules do not interfere with each other), and left-normal (in every left-hand-side the atoms precede the metavariables) [11, pp 126,130,189].

Proposition 3.1 (Klop) *The reduction system defined above has the following properties:*

(i) It is Church-Rosser, i.e. if $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$ then $e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$ for some e'.

(ii) The normal form of a term is unique if it exists, i.e. if $e \downarrow e'$ and $e \downarrow e''$, then $e' \equiv e''$.

(iii) Leftmost reduction leads to the normal form, i.e. if $e \downarrow e'$, then $e \rightarrow^*_{lf} e'$.

Proof:

(i) In Klop [11, p 163].

(ii) From (i).

(iii) In Klop [11, p 194].

□

Proposition 3.2 *If e is in normal form, then e has one of the following forms 1–4. If e is closed and in normal form, then e has the form a or $\lambda x.e$.*

1. a for a an atom

2. $\lambda x.e$

3. $x\vec{e}$

4. $b_n(\ldots(b_1(x\vec{e}_0)\vec{e}_1)\ldots)\vec{e}_n \quad \text{for } n \geq 1$

Proof: By considering cases. $\Box$

We say a term is in *weak head normal form* (whnf) if it is in one of the forms 1–4 of Proposition 3.2. We say that a term of the form (3) or (4) is *blocked on* the variable x.

Proposition 3.3 *If e is in one of the weak head normal forms 2–4, and $e \to e'$ then e' is of the same form, and corresponding sub-vectors have the same length, i.e:*

(i) If $\lambda x.e \to e'$ then $e' \equiv \lambda x.e''$ for some e''..

(ii) If $xe_1 \ldots e_n \to e'$ then $e' \equiv xe'_1 \ldots e'_n$ for some e'_i.

(iii) If $b_n(\ldots(b_1(xe_{01}\ldots e_{0m})e_{11}\ldots e_{1m})\ldots)e_{n1}\ldots e_{nm} \to e'$ then
$e' \equiv b_n(\ldots(b_1(xe'_{01}\ldots e'_{0m})e'_{11}\ldots e'_{1m})\ldots)e'_{n1}\ldots e'_{nm}$ for some e'_{ij}.

Proof: By considering possible redexes. $\Box$

4 Lazy Reduction

The reduction rules define a Church-Rosser system where $e \to e'$ by reduction of any redex in e. We now define one-step *lazy reduction* ($e \to_l e'$) so that e' is a function of e:

1. $(\lambda x.e)\, e'\, \vec{e} \to_l e\,[e'/x]\,\vec{e}$
2. $W e\, \vec{e} \to_l W \vec{e}$
3. $i\, e\, \vec{e} \to_l e\, i\, \vec{e}$
4. $b\, e\, \vec{e} \to_l e'\, \vec{e} \quad$ if $b\, e \to e'$ is a rewrite rule instance
5. $b\, e\, \vec{e} \to_l b\, e'\, \vec{e} \quad$ if $e \to_l e'$

Note that cases (4) and (5) are mutually exclusive.

Proposition 4.1

(i) If $e \to_l e'$ then $e \to_{lf} e'$.

(ii) A term e is in whnf if and only if $e \to_l e'$ for no e'.

Proof:

(i) From the definitions.

(ii) By induction on the structure of e.

$\square$

We write $\rightarrow_l^*$ (*multi-step reduction*) for the transitive reflexive closure of $\rightarrow_l$, and we define:

$$\begin{array}{rcl}
e \rightarrow_l^n e_n & \iff & e \equiv e_0 \rightarrow_l \cdots \rightarrow_l e_n \text{ for some } e_i \\
e \downarrow_l e' & \iff & e \rightarrow_l^* e' \text{ and } e' \text{ is in whnf} \\
e \downarrow_l^n e' & \iff & e \rightarrow_l^n e' \text{ and } e' \text{ is in whnf} \\
e \downarrow_l & \iff & e \downarrow_l e' \text{ for some } e' \\
e \uparrow_l & \iff & \sim e \downarrow_l
\end{array}$$

Only in case (5) of the rewrite rules is the redex head symbol not the leftmost symbol in the term. By the following proposition, if the only use of strict constants in a user program are $e_2(e_1 P)$ and $e_2(e_1 E)$, a lazy functional programming language implementation need not consider case (5), and can use a simple stack-based evaluation mechanism [8]. The introduction of C, however, does require case (5).

Proposition 4.2 *If $e_1 \rightarrow_l^* i$ and $e_2 \rightarrow_l^* j$, then:*

(i) $e_2(e_1 P) \rightarrow_l^* i + j$.

(ii) $e_2(e_1 E) \rightarrow_l^* \lambda x.\lambda y.x$ *if* $i = j$.

(iii) $e_2(e_1 E) \rightarrow_l^* \lambda x.\lambda y.y$ *if* $i \neq j$.

Proof:

(i) $e_2(e_1 P) \rightarrow_l^* j(e_1 P) \rightarrow_l e_1 P j \rightarrow_l^* i P j \rightarrow_l P i j \rightarrow_l P_i j \rightarrow_l i + j$.

(ii) Similarly to (i).

(iii) Similarly to (i).

$\square$

Proposition 4.3

(i) If $e \downarrow e'$ then $e \downarrow_l e''$ for some e'' in whnf, and $e'' \rightarrow^* e'$.

(ii) If $e \uparrow_l$ then $e \uparrow$.

Proof:

(i) By Propositions 4.1 and 3.1 (iii).

(ii) From (i).

$\square$

5 Call-by-Value Reduction

Most modern lazy functional programming languages provide a facility for defining strict (call-by-value) functions. We provide two mechanisms for doing this, using the C constant and the ν rewrite rule. We define *call-by-value abstraction*:

$$\lambda_C x.e \;\equiv\; \lambda y.C y\,((\lambda x.e)\,y)$$

and *numerical abstraction* (which provides call-by-value on numbers only):

$$\lambda_N x.e \;\equiv\; \lambda y.y\,(\lambda x.e)$$

Proposition 5.1 *For closed terms e:*

(i) If $e\!\uparrow_l$ then $(\lambda_C x.e')\,e\!\uparrow_l$ and $(\lambda_N x.e')\,e\!\uparrow_l$.

(ii) If $e\!\downarrow_l W$ then $(\lambda_C x.e')\,e\!\downarrow_l W$ and $(\lambda_N x.e')\,e\!\downarrow_l W$.

(iii) If $e\!\downarrow_l e' \not\equiv W$ in whnf, then $(\lambda_C x.e'')\,e \rightarrow_l^* e''\,[e/x]$.

(iv) If $e\!\downarrow_l i$ then $(\lambda_N x.e')\,e \rightarrow_l^* e'\,[i/x]$.

Proof:

(i) If $e\!\uparrow_l$ then $(\lambda_C x.e')\,e \rightarrow_l C\,e\,((\lambda x.e')\,e)\!\uparrow_l$ and $(\lambda_N x.e')\,e \rightarrow_l e\,(\lambda x.e')\!\uparrow_l$.

(ii) If $e\!\downarrow_l W$ then $(\lambda_C x.e')\,e \rightarrow_l^* CW((\lambda x.e')\,e) \rightarrow_l^* W$ and $(\lambda_N x.e')\,e \rightarrow_l^* W(\lambda x.e') \rightarrow W$.

(iii) If $e \rightarrow_l^* e' \not\equiv W$ in whnf then $(\lambda_C x.e'')\,e \rightarrow_l^* C\,e'\,((\lambda x.e'')\,e) \rightarrow_l (\lambda x.x)((\lambda x.e'')\,e) \rightarrow_l (\lambda x.e'')\,e \rightarrow_l e''\,[e/x]$.

(iv) If $e\!\downarrow_l i$ then $(\lambda_N x.e')\,e \rightarrow_l^* i\,(\lambda x.e') \rightarrow_l (\lambda x.e')\,i \rightarrow_l e'\,[i/x]$.

□

Thus $\lambda_C x.e$ provides a general call-by-value mechanism, but if we use a type system to guarantee that $e'\!\uparrow_l$ or $e'\!\downarrow_l i$ for some i, then we can use $\lambda_N x.e$ instead of $\lambda_C x.e$ for strict functions applied to e'.

6 Natural Semantics for Lazy Reduction

As is the case for the Lazy Lambda Calculus [14, 13], we define a natural semantics for lazy reduction by the following rules:

$$\frac{e \quad \text{is in whnf}}{e \Downarrow_l e} \qquad \frac{e_1 \Downarrow_l \lambda x.e \qquad e\,[e_2/x] \Downarrow_l e'}{e_1 e_2 \Downarrow_l e'} \qquad \frac{e_1 \Downarrow_l W}{e_1 e_2 \Downarrow_l W}$$

$$\frac{e_1 \Downarrow_l i \qquad e_2\, i \Downarrow_l e}{e_1 e_2 \Downarrow_l e} \qquad \frac{e_1 \Downarrow_l b \qquad e_2 \Downarrow_l e \qquad b\,e \rightarrow e' \quad \text{is a rewrite rule instance}}{e_1 e_2 \Downarrow_l e'}$$

We also define:

$$\begin{aligned} e \Downarrow_l \quad &\iff \quad e \Downarrow_l e' \text{ for some } e' \\ e \Uparrow_l \quad &\iff \quad \sim e \downarrow \end{aligned}$$

Proposition 6.1 *For all terms e, $e \Downarrow_l e'$ if and only if $e \downarrow_l e'$.*

Proof: We prove $e \Downarrow_l e' \Longrightarrow e \downarrow_l e'$ by induction on the derivation of $e \Downarrow_l e'$. The base case is trivial, using Proposition 4.1. For the $\lambda x.e$ case, $e_1 \downarrow_l \lambda x.e$ and $e\,[e_2/x] \downarrow_l e'$ by induction, so $e_1 e_2 \to_l^* (\lambda x.e)\, e_2 \to_l e\,[e_2/x] \downarrow_l e'$. For the W case, $e_1 \downarrow_l W$ by induction, so $e_1 e_2 \to_l^* W e_2 \downarrow_l W$. For the i case, $e_1 \downarrow_l i$ and $e_2\, i \downarrow_l e$ by induction, so $e_1 e_2 \to_l^* i\, e_2 \to_l e_2\, i \downarrow_l e$. For the b case, $e_1 \downarrow_l b$ and $e_2 \downarrow_l e$ by induction, so $e_1 e_2 \to_l^* b\, e_2 \to_l^* b\, e \to_l e'$, and e' is in whnf by definition of the rewrite rules.
We prove $e \downarrow_l e' \Longrightarrow e \Downarrow_l e'$ by induction on the length of the reduction $e \to_l^n e'$. For $n = 0$, $e \equiv e'$ must be in whnf, otherwise $e \equiv e_1 e_2$ for some $e_1$and e_2, and there are four possibilities:

1. $e_1 e_2 \to_l^p (\lambda x.e)\, e_2 \to e\,[e_2/x] \to_l^q e'$.

2. $e_1 e_2 \to_l^p W e_2 \to W$.

3. $e_1 e_2 \to_l^p i\, e_2 \to e_2\, i \to_l^q e'$.

4. $e_1 e_2 \to_l^p b\, e_2 \to_l^q b\, e \to_l e'$.

Since $p, q < n$ in each case, the required derivation can be constructed by induction. $\square$

7 Operational Semantics

We define a *context* $C[\cdot]$ to be a term containing a special constant, denoted $[\cdot]$. We write $C[e]$ for the result of replacing all occurrences of $[\cdot]$ in $C[\cdot]$ by e. We define three operational preorders on *closed* terms, $\sqsubseteq^{\mathrm{op}}$, $\sqsubseteq^{\mathrm{pre}}$, and $\sqsubseteq^{\mathrm{cxt}}$, as follows. In this definition and the remainder of the paper, we use v with possible subscripts and superscripts to denote closed whnfs.

$$\begin{array}{lcl}
e_1 \sqsubseteq^{\mathrm{op}} e_2 & \Longleftrightarrow & \forall \vec{e}, v_1.\ e_1\, \vec{e} \Downarrow_l v_1 \Longrightarrow \\
& & \quad \exists v_2.\ e_2\, \vec{e} \Downarrow_l v_2 \wedge (v_1 = W \Leftrightarrow v_2 = W) \wedge \forall i.\, (v_1 = i \Leftrightarrow v_2 = i) \\
e_1 \sqsubseteq^{\mathrm{pre}} e_2 & \Longleftrightarrow & \forall C[\cdot], v_1.\ C[e_1] \Downarrow_l v_1 \Longrightarrow \\
& & \quad \exists v_2.\ C[e_2] \Downarrow_l v_2 \wedge (v_1 = W \Leftrightarrow v_2 = W) \wedge \forall i.\, (v_1 = i \Leftrightarrow v_2 = i) \\
e_1 \sqsubseteq^{\mathrm{cxt}} e_2 & \Longleftrightarrow & \forall C[\cdot].\ C[e_1] \Downarrow_l \Longrightarrow C[e_2] \Downarrow_l
\end{array}$$

Lemma 7.1 *For all closed whnfs v_1 and v_2, if we have $\forall i.\, (v_1 = i \Leftrightarrow v_2 = i)$ and $(v_1 = W \Leftrightarrow v_2 = W)$, then for each strict constant b, $b\, v_1$ and $b\, v_2$ are redexes which rewrite to the same closed whnf.*

Proof: By definition of the rewrite rules. $\square$

Proposition 7.2 *For all closed e_1 and e_2:*

(i) $e_1 \sqsubseteq^{\mathrm{pre}} e_2 \Longrightarrow e_1 \sqsubseteq^{\mathrm{op}} e_2$

(ii) $e_1 \sqsubseteq^{\mathrm{pre}} e_2 \Longrightarrow e_1 \sqsubseteq^{\mathrm{cxt}} e_2$

(iii) $e_1 \sqsubseteq^{\mathrm{pre}} e_2 \Longrightarrow \forall C[\cdot].\, C[e_1] \sqsubseteq^{\mathrm{pre}} C[e_2]$ (i.e. $\sqsubseteq^{\mathrm{pre}}$ is a precongruence)

Proof:

(i) Taking $C[\cdot] \equiv [\cdot]\vec{e}$.

(ii) Trivial.

(iii) Taking $C[\cdot] \equiv D[E[\cdot]]$.

□

Proposition 7.3 *For all closed e_1 and e_2, $e_1 \sqsubseteq^{\mathrm{op}} e_2 \Longrightarrow e_1 \sqsubseteq^{\mathrm{pre}} e_2$*

Proof: Assume $e_1 \sqsubseteq^{\mathrm{op}} e_2$ and $C[e_1] \Downarrow_l v_1$. We prove that $C[e_2] \Downarrow_l v_2$ for some v_2, such that $(v_1 = W \Leftrightarrow v_2 = W)$ and $\forall i.\, (v_1 = i \Leftrightarrow v_2 = i)$. The proof is by induction on the length of the reduction $C[e_1] \rightarrow_l^n v_1$, and considering possible cases of $C[\cdot]$ (based on the method used in [2, p 11]).
If $C[\cdot]$ is one of a, $\lambda x.D[\cdot]$, $W\vec{D}[\cdot]$, or $[\cdot]$ the proof is trivial.
If $C[\cdot] \equiv (\lambda x.D[\cdot])E[\cdot]\vec{F}[\cdot] \rightarrow_l (D[\cdot])[E[\cdot]/x]\vec{F}[\cdot]$ or $C[\cdot] \equiv iD[\cdot]\vec{E}[\cdot] \rightarrow_l D[\cdot]\, i\vec{E}[\cdot]$ the result follows easily by induction.
If $C[\cdot] \equiv bD[\cdot]\vec{E}[\cdot]$, then $D[e_1] \Downarrow_l v'_1$ for some v'_1, and by the induction hypothesis and Lemma 7.1, $D[e_2] \Downarrow_l v'_2$ for some v'_2 and $b\, v_1 \rightarrow_l v$, $b\, v_2 \rightarrow_l v$ for some v, so that the result follows by induction for $v\vec{E}[\cdot]$.
If $C[\cdot] \equiv [\cdot]D[\cdot]\vec{E}[\cdot]$, then $e_1 \Downarrow_l v'_1$ and $C[e_1] \equiv e_1 D[e_1]\vec{E}[e_1] \rightarrow_l^* v'_1 D[e_1]\vec{E}[e_1] \Downarrow_l v_1$ for some v'_1 and v_1. Since $e_1 \sqsubseteq^{\mathrm{op}} e_2$, we have $e_2 \Downarrow_l v'_2$ for some v'_2, and $e_2 D[e_1]\vec{E}[e_1] \rightarrow_l^* v'_2 D[e_1]\vec{E}[e_1] \Downarrow_l v_0$ for some v_0 with $(v_1 = W \Leftrightarrow v_0 = W)$ and $\forall i.\, (v_1 = i \Leftrightarrow v_0 = i)$. Since $C[e_2] \rightarrow_l^* v'_2 D[e_2]\vec{E}[e_2]$, by appeal to other cases for the context $v'_2 D[\cdot]\vec{E}[\cdot]$ we have $v'_2 D[e_2]\vec{E}[e_2] \Downarrow_l v_2$ for some v_2, with $(v_0 = W \Leftrightarrow v_2 = W)$ and $\forall i.\, (v_0 = i \Leftrightarrow v_2 = i)$. The result follows by transitivity.
□

Corollary 7.4 *For all closed e_1 and e_2:*

(i) $e_1 \sqsubseteq^{\mathrm{op}} e_2 \Longleftrightarrow e_1 \sqsubseteq^{\mathrm{pre}} e_2$

(ii) $e_1 \sqsubseteq^{\mathrm{op}} e_2 \Longrightarrow e_1 \sqsubseteq^{\mathrm{cxt}} e_2$

(iii) $e_1 \sqsubseteq^{\mathrm{op}} e_2 \Longrightarrow \forall C[\cdot].\, C[e_1] \sqsubseteq^{\mathrm{op}} C[e_2]$ (i.e. $\sqsubseteq^{\mathrm{op}}$ is a precongruence)

As a consequence of this corollary, we will not use $\sqsubseteq^{\mathrm{pre}}$ in the remainder of the paper. We define two *equivalence relations* corresponding to the two distinct preorders:

$$\begin{aligned} e_1 \approx^{\mathrm{op}} e_2 &\iff e_1 \sqsubseteq^{\mathrm{op}} e_2 \wedge e_2 \sqsubseteq^{\mathrm{op}} e_1 \\ e_1 \approx^{\mathrm{cxt}} e_2 &\iff e_1 \sqsubseteq^{\mathrm{cxt}} e_2 \wedge e_2 \sqsubseteq^{\mathrm{cxt}} e_1 \end{aligned}$$

We can prove that these equivalence relations are identical because we can distinguish numbers and W from other values by convergence or divergence in appropriate contexts:

Lemma 7.5 *For each pair of distinct atoms $a \neq a'$, there is a vector of closed terms $\vec{e}$ such that either $a\,\vec{e} \Downarrow_l$ and $a'\,\vec{e} \Uparrow_l$, or $a\,\vec{e} \Uparrow_l$ and $a'\,\vec{e} \Downarrow_l$.*

Proof: As follows:

$$\begin{array}{ll}
i\,(\lambda x.Eix0\Omega) \Downarrow_l 0 & j\,(\lambda x.Eix0\Omega) \Uparrow_l \text{ for } j \neq i \\
W\Omega \Downarrow_l W & a\Omega \Uparrow_l \text{ for } a \neq W \\
i\,(\lambda x.\Omega) \Uparrow_l & b\,(\lambda x.\Omega) \Downarrow_l \\
CC\Omega \Uparrow_l & bC\Omega \Downarrow_l W \text{ for } b \neq C \\
P_i01 \Uparrow_l & b01 \Downarrow_l \text{ for } b \neq P_i \\
P01\Omega \Uparrow_l & E01\Omega \Downarrow_l \lambda x.x \\
P0\Omega \Uparrow_l & E_i0\Omega \Downarrow_l \\
Ejj\Omega \Downarrow_l \lambda x.\Omega & E_ijj\Omega \Uparrow_l \text{ for } j \neq i
\end{array}$$

□

We cannot distinguish e.g. i and $\lambda x.x\,i$ by such applicative contexts, but other contexts suffice:

Lemma 7.6 *There are closed contexts $D[\cdot]$ and $E[\cdot]$ such that $D[\lambda x.e] \Downarrow_l$ and $D[i] \Uparrow_l$, and also $E[W] \Downarrow_l$ and $E[\lambda x.e] \Uparrow_l$.*

Proof: Since $P(\lambda x.e)\Omega \Downarrow_l W$ and $Pi\Omega \Uparrow_l$, and also $CW\Omega \Downarrow_l \mathrm{W}$ and $C(\lambda x.e)\Omega \Uparrow_l$. □

Note that we cannot distinguish strict constants from abstractions in this way: for example P and $\lambda x.Px$ are operationally indistinguishable. However, this is acceptable, since the two terms denote the same function.

Corollary 7.7 *For all closed e_1 and e_2, $e_1 \approx^{\mathrm{op}} e_2 \Longleftrightarrow e_1 \approx^{\mathrm{cxt}} e_2$*

Proof: By Corollary 7.4, $e_1 \approx^{\mathrm{op}} e_2 \Longrightarrow e_1 \approx^{\mathrm{cxt}} e_2$. Conversely, if $e_1 \approx^{\mathrm{cxt}} e_2$, then for all contexts $C[\cdot]$, $C[e_1] \Downarrow_l \Leftrightarrow C[e_2] \Downarrow_l$. In particular, for each $\vec{e}$, $e_1\,\vec{e} \Downarrow_l v_1 \Leftrightarrow e_2\,\vec{e} \Downarrow_l v_2$ for some v_1 and v_2, and for each context $D[\cdot]$, $D[e_1\,\vec{e}] \Downarrow_l \Leftrightarrow D[e_2\,\vec{e}] \Downarrow_l$. Hence by Lemmas 7.5 and 7.6, $(v_1 = W \Leftrightarrow v_2 = W)$ and $\forall i.\,(v_1 = i \Leftrightarrow v_2 = i)$, i.e. $e_1 \approx^{\mathrm{op}} e_2$. □

As a consequence of this corollary, we will not use $\approx^{\mathrm{cxt}}$ in the remainder of the paper. However, as the following proposition and corollary show, $\sqsubseteq^{\mathrm{op}}$ and $\sqsubseteq^{\mathrm{cxt}}$ are not identical:

Proposition 7.8 *For all closed terms e and all contexts $C[\cdot]$:*

(i) $\forall a.\, C[e] \Downarrow_l a \Longrightarrow C[W] \Downarrow_l a \vee C[W] \Downarrow_l W$

(ii) $\forall e'.\, C[e] \Downarrow_l \lambda x.e' \Longrightarrow \exists e''.\, C[W] \Downarrow_l \lambda x.e'' \vee C[W] \Downarrow_l W$

Proof: By induction on the length of the reduction $C[e] \rightarrow_l^n v$, and considering possible cases of $C[\cdot]$, as for Proposition 7.3.
If $C[\cdot] \equiv bD[\cdot]\vec{E}[\cdot]$ then $D[e] \Downarrow_l v_1$ and $b\,v_1 \rightarrow_l v$ for some v_1 and v, and by the induction hypothesis and the definitions of the rewrite rules, $D[W] \Downarrow_l v_2$ for some v_2 such that $b\,v_2 \rightarrow_l v$ (in which case the result follows by induction for $v\vec{E}[\cdot]$), or $b\,v_2 \rightarrow_l W$ (in which case the result follows immediately).
If $C[\cdot] \equiv [\cdot]D[\cdot]\vec{E}[\cdot]$, then the result follows immediately. The other cases are similar to Proposition 7.3. □

Corollary 7.9 *For all closed terms e, $e \sqsubseteq^{\mathrm{cxt}} W$.*

Thus $\approx^{\mathrm{cxt}}$ is an acceptable operational equivalence (identical to $\approx^{\mathrm{op}}$) even though it makes no explicit reference to constants. However, $\sqsubseteq^{\mathrm{cxt}}$ is not the most suitable preorder (unless we are using a lattice semantics where W corresponds to the top element of the lattice), and $\sqsubseteq^{\mathrm{op}}$ (which makes explicit reference to constants) is necessary.

8 Denotational Semantics

We give a denotational semantics for the Lazy Lambda Calculus with constants using the cpo (not lattice) domain:

$$\mathbf{D} \cong [\mathbf{D} \to \mathbf{D}] + \{\mathit{wrong}, 0, 1, 2, \ldots\}$$

where the sum is a separated sum. Thus an element of $\mathbf{D}$ is one of $\bot$, *wrong*, $i, j \in \{0, 1, 2, \ldots\}$ or $f, g \in [\mathbf{D} \to \mathbf{D}]$. We define a binary operator $\bullet$ (application) on $\mathbf{D}$ as follows:

$$\begin{array}{rcl}
\bot \bullet d & = & \bot \\
\mathit{wrong} \bullet d & = & \mathit{wrong} \\
f \bullet d & = & f(d) \\
i \bullet \bot & = & \bot \\
i \bullet \mathit{wrong} & = & \mathit{wrong} \\
i \bullet j & = & \bot \\
i \bullet f & = & f(i)
\end{array}$$

Note that $i \bullet d = d \bullet i$ for each d, and $i \bullet j = \bot$ since $i\, j \Uparrow_l$. For all terms e and *environments* ρ mapping the free variables $FV(e)$ of e to $\mathbf{D}$, we define $\mathcal{M}[e]_\rho$ (the denotation of e in the environment ρ) as follows. For closed e, ρ is not used, and we write simply $\mathcal{M}[e]$.

$$\begin{array}{rcl}
\mathcal{M}[i]_\rho & = & i \\
\mathcal{M}[W]_\rho & = & \mathit{wrong} \\
\mathcal{M}[x]_\rho & = & \rho(x) \\
\mathcal{M}[e_1 e_2]_\rho & = & \mathcal{M}[e_1]_\rho \bullet \mathcal{M}[e_2]_\rho \\
\mathcal{M}[\lambda x.e]_\rho & = & f \\
 & & \text{where } f(d) = \mathcal{M}[e]_{\rho[d/x]} \\
\mathcal{M}[C]_\rho & = & f \\
 & & \text{where } f(\bot) = \bot, \quad f(\mathit{wrong}) = \mathit{wrong}, \\
 & & f(g) = \mathrm{id}, \quad f(i) = \mathrm{id} \in [\mathbf{D} \to \mathbf{D}] \\
 & & \text{and } \mathrm{id}(d) = d \\
\mathcal{M}[P]_\rho & = & f \\
 & & \text{where } f(\bot) = \bot, \quad f(\mathit{wrong}) = \mathit{wrong},
\end{array}$$

$$
\begin{aligned}
& && f(g) = \mathit{wrong}, \quad f(i) = \mathcal{M}[P_i]_\rho \\
\mathcal{M}[P_i]_\rho &= && f \\
& && \text{where } f(\bot) = \bot, \quad f(\mathit{wrong}) = \mathit{wrong}, \\
& && f(g) = \mathit{wrong}, \quad f(j) = i + j \\
\mathcal{M}[E]_\rho &= && f \\
& && \text{where } f(\bot) = \bot, \quad f(\mathit{wrong}) = \mathit{wrong}, \\
& && f(g) = \mathit{wrong}, \quad f(i) = \mathcal{M}[E_i]_\rho \\
\mathcal{M}[E_i]_\rho &= && f \\
& && \text{where } f(\bot) = \bot, \quad f(\mathit{wrong}) = \mathit{wrong}, \\
& && f(g) = \mathit{wrong}, \quad f(j) = \text{true} \ \text{ if } i = j, \\
& && f(j) = \text{false} \ \text{ if } i \neq j \\
& && \text{and } \text{true}(d)(d') = d, \quad \text{false}(d)(d') = d'
\end{aligned}
$$

Proposition 8.1 *If* $e_1 \rightarrow e_2$ *then* $\mathcal{M}[e_1]_\rho = \mathcal{M}[e_2]_\rho$.

Proof: By induction on the structure of e, noting that if $\mathcal{M}[e']_\rho = d$ then $\mathcal{M}[(\lambda x.e)\, e']_\rho = \mathcal{M}[e]_{\rho[d/x]} = \mathcal{M}[e\,[e'/x]]_\rho$ (by a further induction on the structure of e), and also $\mathcal{M}[W e]_\rho = \mathit{wrong} = \mathcal{M}[W]_\rho$, $\mathcal{M}[i\, e]_\rho = \mathcal{M}[i]_\rho \bullet \mathcal{M}[e]_\rho = \mathcal{M}[e]_\rho \bullet \mathcal{M}[i]_\rho = \mathcal{M}[e\, i]_\rho$, and for each rewrite rule instance $b\, e \rightarrow e'$, $\mathcal{M}[b\, e]_\rho = \mathcal{M}[b]_\rho \bullet \mathcal{M}[e]_\rho = \mathcal{M}[e']_\rho$. □

Proposition 8.2 *The domain* $\mathbf{D}$*, together with the application operator* $\bullet$ *is a* λ*-model* [9, p 112].

Proof: From the definition of $\mathcal{M}$, we can verify that:

(i) $\mathcal{M}[x]_\rho = \rho(x)$.

(ii) $\mathcal{M}[e_1 e_2]_\rho = \mathcal{M}[e_1]_\rho \bullet \mathcal{M}[e_2]_\rho$.

(iii) For each $d \in \mathbf{D}$, $\mathcal{M}[\lambda x.e]_\rho \bullet d = \mathcal{M}[e]_{\rho[d/x]}$.

(iv) If $\rho(x) = \rho'(x)$ for each $x \in FV(e)$, then $\mathcal{M}[e]_\rho = \mathcal{M}[e]_{\rho'}$.

(v) If $y \notin FV(e)$, then $\mathcal{M}[\lambda x.e]_\rho = \mathcal{M}[\lambda y.e\,[y/x]]_\rho$.

(vi) If $\mathcal{M}[e]_{\rho[d/x]} = \mathcal{M}[e']_{\rho[d/x]}$ for each $d \in \mathbf{D}$, then $\mathcal{M}[\lambda x.e]_\rho = \mathcal{M}[\lambda x.e']_\rho$ are the same function.

□

Lemma 8.3 *Let* Ω *be any term without whnf, and let* v *be any closed whnf. Then for all closed terms* e, $e\Omega\Omega \Uparrow_l \iff e\, v\Omega \Uparrow_l \vee e\Omega v \Uparrow_l$.

Proof: By considering possible cases for e, extending the proof in [14, p 154]:

$$
\begin{array}{lll}
e \Uparrow_l & & e\, e_1 e_2 \Uparrow_l \\
e \to^* i & & e\Omega e_2 \Uparrow_l \\
e \to^* b & & e\Omega e_2 \Uparrow_l \\
e \to^* W & & e\, e_1 e_2 \Downarrow_l W \\
e \to^* \lambda x.i & & e\, e_1 \Omega \Uparrow_l \\
e \to^* \lambda x.b & & e\, e_1 \Omega \Uparrow_l \\
e \to^* \lambda x.W & & e\, e_1 e_2 \Downarrow_l W \\
e \to^* \lambda x.e', & \text{where } e' \Uparrow_l & e\, e_1 e_2 \Uparrow_l \\
e \to^* \lambda x.e', & e' \text{ blocked on } x & e\Omega e_2 \Uparrow_l \\
e \to^* \lambda x.\lambda y.a & & e\, e_1 e_2 \Downarrow_l a \\
e \to^* \lambda x.\lambda y.e', & \text{where } e' \Uparrow_l & e\, e_1 e_2 \Uparrow_l \\
e \to^* \lambda x.\lambda y.e', & e' \text{ blocked on } x & e\Omega e_2 \Uparrow_l \\
e \to^* \lambda x.\lambda y.e', & e' \text{ blocked on } y & e\, e_1 \Omega \Uparrow_l \\
e \to^* \lambda x.\lambda y.\lambda z.e' & & e\, e_1 e_2 \Downarrow_l
\end{array}
$$

□

Proposition 8.4 (Ong) *There are closed terms e_1 and e_2 such that $e_1 \approx^{\mathrm{op}} e_2$ but $\mathcal{M}[e_1] \neq \mathcal{M}[e_2]$. In particular:*

$$
\begin{array}{rcl}
e_1 & \equiv & \lambda x.C(C(x\,(\lambda y.\Omega)\Omega)(x\Omega\Omega)) \\
e_2 & \equiv & \lambda x.C(C(x\,(\lambda y.\Omega)\Omega)(x\Omega(\lambda y.\Omega)))
\end{array}
$$

Proof: Similar to [14, p 155]. If $e\,(\lambda y.\Omega)\Omega \Uparrow_l$ then $e_1\, e \Uparrow_l$ and $e_2\, e \Uparrow_l$. If $e\,(\lambda y.\Omega)\Omega \Downarrow_l$, then by Lemma 8.3, either $e\Omega\Omega \Uparrow_l$ and $e\Omega(\lambda y.\Omega) \Uparrow_l$ (in which case $e_1\, e \Uparrow_l$ and $e_2\, e \Uparrow_l$), or $e\Omega\Omega \Downarrow_l$ and $e\Omega(\lambda y.\Omega) \Downarrow_l$ (in which case $e_1\, e \Downarrow_l \lambda x.x$ and $e_2\, e \Downarrow_l \lambda x.x$). Hence $e_1 \approx^{\mathrm{op}} e_2$.
However if we consider the continuous function p defined by:

$$
\begin{array}{rcll}
p(\bot)(\bot) & = & \bot & \\
p(d)(\bot) & = & 0 & \text{for } d \neq \bot \\
p(\bot)(d) & = & 0 & \text{for } d \neq \bot
\end{array}
$$

Then $\mathcal{M}[e_1] \bullet p = \bot$ and $\mathcal{M}[e_2] \bullet p = 0$, so $\mathcal{M}[e_1] \neq \mathcal{M}[e_2]$. □

This disproves a conjecture of Wadsworth [23, p 518] that $e_1 \approx^{\mathrm{op}} e_2 \iff \mathcal{M}[e_1] = \mathcal{M}[e_2]$. Abramsky [1, p 95] proves that this conjecture holds in the pure lazy lambda calculus (i.e. without atoms) extended with a new parallel operator P such that:

$$
\begin{array}{c}
P v\, e \to \lambda x.x \\
P e\, v \to \lambda x.x
\end{array}
$$

where v is one of P, Pe', or $\lambda x.e'$ for some e'. This operator P is also needed to define strict functions on coalesced product types (i.e. where $\bot = (\bot, \bot)$). This is the reason why pairing in lazy functional programming languages with a call-by-value facility (such as Miranda) must be modelled by lifted products. Pairing (e_1, e_2) can be modelled in our calculus as $\lambda x.x e_1 e_2$, and it is clear that $\bot$ and $\lambda x.x\bot\bot$ must be distinguished.

9 Types

To eliminate the semantic problems associated with W (and more importantly to prevent run-time errors in practical programs) we use a *type system* such as that given by MacQueen *et al.* [12]. We consider the following type expressions:

- type variables $\alpha, \beta, \gamma, \ldots$
- number type N
- function types $\sigma \to \tau$
- intersection types $\sigma \cap \tau$
- quantified types $\forall \alpha.\sigma$
- recursive types $\mu\alpha.\sigma$

Here $\mu\alpha.\sigma$ is only well-formed if all occurrences of the bound type variable α occur inside subexpressions of the form $\tau \to \tau'$. For each term e and each *assumption set* A mapping free variables of e to types, we write $A \vdash e : \tau$ to indicate that e has the type τ given the assumptions A. For closed e we also write $\vdash e : \tau$. The types of variables and atoms are given as follows:

$$
\begin{array}{l}
A \vdash x : A(x) \\
A \vdash i : N \\
A \vdash C : \forall\alpha.\alpha \to \forall\beta.\beta \to \beta \\
A \vdash P : N \to N \to N \\
A \vdash P_i : N \to N \\
A \vdash E : N \to N \to \forall\alpha.\alpha \to \alpha \to \alpha \\
A \vdash E_i : N \to \forall\alpha.\alpha \to \alpha \to \alpha
\end{array}
$$

We also have the following type inference rules:

$$
\frac{A \vdash e_1 : \sigma \to \tau \qquad A \vdash e_2 : \sigma}{A \vdash e_1 e_2 : \tau}
\qquad
\frac{A[\sigma/x] \vdash e : \tau}{A \vdash \lambda x.e : \sigma \to \tau}
$$

$$
\frac{A \vdash e : \sigma[\mu\alpha.\sigma/\alpha]}{A \vdash e : \mu\alpha.\sigma}
\qquad
\frac{A \vdash e : \mu\alpha.\sigma}{A \vdash e : \sigma[\mu\alpha.\sigma/\alpha]}
$$

$$
\frac{A \vdash e : \forall\alpha.\sigma}{A \vdash e : \sigma[\tau/\alpha]}
\qquad
\frac{A \vdash e : \sigma \quad \alpha \text{ not free in } A}{A \vdash e : \forall\alpha.\sigma}
$$

$$
\frac{A \vdash e : \sigma \qquad A \vdash e : \tau}{A \vdash e : \sigma \cap \tau}
\qquad
\frac{A \vdash e : \sigma \cap \tau}{A \vdash e : \sigma}
\qquad
\frac{A \vdash e : \sigma \cap \tau}{A \vdash e : \tau}
$$

$$
\frac{A \vdash e : N}{A \vdash e : \forall\alpha.(N \to \alpha) \to \alpha}
$$

The final rule is justified by the functional behaviour of numbers. It is useful to define the following terms:

$$
\begin{array}{rcl}
e_1 \oplus e_2 & \equiv & e_2(e_1 P) \\
e_1 \bowtie e_2 & \equiv & e_2(e_1 E) \\
Y & \equiv & (\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))
\end{array}
$$

Proposition 9.1

(i) *If* $A \vdash e_1 : N$ *and* $A \vdash e_2 : N$ *then* $A \vdash e_1 \oplus e_2 : N$.

(ii) *If* $A \vdash e_1 : N$ *and* $A \vdash e_2 : N$ *then* $A \vdash e_1 \bowtie e_2 : \forall\alpha.\alpha \to \alpha \to \alpha$.

(iii) $\vdash Y : \forall\alpha.(\alpha \to \alpha) \to \alpha$.

(iv) $\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha$.

(v) $\vdash Y(\lambda x.x) : \forall\alpha.\alpha$.

(vi) $\vdash \lambda x.\lambda y.x : \forall\alpha.\alpha \to \alpha \to \alpha$.

(vii) $\vdash \lambda x.\lambda y.y : \forall\alpha.\alpha \to \alpha \to \alpha$.

Proof: Straightforward. For (i) and (ii) infer $A \vdash e_1 : (N \to N \to \sigma) \to N \to \sigma$ and $A \vdash e_2 : (N \to \sigma) \to \sigma$.
For (iii) assume $y : \alpha \to \alpha$ and $x : \mu\beta.\beta \to \beta \to (\alpha \to \alpha) \to \alpha$, so that $\vdash \lambda x.\lambda y.y(xxy) : \mu\beta.\beta \to \beta \to (\alpha \to \alpha) \to \alpha$. □

Following MacQueen *et al.* [12], for each closed type τ we define a subset $\mathbf{D}_\tau$ of $\mathbf{D}$ which is downward-closed (and hence $\bot \in \mathbf{D}_\tau$ for each τ), directed-closed, and does not contain *wrong*. We define:

$$\begin{array}{rcl}
\mathbf{D}_N & = & \{\bot, 0, 1, 2, \ldots\} \\
\mathbf{D}_{\sigma\to\tau} & = & \{d \mid \forall d' \in \mathbf{D}_\sigma.\ d \bullet d' \in \mathbf{D}_\tau\} \\
\mathbf{D}_{\sigma\cap\tau} & = & \mathbf{D}_\sigma \cap \mathbf{D}_\tau \\
\mathbf{D}_{\forall\alpha.\sigma} & = & \bigcap_\tau \mathbf{D}_{\sigma[\tau/\alpha]} \\
\mathbf{D}_{\mu\alpha.\sigma} & = & \mathbf{D}_{\sigma[\mu\alpha.\sigma/\alpha]}
\end{array}$$

The major result of [12] is to show that $\mathbf{D}_{\mu\alpha.\sigma}$ is well-defined, provided that all occurrences of the bound type variable α occur inside subexpressions of the form $\tau \to \tau'$.

Proposition 9.2 (Soundness) *For each term* e*, environment* ρ*, closed type* τ*, and assumption set* A*, if* $A(x)$ *is closed for each* $x \in FV(e)$*,* $A \vdash e : \tau$ *and* $\rho(x) \in \mathbf{D}_{A(x)}$ *for each* $x \in FV(e)$ *then* $\mathcal{M}[e]_\rho \in \mathbf{D}_\tau$.

Proof: By induction on the derivation $A \vdash e : \tau$, as in [12, pp 121–122]. We give the two most difficult cases. If $A[\sigma/x] \vdash e : \tau$ then by induction $\mathcal{M}[e]_{\rho[d/x]} \in \mathbf{D}_\tau$ for each $d \in \mathbf{D}_\sigma$, and hence $\mathcal{M}[\lambda x.e]_\rho \in \mathbf{D}_{\sigma\to\tau}$. If $A \vdash e : N$ then by induction $\mathcal{M}[e]_\rho \in \mathbf{D}_N$, and hence $\mathcal{M}[e]_\rho \bullet d = d \bullet \mathcal{M}[e]_\rho \in \mathbf{D}_\tau$ for each $d \in \mathbf{D}_{N\to\tau}$ Thus $\mathcal{M}[e]_\rho \in \mathbf{D}_{(N\to\tau)\to\tau}$, and since this holds for each τ, $\mathcal{M}[e]_\rho \in \mathbf{D}_{\forall\alpha.(N\to\alpha)\to\alpha}$. □

Corollary 9.3 *If* $\vdash e : N$ *then* $e \Uparrow_l$ *or* $e \Downarrow_l i$ *for some* i.

Proof: Since $\mathcal{M}[e] \in \mathbf{D}_N$. □

10 Programs

For a practical lazy functional programming language such as Miranda [17] or CLEAN [4, 22], intersection types and recursive types are not included. Universally quantified types are used in restricted ways [5] and user-defined types such as lists are introduced. These types are definable using universal quantification [18]. The strict constants P, E, P_i and E_i and the number-swap rule are hidden from the programmer, being replaced by $e_1 \oplus e_2$ and $e_1 \bowtie e_2$ as new syntax for terms, with type inference rules corresponding to Proposition 9.1 (i) to (iii). We call the resulting terms *programs*, and we have the following type axioms and inference rules for programs:

$$\begin{array}{l} A \vdash^p x : A(x) \\ A \vdash^p i : N \\ A \vdash^p C : \forall\alpha.\alpha \to \forall\beta.\beta \to \beta \\ A \vdash^p Y : \forall\alpha.(\alpha \to \alpha) \to \alpha \end{array}$$

$$\frac{A \vdash^p e_1 : N \qquad A \vdash^p e_2 : N}{A \vdash^p e_1 \oplus e_2 : N} \qquad \frac{A \vdash^p e_1 : N \qquad A \vdash^p e_2 : N}{A \vdash^p e_1 \bowtie e_2 : \forall\alpha.\alpha \to \alpha \to \alpha}$$

$$\frac{A \vdash^p e_1 : \sigma \to \tau \qquad A \vdash^p e_2 : \sigma}{A \vdash^p e_1 e_2 : \tau} \qquad \frac{A[\sigma/x] \vdash^p e : \tau}{A \vdash^p \lambda x.e : \sigma \to \tau}$$

$$\frac{A \vdash^p e : \forall\alpha.\sigma}{A \vdash^p e : \sigma[\tau/\alpha]} \qquad \frac{A \vdash^p e : \sigma \quad \alpha \text{ not free in } A}{A \vdash^p e : \forall\alpha.\sigma}$$

Proposition 10.1 *For each term e, if $A \vdash^p e : \tau$, then $A \vdash e : \tau$.*

Proof: By induction on the derivation $A \vdash^p e : \tau$. □

Lemma 10.2 *For each term e, if $A[\sigma/x] \vdash^p e : \tau$ and $A \vdash^p e' : \sigma$, then $A \vdash^p e\,[e'/x] : \tau$.*

Proof: By induction on the derivation $A[\sigma/x] \vdash^p e : \tau$. □

Proposition 10.3 *For each closed term e such that $\vdash^p e : \tau$ for some τ, if $e \Downarrow_l e'$ then $\vdash^p e' : \tau$.*

Proof: By induction on the length of the reduction $e \to_l^n e'$, using Lemma 10.2 and Proposition 4.2. □

Corollary 10.4

(i) If $\vdash^p e : \sigma \to \tau$ then $e \Uparrow_l$ or $e \Downarrow_l \lambda x.e'$ for some e'.

(ii) If $\vdash^p e : \forall\alpha.\alpha$ then $e \Uparrow_l$.

Proof:

(i) Since $\lambda x.e$ is the only whnf of type $\sigma \to \tau$.

(ii) From (i) and Proposition 9.3.

$\square$

We define families of preorders $\sqsubseteq^{\text{op}}_{\sigma}$ and $\sqsubseteq^{\text{cxt}}_{\sigma}$ on closed programs. If $\vdash^p e_1 : \sigma$ and $\vdash^p e_1 : \sigma$, we define:

$$\begin{aligned}
e_1 \sqsubseteq^{\text{op}}_{N} e_2 &\iff \forall i.\ e_1 \Downarrow_l i \Longrightarrow e_2 \Downarrow_l i \\
e_1 \sqsubseteq^{\text{op}}_{\sigma\to\tau} e_2 &\iff (e_1 \Downarrow_l \Longrightarrow e_2 \Downarrow_l) \wedge \forall e.\ \vdash^p e : \sigma \Longrightarrow e_1\, e \sqsubseteq^{\text{op}}_{\tau} e_2\, e \\
e_1 \sqsubseteq^{\text{op}}_{\forall\alpha.\sigma} e_2 &\iff \forall \tau.\ e_1 \sqsubseteq^{\text{op}}_{\sigma[\tau/\alpha]} e_2 \\
e_1 \sqsubseteq^{\text{cxt}}_{\sigma} e_2 &\iff \forall C[\cdot].\ [\sigma/x] \vdash^p C[x] : \tau \Longrightarrow (C[e_1] \Downarrow_l \Longrightarrow C[e_2] \Downarrow_l)
\end{aligned}$$

We will prove that these two families of preorders are identical. We also define a family of equivalence relations:

$$e_1 \approx^{\text{op}}_{\sigma} e_2 \iff e_1 \sqsubseteq^{\text{op}}_{\sigma} e_2 \wedge e_2 \sqsubseteq^{\text{op}}_{\sigma} e_1$$

Proposition 10.5 *For all closed e_1 and e_2 such that $\vdash^p e_1 : \sigma$ and $\vdash^p e_1 : \sigma$:*

(i) If $e_1 \sqsubseteq^{\text{cxt}}_{\sigma} e_2$ and $[\sigma/x] \vdash^p C[x] : \tau$, then $C[e_1] \sqsubseteq^{\text{cxt}}_{\tau} C[e_2]$.

(ii) If $e_1 \sqsubseteq^{\text{cxt}}_{\sigma} e_2$ then $e_1 \sqsubseteq^{\text{op}}_{\sigma} e_2$.

(iii) If $e_1 \sqsubseteq^{\text{op}} e_2$ then $e_1 \sqsubseteq^{\text{op}}_{\sigma} e_2$.

Proof:

(i) Taking $C[\cdot] \equiv D[E[\cdot]]$.

(ii) Taking $C[\cdot] \equiv [\cdot]\vec{e}$. and $C[\cdot] \equiv ([\cdot]\vec{e} \bowtie i)0(Y(\lambda x.x))$ for each i.

(iii) From the definitions of the preorders.

$\square$

Note that the converse to (iii) does not hold, since we have for example $\lambda x.1 \oplus x \approx^{\text{op}}_{N\to N} \lambda x.x \oplus 1$, but the two terms can be distinguished by ill-typed contexts: $(\lambda x.x\,(1P))C \Downarrow_l \lambda x.x$ and $(\lambda x.1\,(xP))C \Downarrow_l 1$.

Proposition 10.6 *For all closed e_1, e_2, and $C[\cdot]$ such that $\vdash^p e_1 : \sigma$, $\vdash^p e_1 : \sigma$, $e_1 \sqsubseteq^{\text{op}}_{\sigma} e_2$, $[\sigma/x] \vdash^p C[x] : \tau$, $C[e_1] \Downarrow_l$, and $C[e_1] \Downarrow_l i$ for some i if $\tau = N$, we have $C[e_2] \Downarrow_l$ and $C[e_2] \Downarrow_l i$ if $\tau = N$.*

Proof: By induction on the length of the reduction $C[e_1] \to^n_l v_1$, and considering possible cases of $C[\cdot]$, as for the untyped case Proposition 7.3.
If $C[\cdot] \equiv (D[\cdot] \bowtie E[\cdot])\vec{F}[\cdot]$ then $D[e_1] \Downarrow_l i$, $E[e_1] \Downarrow_l j$, and $D[e_1] \bowtie E[e_1] \Downarrow_l v$, where v is either $\lambda x.\lambda y.x$ or $\lambda x.\lambda y.y$. Then by the induction hypothesis, $D[e_2] \Downarrow_l i$ and $E[e_2] \Downarrow_l j$, so that $D[e_2] \bowtie E[e_2] \Downarrow_l v$. The result follows by induction for $v\vec{F}[\cdot]$. The case for $C[\cdot] \equiv D[\cdot] \oplus E[\cdot]$ is similar. The other cases are similar to Proposition 7.3. $\square$

Corollary 10.7 *For all closed e_1 and e_2 such that $\vdash^p e_1 : \sigma$ and $\vdash^p e_1 : \sigma$:*

(i) $e_1 \sqsubseteq^{\text{op}}_{\sigma} e_2 \iff e_1 \sqsubseteq^{\text{cxt}}_{\sigma} e_2$.

(ii) If $e_1 \sqsubseteq^{\text{op}}_{\sigma} e_2$ and $[\sigma/x] \vdash^p C[x] : \tau$, then $C[e_1] \sqsubseteq^{\text{op}}_{\tau} C[e_2]$.

11 Towards Full Abstraction

It is possible to define families of denotational preorders $\leq_\sigma$ (corresponding to $\sqsubseteq^{\text{op}}_\sigma$) and equivalence relations $=_\sigma$ (corresponding to $\approx^{\text{op}}_\sigma$) on elements of $\mathbf{D}_\sigma$ as follows:

$$
\begin{array}{rcl}
d_1 \leq_N d_2 & \iff & d_1 = \bot \ \vee \ d_1 = d_2 \\
d_1 \leq_{\sigma\to\tau} d_2 & \iff & d_1 = \bot \ \vee \ \forall d \in \mathbf{D}_\sigma .\ d_1\, d \leq_\tau d_2\, d \\
d_1 \leq_{\forall\alpha.\sigma} d_2 & \iff & \forall \tau .\ d_1 \leq_{\sigma[\tau/\alpha]} d_2 \\
d_1 =_\sigma d_2 & \iff & d_1 \leq_\sigma d_2 \wedge d_2 \leq_\sigma d_1
\end{array}
$$

Proposition 11.1 *There are closed terms e_1 and e_2 such that:*

$$
\begin{array}{l}
\vdash^p e_1 : (N \to N \to N) \to N, \\
\vdash^p e_2 : (N \to N \to N) \to N, \\
e_1 \approx^{\text{op}}_{(N\to N\to N)\to N} e_2,
\end{array}
$$

but $\mathcal{M}[e_1] \neq_{(N\to N\to N)\to N} \mathcal{M}[e_2]$. In particular:

$$
\begin{array}{rcl}
e_1 & \equiv & \lambda x.((x0\Omega) \bowtie 0)(((x\Omega 0) \bowtie 0)(((x11) \bowtie 0)\Omega\, 7)\Omega)\Omega \\
e_2 & \equiv & \lambda x.((x0\Omega) \bowtie 0)(((x\Omega 0) \bowtie 0)(((x11) \bowtie 0)\Omega\, 8)\Omega)\Omega
\end{array}
$$

Proof: The counterexample is taken from Plotkin [16, p 234], but our proof is similar to Proposition 8.4. If $e0\Omega \Downarrow_l 0$ and $e\Omega 0 \Downarrow_l 0$ then $e\Omega\Omega \Downarrow_l 0$ by Lemma 8.3 and continuity. Hence $e11 \Downarrow_l 0$ by continuity, and hence $e_1 e \Uparrow_l$ and $e_2 e \Uparrow_l$ for every e such $\vdash^p e : N \to N \to N$. However if we consider the continuous function q defined by:

$$
\begin{array}{rcll}
q(\bot)(\bot) & = & \bot & \\
q(0)(\bot) & = & 0 & \\
q(i)(\bot) & = & \bot & \text{for } i > 0 \\
q(\bot)(0) & = & 0 & \\
q(\bot)(i) & = & \bot & \text{for } i > 0 \\
q(i)(j) & = & i \times j &
\end{array}
$$

Then $\mathcal{M}[e_1] \bullet q = 7$ and $\mathcal{M}[e_2] \bullet q = 8$. $\square$

In the remainder of this section we will use numbers as truth values, with 0 representing true, and $i > 0$ representing false. These values can be converted to and from boolean functions using $\lambda x.x \bowtie 0$ and $\lambda x.x01$. We also extend our calculus with the following parallel conditional Q (where w with possible subscripts denotes any non-numeric whnf):

$$\begin{array}{ll} Q0ie \to i & \\ Qiej \to j & \text{for } i > 0 \\ Qeii \to i & \\ Qew_1w_2 \to W & \\ Qwij \to W & \text{for } i \neq j \\ Qww_1i \to W & \\ Qwiw_2 \to W & \\ Q0we \to W & \\ Qiew \to W & \text{for } i > 0 \end{array}$$

where $A \vdash^p Q : N \to N \to N \to N$ and $\Uparrow_l$ is extended in the obvious way to describe convergence of Qee_1e_2. We then have the following generalisation of a well-known result:

Theorem 11.2 (Plotkin) *For the Lazy Lambda Calculus with constants including Q, let the type τ be constructed using only $\to$ and N, and let $\vdash^p e_1 : \tau$ and $\vdash^p e_2 : \tau$ with no use of universally quantified types other than that required to infer $A \vdash^p C : \sigma \to \sigma' \to \sigma'$, $A \vdash^p Y : (\sigma \to \sigma) \to \sigma$, and $A \vdash^p e'_1 \bowtie e'_2 : \sigma \to \sigma \to \sigma$. Then*

$$e_1 \sqsubseteq^{\mathrm{op}}_\tau e_2 \iff \mathcal{M}[e_1] \leq_\tau \mathcal{M}[e_2]$$

Proof: Essentially the same as Plotkin [16, pp 237-240]. We define the function $(d_1 \leadsto d_2)$ such that $(d_1 \leadsto d_2)(d) = d_2$ if $d_1 \leq d$, and $(d_1 \leadsto d_2)(d) = \bot$ otherwise. Then the keys to the proof are (i) that every finite element in $\mathbf{D}_{\sigma\to\tau}$ other than $\lambda x.\bot$ can be represented as $\bigsqcup\{(d_1 \leadsto d'_n), \ldots, (d_n \leadsto d'_n)\}$ and (ii) that for each finite $d, d' \in \mathbf{D}_\tau$, there are terms denoting d, $(d \leadsto 0)$, and (if it exists) $(d \leadsto 0) \sqcup (d' \leadsto 1)$. We add to Plotkin's proof the fact that $\lambda x.\Omega$ denotes $\lambda x.\bot$ and $\lambda x.Cx0$ denotes $(\lambda x.\bot \leadsto 0)$ □

We conjecture that this theorem extends to all types, since for example Abramsky's P operator of type $\forall\alpha.\alpha \to \alpha \to \forall\beta.\beta \to \beta$ can be defined by:

$$\lambda x.\lambda y.C(Q(Cx0)\,0\,(Cy0)).$$

However, a proof of $e_1 \sqsubseteq^{\mathrm{op}}_\tau e_2 \iff \mathcal{M}[e_1] \leq_\tau \mathcal{M}[e_2]$ for all closed types τ would need to generalise both Abramsky's and Plotkin's proofs, and it is not clear how this could be accomplished.

It is an open question whether this result extends to the untyped case, i.e. whether $e_1 \sqsubseteq^{\mathrm{op}} e_2 \iff \mathcal{M}[e_1] \leq \mathcal{M}[e_2]$ or at least $e_1 \approx^{\mathrm{op}} e_2 \iff \mathcal{M}[e_1] = \mathcal{M}[e_2]$ for the untyped Lazy Lambda Calculus with constants including Q, and resolving this remains a question for further work.

12 Acknowledgements

The author is deeply indebted to Luke Ong, Andrew Moran, Ed Kazmierczak, and David Turner for useful comments. This work was supported by National University of Singapore Research Project RP900632.

References

[1] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[2] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. Technical Report 259, University of Cambridge Computer Laboratory, June 1992.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[4] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph rewriting. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture: 1987 Proceedings*, number 274 in Lecture Notes in Computer Science, pages 364–384, Berlin, 1987. Springer-Verlag.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[6] A. H. Dekker. Output as reduction in functional programming languages. Technical Report R88-11, Department of Electrical Engineering and Computer Science, University of Tasmania, Hobart, November 1988.

[7] A. H. Dekker. *Subtype Inference in Functional Languages*. PhD thesis, Department of Electrical Engineering and Computer Science, University of Tasmania, Hobart, July 1989.

[8] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture:1987 Proceedings*, number 274 in Lecture Notes in Computer Science, pages 34–45, Berlin, 1987. Springer-Verlag.

[9] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ–Calculus*. Cambridge University Press, 1986.

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

[11] J. W. Klop. *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1980.

[12] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[13] Andrew Moran. Natural semantics for non-determinism. Technical report, Department of Computing Science, Chalmers University of Technology, Göteborg, 1994.

[14] C.-H. Luke Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, Imperial College of Science and Technology, London, May 1988.

[15] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

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

[17] Research Software Limited. *Miranda System Manual: version 2.014*, April 1990.

[18] J. C. Reynolds. Three approaches to type structure. In Hartmut Ehrig et al., editors, *Mathematical Foundations of Software Development: TAPSOFT 1985 Proceedings Volume 1*, number 185 in Lecture Notes in Computer Science, pages 97–138, Berlin, 1985. Springer-Verlag.

[19] J. E. W. Smetsers. *Graph Rewriting and Functional Languages.* PhD thesis, Katholieke Universiteit Nijmegen, Nijmegen, February 1993.

[20] Joseph. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[21] David A. Turner, April 1991. Personal Communication.

[22] Marko van Eekelen, Halbe Huitema, Eric Nöcker, Sjaak Smetsers, and Rinus Plasmeijer. *Concurrent Clean language manual: version 0.8.4.* Katholieke Universiteit Nijmegen, Nijmegen, April 1993.

[23] Christopher P. Wadsworth. The relation between computational and denotational properties for scott's $\mathbf{D}_\infty$-models of the lambda-calculus. *SIAM. J. Comput*, 5(3):488–521, September 1976.