# Böhm Trees for 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`

August 15, 1994

**Abstract**

In this paper we present a Böhm Tree model for the Lazy Lambda Calculus with constants, which extends Abramsky's pure Lazy Lambda Calculus. The Lazy Lambda Calculus with constants forms a basis for modern lazy functional programming languages, which usually provide a call-by-value facility which is able to distinguish between the values $\perp$ and $\lambda x.\perp$, as well as providing strict arithmetic primitives. The Böhm Tree model we present also acts as an improved model for the pure Lazy Lambda Calculus. In addition, the paper provides a framework for studies of Böhm Trees in more general systems.

## 1 Introduction

The *Lazy Lambda Calculus* was first defined by Abramsky [1, 2] and further studied by Ong [10] as a foundation for lazy functional programming languages as they are actually implemented, i.e. taking account of the fact that evaluation of a function terminates with a *weak head normal form* of the form $\lambda x.e$ (or with a pointer to a code block which corresponds to such a weak head normal form). Consequently the values $\perp$ and $\lambda x.\perp$, which are equated in most models of the lambda calculus, must be differentiated. It is possible to argue that $\perp$ and $\lambda x.\perp$ are extensionally equal, but most lazy functional programming languages also provide a call-by-value (strict function) facility which distinguishes between the two values operationally. Thus the Lazy Lambda Calculus is the only possible model for modern lazy functional programming languages.

In [6] we presented the *Lazy Lambda Calculus with constants*, which extends the Lazy Lambda Calculus with natural numbers and arithmetic, and we argued that by providing strict arithmetic primitives, this gives a more realistic foundation for modern lazy functional programming languages than the pure Lazy

Lambda Calculus. In this paper we discuss Böhm Tree models for our calculus. A Böhm Tree model for a combinatory logic version of this calculus was briefly presented in [4]. The treatment of Böhm trees in this paper is based on that of Barendregt [3], which is the only fully detailed study of Böhm Trees available. However, this treatment applies only to the pure Lambda Calculus. The body of this paper provides a general framework for proving Böhm Tree properties for other reduction systems, by identifying the key propositions which require a proof depending on the exact nature of the reduction rules. Once these propositions are proved, the major results of [3] follow.

# 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 [10, p 134] or the *seq* operator of Miranda[1] [11], with the rewrite rule:

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

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).

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 [9, p 122] we consider $\lambda x.e$ as shorthand for $\lambda([x]e)$, where $[x]$ is a variable binding construct, and $\lambda$ 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 $\lambda$.

[1]Miranda is a trademark of Research Software Limited

## 3 Rewrite Rules

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

- $\beta$ ($\beta$-reduction) $(\lambda x.e)\, e' \to e\,[e'/x]$
- $W$ (error-propagation) $W e \to W$
- $\nu$ (number-swap) $i\, e \to e\, i$
- $\delta$ (strict-constant) $b\, e \to e'$ 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. The body of the paper does not depend on the exact details of the table, we only assume that $b\, e$ is a redex for every $b$ and for $e$ of the form $b'$, $i$, $W$, or $\lambda x.e$, and that the contractum is a closed normal form 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. $\lambda$, $W$, $i$ 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:

$$
\begin{aligned}
e \to^n e_n &\iff e \equiv e_0 \to \cdots \to e_n \text{ for some } e_i \\
e \downarrow e' &\iff e \to^* e' \text{ and } e' \text{ is in normal form} \\
e \downarrow &\iff e \downarrow e' \text{ for some } e' \\
e \uparrow &\iff \sim e \downarrow
\end{aligned}
$$

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

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

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

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 [7]. However, there is no such mechanism for 'pre-evaluating' the argument to $C$. Another consequence of the $\nu$ rule is that:

$$00 \to 00$$

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

The reduction system defined by these rules is a combinatory reduction system [9, 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) [9, pp 126,130,189]. Hence the following result applies:

**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) Klop [9, p 163].

(ii) From (i).

(iii) Klop [9, 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. $b_n(\ldots(b_1(x\vec{e}_0)\vec{e}_1)\ldots)\vec{e}_n$ for $n \geq 1$

*Proof*: By considering cases. □

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 $e \rightarrow e'$ then $e'$ is of the same form, and corresponding sub-vectors have the same length, i.e:*

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

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

(iii) If $b_n(\ldots(b_1(xe_{01}\ldots e_{0m})e_{11}\ldots e_{1m})\ldots)e_{n1}\ldots e_{nm} \rightarrow e'$ 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. □

## 4 Lazy Reduction

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

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

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

**Proposition 4.1**

*(i) If $e \rightarrow_l e'$ then $e \rightarrow_{lf} e'$.*

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

*Proof*:

(i) From the definitions.

(ii) By induction on the structure of $e$.

□

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

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

Only in case (5) of the rewrite rules is the redex head symbol not the leftmost symbol in the term. As discussed above, 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 [7]. The introduction of $C$, however, does require case (5).

**Proposition 4.2**

*(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).

□

## 5 Böhm Trees

We define Böhm trees similarly to Barendregt [3], which develops in detail the theory of Böhm trees for the pure lambda calculus. The extension of the Böhm tree concept to other systems (as is done informally in [9, pp 216–220]) has however fallen into the category of folk theorems. We will show that the machinery of [3, sections 10.2, 14.3, and 18.3] extends to the Lazy Lambda Calculus with constants, providing proofs of those propositions that rely on the exact nature of the new rewrite rules. Our treatment will also give a Böhm tree semantics for the pure Lazy Lambda Calculus which is simpler than that of [10].

A Böhm tree is a partially labelled (possibly infinite) tree where subtrees have the same form as whnfs (i.e. the nodes contain atoms and/or variables). We use the symbol $\perp$ to represent an undefined label, and write $\mathcal{B}$ for the set of all such trees. We define $BT(e)$, the Böhm tree of a term $e$, as follows:

$$
\begin{array}{ll}
\text{if } e\uparrow_l & \text{then } BT(e) = \perp \\
\text{if } e \to^* a & \text{then } BT(e) = a \\
\text{if } e \to^* \lambda x.e' & \text{then } BT(e) = \lambda x.BT(e') \\
\text{if } e \to^* x\,\vec{e} & \text{then } BT(e) = x\,\vec{BT}(\vec{e}\,) \\
\text{if } e \to^* b_n(\ldots(b_1(x\,\vec{e}_0)\vec{e}_1)\ldots)\vec{e}_n & \text{then} \\
\quad BT(e) = b_n(\ldots(b_1(x\,\vec{BT}(\vec{e}_0))\,\vec{BT}(\vec{e}_1))\ldots)\,\vec{BT}(\vec{e}_n) & \\
\text{where } \vec{BT}(e_1\ldots e_n) = (BT(e_1))\ldots(BT(e_n)) &
\end{array}
$$

We denote Böhm trees using $p$ and $q$ with possible subscripts and superscripts.

**Proposition 5.1** *For each $e$, $BT(e)$ is uniquely defined.*

*Proof*: By Proposition 4.1 (ii), exactly one of the 5 cases applies. Uniqueness follows from Propositions 3.3 and 3.1 (i). □

As a result of this proposition, it is possible to define Böhm trees using $\to_l^*$ instead of $\to^*$. However, our definition emphasizes the nature of the Böhm tree as a kind of infinitary normal form, as discussed by Klop [9, pp 216–220] Note that since we consider $\lambda x.e$ as shorthand for $\lambda([x]e)$, $\lambda x.\perp$ is a legitimate Böhm tree, as is the infinite Böhm tree:

$$
BT(Y(\lambda x.\lambda y.x)) \quad = \quad \begin{array}{c} \lambda y. \\ | \\ \lambda y. \\ | \\ \vdots \end{array}
$$

This avoids the complexity of Levy-Longo trees introduced by Ong [10, p 39] for the pure Lazy Lambda Calculus.

**Corollary 5.2 (Barendregt 10.1.6)**

*(i) If $e_1 \to^* e_2$ then $BT(e_1) = BT(e_2)$.*

*(ii) If $e_1 = e_2$ then $BT(e_1) = BT(e_2)$.*

*Proof*:

(i) Follows immediately from uniqueness of the definition.

(ii) From (i).

□

We define $p \subseteq q$ if $p$ results from $q$ by replacing zero or more subtrees of $q$ by $\perp$. this is clearly a partial order. We define $p \cup q$ to be the least upper bound of $p$ and $q$, if it exists. We define $p^{(n)}$ to be the result of replacing subtrees of $p$ at depth $n$ by $\perp$, i.e.

$$\begin{array}{rcl} p^{(0)} & = & \perp \\ \perp^{(n)} & = & \perp \\ a^{(n+1)} & = & a \\ (\lambda x.p)^{(n+1)} & = & \lambda x.p^{(n)} \\ (x\,\vec{p}\,)^{(n+1)} & = & x\,\vec{p}^{\,((n))} \\ (b_m(\ldots(b_1(x\,\vec{p}_0)\vec{p}_1)\ldots)\vec{p}_m)^{(n+1)} & = & b_m(\ldots(b_1(x\,\vec{p}_0{}^{((n))})\,\vec{p}_1{}^{((n))})\ldots)\,\vec{p}_m{}^{((n))} \\ (p_1\ldots p_m)^{((n))} & = & p_1{}^{(n)}\ldots p_m{}^{(n)} \end{array}$$

Clearly $p^{(n)} \subseteq p$ for every $n$. It is also useful to define $BT^n(e) = (BT(e))^{(n)}$.

**Proposition 5.3 (Barendregt 10.2.2)** *The set $\mathcal{B}$ of trees under $\subseteq$ forms a consistently complete algebraic cpo with $p = \bigcup_n p^{(n)}$.*

*Proof*: Barendregt [3, p 229]. □

We define a map $M$ from *finite* Böhm trees to terms as follows, and use this to define $e^{(n)} \equiv M(BT^n(e))$.

$$\begin{array}{rcl} M(\perp) & \equiv & \Omega \\ M(a) & \equiv & a \\ M(\lambda x.p) & \equiv & \lambda x.M(p) \\ M(x\,\vec{p}\,) & \equiv & x\,\vec{M}(\vec{p}\,) \\ M(b_n(\ldots(b_1(x\,\vec{p}_0)\vec{p}_1)\ldots)\vec{p}_n) & \equiv & b_n(\ldots(b_1(x\,\vec{M}(\vec{p}_0))\,\vec{M}(\vec{p}_1))\ldots)\,\vec{M}(\vec{p}_n) \\ \vec{M}(p_1\ldots p_n) & \equiv & (M(p_1))\ldots(M(p_n)) \end{array}$$

**Proposition 5.4** *For finite $p$, $BT(M(p)) = p$.*

*Proof*: By induction on the (finite!) structure of $p$. □

The *tree topology* on terms [3, p 230] is the smallest topology such that $BT$ is a continuous map. In particular, it gives the following equivalence relation and preorder on terms:

$$\begin{array}{rcl} e_1 \simeq e_2 & \Longleftrightarrow & BT(e_1) = BT(e_2) \\ e_1 \preceq e_2 & \Longleftrightarrow & BT(e_1) \subseteq BT(e_2) \end{array}$$

**Proposition 5.5 (Barendregt 10.2.7)** *The sets $O_{e,k} = \{e' \mid e^{(k)} \preceq e'\}$ form a basis for the tree topology.*

*Proof*: Barendregt [3, pp 230–231]. $\square$

# 6 Approximate Normal Forms

To examine the continuity properties of Böhm trees, we add a new constant $\square$ with reduction rules:

$$\begin{array}{rcl} \square e & \to & \square \\ b\,\square & \to & \square \end{array}$$

Barendregt [3, p 364] uses the symbol $\perp$ in the same context. We write $\to_\square$ for reduction using these rules in addition to the previous rules. The new system is still a non-ambiguous, left-linear and left-normal combinatory reduction system [9, pp 120,126,130,189] so that Proposition 3.1 still applies. We define $\to^*_\square$, $\downarrow_\square$, and $\uparrow_\square$ analogously to the definitions for the original reduction system. We also extend our earlier definition of *lazy reduction* to the relation $\to_{l\square}$, by adding the rules:

$$\begin{array}{rcl} \square e\,\vec{e} & \to_{l\square} & \square\,\vec{e} \\ b\,\square\,\vec{e} & \to_{l\square} & \square\,\vec{e} \end{array}$$

We define $\to^*_{l\square}$, $\downarrow_{l\square}$, and $\uparrow_{l\square}$ analogously to the earlier definitions. We also extend the definition of Böhm trees to include $\square$, modifying the earlier definition to replace $\uparrow_l$ by $\uparrow_{l\square}$ and $\to^*$ by $\to^*_\square$, and adding:

$$\text{if } e \to^*_\square \square \quad \text{then } BT(e) = \perp$$

In addition we redefine:

$$M(\perp) \quad \equiv \quad \square$$

It is clear that the propositions of the previous section still apply to the extended definitions.

We say that $e$ is an *approximate normal form* if $e$ is in normal form under $\to_\square$, and $e$ is an *approximate normal form of* $e'$ if $e$ is an approximate normal form and $e \preceq e'$. We write $\mathcal{A}(e)$ for the set of approximate normal forms of e. The following proposition shows that approximate normal forms are in one-to-one correspondence with finite Böhm trees, with $\square$ replacing $\perp$:

**Proposition 6.1**

*(i) For each finite Böhm tree $p$, $M(p)$ is an approximate normal form with $BT(M(p)) = p$.*

*(ii) For each approximate normal form $e$, $BT(e)$ is finite and $M(BT(e)) \equiv e$.*

*Proof*:

(i) From the definition of $BT$ and Proposition 5.4.

(ii) From the definitions of $BT$ and $M$.

$\square$

We define an operator $\omega$ which constructs approximate normal forms of terms as follows:

$$\begin{array}{rcl}
\omega(e) & \equiv & \square \quad \text{if } e \text{ is not in whnf} \\
\omega(a) & \equiv & a \\
\omega(\lambda x.e) & \equiv & \lambda x.\omega(e) \\
\omega(x\,\vec{e}\,) & \equiv & x\,\vec{\omega}(\vec{e}\,) \\
\omega(b_n(\ldots(b_1(x\,\vec{e}_0)\vec{e}_1)\ldots)\vec{e}_n) & \equiv & b_n(\ldots(b_1(x\,\vec{\omega}(\vec{e}_0))\,\vec{\omega}(\vec{e}_1))\ldots)\,\vec{\omega}(\vec{e}_n) \\
\vec{\omega}(e_1\ldots e_n) & \equiv & (\omega(e_1))\ldots(\omega(e_n))
\end{array}$$

We also define $\mathcal{A}'(e) = \{\omega(e') \mid e \to^* e'\}$.

**Proposition 6.2 (Barendregt 14.3.7)** *Let $e$ be a term possibly containing $\square$. Then:*

*(i) $\omega(e)$ is an approximate normal form of $e$.*

*(ii) If $e \to^* e'$ then $\omega(e) \preceq \omega(e')$.*

*(iii) $\mathcal{A}'(e) \subseteq \mathcal{A}(e)$.*

*Proof*:

(i) By induction on the structure of $e$, $\omega(e)$ is in normal form under $\to_\square$ and $\omega(e) \preceq e$.

(ii) By induction on the length of the reduction sequence $e \to^* e'$, and by induction on the structure of $e$ for $e \to e'$, using the fact that if $e$ is not in whnf, $\omega(e) \equiv \square \preceq \omega(e')$.

(iii) Let $\omega(e') \in \mathcal{A}'(e)$ with $e \to^* e'$. Then $\omega(e') \preceq e' \simeq e$ is an approximate normal form of $e$ by (i) and Corollary 5.2.

$\square$

**Lemma 6.3** *For each term $e$ possibly containing $\square$, and each $n \geq 0$, there exist $m$ and $e'$ such that $e \to^m e'$ and $e^{(n)} \preceq \omega(e')$.*

*Proof*: By induction on $n$, using Proposition 3.3. For $n = 0$, $e^{(0)} \equiv \square \preceq \omega(e')$. If $e \simeq \square$ the result follows trivially, otherwise $e \to^k e''$ for some $k$ and $e''$ in whnf, and the result follows by induction. $\square$

Since approximate normal forms are in one-to-one correspondence with finite Böhm Trees, we extend the definition of $\cup$ to pairs of approximate normal forms. If $S$ is a set of approximate normal forms, $\bigcup S$ may not be a finite Böhm Tree, but we write $\bigcup S \simeq e$ if $\bigcup S = BT(e)$.

**Proposition 6.4 (Barendregt 14.3.9, 14.3.10)** *Let $e$ be a term possibly containing $\Box$. Then:*

*(i)* *$\mathcal{A}(e)$ is directed.*

*(ii)* *$\mathcal{A}'(e)$ is directed.*

*(iii)* *$e \simeq \bigcup \mathcal{A}(e) = \bigcup \mathcal{A}'(e)$.*

*Proof*:

(i) If $e_1, e_2 \in \mathcal{A}(e)$, then $e_1 \cup e_2 \in \mathcal{A}(e)$.

(ii) If $\omega(e_1), \omega(e_2) \in \mathcal{A}'(e)$ with $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$, then by Proposition 3.1 there exists $e'$ such that $e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$, and hence $\omega(e') \in \mathcal{A}'(e)$. Also $\omega(e_1) \preceq \omega(e')$ and $\omega(e_2) \preceq \omega(e')$ by Proposition 6.2 (ii).

(iii) Clearly $\bigcup \mathcal{A}(e) \simeq e$ by Proposition 5.3, and $\bigcup \mathcal{A}'(e) \subseteq \bigcup \mathcal{A}(e)$ by Proposition 6.2 (iii). Since $e' \in \mathcal{A}(e)$ corresponds to a finite Böhm Tree, $e' \preceq e^{(n)}$ for some $n$, and by Lemma 6.3, $e \rightarrow^* e''$ and $e^{(n)} \preceq \omega(e'')$ for some $e''$. Since $\omega(e'') \in \mathcal{A}'(e)$, $\bigcup \mathcal{A}(e) \subseteq \bigcup \mathcal{A}'(e)$, and hence $\bigcup \mathcal{A}(e) = \bigcup \mathcal{A}'(e)$.

$\Box$

**Corollary 6.5 (Barendregt 14.3.11)** *Let $e_1$ and $e_2$ be terms possibly containing $\Box$. Then $e_1 \preceq e_2$ if and only if for every ${e'}_1$ such that $e_1 \rightarrow^* {e'}_1$ there exists ${e'}_2$ such that $e_2 \rightarrow^* {e'}_2$ and $\omega({e'}_1) \preceq \omega({e'}_2)$.*

*Proof*: Barendregt [3, p 368], using Propositions 5.2, 6.2 and 6.4. $\Box$

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 $e$. Contexts are not considered modulo renaming of bound variables, so that free variables in $e$ may become bound in $C[e]$. The following proposition differs slightly from Lemma 14.3.12 of Barendregt [3, p 369], distinguishing occurrences of $\Box$ substituted into $C[\Box]$ from occurrences in $C[\cdot]$ itself:

**Proposition 6.6 (Barendregt 14.3.12)** *Let $e$ be a term possibly containing $\Box$, and $C[\cdot]$ be a context possibly containing $\Box$. Then $C[\Box] \preceq C[e]$.*

*Proof*: Let $C[\Box] \rightarrow^* D[\Box]$. Then $C[e] \rightarrow^* D[e]$. By induction on the structure of $D[\cdot]$, $\omega(D[\Box]) \preceq \omega(D[e])$, and thus Corollary 6.5 applies. $\Box$

**Corollary 6.7 (Barendregt 14.3.13)** *If $e \in \mathcal{A}(e')$ then $C[e] \preceq C[e']$.*

*Proof*: By repeated application of Proposition 6.6. $\Box$

The following result is non-trivial, since variables in $e_1$ may become bound in $C[e_1]$. The corresponding proposition for the pure lambda calculus was first proved by Welch. The corresponding proposition for the pure Lazy Lambda Calculus is stated but not proved in [10, p 50].

**Proposition 6.8 (Barendregt 14.3.18)** *Let $C[e_1]$ and $e_2$ be terms not containing $\Box$, with $C[e_1] \rightarrow^* e_2$. Then there exist $e'_1$ and $e'_2$ such that $e_1 \rightarrow^* e'_1$, $e_2 \rightarrow^* e'_2$, $C[e'_1] \rightarrow^* e'_2$, and if the redex head symbols in $e'_1$ are marked, the reduction $C[e'_1] \rightarrow^* e'_2$ does not reduce any marked redexes.*

*Proof*: By defining labelled reduction:

$$
\begin{array}{rcl}
(e^n)^m & \rightarrow_{lab} & e^{\min(n,m)} \\
(\lambda x.e)^{n+1} e' & \rightarrow_{lab} & (e[(e')^n/x])^n \\
W^{n+1} e & \rightarrow_{lab} & W^n \\
i^{n+1} e & \rightarrow_{lab} & (e^n i^n)^n \\
b^{n+1} e^{m+1} & \rightarrow_{lab} & e''
\end{array}
$$

where in the last line $b\,e \rightarrow e'$ is an instance of a reduction rule, and $e''$ is the result of labelling $e'$ and all its subterms with $\min(n, m)$.
Labelled reduction is Church-Rosser, since if we interpret $e^n$ as $L_n e$ for a set of new constants $L_n$, Proposition 3.1 still applies. Labelled reduction is also strongly normalising by Klop [9, p 189]. Furthermore any reduction can be lifted to a labelled reduction by taking the initial labels to be sufficiently large (e.g. $\geq k$ where $k$ is the number of reduction steps in the reduction sequence). Conversely if $(e_1)^n \rightarrow^*_{lab} (e_2)^m$, then erasing the labels gives a legal reduction using $\rightarrow^*$.
The proposition is proved by lifting the reduction $C[e_1] \rightarrow^* e_2$ to a labelled reduction $C'[e''_1] \rightarrow^*_{lab} e''_2$, taking $e'''_1$ to be the normal form of $e''_1$ under labelled reduction (which exists by strong normalisation), using the Church-Rosser property for labelled reduction to show $C'[e'''_1] \rightarrow^*_{lab} e'''_2$ and $e''_2 \rightarrow^*_{lab} e'''_2$ for some $e'''_2$, and then erasing the labels in the resulting reductions. $\Box$

This proposition combines with the following lemma to prove the major proposition 6.10, which was first proved for the pure lambda calculus by Wadsworth.

**Lemma 6.9 (Barendregt 14.3.17)** *Let $e_1$ and $e_2$ be terms not containing $\Box$, and let a set of redex head symbols in $e_1$ be marked. Let $e_1 \rightarrow^* e_2$ be a reduction which does not reduce any marked redexes, and let $e'_1$ and $e'_2$ be the result of replacing (maximal under containment) marked redexes in $e_1$ and $e_2$ respectively by $\Box$. Then $e'_1 \rightarrow^* e'_2$.*

*Proof*: By induction on the length of the reduction sequence $e_1 \rightarrow^* e_2$, noting that any redex which is reduced must either (i) be disjoint from the marked redexes, (ii) be properly contained in a marked redex, or (iii) properly contain a maximal marked redex. In the last case, if $e \rightarrow e'$ is an instance of a rewrite rule, then so is the result of replacing a redex inside $e$ by $\Box$. $\Box$

**Proposition 6.10 (Barendregt 14.3.19)** *Let $C[e]$ be a term not containing $\Box$. Then for every $e_1 \in \mathcal{A}'(C[e])$ there exists $e_2 \in \mathcal{A}'(e)$ with $e_1 \preceq C[e_2]$.*

*Proof*: Barendregt [3, pp 370–371], using Propositions 5.2, 6.2 (i) and (ii), 6.8 and 6.9. The proof given there is due to Lévy. $\Box$

**Corollary 6.11 (Barendregt 14.3.20)** *Let $e_1$ and $e_2$ be terms not containing $\Box$, and $C[\cdot]$ a context not containing $\Box$. Then:*

*(i)* $C[e] \simeq \bigcup\{C[e'] \mid e' \in \mathcal{A}(e)\} = \bigcup\{C[e'] \mid e' \in \mathcal{A}'(e)\}$.

*(ii)* $C[e] \simeq \bigcup_n C[e^{(n)}]$.

*(iii)* $e_1 \preceq e_2 \Longrightarrow C[e_1] \preceq C[e_2]$, *i.e.* $\preceq$ *is a precongruence.*

*Proof*: Barendregt [3, pp 371–372], using Propositions 6.4 (iii) and 6.7. $\Box$

From this corollary the major continuity theorem of [3, section 14.3] follows. This result allows the set of Böhm Trees to be made into a $\lambda$-model.

**Proposition 6.12 (Barendregt 14.3.21, 14.3.22)**

*(i) The context $C[\cdot]$, considered as a map on terms not containing $\Box$, is continuous with respect to the tree topology.*

*(ii) The operation of abstraction on terms is continuous with respect to the tree topology.*

*(iii) Application of terms is continuous with respect to the tree topology.*

*Proof*: Barendregt [3, pp 372–373], using Propositions 6.10 and 6.11 (iii). $\Box$

The following proposition gives the relation between $\preceq$ and the operational preorders described in [6].

**Proposition 6.13** *Let $e_1$ and $e_2$ be terms not containing $\Box$, and $C[\cdot]$ a context not containing $\Box$. If $e_1 \preceq e_2$ and $C[e_1]\downarrow_l e'_1$ for some $e'_1$, then $C[e_2]\downarrow_l e'_2$ for some $e'_2$, and $e'_2$ is a whnf of the same form as $e'_1$ (i.e. $a$, $\lambda x.e$, $x\,\vec{e}$, or $b_n(\ldots(b_1(x\vec{e}_0)\vec{e}_1)\ldots)\vec{e}_n$) with corresponding sub-vectors of the same length.*

*Proof*: From the definition of $BT$, using Corollary 6.11 (iii). $\Box$

# 7 The Böhm Tree Model

Proposition 6.12 allows the definition of a Böhm Tree model for the Lazy Lambda Calculus with constants. We can define application and substitution on Böhm Trees by:

$$\begin{aligned} p \bullet q &= \bigcup_n BT((M(p^{(n)}))(M(q^{(n)}))) \\ p[q/x] &= \bigcup_n BT(M(p^{(n)}[q^{(n)}/x])) \end{aligned}$$

Note that the substitution operation also reduces the redexes created after substitution.

**Proposition 7.1 (Barendregt 18.3.3, 18.3.4)** *The limits in the above definition exist, the operations defined are continuous, and:*

*(i)* $BT(e_1e_2) = BT(e_1) \bullet BT(e_2)$.

*(ii)* $BT(e[e'/x]) = (BT(e))[BT(e')/x]$.

*Proof*: Barendregt [3, pp 486–488], using Corollary 6.11 (iii) and Proposition 6.12. $\square$

For all terms $e$ not containing $\square$ and *environments* $\rho$ mapping the free variables $FV(e)$ of $e$ to Böhm Trees, we define $\mathcal{M}[e]_\rho$ (the denotation of $e$ as a Böhm Tree in the environment $\rho$) as follows. For closed $e$, $\rho$ is not used, and we write simply $\mathcal{M}[e]$.

$$\begin{aligned}
\mathcal{M}[a]_\rho &= a \\
\mathcal{M}[x]_\rho &= \rho(x) \\
\mathcal{M}[e_1e_2]_\rho &= \mathcal{M}[e_1]_\rho \bullet \mathcal{M}[e_2]_\rho \\
\mathcal{M}[\lambda x.e]_\rho &= \lambda x.\mathcal{M}[e]_{\rho[x/x]}
\end{aligned}$$

In the last line the modified environment maps the variable $x$ to the Böhm Tree $x$.

**Proposition 7.2 (Barendregt 18.3.10)**

*(i) The set of trees $\mathcal{B}$ together with $\bullet$ and $\mathcal{M}$ forms a $\lambda$-model.*

*(ii)* $BT(e_1) = BT(e_2) \iff \forall\rho.\ \mathcal{M}[e_1]_\rho = \mathcal{M}[e_2]_\rho$.

*(iii)* $\forall\rho.\ \mathcal{M}[We]_\rho = W \bullet \mathcal{M}[e]_\rho = W$.

*(iv)* $\forall\rho.\ \mathcal{M}[i\,e]_\rho = i \bullet \mathcal{M}[e]_\rho = \mathcal{M}[e]_\rho \bullet i$.

*(v)* $\forall\rho.\ \mathcal{M}[b\,e]_\rho = b \bullet \mathcal{M}[e]_\rho = \mathcal{M}[e']$, *if* $b\,e \to e'$ *is a rewrite rule instance.*

*Proof*: For (i) and (ii) the proof is given in Barendregt [3, pp 488–491], using Proposition 7.1. Cases (iii), (iv) and (v) follow from the definition of $\bullet$, noting that in case (v) $e'$ is closed and does not depend on $\rho$. $\square$

Thus we have constructed a Böhm Tree $\lambda$-model which is also a model for our new reduction rules, having modified the framework of Barendregt [3, sections 10.2, 14.3, and 18.3] to deal with our new reduction system. This required proving the key propositions 5.1, 5.2, 6.1, 6.2, 6.3, 6.4, 6.8, and 6.9 to take account of the additional reduction rules and the altered definition of Böhm Trees.

Proposition 6.13 shows that the Böhm Tree preorder $\preceq$ implies the operational preorders described in [6]. The continuity properties which follow from the key propositions allow the definition of application and substitution on Böhm Trees, and hence the semantic function $\mathcal{M}[e]_\rho$. This provides us with a (non-extensional) semantics for lazy functional programs.

# 8 Acknowledgements

The author is deeply indebted to Henk Barendregt, 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] 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.

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

[6] A. H. Dekker. The lazy lambda calculus with constants. Technical Report TRA7/94, Department of Information Systems and Computer Science, National University of Singapore, July 1994.

[7] 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.

[8] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and $\lambda$–Calculus*. Cambridge University Press, 1986.

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

[10] 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.

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