Operational Interpretations of Linear Logic

David N. Turner, An Teallach Limited, dnt@an-teallach.com
Philip Wadler, Bell Labs, Lucent Technologies, wadler@research.bell-labs.com

Abstract

Two different operational interpretations of intuitionistic linear logic have been proposed in the literature. The simplest interpretation recomputes non-linear values every time they are required. It has good memory-management properties, but is often dismissed as being too inefficient. Alternatively, one can memoize the results of evaluating non-linear values. This avoids any recomputation, but has weaker memory-management properties. Using a novel combination of type-theoretic and operational techniques we give a concise formal comparison of the two interpretations. Moreover, we show that there is a subset of linear logic where the two operational interpretations coincide. In this subset, which is sufficiently expressive to encode call-by-value lambda-calculus, we can have the best of both worlds: a simple and efficient implementation, and good memory-management properties.

Keywords: linear logic, operational semantics, call-by-value lambda calculus, memory management.

1 Introduction

Two different operational interpretations of linear logic have been proposed in the literature. The two interpretations differ primarily in their treatment of non-linear values. The simplest interpretation recomputes non-linear values every time they are required. This strategy is easy to implement, and preserves the single-pointer property: values of linear type are guaranteed to have exactly one pointer to them. However, it is too inefficient for many purposes because of the costs of recomputing non-linear values.

An alternative strategy is to share the results of evaluating non-linear values (in the same way that call-by-need languages share the results of evaluating closures). This avoids any copying or recomputation, but unfortunately loses the single-pointer property: values of linear type are no longer guaranteed to have exactly one pointer to them.

This paper uses a novel combination of type-theoretic and operational techniques to make a concise formal comparison of the memory-management properties of the above two operational interpretations of intuitionistic linear logic. Moreover, we show that there is a subset of linear logic where the two operational interpretations coincide. In this subset, which is sufficiently expressive to encode call-by-value lambda-calculus, we can have the best of both worlds: a simple and efficient implementation, and good memory-management properties.

1.1 Operational Models

The memory-management properties of linear logic are sufficiently subtle that it is well worth writing down a formal semantics for linear logic at a level of detail sufficient to describe storage allocation and deallocation. Chirimar, Gunter and Riecke [CGR92, CGR95] have already given one such formal semantics, which uses a reference-counting implementation of linear logic. They prove that, in their implementation, not all values of linear type have exactly one pointer to them. Instead, they identify a weaker property: a particular subset of the values of linear type which do have exactly one pointer to them. Our formalism is similar to that used by Morrisett, Felleisen, and Harper [MFH95] to model garbage collection algorithms.

Our work extends Chirimar, Gunter and Riecke's work in three respects: (1) We compare two different operational interpretations of linear logic. (In particular, we consider the interpretation of linear logic which recomputes non-linear values, and prove that it has strictly stronger memory-management properties than the interpretation considered by Chirimar, Gunter and Riecke.) (2) We formalise our operational model at a higher level of abstraction, where we do not keep explicit reference counts. This significantly simplifies our semantics and our subsequent formalisations of memory-management properties: their interpreter requires fifteen rules, the largest of which is fifteen lines long (Section 4.2 of Chirimar, Gunter and Riecke [CGR95]) whereas we have nine reduction rules, the longest of which is two lines (Definition 4.1). (3) Our use of type-theoretic techniques to formalise memory-management properties yields concise statements of the properties we are interested in, and lets us reuse standard proof techniques from type theory. Moreover, it allows us to relate our memory-management properties directly to linear logic.

1.2 Overview

Section 2 presents the syntax, typing rules and a simple operational semantics for intuitionistic linear logic. In Section 3 we refine our operational semantics to capture the storage-allocation properties of programs. In Section 4 we

To appear in *Theoretical Computer Science*, special issue on linear logic. This work was performed at Glasgow University.

modify our operational semantics so that the results of evaluating non-linear values are shared, and then compare the storage allocation properties of the resulting interpretation of linear logic. Section 5 describes related work. Section 6 offers some conclusions and ideas for further work.

2 Intuitionistic Linear Logic

We use (a slight variant of) Plotkin's formulation of intuitionistic linear logic [Plo93] which, unlike many formulations of linear logic, has no explicit syntax for duplicating or discarding values. Such term constructs are important for models of reference-counting implementations, but are unnecessary here since our operational semantics is formulated at a slightly higher level of abstraction where we do not keep explicit reference counts. (Other closely related formulations of linear logic can be found in [Wad93, BBdPH93, MOTW95].)

2.1 Syntax

Definition 2.1 describes the syntax of terms. We differ from Plotkin in that we distinguish between two kinds of variables: non-linear variables (ranged over by x, y and z), which may be freely duplicated or discarded, and linear variables (ranged over by a, b, c and d), which must be used exactly once. It is useful to make this distinction syntactically, rather than semantically, since we need to treat linear and non-linear variables differently in our operational model.

Definition 2.1 (Term syntax)

$$\begin{array}{llll}
\text{Terms} & e, f, g & ::= & x \mid a \mid e_1 \otimes e_2 \mid \text{let}\, a_1 \otimes a_2 = e \,\text{in}\, f \mid \\
 & & & \star \mid \text{let}\, \star = e \,\text{in}\, e \mid \lambda a.e \mid e\ f \mid \\
 & & & !e \mid \text{let}\, !x = e \,\text{in}\, f \\
 & & & \\
\text{Values} & v & ::= & \star \mid \lambda a.e \mid v_1 \otimes v_2 \mid !e
\end{array}$$

For simplicity, we do not include recursion, or sum types, in our calculus (their absence has no effect on the operational interpretation of non-linear values, which is the key feature we intend to model).

2.2 Operational Semantics

Definition 2.2 presents a simple operational semantics for closed terms: evaluation judgments have the form $e \Downarrow v$, meaning "e terminates, returning a value v." (Note that the distinction between linear and non-linear variables is not apparent in this operational semantics. In Section 3, when we refine these rules to capture the storage-allocation properties of programs, the distinction will become clear.)

The expression $\star$ denotes the linear unit value. The term 'let $\star = e$ in f' eliminates a unit value: it evaluates e to a unit value, and then evaluates f.

The expression $e_1 \otimes e_2$ constructs a linear pair (the components of which are evaluated strictly). The expression 'let $a_1 \otimes a_2 = e$ in f' destructs the linear pair computed by e, binding the resulting values v_1 and v_2 to the linear variables a_1 and a_2.

The expression $\lambda a.e$ constructs a linear function. Function application, written $e\,f$, evaluates e to some linear function $\lambda a.g$ and then evaluates the argument expression f, substituting the resulting value v for the lambda-bound variable a in the function body g. Note that the bound variable in a linear function is always a linear variable, which much be used exactly once, so it makes sense to evaluate function arguments strictly.

Definition 2.2 (Operational semantics)

$$\star\text{-I}\ \frac{}{\star \Downarrow \star} \qquad \star\text{-E}\ \frac{e \Downarrow \star \qquad f \Downarrow v}{\text{let}\, \star = e \,\text{in}\, f \Downarrow v}$$

$$\otimes\text{-I}\ \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \otimes e_2 \Downarrow v_1 \otimes v_2} \qquad \multimap\text{-I}\ \frac{}{\lambda a.e \Downarrow \lambda a.e}$$

$$\otimes\text{-E}\ \frac{e \Downarrow v_1 \otimes v_2 \qquad [v_1/a_1, v_2/a_2]f \Downarrow v}{\text{let}\, a_1 \otimes a_2 = e \,\text{in}\, f \Downarrow v}$$

$$\multimap\text{-E}\ \frac{e \Downarrow \lambda a.g \qquad f \Downarrow v \qquad [v/a]g \Downarrow v'}{e\ f \Downarrow v'}$$

$$!\text{-I}\ \frac{}{!e \Downarrow !e} \qquad !\text{-E}\ \frac{e \Downarrow !g \qquad [g/x]f \Downarrow v}{\text{let}\, !x = e \,\text{in}\, f \Downarrow v}$$

The replicated expression $!e$ represents a non-linear value. The simplest interpretation of $!e$ is as a suspended computation which, when forced, will return some value (for this reason, we often refer to expressions of the form $!e$ as !-closures). References to !-closures may be freely duplicated or discarded. The expression 'let $!x = e$ in f' evaluates e until we get a closure $!g$, and then substitutes the expression g for the variable x in f.

2.3 Typing Rules

Our syntax of types is standard: $\star$ is the linear unit type, $A \otimes B$ is the type of linear pairs, $A \multimap B$ is the type of linear functions, and $!A$ is the type of non-linear values.

Definition 2.3 (Type syntax)

Types	$A, B, C ::= \star \mid A \otimes B \mid A \multimap B \mid !A$
Linear contexts	$\Gamma, \Delta ::= a_1 : A_1, \ldots, a_n : A_n$
Non-linear contexts	$\Theta ::= x_1 : A_1, \ldots, x_n : A_n$

Figure 1 presents our typing rules. We control duplication and discard by maintaining two separate type contexts: in the typing judgment $\Theta\ ;\ \Gamma \vdash e : A$ the context Θ contains those bindings which are used non-linearly (that is, those bindings which may be either duplicated or discarded), while the context Γ contains those bindings which are used exactly once. Variables bound in either kind of context must be pairwise distinct. We write an empty context as -.

The Var rule looks up the type of x in the non-linear context. The linear context is forced to be empty, since the term x does not consume any linear capabilities. The LVar rule looks up the type of a in the linear context (and ensures that there are no other linear bindings). No constraint is made on Θ, since we allow non-linear bindings to be freely discarded.

The unit value consumes no linear resources, so we force the linear part of the context to be empty in the $\star$-I rule. When we eliminate an expression of unit type using the $\star$-E rule, we check that the linear bindings we are expected to consume can be partitioned into two parts: one part which is

$$\text{Var}\ \frac{}{\Theta, x : A\ ;\ \text{-} \vdash x : A} \qquad \text{LVar}\ \frac{}{\Theta\ ;\ a : A \vdash a : A}$$

$$\star\text{-I}\ \frac{}{\Theta\ ;\ \text{-} \vdash \star : \star} \qquad \star\text{-E}\ \frac{\Theta\ ;\ \Gamma \vdash e : \star \qquad \Theta\ ;\ \Delta \vdash f : A}{\Theta\ ;\ \Gamma, \Delta \vdash \text{let}\ \star = e\ \text{in}\ f : A}$$

$$\otimes\text{-I}\ \frac{\Theta\ ;\ \Gamma \vdash e_1 : A_1 \qquad \Theta\ ;\ \Delta \vdash e_2 : A_2}{\Theta\ ;\ \Gamma, \Delta \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \qquad \otimes\text{-E}\ \frac{\Theta\ ;\ \Gamma \vdash e : A_1 \otimes A_2 \qquad \Theta\ ;\ \Delta, a_1 : A_1, a_2 : A_2 \vdash f : B}{\Theta\ ;\ \Gamma, \Delta \vdash \text{let}\ a_1 \otimes a_2 = e\ \text{in}\ f : B}$$

$$\multimap\text{-I}\ \frac{\Theta\ ;\ \Gamma, a : A \vdash e : B}{\Theta\ ;\ \Gamma \vdash \lambda a.e : A \multimap B} \qquad \multimap\text{-E}\ \frac{\Theta\ ;\ \Gamma \vdash e : A \multimap B \qquad \Theta\ ;\ \Delta \vdash f : A}{\Theta\ ;\ \Gamma, \Delta \vdash e\ f : B}$$

$$!\text{-I}\ \frac{\Theta\ ;\ \text{-} \vdash e : A}{\Theta\ ;\ \text{-} \vdash\ !e : !A} \qquad !\text{-E}\ \frac{\Theta\ ;\ \Gamma \vdash e : !A \qquad \Theta, x : A\ ;\ \Delta \vdash f : B}{\Theta\ ;\ \Gamma, \Delta \vdash \text{let}\ !x = e\ \text{in}\ f : B}$$

Figure 1: Intuitionistic linear logic

consumed by e, the other part by f. All non-linear bindings, however, are available to both e and f. The $\otimes$-I and $\otimes$-E rules partition the linear context into two parts in a similar way to the $\star$-E rule.

All function arguments are bound to linear variables, so the $\multimap$-I rule checks that the variable a is guaranteed to be used linearly in the body of the lambda-abstraction $\lambda a.e$. The $\multimap$-E rule behaves in a similar way to the $\otimes$-E rule, splitting any linear capabilities between the function and argument expressions.

Expressions of the form $!e$ may be freely duplicated or discarded. The !-I rule therefore ensures that the expression e does not refer to any linear bindings (otherwise discarding or duplicating $!e$ would have the effect of discarding or duplicating some linear bindings). The expression 'let $!x = e$ in f' evaluates e until it gets a !-closure. The !-E rule therefore places the binding for x in the non-linear part of the context, since we can be sure that it is safe to duplicate or discard x.

2.4 Type Soundness

Our typing rules and operational semantics are related by the following type-soundness theorem:

Theorem 2.4 (Type soundness)

If $\text{-}\ ;\ \text{-} \vdash e : A$ and $e \Downarrow v$ then $\text{-}\ ;\ \text{-} \vdash v : A$.

Proof A simple induction on the derivation of $e \Downarrow v$. □

3 Storage Model: !-Closures Recompute

The operational semantics which we gave in Definition 2.2 serves well as an abstract description of the behaviour of programs, but gives no insight into their storage-allocation properties. In this section we refine our semantics so that we store all values in an explicit heap. We then use our refined semantics to formalise the single-pointer property mentioned in the introduction.

Since we are interested in formalising both memory-management properties and lazy evaluation, it should not be surprising that our operational semantics is closely related to both Launchbury's natural semantics for lazy evaluation [Lau93] and Morrisett, Felleisen and Harper's abstract models of memory management [MFH95].

3.1 Heaps

Heaps associate non-linear variables with expressions (which can be thought of as suspended computations) and linear variables with stored values. Note in particular the format of tensor pair values, whose sub-terms are forced to be linear variables (i.e. addresses of other linear values). The value $!x$ represents an indirect reference from a linear address to a non-linear address. We ignore the order of entries in a heap.

Definition 3.1 (Heaps and values)

Heaps	H	$::=$	$B_1, \ldots, B_n$
Heap bindings	B	$::=$	$x = e \mid a = sv$
Stored values	sv	$::=$	$\star \mid \lambda a.e \mid a_1 \otimes a_2 \mid\ !x$

3.2 Evaluation Rules

Figure 2 presents our heap-based operational semantics. Evaluation judgments take the form $\{H\}e \Downarrow \{H'\}a$, meaning that, given an initial heap H, the expression e terminates, returning a new heap H' and result a (the address of a value stored in H').

The LVar rule evaluates linear variables: whenever we encounter a linear variable a there must already be a value stored in the heap at address a, so we can therefore simply return a without any further evaluation.

The Var rule is the crucial rule to note, since it recomputes non-linear values. Whenever we require the value of x, we look up the closure associated with x, evaluate it to some result a, and then return a as our own result. Subsequent uses of x will not benefit from the fact that we have already evaluated x, since the closure for x is not updated in any way.

The $\star$-I rule allocates a fresh unit value in the heap. The $\star$-E rule evaluates e to an address a, which must be the address of a unit value in the heap. We then continue to evaluate f, after deallocating the heap binding $a = \star$ (we shall prove later that it is safe to deallocate $a = \star$, since no other part of the program requires it). Note that it is not strictly necessary to heap-allocate unit values, but it is simpler to do so here since we wish to keep the convention that the result of evaluating an expression is always a linear variable (i.e. a pointer to a heap-allocated linear value).

$$\text{LVar}\ \frac{}{\{H\}a \Downarrow \{H\}a} \qquad \text{Var}\ \frac{\{H, x = e\}e \Downarrow \{H'\}a}{\{H, x = e\}x \Downarrow \{H'\}a}$$

$$\star\text{-I}\ \frac{a \text{ fresh}}{\{H\}\star \Downarrow \{H, a = \star\}a} \qquad \star\text{-E}\ \frac{\{H\}e \Downarrow \{H', a = \star\}a \qquad \{H'\}f \Downarrow \{H''\}b}{\{H\}\text{let}\star = e \text{ in } f \Downarrow \{H''\}b}$$

$$\otimes\text{-I}\ \frac{\{H\}e_1 \Downarrow \{H'\}a_1 \qquad \{H'\}e_2 \Downarrow \{H''\}a_2 \qquad b \text{ fresh}}{\{H\}e_1 \otimes e_2 \Downarrow \{H'', b = a_1 \otimes a_2\}b}$$

$$\otimes\text{-E}\ \frac{\{H\}e \Downarrow \{H', c = b_1 \otimes b_2\}c \qquad \{H'\}[b_1/a_1, b_2/a_2]f \Downarrow \{H''\}d}{\{H\}\text{let } a_1 \otimes a_2 = e \text{ in } f \Downarrow \{H''\}d}$$

$$\multimap\text{-I}\ \frac{b \text{ fresh}}{\{H\}\lambda a.e \Downarrow \{H, b = \lambda a.e\}b} \qquad \multimap\text{-E}\ \frac{\{H\}f \Downarrow \{H'\}c \qquad \{H'\}g \Downarrow \{H'', c = \lambda a.e\}b \qquad \{H''\}[b/a]e \Downarrow \{H'''\}d}{\{H\}f\ g \Downarrow \{H'''\}d}$$

$$!\text{-I}\ \frac{x, a \text{ fresh}}{\{H\}!e \Downarrow \{H, x = e, a = !x\}a} \qquad !\text{-E}\ \frac{\{H\}e \Downarrow \{H', a = !y\}a \qquad \{H'\}[y/x]f \Downarrow \{H''\}b}{\{H\}\text{let } !x = e \text{ in } f \Downarrow \{H''\}b}$$

Figure 2: Evaluation rules

The $\otimes$-I rule creates a $\otimes$-pair. We first evaluate the sub-terms e_1 and e_2, yielding two pointers a_1 and a_2. We then pick a fresh address b, store the pair $a_1 \otimes a_2$ at address b, and return b. The $\otimes$-E rules decomposes $\otimes$-pairs. We first evaluate e, yielding c, a pointer to a pair. We then extract the pointers b_1 and b_2 which are stored at c, substitute them for the bound variables a_1 and a_2, and continue executing f. Note that the binding $c = b_1 \otimes b_2$ is deallocated as a result of executing the $\otimes$-E rule.

The $\multimap$-I rule evaluates lambda abstractions: the term $\lambda a.e$ is already a value, so we simply store it at a fresh address b in the heap. The $\multimap$-E rule evaluates function application left-to-right: evaluating f yields a pointer to a function, evaluating g yields a pointer to the argument value which we then substitute for the bound variable a when executing the function body. Note that the binding $c = \lambda a.e$ is deallocated as a result of executing the $\multimap$-E rule.

The !-I rule interprets replication as a closure-building operation. Therefore, since the expression $!e$ is already fully evaluated, we simply store it at a fresh non-linear address in the heap. Rather than returning the address x directly, we create an indirection node to x at a fresh linear address a and return that as our result. This maintains the invariant that the result of evaluating a term is always a linear variable. The !-E rule evaluates the let-bound expression e to a, an indirection node which points to a non-linear address y. We substitute y for x in f, enabling f to make unlimited use of y. Note that the indirection node $a = !y$ is deallocated as a result of evaluating the !-E rule.

3.3 Reduction Rules

We wish to ensure that our memory-management invariants are preserved throughout execution. We therefore transform our operational semantics into an equivalent set of reduction rules (Definition 3.3) so that we can observe the intermediate states of a computation. (We simplify the reduction rules by introducing a new syntactic category, E, of evaluation contexts, which describes those contexts in which we may evaluate sub-terms.)

It is easy to check that our reduction rules are an adequate model of our operational semantics, since whenever our operational semantics can evaluate a term, our reduction rules can simulate that evaluation:

Proposition 3.2 (Adequacy)

If $\{H\}e \Downarrow \{H'\}a$ then $\{H\}e \to^* \{H'\}a$.

Proof A simple induction on the depth of the derivation of $\{H\}e \Downarrow \{H'\}a$. □

Definition 3.3 (Reduction rules)

$$E \quad ::= \quad [\,] \otimes e \mid a \otimes [\,] \mid [\,]\ e \mid a\ [\,] \mid \text{let } a_1 \otimes a_2 = [\,] \text{ in } e \mid \text{let } !x = [\,] \text{ in } e \mid \text{let} \star = [\,] \text{ in } e$$

$$\text{Context}\ \frac{\{H\}e \to \{H'\}e'}{\{H\}E[e] \to \{H'\}E[e']}$$

Var	$\{H, x = e\}x \to \{H, x = e\}e$	
$\star$-I	$\{H\}\star \to \{H, a = \star\}a$	a fresh
$\star$-E	$\{H, a = \star\}\text{let}\star = a \text{ in } e \to \{H\}e$	
$\multimap$-I	$\{H\}\lambda a.e \to \{H, b = \lambda a.e\}b$	b fresh
$\multimap$-E	$\{H, c = \lambda a.e\}c\ b \to \{H\}[b/a]e$	
!-I	$\{H\}!e \to \{H, x = e, a = !x\}a$	x, a fresh
!-E	$\{H, a = !y\}\text{let } !x = a \text{ in } e \to \{H\}[y/x]e$	
$\otimes$-I	$\{H\}a_1 \otimes a_2 \to \{H, b = a_1 \otimes a_2\}b$	b fresh
$\otimes$-E	$\{H, c = b_1 \otimes b_2\}\text{let } a_1 \otimes a_2 = c \text{ in } e \to \{H\}[b_1/a_1, b_2/a_2]e$	

Before proving the soundness of our reduction rules, we first prove that executing a single reduction step never affects the behaviour of a term under our original semantics:

Lemma 3.4 (Single-step reduction)

If $\{H\}e \to \{H'\}e'$ and $\{H'\}e' \Downarrow \{H''\}a$ then $\{H\}e \Downarrow \{H''\}a$.

Proof A simple induction on the depth of the inference of $\{H\}e \to \{H'\}e'$. □

Proposition 3.5 (Soundness)
If $\{H\}e \to^* \{H'\}a$ then $\{H\}e \Downarrow \{H'\}a$.

Proof We prove the result by induction on the length of the reduction sequence $\{H\}e \to^* \{H'\}a$. In the case where we make no reduction steps, $H' = H$ and $a = e$ and the LVar rule lets us infer $\{H\}a \Downarrow \{H\}a$ as required. Otherwise, it must be the case that $\{H\}e \to \{H'\}e' \to^* \{H''\}a$. We can use induction to prove that $\{H'\}e' \Downarrow \{H''\}a$ and the result follows using the Lemma 3.4. □

3.4 The Single-Pointer Property

We are now in a position to formalise the single-pointer property and prove that it is preserved during evaluation. The following rules generalise the typing rules for terms (from Figure 1) so that they can be applied to heaps. The judgment $\vdash H : \Theta \; ; \; \Gamma$ states that the heap H provides the non-linear bindings Θ and linear bindings Γ.

Definition 3.6 (Heap typing rules)

$$\text{Empty}\ \frac{}{\vdash \text{-} \ : \ \text{-} \,;\, \text{-}}$$

$$\text{Closure}\ \frac{\vdash H : \Theta \,;\, \Gamma \qquad \Theta \,;\, \text{-} \vdash e : A}{\vdash (H, x = e) \ : \ (\Theta, x : A) \,;\, \Gamma}$$

$$\text{Value}\ \frac{\vdash H : \Theta \,;\, (\Gamma, \Delta) \qquad \Theta \,;\, \Delta \vdash v : A}{\vdash (H, a = v) \ : \ \Theta \,;\, (\Gamma, a : A)}$$

The Closure rule first determines Θ and Γ, the non-linear and linear bindings provided by H, and then checks that the closure e is well-typed in the non-linear environment Θ and empty linear environment. This ensures that e can only refer to non-linear variables which are provided by H (this rule is very similar to the !-I typing rule for terms). We add the type for x to Θ.

The Value rule checks that the heap H provides non-linear bindings Θ and linear bindings (Γ, Δ). We partition the linear bindings into two parts since not all of H's linear capabilities will be consumed by v. The overall linear capabilities provided by H are therefore $(\Gamma, a : A)$, the remainder of H's linear bindings plus the binding for a.

3.5 Type Soundness

The following two lemmas prove that we can rearrange a heap typing derivation so that the derivation of the types of the bindings $a = v$ and $x = e$ are proved last. These is required in the proof of Lemma 3.9.

Lemma 3.7 (Heap Typing: Linear)
If $\vdash (H, a = v) : \Theta \,;\, (\Gamma, a : A)$ then there exists a Δ such that $\vdash H : \Theta \,;\, (\Gamma, \Delta)$ and $\Theta \,;\, \Delta \vdash v : A$.

Proof Uses induction on the depth of the inference of $\vdash (H, a = v) : \Theta \,;\, (\Gamma, a : A)$. □

Lemma 3.8 (Heap Typing: Non Linear)
If $\vdash (H, x = e) : \Theta \,;\, \Gamma$ then $\Theta \,;\, \text{-} \vdash e : A$.

Proof Uses induction on the depth of the inference of $\vdash (H, a = v) : \Theta \,;\, (\Gamma, a : A)$. □

The following lemma states that if H provides a superset of the linear capabilities required by e, then whenever the configuration $\{H\}e$ reduces to $\{H'\}e'$, the heap H' continues to provide enough capabilities for e' (and leaves the additional capabilities in Δ untouched).

Lemma 3.9 (Invariance)
If $\vdash H : \Theta \,;\, (\Gamma, \Delta)$ and $\Theta \,;\, \Gamma \vdash e : A$ then whenever $\{H\}e \to \{H'\}e'$ there exist Θ' and Γ' such that $\vdash H' : (\Theta, \Theta') \,;\, (\Gamma', \Delta)$ and $(\Theta, \Theta') \,;\, \Gamma' \vdash e' : A$.

Proof Uses induction on the depth of the inference of $\{H\}e \to \{H'\}e'$. Uses Lemmas 3.7 and 3.8. In the base case we use a case analysis on the basic reduction rules. □

Note that the above lemma also makes it clear that no non-linear bindings are ever removed from the heap, unlike linear bindings which are consumed during evaluation.

Theorem 3.10 (Type soundness)
If $\vdash H : \Theta \,;\, \Gamma$ and $\Theta \,;\, \Gamma \vdash e : A$ then whenever $\{H\}e \to \{H'\}e'$ there exist Θ' and Γ' such that $\vdash H' : \Theta' \,;\, \Gamma'$ and $\Theta' \,;\, \Gamma' \vdash e' : A$.

Proof Type soundness is an immediate consequence of Lemma 3.9. □

Our type soundness theorem also proves that all linear data will be correctly deallocated during evaluation (i.e. there are no dangling pointers or space leaks) since after any reduction step $\{H\}e \to \{H'\}e'$, the linear heap bindings provided by H' are exactly those which are required by e'. This means, for example, that we can use a garbage collection scheme to manage the storage allocated to non-linear values (which in our model is never explicitly deallocated), but rely on the program to explicitly deallocate all linear values.

4 Storage Model: !-Closures Update

Figure 3 describes the changes we must make to our operational semantics so that !-closures are updated after they are evaluated.

The key change is in the Var rule where, instead of just evaluating e, we update x with whatever result e returns. Subsequent uses of x will therefore return the result a without recomputing e. Unfortunately, changing the Var rule is by no means the only modification we must make. The observant reader should already be suspicious, since the linear variable a is clearly duplicated when we update x. In fact, this is precisely the reason why this interpretation of linear logic fails to have the single-pointer property: the result of evaluating the closure e is a linear value (pointed to by a), but we have used that value for two separate purposes (updating the binding for x and returning as our result).

Recall that in our original evaluation rules (Figure 2) whenever we consume a linear value we explicitly remove it from the heap. In a calculus which fails to have the single-pointer property, this is no longer safe, since we cannot guarantee that there are no remaining references to the linear value we are consuming. Our modification to the Var rule therefore forces us to also modify all those evaluation rules which deconstruct a value of linear type, since they can no longer explicitly remove the heap binding for the value they just consumed.

$$\text{Var}\ \frac{\{H\}e \Downarrow \{H'\}a}{\{H, x = e\}x \Downarrow \{H', x = a\}a}$$

$$\otimes\text{-E}\ \frac{\{H\}e \Downarrow \{H', c = b_1 \otimes b_2\}c \qquad \{H', c = b_1 \otimes b_2\}[b_1/a_1, b_2/a_2]f \Downarrow \{H''\}d}{\{H\}\mathrm{let}\ a_1 \otimes a_2 = e\ \mathrm{in}\ f \Downarrow \{H''\}d}$$

$$\multimap\text{-E}\ \frac{\{H\}f \Downarrow \{H'\}c \qquad \{H'\}g \Downarrow \{H'', c = \lambda a.e\}b \qquad \{H''\}[b/a]e \Downarrow \{H'''\}d}{\{H\}f\ g \Downarrow \{H'''\}d}$$

$$!\text{-E}\ \frac{\{H\}e \Downarrow \{H', a = !y\}a \qquad \{H', a = !y\}[y/x]f \Downarrow \{H''\}b}{\{H\}\mathrm{let}\ !x = e\ \mathrm{in}\ f \Downarrow \{H''\}b}$$

Figure 3: !-closures update

4.1 Reduction Rules

Just as before, we transform our operational semantics into an equivalent set of reduction rules, so that we can observe the intermediate states of a computation. (The syntax of evaluation contexts remains unchanged from Definition 3.3.)

Definition 4.1 (Reduction rules)

$$\text{Context}\ \frac{\{H\}e \to \{H'\}e'}{\{H\}E[e] \to \{H'\}E[e']}$$

$$\text{Binding}\ \frac{\{H\}e \to \{H'\}e'}{\{H, x = e\}x \to \{H', x = e'\}x}$$

$$\text{Lookup}\ \frac{}{\{H, x = a\}x \to \{H', x = a\}a}$$

$\multimap$-I	$\{H\}\lambda a.e \to \{H, b = \lambda a.e\}b$	b fresh
$\multimap$-E	$\{H, c = \lambda a.e\}c\ b \to \{H, c = \lambda a.e\}[b/a]e$	
!-I	$\{H\}!e \to \{H, x = e, a = !x\}a$	x, a fresh
!-E	$\{H, a = !y\}\mathrm{let}\ !x = a\ \mathrm{in}\ e \to$ $\{H, a = !y\}[y/x]e$	
$\star$-I	$\{H\}\star \to \{H, a = \star\}a$	x, a fresh
$\star$-E	$\{H, a = \star\}\mathrm{let}\ \star = a\ \mathrm{in}\ e \to$ $\{H, a = \star\}e$	
$\otimes$-I	$\{H\}a_1 \otimes a_2 \to \{H, b = a_1 \otimes a_2\}b$	b fresh
$\otimes$-E	$\{H, c = b_1 \otimes b_2\}\mathrm{let}\ a_1 \otimes a_2 = c\ \mathrm{in}\ e \to$ $\{H, c = b_1 \otimes b_2\}[b_1/a_1, b_2/a_2]e$	

4.2 Single-Pointer Property

It is easy to check that our new interpretation of !-closures fails to preserve the single-pointer property. However, all is not lost since we can identify a subset of the values of linear type which are pointed to exactly once [CGR92, CGR95].

To capture the fact that not all linear values are pointed to exactly one, we must refine our heap and expression typing rules. The judgement $\Theta\ ;\ \Phi\ ;\ \Gamma \vdash e : A$ indicates that e has type A, where Θ contains type bindings for all the non-linear variables (as before), Φ contains type bindings for all the linear variables that are used 'badly' (i.e. more than once, not at all, or inside a !-closure), while Γ contains type bindings for all linear variables that are used exactly one (as before). (The domains of the type contexts Φ and Γ must always be distinct.)

The heap typing rules are weakened in a similar way to the expression typing rules, except that they contain an extra component Δ which records those variables which are not used at all. Note in particular the Bad rule, which forces the free linear variables of any bad heap binding to be marked bad.

Definition 4.2 (Heap typing rules)

$$\text{Empty}\ \frac{}{\vdash \text{-}\ :\ \text{-}\ ;\ \text{-}\ ;\ \text{-}\ ;\ \text{-}}$$

$$\text{Closure}\ \frac{\vdash H : \Theta\ ;\ \Phi\ ;\ \Gamma\ ;\ \Delta \qquad \Theta\ ;\ \Phi\ ;\ \text{-} \vdash e : A}{\vdash (H, x = e)\ :\ (\Theta, x : A)\ ;\ \Phi\ ;\ \Gamma\ ;\ \Delta}$$

$$\text{Unused}\ \frac{\vdash H : \Theta\ ;\ \Phi\ ;\ \Gamma\ ;\ \Delta}{\vdash (H, a = v)\ :\ \Theta\ ;\ \Phi\ ;\ \Gamma\ ;\ (\Delta, a : A)}$$

$$\text{Linear}\ \frac{\vdash H : \Theta\ ;\ \Phi\ ;\ (\Gamma_1, \Gamma_2)\ ;\ \Delta \qquad \Theta\ ;\ \Phi\ ;\ \Gamma_2 \vdash v : A}{\vdash (H, a = v)\ :\ \Theta\ ;\ \Phi\ ;\ (\Gamma_1, a : A)\ ;\ \Delta}$$

$$\text{Bad}\ \frac{\vdash H : \Theta\ ;\ \Phi\ ;\ \Gamma\ ;\ \Delta \qquad \Theta\ ;\ \Phi\ ;\ \text{-} \vdash v : A}{\vdash (H, a = v)\ :\ \Theta\ ;\ (\Phi, a : A)\ ;\ \Gamma\ ;\ \Delta}$$

We need the extra component Δ in our heap typing rules, since we need to explicitly detect when a linear heap binding becomes unused (recall that we no longer explicitly deallocate linear bindings, since we cannot guarantee that they are always pointed to exactly once). We could mark unused linear heap bindings 'bad', since 'bad' bindings are allowed to be discarded. However, we would then not be able to distinguish the case where a linear binding is consumed (which we are happy to allow) from the case where a linear binding changes into a 'bad' heap binding (which we certainly don't want to allow).

It is simple to check that the strong single-pointer property from Section 3.4 implies the above weaker property:

Proposition 4.3 (Strong and weak properties)

1. If $\Theta\ ;\ \Gamma \vdash e : A$ then $\Theta\ ;\ \text{-}\ ;\ \Gamma \vdash e : A$.

2. If $\vdash H : \Theta\ ;\ \Gamma$ then $\vdash H : \Theta\ ;\ \text{-}\ ;\ \Gamma\ ;\ \text{-}$.

Proof In both cases, a simple induction on the depth of the typing derivation. □

$$\text{Var}\ \frac{}{\Theta, x : A\,;\,\Phi\,;\,\text{-} \vdash x : A} \qquad \text{LVar}\ \frac{}{\Theta\,;\,\Phi\,;\,a : A \vdash a : A} \qquad \text{BadVar}\ \frac{}{\Theta\,;\,\Phi, a : A\,;\,\text{-} \vdash a : A}$$

$$\star\text{-I}\ \frac{}{\Theta\,;\,\Phi\,;\,\text{-} \vdash \star : \star} \qquad \star\text{-E}\ \frac{\Theta\,;\,\Phi\,;\,\Gamma_1 \vdash e : \star \qquad \Theta\,;\,\Phi\,;\,\Gamma_2 \vdash f : A}{\Theta\,;\,\Phi\,;\,\Gamma_1, \Gamma_2 \vdash \text{let}\,\star = e\,\text{in}\,f : A}$$

$$\otimes\text{-I}\ \frac{\Theta\,;\,\Phi\,;\,\Gamma_1 \vdash e_1 : A_1 \qquad \Theta\,;\,\Phi\,;\,\Gamma_2 \vdash e_2 : A_2}{\Theta\,;\,\Phi\,;\,\Gamma_1, \Gamma_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \qquad \otimes\text{-E}\ \frac{\Theta\,;\,\Phi\,;\,\Gamma_1 \vdash e : A_1 \otimes A_2 \qquad \Theta\,;\,\Phi\,;\,\Gamma_2, a_1 : A_1, a_2 : A_2 \vdash f : B}{\Theta\,;\,\Phi\,;\,\Gamma_1, \Gamma_2 \vdash \text{let}\,a_1 \otimes a_2 = e\,\text{in}\,f : B}$$

$$\multimap\text{-I}\ \frac{\Theta\,;\,\Phi\,;\,\Gamma, a : A \vdash e : B}{\Theta\,;\,\Phi\,;\,\Gamma \vdash \lambda a.e : A \multimap B} \qquad \multimap\text{-E}\ \frac{\Theta\,;\,\Phi\,;\,\Gamma_1 \vdash e : A \multimap B \qquad \Theta\,;\,\Phi\,;\,\Gamma_2 \vdash f : A}{\Theta\,;\,\Phi\,;\,\Gamma_1, \Gamma_2 \vdash e\,f : B}$$

$$!\text{-I}\ \frac{\Theta\,;\,\Phi\,;\,\text{-} \vdash e : A}{\Theta\,;\,\Phi\,;\,\text{-} \vdash\, !e : !A} \qquad !\text{-E}\ \frac{\Theta\,;\,\Phi\,;\,\Gamma_1 \vdash e : !A \qquad \Theta, x : A\,;\,\Phi\,;\,\Gamma_2 \vdash f : B}{\Theta\,;\,\Phi\,;\,\Gamma_1, \Gamma_2 \vdash \text{let}\,!x = e\,\text{in}\,f : B}$$

Figure 4: Weakened linear logic

4.3 Type Soundness

Using our refined typing rules we can prove Chirimar, Gunter and Riecke's weakened single-pointer property: if a linear heap binding is pointed to exactly once (and as long as it is not pointed to from within a !-closure), it will continue to have that property throughout execution of the program (until the value is consumed, after which we are guaranteed that there will be no remaining references to the value).

The above property is clearly much weaker than the single-pointer property we gave earlier. In particular, it requires information from outside the type system to determine whether a heap binding is initially pointed to exactly once. However, this weak single-pointer property is sufficient to enable update-in-place optimisations on arrays (since it is possible to design array primitives in such a way that arrays always start life with exactly one pointer to them: the linear type system ensures that references to the array will never subsequently be duplicated, and so it is safe to update an array in place).

Lemma 4.4 proves that the linear bindings Γ_1 required by e before the reduction step can be split into two parts: Γ_1', the bindings required after the reduction step; and Δ', the bindings consumed during the reduction step. Bindings which start out linear never need to subsequently be marked bad. Γ_2 is the set of linear bindings used elsewhere in the (enclosing) program, and is left untouched. Γ_3 is the set of new linear bindings created during the reduction.

Lemma 4.4 (Invariance)

If $\vdash H : \Theta\,;\,\Phi\,;\,(\Gamma_1, \Gamma_2)\,;\,\Delta$ and $\Theta\,;\,\Phi\,;\,\Gamma_1 \vdash e : A$ and $\{H\}e \to \{H'\}e'$ then there exist Θ', Φ', Γ_1', Γ_3 and Δ' such that $\vdash H : (\Theta, \Theta')\,;\,(\Phi, \Phi')\,;\,(\Gamma_1', \Gamma_2, \Gamma_3)\,;\,(\Delta, \Delta')$ and $(\Theta, \Theta')\,;\,(\Phi, \Phi')\,;\,\Gamma_1', \Gamma_3 \vdash e : A$ where $\Gamma_1', \Delta' = \Gamma_1$.

Proof By induction on the depth of the inference of $\{H\}e \to \{H'\}e'$. □

Theorem 4.5 (Type soundness)

If $\vdash H : \Theta\,;\,\Phi\,;\,\Gamma\,;\,\Delta$ and $\Theta\,;\,\Phi\,;\,\Gamma \vdash e : A$ and $\{H\}e \to \{H'\}e'$ then there exist Θ', Φ', Γ_1, Γ_2 and Δ' such that $\vdash H' : (\Theta, \Theta')\,;\,(\Phi, \Phi')\,;\,\Gamma_1, \Gamma_2\,;\,(\Delta, \Delta')$ and $(\Theta, \Theta')\,;\,(\Phi, \Phi')\,;\,\Gamma_1, \Gamma_2 \vdash e' : A$ where $\Gamma_1, \Delta' = \Gamma$.

Proof A corollary of Lemma 4.4. □

4.4 Encoding Call-by-value Lambda-Calculus

One might think that updating !-closures is necessarily more efficient that recomputing !-closures, since more computation is shared. However, there are some applications where there is no benefit to be gained from updating !-closures. In particular, if we consider the encoding of call-by-value lambda-calculus into linear logic, we find that it is reasonable to recompute !-closures (especially since recomputing !-closures is simpler to implement and guarantees a stronger single-pointer property).

We first recall the encoding of call-by-value lambda-calculus into intuitionistic linear logic:

Definition 4.6 (Encoding call-by-value λ-calculus)

$$\begin{array}{rcl} x^* & = & !x \\ (\lambda x.e)^* & = & !(\lambda a.\text{let}\,!x = a\,\text{in}\,e^*) \\ (e_1\,e_2)^* & = & \text{let}\,!f = e_1^*\,\text{in}\,f\,e_2^* \\ (e_1, e_2)^* & = & \text{let}\,!x_1 = e_1^*\,\text{in let}\,!x_2 = e_2^*\,\text{in}\,!(!x_1 \otimes !x_2) \end{array}$$

$$\begin{array}{l} (\text{let}\,(x_1, x_2) = e_1\,\text{in}\,e_2)^* = \\ \quad \text{let}\,!x = e_1^*\,\text{in let}\,a_1 \otimes a_2 = x\,\text{in} \\ \qquad \text{let}\,!x_1 = a_1\,\text{in let}\,!x_2 = a_2\,\text{in}\,e_2^* \end{array}$$

Note that the encoding only builds !-closures around three kinds of expression: x, $\lambda a.(\text{let}!x = a\text{in}e^*)$ and $!x_1 \otimes !x_2$. The latter two expression forms encode non-linear function and pair values, and are already fully evaluated. In a closed program, the non-linear variable x can only become bound to the contents of a !-closure, and since !-closures only contain function or pair values this implies that x must also be either a function or a pair value. A let-expression of the form 'let$!x = e_1\,\text{in}\,e_2$' therefore has the effect of forcing e_1 to evaluate to a value (since the only !-closures are encodings of values). For example, the encoding of the pair (e_1, e_2) forces the encodings of the sub-expressions e_1 and e_2 to be evaluated to values before building the pair.

Thus, in the subset of linear logic which includes just the image of the call-by-value encoding of lambda-calculus there is no significant benefit to be gained from using an interpretation of linear logic which updates !-closures, since every !-closure is already fully evaluated.

5 Related Work

Chirimar, Gunter and Riecke [CGR92, CGR95] provided the inspiration for this paper with their reference-counting model of linear logic (we have already described in the introduction how our work extends theirs). They prove that, even though their evaluation strategy does not guarantee the single-pointer property for all values of linear type, values of linear type which already have the single-pointer property will continue to do so throughout evaluation. With careful design, this enables one to, for example, write array primitives which do in-place update of array entries. However, we believe that the lack of a strong single-pointer property complicates both the reference counting interpretation of linear logic and the proofs of memory management invariants. For instance, a compiler cannot make optimisations solely on the basis of linear type information, since extra information is required to ensure that a value has the single-pointer property.

Lincoln and Mitchell [LM92] describe an implementation of a linear language which allocates non-linear values in a garbage collected heap and linear values in a separate area which is never garbage collected. However, they do not specify exactly how they evaluate non-linear values, so it is unclear whether they can avoid garbage collection for all linear values. We give a formal proof that, in the case where our operational interpretation of linear logic satisfies the strong single-pointer property, it is possible to allocate linear values in a separate area which need never be garbage collected.

Lafont [Laf88] and Abramsky [Abr92] both describe abstract machines for intuitionistic linear logic. Both their abstract machine recompute non-linear values, so they should preserve the strong single-pointer property. However, their abstract machines are formulated slightly too abstractly to capture memory management properties.

Wakeling and Runciman [WR91] briefly mention a variant of the G-machine [Aug87, Joh87] which they use to implement a linear language based on the work of Wadler [Wad90, Wad91]. They do not prove any memory-management properties, but their abstract machine would have been ideal for this purpose, since it represents the linear and non-linear storage graphs explicitly. Even though Wakeling and Runciman implement a call-by-need language, which updates closures, their language preserves the strong single-pointer property, since they include non-linear values as primitives, rather than using !-closures to define non-linear values in terms of linear values.

Baker [Bak92, Bak94] has developed a simple untyped language where there is exactly one pointer to each storage cell, and demonstrated that such a language can have substantially improved performance over more traditional languages for small algorithms such as quicksort.

Barendsen and Smetsers [BS93] have developed a complex type system and type reconstruction algorithm for so-called *uniqueness types* which possess exactly one pointer to them, and this type system has been applied to structure IO operations in the lazy functional language Clean by Achten and Plasmeijer [AP95].

Benton's mixed linear and non-linear logic [Ben95], unlike linear logic, incorporates primitive linear and non-linear term constructs, rather than interpreting non-linear values in terms of !-closures. This means that the issue of how to interpret replicated linear values becomes less critical (since one can just use the primitive non-linear term constructs). However, if it turns out that a significant number of values need to move across the interface between the classical and linear parts of the calculus, then all the problems we have described here arise again. We conjecture that, just in the case of linear logic, the single-pointer property for Benton's calculus depends crucially on how much recomputation one is willing to do. Benton and Wadler [BW96] use Benton's model to relate three standard mappings from lambda calculus (direct, call-by-name, and call-by-value) into Benton's linear calculus and Moggi's monadic metalanguage.

Morrisett, Felleisen, and Harper [MFH95] present models of memory management which allow them to formulate and prove properties of garbage collection models at a high level of abstraction. Although formulated independently from our work, the models are strikingly similar, offering the hope that their techniques could be applied in our framework.

6 Conclusions and Further Work

Our work highlights the tradeoff between an interpretation of linear logic which recomputes !-closures and an interpretation which updates !-closures. In the case where we update !-closures, we believe that the lack of a strong single-pointer property poses a significant problem, since the linear type information no longer agree with what happens during execution. This makes it much more difficult to base optimisations on linear type information, since the fact that a value has a linear type no longer means that it is guaranteed to be used exactly once.

It remains to be seen whether an implementation of linear logic can make sufficiently good use of linear type information to make it worth treating linear values specially. For instance, even though linear values can be explicitly deallocated during execution of a linear program, there are other overheads which must be taken into account. The cost of allocating storage in a garbage-collected language is usually very low, so we had better make sure that we can allocate linear values efficiently (this would seem to rule out techniques based on free-lists). We must also ensure that deallocation of linear values is extremely cheap, since deallocation has no cost in a garbage-collected storage management scheme. For this reason, linear type information is often only used to implement update-in-place, rather than to allow explicit deallocation. Unfortunately, it is hard to predict how often opportunities to use update-in-place will arise in real programs (since the storage being updated must usually be deallocated 'close' to where the new piece of memory is required).

An interesting area of further work would be to investigate whether linear type information allows more efficient garbage collection algorithms to be used. For example, copying garbage collectors have to take care not to copy the same object twice (this problem is usually handled by overwriting every copied object with a forwarding pointer, which points to where the new copy has been allocated). This work is unnecessary if a value has the single-pointer property since, by definition, there are no other pointers to the value and the garbage collector will therefore never visit it again. We might therefore be able to use linear type information to help in garbage collection.

Update added in revision (June 1998): Keith Wansbrough of Glasgow University, in collaboration with Simon Peyton and David Turner, is currently experimenting to determine how linear logic can be applied to improve the implementation of the lazy functional language Haskell. While our theories look useful, the tradeoffs are unclear, and experiments such as this are essential to determine if they have value in practice.

References

[Abr92] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 1992.

[AP95] Peter Achten and Rinus Plasmeijer. The ins and outs of clean i/o. *Journal of Functional Programming*, 5(1):81–110, January 1995.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languagess, Part II.* PhD thesis, Chalmers University of Technology, 1987.

[Bak92] Henry G. Baker. Lively linear Lisp–'Look Ma, no garbage!'. *ACM Sigplan Notices*, 27(8):89–98, August 1992.

[Bak94] Henry G. Baker. A 'linear logic' quicksort. *ACM Sigplan Notices*, 29(2):13–18, February 1994.

[BBdPH93] Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. A term calculus for intuitionstic linear logic. In *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[Ben95] Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Proceeding of Computer Science Logic '94*, volume 933 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[BS93] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on the Foundations of Software Techology and Theoretical Computer Science*, number 761 in LNCS. Springer-Verlag, 1993. (Technical report version).

[BW96] Nick Benton and Philip Wadler. Linear logic, monads, and the lambda calculus. In *Symposium on Logic in Computer Science*. IEEE Computer Society, 1996.

[CGR92] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming*. ACM Press, April 1992.

[CGR95] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1995.

[Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, 1987.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59, 1988.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages*, 1993.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *IEEE Symposium on Logic in Computer Science.* IEEE Computer Society, 1992.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Functional Programming Languages and Computer Architecture*, 1995.

[MOTW95] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda-calculus. In *Eleventh Conference on the Mathematical Foundations of Programming Semantics*, number 1 in Electronic Notes in Theoretical Computer Science. Elsevier, 1995.

[Plo93] Gordon Plotkin. Type theory and recursion (abstract). In *Logic in Computer Science.* IEEE Press, 1993.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990.

[Wad91] Philip Wadler. Is there a use for linear logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation*, June 1991.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematic Foundations of Computing Science*, number 711 in LNCS. Springer-Verlag, August 1993.

[WR91] David Wakeling and Colin Runciman. Linearity and laziness. In *Functional Programming Languages and Computer Architecture*, 1991.