Proving Memory Management Invariants for a Language Based on Linear Logic

Jawahar Chirimar* Carl A. Gunter* Jon G. Riecke*
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

April 9, 1992

Abstract: We develop tools for the rigorous formulation and proof of properties of runtime memory management for a sample programming language based on a linear type system. Two semantics are described, one at a level of observable results of computations and one describing linear connectives in terms of memory-management primitives. The two semantics are proven equivalent and the memory-management model is proven to satisfy fundamental correctness criteria for reference counts.

1 Introduction

Although much literature on optimization of programs discusses the interaction of program execution and memory management, one does not often find formal statements and proofs regarding this interaction. For instance, a compiler for a functional language may generate code that updates an array in-place instead of copying the entire array, even though this optimization is not always safe. The compiler writer probably does not *prove* the optimization is safe: the size and complexity of real compilers is one serious impediment, but another problem is the lack of an appropriate level of abstraction at which to carry out the proofs. Most practical interpreters and compilers are written for specific machines and hence are too low-level and specialized to serve as tractable models, whereas the usual abstract operational semantics (*e.g.*, natural semantics or structural operational semantics [21, 24]) do not provide any information about memory management.

In this paper we investigate the interaction of memory management and interpretation, defining two operational semantics of a functional language: one at a high level ordinarily used for specifying the formal operational semantics of a language, and one at a lower level that explicitly manages memory. We give proof techniques for showing that the two interpreters yield equivalent results on programs, and for showing that the desired invariants (*e.g.*, correctness of the reference counts) hold for the low-level specification.

Our main interest in developing such lower-level operational semantics is to test theses about type systems and memory management. Many new type systems based on *linear logic* [15, 16] have been proposed that hypothesize simpler garbage collection or in-place updating. (Others have proposed related type systems [10, 26].) The particular language we study is essentially the linear logic-based language of [1, 2], enhanced with the operations of PCF [9, 23]. Types are formulas in a fragment of linear logic, and terms encode proofs in the sequent calculus for this fragment (*cf.* [8, 12]).

We define two operational semantics for this language. The first is a high-level natural semantics resembling that of Abramsky [1, 2], which delays evaluation for terms of !-type. This semantics describes the expected observational semantics of type-correct programs, but it does not explain the computational significance of some of the linear connectives: the weakening rule of linear logic, which discards one of its two arguments, is essentially interpreted as a "no-op". The second operational semantics, one that mirrors current implementation technology, interprets linear connectives as memory-management primitives which carry

*Partially supported by an ONR Young Investigator Award number N00014–88–K–0557, NSF grant number CCR–8912778, NRL grant number N00014–91–J–2022, and NOSC grant number 19–920123–31.

out pointer manipulations and reference-counting. For instance, the weakening rule is interpreted as a *dispose* command which deallocates memory (or, more precisely, decrements reference counts).

There are three principal technical contributions of the paper. The first shows that our reference-counting interpreter implements the more abstract semantics. The second result is a proof that the reference-counting interpreter maintains a correct reference count for memory cells. The third and final result uses the model to clarify some hypotheses regarding memory management in a language with a linear logic-based type system. In particular, we find that simplification of garbage collection, along the lines suggested in [1, 15, 16], is *not* possible.

The paper is organized as follows. In Section 2 we abstractly describe the reference-counting operational semantics for λ-calculus-based functional languages. The memory model is familiar, using closures to implement λ-abstractions with environments that point to locations in memory; this implementation strategy permits some obvious optimizations through the sharing of data structures. Section 3 defines the syntax and type-checking rules of the linear logic-based language. In Section 4 we prove the correctness of the reference-counting operational semantics, and in Section 5 discuss the role of the linear type system with respect to memory management. Finally, Sections 6 and 7 conclude with a discussion of related work and possible extensions.

2 A Reference-Counting Operational Model

We first describe the low-level model in some generality, leaving vague the functional language to be interpreted.

Memory in the model is structured using environments and closures in much the same way as the SECD machine [17, 22]. Fix an infinite set of locations Loc, with the letter l denoting elements of this set. Then

- An **environment** is a finite function from variables to locations; ρ denotes an environment, and Env denotes the set of all environments. The notation $\rho(x)$ returns the location associated with variable x in ρ, and

$$(\rho[x \mapsto l])(y) = \begin{cases} l & \text{if } x = y \\ \rho(y) & \text{otherwise} \end{cases}$$

  The symbol $\emptyset$ denotes the empty environment.

- A **value** is either a numeral k; a boolean b; a closure tuple $\langle x, M, \rho \rangle$, where ρ is an environment; or a suspended object $\mathsf{susp}(l)$—a boxed value—where l is a location. The letter V denotes a value, and Value denotes the set of values.

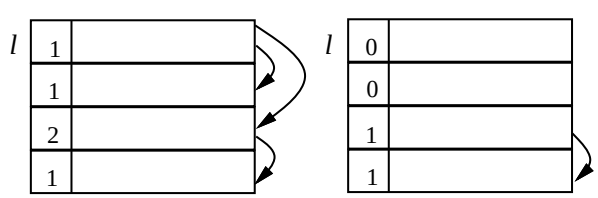
- A **storable object** is either a value or has the form $\mathsf{thunk}(M, \rho)$. We use S to denote a storable object, and $\mathsf{Storable}$ to denote the set of storable values.

- A **store** is a function $\sigma : \mathsf{Loc} \to (\mathbb{N} \times \mathsf{Storable})$; the left part of the returned pair denotes a reference count. Moreover, a store must contain no loops of references—*e.g.*, a function which mapped location l_1 to $\langle x, P, \rho \rangle$ where l_2 is in the range of ρ, and mapped location l_2 to $\mathsf{susp}(l_1)$, would be illegal. Finally, the **domain** of a store σ must be finite, *viz.*, the set

$$\mathsf{dom}(\sigma) = \{l \in \mathsf{Loc} : \mathsf{refcount}(l, \sigma) \geq 1\}$$

  is finite. Our operational rules will always maintain these invariants. The symbol σ denotes a store, $\emptyset$ denotes the empty store, and Store denotes the set of stores. Abusing notation, we use $\sigma(l)$ to denote the storable object associated with location l, and $\mathsf{refcount}(l, \sigma)$ to denote the reference count stored at l.

A few operations are needed for manipulating reference counts, environments, and stores. The operation $\mathsf{update}(l, S, \sigma)$ updates the store σ and binds l to the storable object S but leaves its reference count unchanged; $\mathsf{incr}(l, \sigma)$ increments the reference count of location l in σ and returns the resultant store; and $\mathsf{dec}(l, \sigma)$ decrements the reference count of l and returns the resultant store. Other operations include $\mathsf{increnv}(\rho, \sigma)$, which for every variable $x \in \mathsf{dom}(\rho)$, increments the reference count associated with location $\rho(x)$; and $\mathsf{dec\text{-}ptrs}(l, \sigma)$, which decrements the reference count associated with location l, then recursively decrements the reference counts of all locations stored in location l if the reference count of l falls to zero. For instance, $\mathsf{dec\text{-}ptrs}(l, \sigma)$ has the following action:



Note that $\mathsf{dec\text{-}ptrs}(l, \sigma)$ is easier to define if there is no loop within σ starting from location l; this explains our earlier restriction on stores. Complete definitions of $\mathsf{increnv}(\rho, \sigma)$ and $\mathsf{dec\text{-}ptrs}(l, \sigma)$ may be found in Appendix A.

The last operation we need is a relation for allocating memory cells. A subset R of the product $(\mathsf{Storable} \times \mathsf{Store}) \times (\mathsf{Loc} \times \mathsf{Store})$ is an **allocation relation** if for any store σ and storable value S, there is an l' and σ' where $(S, \sigma)\ R\ (l', \sigma')$ and

- $l' \notin \mathsf{dom}(\sigma)$;
- $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$;
- $\sigma'(l') = S$ and $\mathsf{refcount}(l', \sigma') = 1$; and
- For all locations $l \in \mathsf{dom}(\sigma)$, $\sigma(l) = \sigma'(l)$ and $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(l, \sigma')$.

In particular, an allocation relation can reuse cells immediately when their reference count falls to zero—the usual reference-counting garbage collection scheme—or may occasionally invoke a *stop-and-copy* or *mark-and-sweep* collection [4] to determine those cells with reference count zero for use as the next cells. For specificity, we choose some specific allocation relation new, and by abuse of notation write $\mathsf{new}(S, \sigma)$ for some pair (l', σ') such that $(S, \sigma)\ \mathsf{new}\ (l', \sigma')$. Of course, our operational rules should be independent of the choice of allocation relation, a point formalized later.

The operational rules for a reference-counting interpreter are written in *natural* style [21]. The rules derive conclusions of the form

$$(M, \rho, \sigma) \Downarrow_{rc} (l, \sigma')$$

where the domain of ρ is exactly the free variables of M, and l is a location, in the domain of σ', that holds the result of evaluation; [13] gives a *denotational* semantics in essentially the same form. As an example of an operational rule, suppose the language has a construct $(\mathsf{succ}\ M)$ for computing the successor of a number. Then the operational rule is

$$((\mathsf{succ}\ P), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \begin{cases} \mathsf{suppose}\ (P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad \sigma_0(l_0) = n \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0, \sigma_0) \\ \mathsf{in}\ \ (l', \sigma') = \mathsf{new}(n+1, \sigma_1) \end{cases}$$

Note that the reference count of location l_0 is decremented; this maintains a correct reference count on l_0 in a way described more precisely in Section 4.

The reference-counting model and corresponding evaluation of terms should seem quite familiar, but like the SECD machine we probably would not implement the model *directly*: for example, storable values in both models may occupy more than one word of storage [11, 17, 22]. The SECD machine is, in some ways, easier to implement: our rules implicitly involve stack operations, *e.g.*, the evaluation of the body of succ in the rule above, which the SECD machine would make explicit. On the other hand, implementation details lacking from the SECD model have been made explicit in our model, in particular the management of memory.

3 A Functional Language Based on Linear Logic

Syntax: The syntax of the language is based on PCF [23] and Abramsky's linear functional language [1, 2]. Types are built over the grammar

$$s \quad ::= \quad \mathsf{Nat} \mid \mathsf{Bool} \mid (s \multimap s) \mid !s$$

where $\multimap$ denotes linear implication or linear function space, and ! denotes the possibility of using a value zero or more times. (A complete description of linear logic may be found in [2, 7, 8].) We use the letters s, t, u, and v to denote types. Types without leading !'s, *e.g.*, Nat and $(\mathsf{Nat} \multimap \mathsf{Bool})$, are called *linear*; a variable of linear type must be used exactly once in a term, whereas variables of !-type may be used zero or more times.

The set of raw terms in the language is given by

$$\begin{aligned} M \quad ::= \quad & x \mid (\lambda x : s.\ M) \mid (M\ M) \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \\ & (\mathsf{succ}\ M) \mid (\mathsf{pred}\ M) \mid (\mathsf{zero?}\ M) \mid \\ & (\mathsf{if}\ M\ \mathsf{then}\ M\ \mathbf{else}\ M) \mid (\mu x : s.\ M) \mid \\ & (\mathsf{delay}\ M) \mid \\ & (\mathsf{fetch}\ x\ \mathsf{from}\ M\ \mathsf{in}\ M) \mid \\ & (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ M) \mid \\ & (\mathsf{dispose}\ M\ \mathsf{before}\ M) \end{aligned}$$

where the letter x denotes any variable, and n denotes a numeral in $\{0, 1, 2, \ldots\}$. The last four operations correspond to the special rules of linear logic, which we will explicate shortly; the other operations are the operations of PCF. The usual definitions of free and bound variables apply here, where $(\mathsf{fetch}\ x\ \mathsf{from}\ M\ \mathsf{in}\ N)$ and $(\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ N)$ bind the variables x and x, y in N respectively. Terms are identified up to renaming of bound variables, and syntactic substitution is written $M[x := N]$ [3].

The type-checking rules of the language appear in Table 1, and are essentially those given by Abramsky in [1, 2]. The symbols Γ and Δ denote **type contexts**, which are lists of pairs $x_1 : s_1, \ldots, x_n : s_n$, where each x_i is a distinct variable and each s_i is a type. In the rules, $!\Gamma$ denotes any context of the form $(x_1 : !s_1), \ldots, (x_n : !s_n)$, *i.e.*, where the primary type constructor of all the types mentioned in $!\Gamma$ is !. The two constructs $\mathsf{dispose}$ and share require their arguments be of !-type; these are the only constructs available for using a value zero or more times. The rule

$$x : s \vdash x : s \qquad \frac{\Gamma, x : s, y : t, \Delta \vdash M : t}{\Gamma, y : t, x : s, \Delta \vdash M : t}$$

$$\frac{\Gamma \vdash N : u \qquad \Delta, x : s \vdash M : t}{\Gamma, \Delta \vdash M[x := N] : u}$$

$$\frac{\Gamma, x : s \vdash M : t}{\Gamma \vdash (\lambda x : s.\, M) : (s \multimap t)} \qquad \frac{\Gamma \vdash N : s \quad \Delta, x : t \vdash M : u}{\Gamma, \Delta, f : (s \multimap t) \vdash M[x := (f\ N)] : u}$$

$$\frac{!\Gamma \vdash M : s}{!\Gamma \vdash (\mathsf{delay}\ M) : !s} \qquad \frac{\Gamma, x : s \vdash M : t}{\Gamma, z : !s \vdash (\mathsf{fetch}\ x\ \mathsf{from}\ z\ \mathsf{in}\ M) : t}$$

$$\frac{\Gamma \vdash M : t}{\Gamma, x : !s \vdash (\mathsf{dispose}\ x\ \mathsf{before}\ M) : t} \qquad \frac{\Gamma, x : !s, y : !s \vdash N : t}{\Gamma, z : !s \vdash (\mathsf{share}\ x, y\ \mathsf{as}\ z\ \mathsf{in}\ N) : t}$$

$$\vdash n : \mathsf{Nat} \qquad \vdash \mathsf{true}, \mathsf{false} : \mathsf{Bool}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{succ}\ M) : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{pred}\ M) : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{zero?}\ M) : \mathsf{Bool}} \qquad \frac{\Gamma \vdash L : \mathsf{Bool} \quad \Delta \vdash M : s \quad \Delta \vdash N : s}{\Gamma, \Delta \vdash (\mathsf{if}\ L\ \mathsf{then}\ M\ \mathbf{else}\ N) : s}$$

$$\frac{!\Gamma, x : !s \vdash M : s}{!\Gamma \vdash (\mu x : !s.\, M) : s}$$

Table 1: The type-checking rules for the linear logic-based language.

for checking recursions was suggested to us by Samson Abramsky, and will be justified in the next section when we consider the operational semantics of the language.

The rules in Table 1 have a rather different form than most type systems: our system corresponds to a *sequent-style* formulation of linear logic, whereas most type systems correspond to a *natural deduction-style* formulation [8]. One main difference between the two styles arises in the form of the rules. In natural deduction systems, one uses introduction and elimination rules, *e.g.*, the rule

$$\frac{\Gamma \vdash M : (s \multimap t) \qquad \Delta \vdash N : s}{\Gamma, \Delta \vdash (M\ N) : t},$$

is an elimination operation for $\multimap$. Natural deduction type systems must also satisfy another criterion, the *substitutivity property* [8]: if both $\Gamma \vdash M : s$ and $\Delta, x : s \vdash N : t$ are provable, then the *proofs* may be combined by concatenation to yield a proof of $\Gamma, \Delta \vdash N[x := M] : t$. It is unclear whether there is a true natural deduction formulation of the $\{\multimap, !\}$-fragment for our language, although a melding of natural deduction and sequent-style typing rules is discussed in [18]. In the interest of retaining the elegant proof-theoretic properties of linear logic, we use the sequent-style rules.

Examples of Programs: To get a feel for how one writes programs, it is useful to compare the language to Standard ML [20]. For instance, consider the following program written in Standard ML:

```
let fun add x y =
    if (x = 0) then y
               else (add (x-1) (y+1))
in  add 2 1
end;
```

There are two reasons that make this program *not* immediately translatable to our language. First, in the body of the recursive `add` function, the variable `x` is used more than once: in the test for `0` and in the

recursive call to `add`. Multiple uses of variables are not allowed in well-typed terms; instead, the variable names of the two uses must be made distinct through the operation share. On the other hand, even though the variable `y` is *mentioned* twice, it is not *used* twice since at most one branch of the `if` is executed. Second, the variable `x` appears in the `else` branch but not in the `then` branch. Again, this is not allowed; the programmer must explicitly annotate the fact that the variable `x` will not be used in the `then` branch.

The addition program in our language is

```
[μ Add :  !(!Nat ⊸ Nat ⊸ Nat).  λx : !Nat. λy : Nat.
        share w, z as x in
        if zero? (fetch n from w in n)
          then dispose z before (dispose Add before y)
          else (fetch a from Add in a)
                 (delay (pred (fetch x from z in x)))
                 (succ y)]
(delay 2) 1
```

There are some minor syntactic differences between this program and the above Standard ML code, *e.g.*, the recursive function is explicitly declared using μ- and λ-notation. But there are also four new constructs in this program: delay, fetch, share, and dispose. The operations delay and fetch create and destroy *objects* of !-type, which are the only objects that may shared; the operations share and dispose, on the other hand, create and destroy *references* to objects of !-type.

4 Two Operational Semantics

A high-level interpreter for our language, written in natural style, appears in Table 2. The notation $M \Downarrow c$ is read "M halts at canonical term c"; the canonical terms have the form n, true, false, $(\lambda x.\, M)$, and (delay M). We write $M \Downarrow$ when there exists a c such that $M \Downarrow c$. The rules maintain the invariant that only canonical forms are substituted for variables. To preserve this invariant, $(\mu x.\, M)$ is reduced by substituting (delay $(\mu x.\, M)$) in the body M and reducing the resultant term; this operational rule justifies our choice of type-checking rule for recursions.

The rules for the lower-level semantics of our language appear in Appendix A; the rules derive conclusions of the form $(M, \rho, \sigma) \Downarrow_{rc} (l', \sigma')$, where $\mathsf{dom}(\rho)$ is *exactly* the free variables of M. The operations of our language take their names from the intuitive actions of the reference-counting interpreter. For instance, (share x, y as M in N) first evaluates M, then binds the variables x and y to the location returned, and finally evaluates N in the new environment. This creates multiple references to cells. Another important collection of rules are the rules for reducing (fetch x from P in Q); the interpreter evaluates P to a susp'ended object, and returns the result of evaluating the location under the susp, memoizing the result of computation if the susp'ended object points to a thunk. This memoization saves steps when a subsequent fetch of the susp'ended object is done, and hence resembles a *call-by-need* reduction strategy. (Accounts of operational semantics with call-by-need evaluation appear in [5, 25].)

The reference-counting interpreter must satisfy a number of invariants in order for it to be correct:

1. The evaluation of a term in a legal store must always return a store as one of its results, *i.e.*, the resultant store must contain no cycles of references and must have a finite domain.

2. A store σ is **thunk-correct** if for all l such that $\sigma(l) = \mathsf{thunk}(M, \rho)$, $\mathsf{refcount}(l, \sigma) = 0$ or 1. We could reformulate the rules so that this invariant is not required, but maintaining it allows us to optimize the rules. We also say that a tuple $(l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma)$ is **thunk-correct** if σ is.

3. An environment-store pair (ρ, σ) is called **count-correct** if for any location l' in $\mathsf{dom}(\sigma)$, the number of references to l' in $\mathsf{dom}(\sigma)$ plus the number of references to l' in ρ equals $\mathsf{refcount}(l', \sigma)$. Similarly, a location-store pair (l, σ) is count-correct if for any l' in $\mathsf{dom}(\sigma)$, the number of references to l' in $\mathsf{dom}(\sigma)$ plus the number of references to l' in l (obviously either 0 or 1) equals $\mathsf{refcount}(l', \sigma)$. We may also generalize the notion of count-correctness to tuples $(l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma)$ in the obvious way. Intuitively, $(l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma)$ is count-correct if we regard the locations mentioned by $(l_1, \ldots, l_k, \rho_1, \ldots, \rho_n)$ as a "root set" of pointers, and the reference counts of cells take into account these root set pointers.

4. Finally, all environments should hold pointers to *values*. First, we say that σ is **well-formed** if for every location l', $\sigma(l') = \langle y, P, \rho' \rangle$ or $\mathsf{thunk}(P, \rho')$ implies that for each $x \in \mathsf{dom}(\rho')$, $\sigma(\rho'(x))$ is a value, *i.e.*, not a thunk. Again, we may extend this in the straightforward way to tuples: a tuple $(l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma)$ is **well-formed** if σ is well-formed, each $\sigma(l_i)$ is a value, and for every $x \in \mathsf{dom}(\rho_j)$, $\sigma(\rho_j(x))$ is a value.

The fact that the interpreter maintains these invariants can be verified formally. For instance,

Theorem 1 *If σ is a store and $(M, \rho, \sigma) \Downarrow_{rc} (l', \sigma')$, then σ' is a store.*

$$\frac{}{n \Downarrow n} \qquad \frac{}{\mathsf{true} \Downarrow \mathsf{true}} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}$$

$$\frac{M \Downarrow n}{(\mathsf{succ}\ M) \Downarrow (n+1)} \qquad \frac{M \Downarrow (n+1)}{(\mathsf{pred}\ M) \Downarrow n} \qquad \frac{M \Downarrow 0}{(\mathsf{pred}\ M) \Downarrow 0}$$

$$\frac{M \Downarrow 0}{(\mathsf{zero?}\ M) \Downarrow \mathsf{true}} \qquad \frac{M \Downarrow (n+1)}{(\mathsf{zero?}\ M) \Downarrow \mathsf{false}}$$

$$\frac{L \Downarrow \mathsf{true} \quad M \Downarrow c}{(\mathsf{if}\ L\ \mathsf{then}\ M\ \mathbf{else}\ N) \Downarrow c} \qquad \frac{L \Downarrow \mathsf{false} \quad N \Downarrow c}{(\mathsf{if}\ L\ \mathsf{then}\ M\ \mathbf{else}\ N) \Downarrow c}$$

$$\frac{M[x := (\mathsf{delay}\ \mu x.\ M)] \Downarrow c}{\mu x.\ M \Downarrow c}$$

$$\frac{}{\lambda x.\ M \Downarrow \lambda x.\ M} \qquad \frac{M \Downarrow \lambda x.\ P \quad N \Downarrow d \quad P[x := d] \Downarrow c}{(M\ N) \Downarrow c}$$

$$\frac{}{(\mathsf{delay}\ M) \Downarrow (\mathsf{delay}\ M)} \qquad \frac{M \Downarrow (\mathsf{delay}\ P) \quad P \Downarrow c' \quad N[x := c'] \Downarrow c}{(\mathsf{fetch}\ x\ \mathsf{from}\ M\ \mathsf{in}\ N) \Downarrow c}$$

$$\frac{M \Downarrow d \quad N \Downarrow c}{(\mathsf{dispose}\ M\ \mathsf{before}\ N) \Downarrow c} \qquad \frac{M \Downarrow d \quad P[x, y := d] \Downarrow c}{(\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ P) \Downarrow c}$$

Table 2: The high-level operational semantics of the language.

The proof proceeds by a simple induction on the number of steps in $(M, \rho, \sigma) \Downarrow_{rc} (l', \sigma')$. One may also prove that the interpreter maintains correct reference counts.

Theorem 2 *If (ρ, σ) is count-correct, thunk-correct, and well-formed, and $(M, \rho, \sigma) \Downarrow_{rc} (l', \sigma')$, then (l', σ') is count-correct, thunk-correct, and well-formed.*

Proof: (Hint) By induction on the height of $\Downarrow_{rc}$, where we prove the following generalization:

> Suppose the tuple $(l_1, \ldots, l_k, \rho, \rho_1, \ldots, \rho_n, \sigma)$ is well-formed, count-correct, and thunk-correct. If $(M, \rho, \sigma) \Downarrow_{rc} (l', \sigma')$, then the tuple $(l', l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma')$ is well-formed, count-correct, and thunk-correct.

It is worth considering part of one of the cases of this induction. Recall the rule for reducing successors:

$$((\mathsf{succ}\ P), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \begin{cases} \mathsf{suppose}\ (P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad \sigma_0(l_0) = n \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0, \sigma_0) \\ \mathsf{in}\ \ (l', \sigma') = \mathsf{new}(n+1, \sigma_1) \end{cases}$$

The rule first reduces the operand of succ using the reduction $(P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0)$. By the induction hypothesis, we assume that if $(l_1, \ldots, l_k, \rho, \rho_1, \ldots, \rho_n, \sigma)$ is count-correct, thunk-correct, and well-formed, then so is $(l_0, l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma_0)$. The result is then tested to make sure it is a numeral n. Next, the reference count of location l_0 is decremented yielding a new store σ_1; thus, $(l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma_1)$ is count-correct. Finally, a new cell l' is allocated to store the result; thus, the result $(l', l_1, \ldots, l_k, \rho_1, \ldots, \rho_n, \sigma')$ is count-correct. Proving that the result is well-formed and thunk-correct is easier, and the other cases of the induction are similar to this one. ■

Finally, one may prove that alternative choices of the allocation relation "new" do not result in different results. For specificity, if f is an allocation relation, let $\Downarrow_{rc,f}$ be the relation defined (as in the Appendix) by using f in the place of new. Then

Theorem 3 *Suppose f and g are allocation relations, and $(M, \rho, \sigma) \Downarrow_{rc,f} (l', \sigma')$ and $\sigma'(l') = n$,* true, *or* false. *Then $(M, \rho, \sigma) \Downarrow_{rc,g} (l'', \sigma'')$ and $\sigma''(l'') = \sigma'(l')$.*

The proof again follows by an induction on the proof of $(M, \rho, \sigma) \Downarrow_{rc,f} (l', \sigma')$, using a strengthened induction hypothesis describing the equivalence of two stores.

It is harder to prove that the reference-counting interpreter correctly implements the high-level interpreter.

Theorem 4 *For any closed term M, $M \Downarrow c$ iff $(M, \emptyset, \emptyset) \Downarrow_{rc} (l', \sigma')$. Moreover, $c = n$ iff $\sigma'(l') = n$, and similarly for* true, false.

Proof: (Hint) The main difficulty in proving this theorem lies in the memoization done under susp'ended objects in the reference-counting interpreter. This memoization causes the reference-counting interpreter to return results that are "more evaluated" than results returned by the high-level interpreter. To overcome these difficulties, we first define a relation $M \leq N$, read "M is more evaluated than N," in which closed subterms of N may be reduced via $\Downarrow$ to obtain M. We also define the functions $\mathsf{valof}(M, \rho, \sigma)$ and $\mathsf{valofcell}(l, \sigma)$ for extracting closed terms out of an environment-store pair or out of a cell-store pair. The definition of $\leq$ can be extended to stores in the obvious way. We then prove the following by induction on the height of $\Downarrow_{rc}$ and $\Downarrow$:

> Suppose the tuples $(l_1, \ldots, l_k, \rho, \rho_1, \ldots, \rho_n, \sigma)$ and $(l_1, \ldots, l_k, \rho, \rho_1, \ldots, \rho_n, \sigma')$ are well-formed, thunk-correct, and count-correct, and $\sigma' \leq \sigma$. Then $\mathsf{valof}(M, \rho, \sigma) \Downarrow c$ iff $(M, \rho, \sigma') \Downarrow_{rc} (l'', \sigma'')$ where $\sigma'' \leq \sigma'$ and $\mathsf{valofcell}(l'', \sigma'') \leq c$.

Since the only terms $\leq$ basic constants are the constants themselves, the theorem follows directly. ■

5 The Linear Type System and Memory Management

It would seem that only susp'ended objects may have more than one pointer to them. This would lead to many potential optimizations: succ and pred could, for instance, be implemented using in-place updating. Unfortunately, this is not the case. Consider the closed typable term

share y, z as (delay 2)
  in (fetch g from y in (fetch h from z in M))

for some term M. During evaluation, the value 2 is placed in some cell l_1, and y and z are bound to l_0 which holds $\mathsf{susp}(l_1)$. When the evaluation continues, the variable g is also bound to l_1, so l_1 has more than one pointer to it at some intermediate stage. We could, of course, explicitly copy the cell containing 2 and bind g to a new location holding 2, and thus avoid having more than one pointer to a non-susp'ended cell. But if 2 were instead some closure, we would have to copy all cells referenced by the environment of the closure, and then recursively copy those cells referenced by those cells copied. It is clear that this would be a very expensive operation. There may well be other general ways to guarantee that the reference counts of non-susp'ended objects are always 1, but we have not found any honest, efficient means yet.

Linear constructs do provide information, though, about *when* certain data structures may be garbage collected. Consider, for example, the terms

$N_1 =$ ((λx. if B then (dispose x before P)
  else (dispose x before Q)) M)
$N_2 =$ ((λx. dispose x before if B then P else Q) M)

N_2 dispose's the cell bound to M as early as possible; this may potentially free enough space so that the computation of B does not run into problems. This gives the programmer fine control over memory management with some degree of safety: type-correct programs do not dispose pointers unless it is safe to do so, insuring the absence of dangling pointers which may happen in languages like C with memory-management primitives.

6 Relation to Previous Work

Our study has focused primarily on abstract models of memory management for the purposes of detecting certain optimizations. Jones and Muchnick [14], Hudak [13], and Deutsch [6] use abstract interpretation to deduce the correctness of certain compiler optimizations, *e.g.*, in-place updating of arrays. Only Hudak's work explicitly incorporates memory management (through reference-counting), although it probably could be added to the other models. The main difference between these papers and ours is that we prove more properties of our low-level operational semantics: first, that it matches the higher-level description, and second, that it satisfies the necessary invariants. Such principles may be harder to state and prove in the models of these papers.

Others have discovered that objects with linear type do not necessarily have one pointer to them. For instance, Wadler [26] noticed (although informally) in the context of graph reduction, subgraphs corresponding to terms with linear type do not necessarily have one pointer to them. Wadler's language, though, is slightly different than ours—he considers terms without the linear constructs of delay, fetch, dispose, and share, and attempts to infer the position of these connectives in raw λ-terms.

Lincoln and Mitchell [19], on the other hand, use almost exactly the same language as ours with slightly

different typing rules. They also define an abstract machine with two heaps, one for objects with only one pointer to them, and one for objects with possibly more than one pointer to them. As with our work, Lincoln and Mitchell find that objects with linear type cannot be guaranteed to have exactly one pointer to them; they may need to be placed in the second heap. It seems that further work is needed to prove that their abstract machine exploits the linear type system in any significant way.

7 Conclusion

The goal of our work here is to develop tools for formally *establishing* applications for linear logic, as well as proving memory-related optimizations correct. What we have done should also be relevant to other investigations that involve rigorous demonstrations of properties at several levels of semantic abstraction, such as work on efficient implementations of call-by-name evaluation.

The linear language may have uses beyond guaranteeing a safe "dispose" operation. For instance, programs in functional languages, *e.g.*, PCF, can be translated into the syntax of our language in a relatively straightforward manner [7]. Our language may therefore provide a suitable intermediate representation for finding memory management optimizations, just as continuation-passing style is used to find control optimizations. The language may also be useful as a tool for profiling memory usage in standard functional languages.

Acknowledgments: We thank Andre Scedrov for describing the differences between sequent and natural deduction-style proof systems, and James Hicks for pointing us to the literature on abstract interpretation and memory management. Dave MacQueen and Stuart Feldman provided some helpful views on programming languages design and memory management.

References

[1] Samson Abramsky. Computational interpretations of linear logic. Unpublished manuscript, Imperial College, University of London, October 1990.

[2] Samson Abramsky. Intuitionistic linear logic considered as a programming language. Slides for a talk given at MFPS '90, Toronto, May 1990.

[3] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 1981. Revised Edition, 1984.

[4] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, pages 341–367, 1981.

[5] Eric Crank and Matthias Felleisen. Parameter-passing and the lambda calculus. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 233–244. ACM, 1991.

[6] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168. ACM, 1990.

[7] Jean-Yves Girard. Linear logic. *Theoretical Computer Sci.*, 50:1–102, 1987.

[8] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[9] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. To be published by the MIT Press, 1992.

[10] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

[11] Peter Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall, 1980.

[12] William A. Howard. The formulae-as-types notion of construction. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[13] Paul Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363. ACM, 1986.

[14] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74. ACM, 1982.

[15] Yves Lafont. The linear abstract machine. *Theoretical Computer Sci.*, 59:157–180, 1988.

[16] Yves Lafont. *Logiques, Categories & Machines.* PhD thesis, University of Paris VII, 1988.

[17] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.

[18] Patrick Lincoln and John C. Mitchell. Intuitionistic linear logic propositions as types. Unpublished manuscript, March 1991.

[19] Patrick Lincoln and John C. Mitchell. Operational aspects of linear lambda calculus. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, 1992. To appear.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[21] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction for Computer Science.* To be published by John Wiley & Sons, next to final edition, April 1991.

[22] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Sci.*, 1:125–159, 1975.

[23] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[24] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[25] S. Purushothaman and Jill Seaman. An adequate operational semantics of sharing in lazy evaluation. Technical Report PSU-CS-91-18, Pennsylvania State University, 1991.

[26] Philip Wadler. Is there a use for linear logicΓ In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273. ACM, 1991.

A Reference-Counting Interpreter

$$\mathsf{increnv}(\rho,\sigma) = \begin{cases} \sigma_n, & \text{where the domain of } \rho \text{ is } \{x_1,\ldots,x_n\} \\ & \sigma_1 = \mathsf{incr}(\rho(x_1),\sigma) \\ & \vdots \\ & \sigma_n = \mathsf{incr}(\rho(x_n),\sigma_{n-1}) \end{cases}$$

$$\mathsf{decenv\text{-}ptrs}(\rho,\sigma) = \begin{cases} \sigma_n, & \text{where the domain of } \rho \text{ is } \{x_1,\ldots,x_n\} \\ & \sigma_1 = \mathsf{dec\text{-}ptrs}(\rho(x_1),\sigma) \\ & \vdots \\ & \sigma_n = \mathsf{dec\text{-}ptrs}(\rho(x_n),\sigma_{n-1}) \end{cases}$$

$$\mathsf{dec\text{-}ptrs}(l,\sigma) = \begin{cases} \mathsf{dec}(l,\sigma) & \text{if } \sigma(l) = n, \mathsf{true}, \text{ or } \mathsf{false} \\ \mathsf{dec}(l,\sigma) & \text{if } \mathsf{refcount}(l,\sigma) > 1 \\ \mathsf{dec\text{-}ptrs}(l',\sigma') & \text{if } \mathsf{refcount}(l,\sigma) = 1,\ \sigma' = \mathsf{dec}(l,\mathsf{update}(l,0,\sigma)), \text{ and} \\ & \sigma(l) = \mathsf{susp}(l') \\ \mathsf{decenv\text{-}ptrs}(\rho,\sigma') & \text{if } \mathsf{refcount}(l,\sigma) = 1,\ \sigma' = \mathsf{dec}(l,\mathsf{update}(l,0,\sigma)), \text{ and} \\ & \sigma(l) = \langle x, M, \rho\rangle \\ \mathsf{decenv\text{-}ptrs}(\rho,\sigma') & \text{if } \mathsf{refcount}(l,\sigma) = 1,\ \sigma' = \mathsf{dec}(l,\mathsf{update}(l,0,\sigma)), \text{ and} \\ & \sigma(l) = \mathsf{thunk}(M,\rho) \end{cases}$$

(1) $$(x,\rho,\sigma) \Downarrow_{rc} (\rho(x),\sigma)$$

(2) $$(n,\rho,\sigma) \Downarrow_{rc} \mathsf{new}(n,\sigma)$$

(3) $$(\mathsf{true},\rho,\sigma) \Downarrow_{rc} \mathsf{new}(\mathsf{true},\sigma)$$

(4) $$(\mathsf{false},\rho,\sigma) \Downarrow_{rc} \mathsf{new}(\mathsf{false},\sigma)$$

(5) $$((\lambda x.\ P),\rho,\sigma) \Downarrow_{rc} \mathsf{new}(\langle x,P,\rho\rangle,\sigma)$$

(6) $$((\mathsf{delay}\ P),\rho,\sigma) \Downarrow_{rc} \mathsf{new}(\mathsf{susp}(l_0),\sigma_0), \quad \text{where } (l_0,\sigma_0) = \mathsf{new}(\mathsf{thunk}(P,\rho),\sigma)$$

(7) $$((\mathsf{succ}\ P),\rho,\sigma) \Downarrow_{rc} (l',\sigma') \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P,\rho,\sigma) \Downarrow_{rc} (l_0,\sigma_0) \\ \qquad \sigma_0(l_0) = n \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0,\sigma_0) \\ \mathsf{in}\ \ (l',\sigma') = \mathsf{new}(n+1,\sigma_1) \end{cases}$$

(8) $$((\mathsf{pred}\ P),\rho,\sigma) \Downarrow_{rc} (l',\sigma') \quad \text{if} \quad (P,\rho,\sigma) \Downarrow_{rc} (l',\sigma') \text{ and } \sigma'(l') = 0$$

(9) $$((\mathsf{pred}\ P),\rho,\sigma) \Downarrow_{rc} (l',\sigma') \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P,\rho,\sigma) \Downarrow_{rc} (l_0,\sigma_0) \\ \qquad \sigma_0(l_0) = n+1 \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0,\sigma_0) \\ \mathsf{in}\ \ (l',\sigma') = \mathsf{new}(n,\sigma_1) \end{cases}$$

(10) $$((\mathsf{zero?}\ P),\rho,\sigma) \Downarrow_{rc} (l',\sigma') \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P,\rho,\sigma) \Downarrow_{rc} (l_0,\sigma_0) \\ \qquad \sigma_0(l_0) = 0 \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0,\sigma_0) \\ \mathsf{in}\ \ (l',\sigma') = \mathsf{new}(\mathsf{true},\sigma_1) \end{cases}$$

(11) $$((\mathsf{zero?}\ P),\rho,\sigma) \Downarrow_{rc} (l_0,\sigma_1) \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P,\rho,\sigma) \Downarrow_{rc} (l_0,\sigma_0) \\ \qquad \sigma_0(l_0) = n+1 \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0,\sigma_0) \\ \mathsf{in}\ \ (l',\sigma') = \mathsf{new}(\mathsf{false},\sigma_1) \end{cases}$$

$$(12)\qquad (\text{if } N \text{ then } P \textbf{ else } Q, \rho_1 \cup \rho_2, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (N, \rho_1, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \sigma_0(l_0) = \text{true} \\ \text{let } \sigma_1 = \text{dec}(l_0, \sigma_0) \\ \text{in } \ (P, \rho_2, \sigma_1) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(13)\qquad (\text{if } N \text{ then } P \textbf{ else } Q, \rho_1 \cup \rho_2, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (N, \rho_1, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \sigma_0(l_0) = \text{false} \\ \text{let } \sigma_1 = \text{dec}(l_0, \sigma_0) \\ \text{in } \ (Q, \rho_2, \sigma_1) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(14)\qquad ((P\ Q), \rho_1 \cup \rho_2, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (P, \rho_1, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \text{refcount}(l_0, \sigma_0) = 1 \\ \qquad\quad \sigma_0(l_0) = \langle x, N, \rho' \rangle \\ \text{let } \sigma_2 = \text{dec}(l_0, \sigma_0) \\ \quad (Q, \rho_2, \sigma_2) \Downarrow_{rc} (l_3, \sigma_3) \\ \text{in } \ (N, \rho'[x \mapsto l_3], \sigma_3) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(15)\qquad ((P\ Q), \rho_1 \cup \rho_2, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (P, \rho_1, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \text{refcount}(l_0, \sigma_0) > 1 \\ \qquad\quad \sigma_0(l_0) = \langle x, N, \rho' \rangle \\ \text{let } \sigma_1 = \text{dec}(l_0, \sigma_0) \\ \quad \sigma_2 = \text{increnv}(\rho', \sigma_1) \\ \quad (Q, \rho_2, \sigma_2) \Downarrow_{rc} (l_3, \sigma_3) \\ \text{in } \ (N, \rho'[x \mapsto l_3], \sigma_3) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(16)\qquad ((\mu x.\, P), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{let } (l_0, \sigma_0) = \text{new}(\text{thunk}((\mu x.\, P), \rho), \sigma) \\ \quad (l_1, \sigma_1) = \text{new}(\text{susp}(l_0), \sigma_0) \\ \quad \sigma_2 = \text{increnv}(\rho, \sigma_1) \\ \text{in } \ (P, \rho[x \mapsto l_1], \sigma_2) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(17)\qquad ((\text{dispose } P \text{ before } Q), \rho_1 \cup \rho_2, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (P, \rho_1, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \text{let } \sigma_1 = \text{dec-ptrs}(l_0, \sigma_0) \\ \text{in } \ (Q, \rho_2, \sigma_1) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(18)\qquad ((\text{share } x, y \text{ as } P \text{ in } Q), \rho_1 \cup \rho_2, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (P, \rho_1, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \text{let } \sigma_1 = \text{incr}(l_0, \sigma_0) \\ \text{in } \ (Q, \rho_2[x, y \mapsto l_0], \sigma_1) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$(19)\qquad ((\text{fetch } x \text{ from } P \text{ in } Q), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \text{suppose } (P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \sigma_0(l_0) = \text{susp}(l_1) \\ \qquad\quad \text{refcount}(l_0, \sigma_0) = 1 \\ \qquad\quad \sigma_0(l_1) = V \\ \text{let } \sigma_1 = \text{dec}(l_0, \sigma_0) \\ \text{in } \ (Q, \rho[x \mapsto l_1], \sigma_1) \Downarrow_{rc} (l', \sigma') \end{cases}$$

$$
(20) \qquad ((\mathsf{fetch}\ x\ \mathsf{from}\ P\ \mathsf{in}\ Q), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \sigma_0(l_0) = \mathsf{susp}(l_1) \\ \qquad\quad \mathsf{refcount}(l_0, \sigma_0) > 1 \\ \qquad\quad \sigma_0(l_1) = V \\ \mathsf{let}\ \sigma_1 = \mathsf{incr}(l_1, \sigma_0) \\ \quad\ \ \sigma_2 = \mathsf{dec}(l_0, \sigma_1) \\ \mathsf{in}\ \ (Q, \rho[x \mapsto l_1], \sigma_2) \Downarrow_{rc} (l', \sigma') \end{cases}
$$

$$
(21) \qquad ((\mathsf{fetch}\ x\ \mathsf{from}\ P\ \mathsf{in}\ Q), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \sigma_0(l_0) = \mathsf{susp}(l_1) \\ \qquad\quad \mathsf{refcount}(l_0, \sigma_0) = 1 \\ \qquad\quad \sigma_0(l_1) = \mathsf{thunk}(R, \rho') \\ \mathsf{let}\ \sigma_1 = \mathsf{dec}(l_0, \sigma_0) \\ \quad\ \ \sigma_2 = \mathsf{dec}(l_1, \sigma_1) \\ \quad\ \ (R, \rho', \sigma_2) \Downarrow_{rc} (l_3, \sigma_3) \\ \mathsf{in}\ \ (Q, \rho[x \mapsto l_3], \sigma_3) \Downarrow_{rc} (l', \sigma') \end{cases}
$$

$$
(22) \qquad ((\mathsf{fetch}\ x\ \mathsf{from}\ P\ \mathsf{in}\ Q), \rho, \sigma) \Downarrow_{rc} (l', \sigma') \quad \text{if} \quad \begin{cases} \mathsf{suppose}\ (P, \rho, \sigma) \Downarrow_{rc} (l_0, \sigma_0) \\ \qquad\quad \sigma_0(l_0) = \mathsf{susp}(l_1) \\ \qquad\quad \mathsf{refcount}(l_0, \sigma_0) > 1 \\ \qquad\quad \sigma_0(l_1) = \mathsf{thunk}(R, \rho') \\ \mathsf{let}\ \sigma_1 = \mathsf{increnv}(\rho', \sigma_0) \\ \quad\ \ (R, \rho', \sigma_1) \Downarrow_{rc} (l_2, \sigma_2) \\ \quad\ \ \sigma_3 = \mathsf{update}(l_0, \mathsf{susp}(l_2), \sigma_2) \\ \quad\ \ \sigma_4 = \mathsf{dec\text{-}ptrs}(l_1, \sigma_3) \\ \quad\ \ \sigma_5 = \mathsf{dec}(l_0, \mathsf{incr}(l_2, \sigma_4)) \\ \mathsf{in}\ \ (Q, \rho[x \mapsto l_2], \sigma_5) \Downarrow_{rc} (l', \sigma') \end{cases}
$$