J. Functional Programming **6** (2): 195–244, March 1996 © 1996 Cambridge University Press

Reference counting as a computational interpretation of linear logic

JAWAHAR CHIRIMAR, CARL A. GUNTER
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104, USA

JON G. RIECKE
Bell Laboratories
700 Mountain Avenue, Murray Hill, NJ 07974, USA

Abstract

We develop an operational model for a language based on linear logic. Our semantics is 'low-level' enough to express sharing and copying while still being 'high-level' enough to abstract away from details of memory layout, and thus can be used to test potential applications of linear logic for analysis of programs. In particular, we demonstrate a precise relationship between type correctness for the linear-logic-based language and the correctness of a reference-counting interpretation of the primitives, and formulate and prove a result describing the possible run-time reference counts of values of linear type.

Capsule Review

A number of languages based on linear logic have been proposed, and for some of these it has been claimed that 'values of linear type have exactly one pointer to them'. This claim is reasonable for call-by-name implementations of linear logic, but less reasonable for call-by-need implementations. Formalising a correct counterpart in the latter case turns out to be somewhat tricky, and is the contribution of this paper.

The paper considers a language based on linear logic, with explicit store and fetch operations. An operational semantics is presented, detailed enough to record sharing and duplication of values stored in memory. The main result is a connection between linear logic types for the functional language and reference counts to stored linear values.

1 Introduction

Many people have attempted to exploit type systems based on linear logic to discover optimizations for programs. Linear logic (Girard, 1987), hereafter called LL, is a resource-conscious logic in which proofs use hypotheses in very limited ways. An LL-based type system thus may, or so the informal reasoning goes, have some bearing on the way that resources are used in the evaluation of programs. Our goal in this work is to figure out precisely what this connection is to the memory usage of a program. We argue that an LL-based language yields fine-grained information about how the *memory graph* – a graph (whose nodes represent memory cells and

Table 1. *Translating to a LL-Based Language.*

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

```
let fun add x y =
      share w,z as x in
        if fetch w = 0
        then dispose z, add before y
        else (fetch add)
               (store ((fetch z)-1))
               (y+1)
in add (store 2) 1
end
```

edges represent pointers) together with a finite set of roots – evolves at run-time. This information can be exploited in program analysis; in particular, we show that LL primitives can be interpreted as manipulating the *reference counts* of nodes in the memory graph, and thus given suitable restrictions, programs in our LL-based language can be optimized using in-place updating or other memory-based optimizations.

Attempts to study LL-based programming languages fall roughly into two groups. The first group remains faithful to LL using some analog of the Curry-Howard correspondence (Howard, 1980) as the basis (*cf.* (Abramsky, 1993; Holmström, 1988; Lafont, 1988; Lincoln and Mitchell, 1992; Mackie, 1991)); the second group uses systems more or less similar to LL for specific applications (for instance, Guzmán and Hudak (1990) and Wadler (1991) consider systems to detect single-threading). The system presented in this paper falls into the first group.

It is helpful to begin with an example. Consider the program on the left of Table 1, which implements an addition function from primitives for incrementing and decrementing. The syntax is that of SML (Milner *et al.*, 1990) using numerals, recursive definition, conditionals, and local definition. Note the differences in the use of the formal parameters x and y: first, variables may be *used* a different number of times (y exactly once and x either once or twice) and, second, the value of a variable (in this case x) may be *shared* between its two separate uses. More precisely, the value of x is needed in the test of the conditional, which is always evaluated, and in the else branch of the conditional, which may not be evaluated, but not in the then branch of the conditional. On the other hand, the variable y is needed regardless of whether the then or else branch of the conditional in the body is taken.

The program on the right of Table 1 is a version of the addition function written in a program with LL annotations (using a slight simplification of the notation that we will define precisely later). There are four new primitives used here: share, dispose, store, and fetch. The share primitive indicates that x is needed twice: the first use is bound to the variable w and the second to the variable z. These two variables *share* the value to which x is bound. The dispose primitive indicates that one of these sharing variables, z, is not used in the first branch of the conditional. The primitive store creates a sharable value and fetch obtains a shared value. The LL annotations thus make clear the distinctions in use between x and y from the original SML program.

In our interpretation, the LL-specific operations share and dispose explicitly manage reference counts of the share'able and dispose'able created by store. Reference counting has a long (Collins, 1960; Deutsch and Bobrow, 1976) and controversial (Wise, *et al.*, 1993; Baker, 1978; Appel, 1992) history as a garbage collection methodology, but we use reference counting only for the sake of program analysis; the implementer of the language is free to use any garbage-collection technique. The share operation indicates that two pointers are needed for the value associated with x, so the reference count of the associated value is incremented. The dispose primitive has an analog (and namesake) in several programming languages. In our LL language the primitive dispose will decrement a reference count; deallocation only happens when this count falls to zero. The operation dispose thus never creates dangling pointers which may happen in other languages with dispose (like C). Analogs to the store and fetch operations are the *delay* and *force* operations that appear in many functional programming languages. In such languages, the delay primitive postpones the evaluation of a term until it is supplied to the force primitive as an argument. When this happens, the value of the delayed term is computed, returned, and memoized for any other applications of force. Abramsky (1993) has argued that this is a natural way to view the operational semantics of the store and fetch operations of LL.

One of the primary goals of this paper is to offer a framework for rigorously expressing and proving optimizations obtained by analyzing an LL-based language. For instance, we may test the claim that 'linear values have only one pointer to them' or 'linear values can be updated in place'. Wadler (1990) has informally observed that these claims must be stated with some care. For instance, Lafont (1988) has developed an operational semantics maintaining the invariant that linear values have only one pointer to them (and thus a reference count of one). Linear values may therefore be overwritten – or subject to in-place updates – once they are used. Lafont's operational interpretation maintains the invariant by copying linear values. Lincoln and Mitchell (1992) take a slightly different approach. Their operational semantics divides memory into two subspaces: the first contains objects with exactly one pointer, the second contains objects with possibly many pointers. Non-linear objects – those created by share – live in the second subspace, whereas linear objects, when created, live in the first subspace. Objects from the first subspace that fall within the scope of share get copied to the second subspace.

Our operational semantics fits more with the tradition of functional languages and seems to formalize the model implicit in Wadler's work. Copying is never done; instead, multiple pointers are created to objects, and in doing so we avoid the time penalty of copying. In our semantics, we find that linear variables may *fail* to have a count of one – in other words, a linear variable may not have a unique pointer to it and thus may not be amenable to updating in place. This is reminiscent of (Lincoln and Mitchell, 1992) when linear objects get copied to non-linear space when they fall under the scope of share. On the other hand, all is not lost – we can state a theorem stating precisely when a linear object maintains a reference count of at most one, and thus may be subject to in-place updating. Such a theorem seems difficult to state in a language without LL-like primitives, and thus the LL-based language may yield more opportunities for memory-based optimizations. A broader theme of our investigation is developing a level of abstraction in the semantics of programming languages that permits 'low-level' concepts to be formalized in a clear and relevant way. There has been significant progress in formulating theorems about programming languages and memory – Goldberg and Gloger (1992) and Wand and Oliva (1992) are recent examples treating garbage collection and run-time storage representation, respectively. It is our hope that we can contribute to a foundation for further advances in this direction.

We will be concerned only with the question of a computational interpretation of *intuitionistic* LL, the fragment of the language without negation and the 'par' operation. In fact, we will restrict ourselves to the language obtained from the linear implication ($s \multimap t$) and 'of course' ($!s$) operations, although our results can be extended to all of intuitionistic LL. For the rest of the paper, read 'LL' to mean the implicational fragment of intuitionistic LL. We present our language and its properties in stages. The second section of the paper discusses the operational semantics of memoization with the aim of putting in place the basic notation and approach that will be used in subsequent sections. The third section describes the syntax, typing rules, and 'high-level' operational semantics for our LL-based language. The fourth section describes the 'low-level' operational semantics of the language. The invariants that express the basic properties of the memory graph in this semantics are precisely expressed and proved. The fifth section of the paper demonstrates further basic properties of this semantics, including its correspondence to the high-level semantics and its independence from the scheme used to allocate new memory. The sixth section uses the operational semantics to prove a static condition under which a linear value will always have a reference count of one; this shows that the LL-based language is indeed amenable to analysis about memory usage. The seventh section discusses various aspects of the technical results of the paper and attempts to provide additional perspective. Some of the most technical proofs have been deferred to an appendix.

2 Operational semantics with memory

Here we give a preview of the operational semantics of the LL-based language by describing the familiar operational semantics of a simple functional language with store (delay) and fetch (force) operations. We base this preliminary discussion on a language with the grammar

$$\begin{aligned} M ::= {} & x \mid (\lambda x.\, M) \mid (M\ M) \mid \\ & n \mid \mathsf{true} \mid \mathsf{false} \mid (\mathsf{succ}\ M) \mid (\mathsf{pred}\ M) \mid (\mathsf{zero?}\ M) \mid \\ & (\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M) \mid (\mathsf{fix}\ M) \mid \\ & (\mathsf{store}\ M) \mid (\mathsf{fetch}\ M) \end{aligned}$$

where x and n are from primitive syntax classes of variables and numerals respectively. This is a variant of PCF (Scott, 1993; Plotkin, 1977; Breazu-Tannen *et al.*,

1990) augmented by primitive operations for forcing and delaying evaluations. The expression (fix M) is used for recursive definitions.

The key to providing a semantics for this language is to represent the memoization used in computing the `fetch` primitive so that certain recomputation is avoided. We aim to provide a semantics at a fairly high level of abstraction using what is sometimes known as a *natural semantics* (Despeyroux, 1986; Kahn, 1987). Such a semantics has been described in Purushothaman and Seaman (1991) using explicit substitution and in Launchbury (1993) through the use of an intermediate representation in which all function applications have variables as arguments. Both of these approaches are appealingly simple but slightly more abstract than we would like. Our approach, first described in a preliminary version (Chirimar *et al.*, 1992), uses the traditional distinction between an *environment*, which is an association of variables with locations, and a *store*, which is an association of values with locations. Sharing of computation results is achieved through creating multiple references to a location that holds a delayed computation called a *thunk*. When the value delayed in the thunk is needed, it is calculated and memoized for future reference. To define this precisely we must begin with some notation and basic operations for environments, stores, and memory allocation.

Fix an infinite set of locations Loc, with the letter l denoting elements of this set. Let us say that a partial function is finite just in case its domain of definition is finite.

- An **environment** is a finite partial 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 to update an environment, we use the notation

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

  The symbol $\emptyset$ denotes the empty environment; we also use $[x \mapsto l]$ as shorthand for $\emptyset[x \mapsto l]$.
- A **value** is a
  - — numeral k,
  - — boolean b,
  - — pointer $\mathsf{susp}(l)$ or $\mathsf{rec}(l, f)$, or
  - — closure $\mathsf{closure}(\lambda x.\, M, \rho)$ or $\mathsf{recclosure}(\lambda x.\, M, \rho)$.

  The letter V denotes a value, and Value denotes the set of values.
- A **storable object** is either a value or a thunk $\mathsf{thunk}(M, \rho)$. We use $\mathsf{Storable}$ to denote the set of storable objects.
- A **store** is a finite partial function σ from Loc to $\mathsf{Storable}$. The symbol σ denotes a store, $\emptyset$ denotes the empty store, and Store denotes the set of stores. We will use the same notation to update stores as for updating environments.

We also need 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 object S, there is an l' and σ' where $(S, \sigma)\ R\ (l', \sigma')$ and

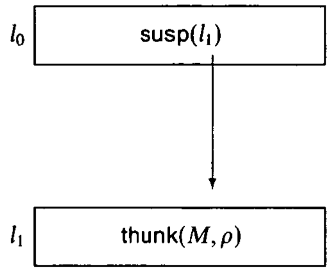


Fig. 1. Structure generated by (store M).

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

This definition abstracts away from the issue of exactly how new locations are found. For specificity, we choose an allocation relation new that is a function, and write new(S, σ) for the pair (l', σ') such that (S, σ) new (l', σ'). Of course, our operational semantics should be independent of the choice of allocation relation, a point we will formalize after describing the semantics of our LL-based language below.

The operational rules for our language could be given using a natural semantics with rules of the form $(M, \rho, \sigma) \Downarrow (l, \sigma')$ where the domain of ρ contains the set of free variables of M, and l is a location in the domain of σ' that holds the result of evaluation. Writing the semantics in the form of rules (e.g., as in the appendix of (Chirimar *et al.*, 1992)) becomes somewhat cumbersome, so we use a kind of primitive pseudo-code that can readily be translated into a natural semantics. As a first example, consider how the store primitive is evaluated:

```
meminterp((store $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = new(thunk($M$, $\rho$), $\sigma$)
  in new(susp($l_0$), $\sigma_0$)
```

Read this as follows. To evaluate (store M) in the environment ρ and store σ, first allocate a new location holding a thunk composed of M and the environment ρ. Let σ_0 be the new store and l_0 be the location in which the thunk is held. The result of the evaluation is a store obtained from σ_0 by allocating a new location holding the storable value susp(l_0) paired with this new location. Note, in particular, that M is not evaluated. The structure that has been added to the memory is depicted in Figure 1.

The interesting part of the evaluator and the essence of memoization is given by the way in which the fetch primitive is handled. The argument of fetch is evaluated to return a storable value of the form susp(l_1). The content of location l_1 is then examined to determine whether the suspension has been evaluated to a value or whether it has not yet been evaluated, in which case it has the form thunk(N, ρ). If the content is a value, a pointer to the value is returned, otherwise the thunk is evaluated and the susp is duly updated. A pointer to the value of the thunk is then returned as the result. Here is the pseudo-code description:

```
meminterp((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp($M$, $\rho$, $\sigma$)
  in  case $\sigma_0(l_0)$ of susp($l_1$) =>
        case $\sigma_0(l_1)$
          of thunk($N$, $\rho'$) =>
               let ($l_2$, $\sigma_1$) = meminterp($N$, $\rho'$, $\sigma_0$)
               in  ($l_2$, $\sigma_1[l_0 \mapsto$ susp($l_2$)])
           | _ => ($l_1$, $\sigma_0$)
```

In the case that $\sigma_0(l_0)$ is *not* a suspension, we assume that the behavior of the interpreter on (fetch M) is undefined. This assumption simplifies the rules, and allows us to ignore what are, in effect, run-time type errors. The type systems we will introduce later will prevent these run-time type errors.

There is another approach we might have taken to modeling memoization. The interpretation of (store M) allocates a location l_0 that holds a thunk, and returns a location l_1 that holds a pointer susp(l_0) to this location. Could we instead have returned l_0 as the value? That is, the rule could read

```
meminterp'((store $M$), $\rho$, $\sigma$) = new(thunk($M$, $\rho$), $\sigma$)
```

The answer to this question is instructive, since it relates to the way in which we will represent the distinction between copying and sharing in our model. If we choose to return the location holding the thunk as the value of the store (as opposed to returning a location holding the pointer to this thunk), then this would require a change in the fetch command. In particular, when the location l_2 is obtained there, it would be essential to put the value $\sigma(l_2)$ in the location where the value of the thunk may be sought later:

```
meminterp'((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp'($M$, $\rho$, $\sigma$)
  in  case $\sigma_0(l_0)$
        of thunk($N$, $\rho'$) =>
             let ($l_2$, $\sigma_1$) = meminterp'($N$, $\rho'$, $\sigma_0$)
             in  ($l_0$, $\sigma_1[l_0 \mapsto \sigma_1(l_2)]$)
         | _ => ($l_0$, $\sigma_0$)
```

Note that in the second line from the bottom of the program, the values of l_0 and l_2 in the store are the same and we will say that the value of the thunk has been *copied* from location l_2 to l_0. In the case that $\sigma_1(l_2)$ is a 'small' value, like an integer that occupies only a word of storage, there is little difference between copying the value from l_2 to l_0 versus returning a pointer to l_2 as we did in the earlier implementation. If the value $\sigma_1(l_2)$ is 'large', however, then copying may be expensive. In the language we are considering, this might involve copying a closure, which would be a modest expense, but in a fuller language it might involve copying a string or functional array, which could be very expensive. (If $\sigma_1(l_2)$ is a mutable value, then the copying is probably incorrect – but this is not a problem for the functional language at hand.) Our semantics does not directly represent the cost associated with copying because it abstracts away from a measure of the size of a value; instead, we will treat copying as if it is something to be avoided in favor of sharing (indirection) whenever this is feasible. This suggests yet a third approach to the semantics of fetch where store is implemented as with meminterp$'$ but where the interpretation of fetch uses an indirection for the returned value:

```
meminterp''((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp''($M$, $\rho$, $\sigma$)
  in  case $\sigma_0(l_0)$
        of thunk($N$, $\rho'$) =>
             let ($l_2$, $\sigma_1$) = meminterp''($N$, $\rho'$, $\sigma_0$)
             in  ($l_0$, $\sigma_1[l_0 \mapsto @l_2]$)
         | _ => ($l_0$, $\sigma_0$)
```

where $@l_2$ is to be viewed as a boxed value. This is possibly closer to the way memoization would be implemented in most compilers. However, this approach complicates the approach to the LL-based semantics in the next section.

The implementation of memoization involves the idea of mutating a store. Even the 'functional' parts of the language must respect the potential side effects to the store that memoization may cause, and hence these operations must pass the store along in an appropriate manner. Doing this correctly may save recomputation. Here, for instance, is how the application operation is described:

```
meminterp(($M$ $N$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp($M$, $\rho$, $\sigma$)
      ($l_1$, $\sigma_1$) = meminterp($N$, $\rho$, $\sigma_0$)
  in  case $\sigma_1(l_0)$ of closure($\lambda x. N$, $\rho'$) =>
        meminterp($N$, $\rho'[x \mapsto l_1]$, $\sigma_1$)
```

The store resulting from evaluating M is used in evaluating N; similarly, the store resulting from evaluating N is used in evaluating the application.

The general approach we are taking to the form of this semantic definition is different in a significant way from the kind one finds elsewhere for the operational or denotational semantics of a language like this one. In particular, even the *immutable* values used in the semantics give rise to allocations in the store. For example, the semantics of lambda abstractions:

```
meminterp($\lambda x. P$, $\rho$, $\sigma$) = new(closure($\lambda x. P$, $\rho$), $\sigma$)
```

allocates a new location where a closure is held. In our treatment, this way of describing the meanings is crucial for the purposes of modelling sharing and copying of run-time entities.

There are a variety of ways to implement recursion. A reasonably efficient approach is to create a circular structure. This approach is simplified by restricting the interpreter to programs such that, in constructs of the form (fix N), the term N has the form $\lambda f. \lambda x. M$. The restriction is not necessary, but it is typical for call-by-value programming languages. The semantics for such recursions is given by

```
meminterp((fix $\lambda f. \lambda x. M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = new(0, $\sigma$)
      ($l_1$, $\sigma_1$) = new(recclosure($\lambda x. M$, $\rho[f \mapsto l_0]$), $\sigma_0$)
  in  ($l_0$, $\sigma_1[l_0 \mapsto$ rec($l_1$, $f$)])
```

which creates the circular structure in Figure 2. For this language we could create a single cell holding the recclosure that looped back to itself; we use two cells,

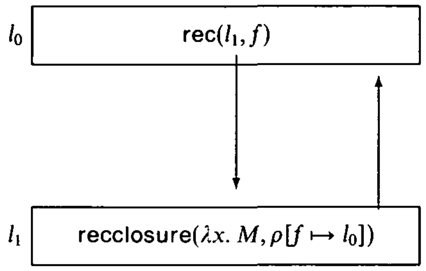


Fig. 2. Structure generated by (fix $\lambda f.\ \lambda x.\ M$).

though, since the additional cell holding rec will be used in the semantics of the LL-based language to facilitate connections with the type system. We also need here to change the semantics of applications so that if the operator evaluates to a rec, the pointer is traced to a recclosure; in turn, if the operator evaluates to a recclosure, the operator is used in the same way as a closure.

In the implementation of actual functional programming languages, a single recursion such as the one above would probably make its recursive calls through a jump instruction. This would be difficult to formalize with the source-code-based approach we are using to describe the interpreter. The important thing, for our purposes, is that recursive calls to f do not allocate further memory for the recursive closure. This means that, as far as memory is concerned, there is little difference between implementing the recursion with the jump and implementing it with a circular structure. The cycle created in this way introduces extra complexity into the structure of memory, of course, but the cycles must have precisely the form pictured in Figure 2.

3 Programming language based on linear logic

3.1 Term assignment for linear logic

If a programming language L is to be based on LL, it seems reasonable to attempt the completion of an analogy based on the Curry-Howard correspondence: intuitionistic logic is to traditional functional programming languages (such as ML or Haskell) as LL is to L. Basing a language on the Curry–Howard correspondence for LL immediately becomes problematic, as LL was originally described by Girard (1987) using a sequent calculus. Most programming languages have a syntax and typing system like the natural deduction (hereafter called 'ND') formulation of intuitionistic logic rather than its sequent calculus formulation, mostly because the ND formulation leads to a familiar syntax for application of functions. Progress on an ND form for intuitionistic LL has been gradual, in part because **substitutivity** fails for the obvious formulations:

Definition
A type system satisfies the **substitutivity** *property if well-typed programs are closed under substitution, i.e., if* $\Gamma \vdash N : t$ *and* $\Delta, x : t \vdash M : u$ *and all variables in* Γ *and* Δ *are distinct, then* $\Gamma, \Delta \vdash M[x := N] : u$.

Here $M[x := N]$ denotes substitution of N for x in M with the bound variables of M renamed to avoid capture of the free variables of N. SML (Milner *et al.*, 1990; Milner and Tofte, 1991) is a prototypical example of a language based on an ND presentation that satisfies the substitutivity property.

Merely coming up with a ND presentation of LL that satisfies substitutivity has been an outstanding problem. In the absence of such a system, Lincoln and Mitchell (1992), Mackie (1991), Wadler (1993), and the authors of this paper in a preceding work (Chirimar *et al.*, 1992) employed approaches that obtain some of the virtues of an ND system for LL. The system used in this paper is based on a proposal of Benton, Bierman, de Paiva, and Hyland (1992; 1993) that *does* satisfy the substitutivity property. We refer the reader to their paper for a fuller discussion.

The propositions of the fragment of linear logic we consider are given by the grammar

$$s ::= a \mid (s \multimap s) \mid !s$$

where a ranges over atomic propositions. The proofs of linear propositions are encoded by terms in the grammar

$$\begin{aligned} M ::= {} & x \mid (\lambda x : s.\, M) \mid (M\ M) \mid \\ & (\mathsf{store}\ M\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) \mid (\mathsf{fetch}\ M) \mid \\ & (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ M) \mid (\mathsf{dispose}\ M\ \mathsf{before}\ M). \end{aligned}$$

Our notation here essentially corresponds to that in Chirimar *et al.* (1992) and Lincoln and Mitchell (1992) modulo incorporating adjustments from Benton *et al.* (1992, 1993). The store operation,

$$(\mathsf{store}\ M\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n),$$

binds the variables $x_1, \ldots, x_n$ in the expression M and the share operation

$$(\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ N)$$

binds the variables x and y in N. The notation for store can be somewhat unwieldy when writing programs, but most programs involving store bind the variables in the where clause to other variables. Thus, if the free variables of M are $x_1, \ldots, x_n$, then $(\mathsf{store}\ M)$ is shorthand for the expression $(\mathsf{store}\ M\ \mathsf{where}\ x_1 = x_1, \ldots, x_n = x_n)$.

The typing rules for the language appear in Table 2, where the symbols Γ and Δ denote **type assignments**, 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. Each of the rules is built on the assumption that all left-hand sides of the $\vdash$ symbol are legal type assignments, e.g. in the rule for typing applications, the type assignments Γ and Δ, which appear concatenated together in the conclusion of the rule, must have disjoint variables. Each type-checking rule corresponds to a proof rule in the ND presentation of

Table 2. *Natural deduction rules and term assignment for linear logic.*

$$x : s \vdash x : s$$

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

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

$$\frac{\Gamma_1 \vdash M_1 : !s_1 \quad \dots \quad \Gamma_n \vdash M_n : !s_n \qquad x_1 : !s_1, \dots, x_n : !s_n \vdash N : t}{\Gamma_1, \dots, \Gamma_n \vdash (\textsf{store } N \textsf{ where } x_1 = M_1, \dots, x_n = M_n) : !t}$$

$$\frac{\Gamma \vdash M : !s}{\Gamma \vdash (\textsf{fetch } M) : s}$$

linear logic. For instance, the rules for share and dispose correspond to the proof rules generally called *contraction* and *weakening* respectively, while those for store and fetch correspond to the LL rules called *promotion* and *dereliction*. Due to the presence of explicit rules for weakening and contraction – the rules for type-checking dispose and share – one can easily see that the free variables of a well-typed term are *exactly* those contained in the type assignment. A particular note should be taken of the form of the rule for store; this operation puts the value of its body with bindings for its free variables in a location that can be shared by different terms during reduction – the type changes correspondingly from t to $!t$. The construct (fetch M) corresponds to reading the stored value – the type changes from $!t$ to t.

There may be other ND presentations of LL on which one could base a type system. It is our belief that results in this paper are robust with respect to the exact choice of term assignment and type-checking rules. All of the results in this paper – including negative results that say that values of linear type may have more than one pointer to them – hold in the system described in Chirimar *et al.* (1992), and we expect that they are true for the languages described in Lincoln and Mitchell (1992) and Mackie (1991).

3.2 *Programming language based on linear logic*

To fully realize the ideas of LL as the basis for a programming language, it is essential to go beyond the core language. First of all, the language could be extended to one that includes the linear logic connectives for pairing and sums, namely *tensor* $\otimes$, *plus* $\oplus$, and *with* &. Suitable ND proof rules for these connectives and term assignments for proofs using these rules are described in several places (Mackie, 1991; Lincoln and Mitchell, 1992; Benton *et al.*, 1992, 1993). A more challenging question is how to extend the language to include constructs for which the use of the Curry–Howard correspondence is less useful as a guide. Examples that fall in this category are arrays, general recursive datatypes involving linear implication, and recursive definitions of

Table 3. *Typing rules for non-logical constructs.*

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

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

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{zero?}\ M) : \mathsf{Bool}} \qquad \frac{\Gamma \vdash M : !(!s \multimap s)}{\Gamma \vdash (\mathsf{fix}\ M) : s}$$

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

functions. In this paper we treat only recursive function definitions; the question of the proper treatment of recursive definitions in an LL-based language is likely to be simpler than that of general recursive datatypes, and more fundamental than that of arrays.

Our language is essentially a synthesis of PCF (Plotkin, 1977; Scott, 1993) and the term language for encoding LL natural deduction proofs. The types are given by the following grammar:

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

Types without leading !'s, e.g., Nat and (Nat $\multimap$ Bool), are called **linear** and those of the form $!s$ are called **non-linear**. We use the letters s, t, u, and v to denote types. The set of raw terms in the language is given by the grammar

$$\begin{aligned} M ::= {} & x \mid (\lambda x : s.\, M) \mid (M\ M) \mid \\ & n \mid \mathsf{true} \mid \mathsf{false} \mid (\mathsf{succ}\ M) \mid (\mathsf{pred}\ M) \mid (\mathsf{zero?}\ M) \mid \\ & (\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M) \mid (\mathsf{fix}\ M) \mid \\ & (\mathsf{store}\ M\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) \mid (\mathsf{fetch}\ M) \mid \\ & (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ M) \mid (\mathsf{dispose}\ M\ \mathsf{before}\ M) \end{aligned}$$

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; the other term constructors are those of PCF. The usual definitions of free and bound variables for PCF also apply here for the first three lines of the grammar.

The typing rules for our language are given by combining Tables 2 and 3. Two of these rules deserve special explanation. First, the rule for checking the expression if L then M else N checks both branches in the same type assignment, i.e. the terms M and N must contain the same free variables. This is the only type-checking rule that allows variables to *appear* multiple times; it does not, however, violate the intuition that variables are *used* once, since only one branch will be taken during the execution of the program. (If we had the linear connective $\oplus$ in the language, a similar type-checking rule would be needed.) Second, the typing rule for

Table 4. *Interpreting the linear core.*

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

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

$$\frac{M_1 \Downarrow c_1 \quad \dots \quad M_n \Downarrow c_n}{(\text{store } N \text{ where } x_1 = M_1, \dots, x_n = M_n) \Downarrow (\text{store } N[x_1 := c_1, \dots, x_n := c_n])}$$

$$\frac{M \Downarrow (\text{store } N) \quad N \Downarrow c}{(\text{fetch } M) \Downarrow c}$$

recursions has a slightly mysterious form: the type of M must be a !'ed function whose argument itself must be of ! type. Both !'s are needed because of a simple observation: the formal parameter of a recursive definition must be share'd and dispose'd if there is to be anything interesting about it. Consider, for example, the rendering of the program of Table 1 into our language:

```
(fix (store (λadd : !(!Nat⊸Nat⊸Nat). λx : !Nat. λy : Nat.
   share w, z as x in
      if zero? (fetch w)
      then dispose z before dispose add before y
      else (fetch add) (store (pred (fetch z))) (succ y))))
(store 2) 1
```

(where some liberties have been taken in dropping a few of the parentheses to improve readability). The recursive function *add*, whose type is (!Nat⊸Nat⊸Nat), gets used only in one of the branches; thus, the recursive call must have a non-linear type, accounting for the second ! in the type of M. The first ! in the type of M is needed because we will interpret recursion via a cycle as in Figure 2.

The definition of the addition function is a prototypical example of how one programs recursive functions in this language. In fact, both the high-level and low-level semantics will only interpret recursions (fix M) where M has the form

$$(\text{store } (\lambda f : !s \multimap t.\, \lambda x : s.\, M) \text{ where } x_1 = M_1, \dots, x_n = M_n).$$

This restriction is closely connected to the restriction on interpreting recursion mentioned in the previous section; the only difference here is the occurrence of the store. As before, this restriction is not essential, but it does simplify the semantic clause for the recursion without compromising the way programs are generally written.

3.3 Natural semantics

Tables 4 and 5 give a high-level description of an interpreter for our language, written using natural semantics. A natural semantics describes a partial function $\Downarrow$ via proof trees. The notation $M \Downarrow c$, read 'the term M halts at the final result c',

Table 5. *Interpreting the PCF extensions.*

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

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

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

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

$$\frac{M_1 \Downarrow c_1 \quad \dots \quad M_n \Downarrow c_n \qquad M' \equiv M[x_1 := c_1, \dots, x_n := c_n]}{\begin{array}{c}(\mathsf{fix}\ (\mathsf{store}\ (\lambda f.\, \lambda x.\, M)\ \mathsf{where}\ x_1 = M_1, \dots, x_n = M_n)) \\ \Downarrow (\lambda x.\, M')[f := (\mathsf{store}\ (\mathsf{fix}\ (\mathsf{store}\ \lambda f.\, \lambda x.\, M')))]\end{array}}$$

is used when there is a proof from the rules with the conclusion being $M \Downarrow c$. The terms at which the interpreter function halts are called **canonical forms**; it is easy to see from the form of the rules that the canonical forms are n, true, false, $(\lambda x.\, M)$, and (store M).

The natural semantics in Tables 4 and 5 describes a *call-by-value* evaluation strategy. That is, operands in applications are evaluated to canonical form before the substitution takes place. A basic property of the semantics is that types are preserved under evaluation:

Theorem 1 (Subject reduction)
If $\vdash M : s$ *and* $M \Downarrow c$*, then* $\vdash c : s$.

The proof can be carried out by an easy induction on the height of the proof tree of $M \Downarrow c$.

4 Semantics

The high-level natural semantics is useful as a specification for an interpreter for our language, and for proving facts like Theorem 1. One would not want to implement the semantics directly, however: explicit substitution into terms can be expensive, and one would therefore use some standard representation of terms like closures or graphs in order to perform substitution more efficiently. But there is another problem with the high-level semantics: it does not go very far in providing a computational intuition for the LL primitives in the language. For example, the dispose operation is treated essentially as 'no-op'. As such, there is no apparent relationship between these connectives and memory; indeed, the semantics entirely suppresses the concept of memory.

In order to understand what the constructs of linear logic have to do with memory, we construct a semantics that relates the LL primitives to reference counting. In this semantics, the linear logic primitives dispose and share maintain reference counts. The basic structure of the reference-counting interpreter is the same as the one outlined earlier. Environments, values, and storable objects have the same definition as before. Because we now want to maintain reference counts, however, the definition of stores must change. A **store** is now a function

$$\sigma : \mathsf{Loc} \rightarrow (\mathsf{N} \times \mathsf{Storable}),$$

where the left part of the returned pair denotes a reference count. Abusing notation, we use $\sigma(l)$ to denote the storable object associated with location l, and $\sigma[l \mapsto S]$ to denote a new store which is the same as σ except at location l, which now holds the storable object S with the reference count of l left unaffected. The reference count of a cell is denoted by $\mathsf{refcount}(l, \sigma)$. The **domain** of a store σ is the set

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

The change in the definition of 'store' forces an adjustment in the definition of 'allocation relation'. 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 object S, there is an l' and σ' where $(S, \sigma)\ R\ (l', \sigma')$ and

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

The basic structure underlying a store may be captured abstractly by a graph. Formally, a **graph** is a tuple (V, E, s, t) where V and E are sets of **vertices** and **edges** respectively and s, t are functions from E to V called the **source** and **target** functions respectively. (Note that there may be more than one edge with the same source and target; such 'multiple edge' graphs are sometimes called *multigraphs.*) Given $v \in V$, the **in-degree** of v is the number of elements $e \in E$ such that $t(e) = v$. A vertex v is **reachable** from a vertex v' if $v = v'$ or there is a path between them, that is, there is a list of edges $e_1, \ldots, e_n$ such that $v' = s(e_1)$, $v = t(e_n)$ and $t(e_i) = s(e_{i+1})$ for each $i < n$.

A **memory graph** $\mathscr{G}$ is a tuple $(V, E, s, t, [\rho_1, \ldots, \rho_n])$ where (V, E, s, t) is a graph together with a list of functions ρ_i such that each ρ_i is a function whose domain is a finite subset of variables and whose codomain is V. The functions ρ_i are called the **root set** of the memory graph. Given $v \in V$ and ρ_i, let $|\rho_i^{-1}(v)|$ be the number of elements x in the domain of ρ_i such that $\rho_i(x) = v$. The **reference count** of a vertex $v \in V$ is the sum

$$\textbf{in-degree}(v) + \sum_{i=1}^{n} |\rho_i^{-1}(v)|.$$

A vertex in a memory graph is said to be reachable from ρ_i if it is reachable from an element in the range of ρ_i.

A **state** is a triple $(\bar{l}, \bar{\rho}, \sigma)$ where $\bar{l}$ is a list of locations, $\bar{\rho}$ is a list of environments and σ is a store. It is assumed that the set of locations in $\bar{l}$ and the range of each environment in $\bar{\rho}$ are contained in $\mathsf{dom}(\sigma)$.

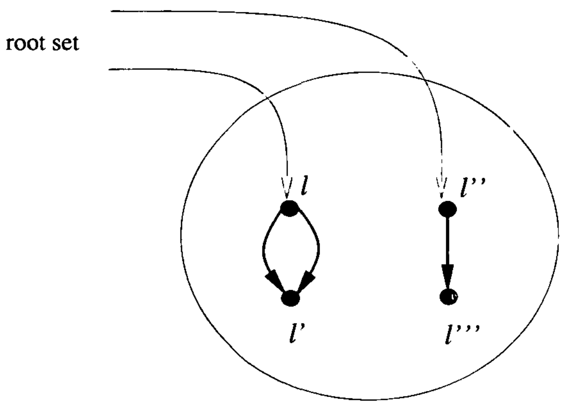


Fig. 3. A memory graph.

Definition
If $S = (\bar{l}, \bar{\rho}, \sigma)$ is a state where $\bar{l} = [l_1, \ldots, l_n]$ and $\bar{\rho} = [\rho_1, \ldots, \rho_m]$, then the **memory graph** $\mathscr{G}(S)$ **induced by** *S is defined as follows. The vertices of the graph are the locations in* $\mathsf{dom}(\sigma)$*, and the edges are determined by the following definition.*

- *If $l \in \mathsf{dom}(\sigma)$ is such that $\sigma(l) = \mathsf{susp}(l')$ or $\sigma(l) = \mathsf{rec}(l', f)$, there is an edge from l to l'.*
- *Suppose $l \in \mathsf{dom}(\sigma)$ is such that $\sigma(l) = \mathsf{closure}(N, \rho)$, $\mathsf{recclosure}(N, \rho)$ or $\mathsf{thunk}(N, \rho)$. Then for every $x \in \mathsf{dom}(\rho)$, there is an edge from l to $\rho(x)$.*

Let $f : \{x_1, \ldots, x_n\} \to V$ be given by $f : x_i \mapsto l_i$. The root set of the induced memory graph is represented by the list $[f, \rho_1, \ldots, \rho_m]$.

For instance, (l, ρ, σ) where $\mathsf{dom}(\rho) = \{x\}$, $\rho(x) = l''$, $\sigma(l) = \mathsf{thunk}(M, [y, z \mapsto l'])$, $\sigma(l') = 3$, $\sigma(l'') = \mathsf{susp}(l''')$, and $\sigma(l''') = \mathsf{true}$ induces the memory graph in Figure 3. We will abuse notation and sometimes write $\mathscr{G}(\sigma)$ for the graph induced by σ alone (with no root set).

We are primarily concerned with states that satisfy a collection of basic invariants.

Definition
A state $S = (\bar{l}, \bar{\rho}, \sigma)$ is **count-correct** *if, for each $l \in \mathsf{dom}(\sigma)$, $\mathsf{refcount}(l, \sigma)$ is equal to the reference count of l in $\mathscr{G}(S)$.*

Definition
A state $S = (\bar{l}, \bar{\rho}, \sigma)$ is called **regular**, *written $\mathfrak{R}(S)$, provided the following conditions hold:*

$\mathfrak{R}1$ *S is count-correct.*
$\mathfrak{R}2$ $\mathsf{dom}(\sigma)$ *is finite.*
$\mathfrak{R}3$ *For each $l \in \mathsf{dom}(\sigma)$, if $\sigma(l) = \mathsf{thunk}(M, \rho)$, then $\mathsf{refcount}(l, \sigma) = 1$.*
$\mathfrak{R}4$ *A cycle in the memory graph induced by S arises only in the form of a* rec *and* recclosure *as in Figure 2: that is, it has two nodes l_0 and l_1 such that $\sigma(l_0) = \mathsf{rec}(l_1, f)$ and $\sigma(l_1) = \mathsf{recclosure}(\lambda x.\, M, \rho[f \mapsto l_0])$ for some f, x, M, and ρ.*

$\mathfrak{R}5$ *For each* $l \in \mathsf{dom}(\sigma)$, *if* $\sigma(l) = \mathsf{thunk}(M, \rho)$, *then the domain of* ρ *is the set of free variables of* M, *and* M *is typable. Similarly, if* $\sigma(l) = \mathsf{closure}(\lambda x.\, M, \rho)$ *or* $\mathsf{recclosure}(\lambda x.\, M, \rho)$, *then the domain of* ρ *is the set of free variables of* $\lambda x.\, M$, *and* $\lambda x.\, M$ *is typable.*

$\mathfrak{R}6$ *For each* l *in the list* $\bar{l}$, $\sigma(l)$ *is not a thunk. For each* ρ *in the list* $\bar{\rho}$ *and* $x \in \mathsf{dom}(\rho)$, $\sigma(\rho(x))$ *is not a thunk. Finally, if* $\sigma(l) = \mathsf{closure}(N, \rho)$, $\mathsf{recclosure}(N, \rho)$, *or* $\mathsf{thunk}(N, \rho)$ *and* $x \in \mathsf{dom}(\rho)$, $\sigma(\rho(x))$ *is not a thunk.*

Here, a term M is said to be **typable** if there is some type context Γ and type t such that $\Gamma \vdash M : t$. Most of the invariants are self-explanatory with the possible exceptions of $\mathfrak{R}3$ and $\mathfrak{R}4$. The invariant $\mathfrak{R}3$ allows us to perform an 'in-place update' when evaluating a thunk, and thus allows for some simplification of the operational semantics. The invariant $\mathfrak{R}4$ restricts the kind of cycles that may appear in the store to be only those created by evaluating recursive function definitions. Arbitrary cycles cause a problem in the definition of some of our primitives for reference-counting (*cf.* dec-ptrs below), but fortunately the functional character of the language prevents the programmer from creating arbitrary cycles.

It is convenient to abuse notation slightly in denoting states by writing locations, environments, and store without grouping them as in the official definition. For example, $(l_1, l_2, \rho, \sigma, \bar{l}, \bar{\rho})$ should be read as $(l_1 :: l_2 :: \bar{l}, \rho :: \bar{\rho}, \sigma)$ (where :: is the 'cons' operation that puts a datum at the head of a list). There is no chance of confusion so long as the lexical conventions distinguish the parts of the tuple, and the locations and environments are properly ordered from left to right. However, the order of these lists is irrelevant for regularity: if $\mathfrak{R}(\bar{l}, \bar{\rho}, \sigma)$ and $\bar{l}', \bar{\rho}'$ are permutations of $\bar{l}$ and $\bar{\rho}$ respectively, then $\mathfrak{R}(\bar{l}', \bar{\rho}', \sigma)$. We will use this fact without explicit mention.

4.1 Basic reference-counting operations

Our interpreter will need four auxiliary functions to manipulate reference counts. Two of these functions, inc and dec, increment and decrement reference counts. More formally, $\mathsf{inc}(l, \sigma)$ increments the reference count of l and returns the resultant store, while $\mathsf{dec}(l, \sigma)$ decrements the reference count of l and returns the resultant store. The other two operations, $\mathsf{inc\text{-}env}(\rho, \sigma)$ and $\mathsf{dec\text{-}ptrs}(l, \sigma)$, increment or decrement the reference counts of multiple cells. The formal definition of the first of these is

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

In words, $\mathsf{inc\text{-}env}(\rho, \sigma)$ increments the reference counts of the locations in the range of ρ and returns the resultant store. Note that a location's reference count may be incremented more than once by this operation, since two variables x_i, x_j may map to the same location l according to ρ.

The operation $\mathsf{dec\text{-}ptrs}(l, \sigma)$, which also returns an updated store, first decrements the reference count of location l. If the reference count falls to zero, it then recursively

Table 6. *The Definition of* dec-ptrs.

$$
\mathsf{dec\text{-}ptrs}(l,\sigma) = \begin{cases}
\mathsf{dec}(l,\sigma) & \text{if } \sigma(l) = n, \text{ true, or false} \\
\mathsf{dec\text{-}ptrs}(l',\mathsf{dec}(l,\sigma)) & \text{if } \sigma(l) = \mathsf{susp}(l'),\ \mathsf{refcount}(l,\sigma) = 1 \\
\mathsf{dec\text{-}ptrs\text{-}env}(\rho,\mathsf{dec}(l,\sigma)) & \text{if } \sigma(l) = \mathsf{thunk}(M,\rho),\ \mathsf{refcount}(l,\sigma) = 1 \\
\mathsf{dec\text{-}ptrs\text{-}env}(\rho,\mathsf{dec}(l,\sigma)) & \text{if } \sigma(l) = \mathsf{closure}(\lambda x.\, M,\rho), \\
 & \quad \mathsf{refcount}(l,\sigma) = 1 \\
\mathsf{dec\text{-}ptrs\text{-}env}(\rho - f, & \text{if } \sigma(l) = \mathsf{recclosure}(\lambda x.\, N,\rho), \\
\quad \mathsf{dec}(l',\mathsf{dec}(l,\mathsf{dec}(l,\sigma)))) & \quad \sigma(l') = \mathsf{rec}(l,f), \\
 & \quad \mathsf{refcount}(l,\sigma) = 2,\ \mathsf{refcount}(l',\sigma) = 1 \\
\mathsf{dec\text{-}ptrs\text{-}env}(\rho - f, & \text{if } \sigma(l) = \mathsf{rec}(l',f), \\
\quad \mathsf{dec}(l',\mathsf{dec}(l,\mathsf{dec}(l,\sigma)))) & \quad \sigma(l') = \mathsf{recclosure}(\lambda x.\, N,\rho), \\
 & \quad \mathsf{refcount}(l,\sigma) = 2,\ \mathsf{refcount}(l',\sigma) = 1 \\
\mathsf{dec}(l,\sigma) & \text{otherwise}
\end{cases}
$$

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

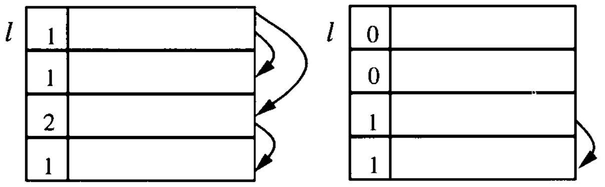


Fig. 4. An example of the dec-ptrs operation.

decrements the reference counts of all cells pointed to by l. The formal definition appears in Table 6; an example appears in Figure 4 where the left side of Figure 4 (assumed to be part of the graph of the store σ) is transformed into the right side by calling $\mathsf{dec\text{-}ptrs}(l,\sigma)$. In the definition, $\rho \mid P$ is the restriction of the environment ρ to the free variables of P, and $\rho - f$ is the environment with domain $\mathsf{dom}(\rho) - \{f\}$ such that $\rho(x) = (\rho - f)(x)$ for all $x \in \mathsf{dom}(\rho) - \{f\}$. The operation $\mathsf{dec\text{-}ptrs}(l,\sigma)$ is the single most complex operation used in the interpreter. Other operations are 'local' to parts of the memory graph and do not require a recursive definition. A key characteristic of our semantics is the fact that $\mathsf{dec\text{-}ptrs}(l,\sigma)$ is only used in the rule for evaluating ($\mathsf{dispose}$ M before N).

The basic laws that capture the relationships maintained by the allocation, reference-counting, and update operations on states are given in Table 7. Most of the laws are proven in the appendix, but we give the proof for the Attenuation Law A1 here to show how the proofs go. Suppose $\mathfrak{R}(l,\bar{l},\bar{\rho},\sigma)$, $\mathsf{refcount}(l,\sigma) = 1$ and $\sigma(l) = \mathsf{closure}(N,\rho)$, $\mathsf{recclosure}(N,\rho)$, or $\mathsf{thunk}(N,\rho)$. Note first that the state $S' = (\bar{l},\rho,\bar{\rho},\mathsf{dec}(l,\sigma))$ is count-correct: the environment ρ has been placed in the root set, accounting for the edges coming out of the closure or thunk which has now disappeared from the memory graph. Thus, property $\mathfrak{R}1$ holds of state S'. Since $\mathsf{dom}(\sigma) \supseteq \mathsf{dom}(\mathsf{dec}(l,\sigma))$, each of the properties $\mathfrak{R}2$–$\mathfrak{R}5$ follow directly from the

Table 7. *Memory graph laws.*

Attenuation Laws Suppose $\mathfrak{R}(l,\bar{l},\bar{\rho},\sigma)$ and $\mathsf{refcount}(l,\sigma)=1$.

A1 If $\sigma(l)=\mathsf{closure}(N,\rho)$, $\mathsf{recclosure}(N,\rho)$, or $\mathsf{thunk}(N,\rho)$, then $\mathfrak{R}(\bar{l},\rho,\bar{\rho},\mathsf{dec}(l,\sigma))$.

A2 If $\sigma(l)=\mathsf{susp}(l')$ and $\sigma(l')=\mathsf{thunk}(N,\rho)$, then $\mathfrak{R}(\bar{l},\rho,\bar{\rho},\mathsf{dec}(l',\mathsf{dec}(l,\sigma)))$.

Laws of Decrement

D1 If $\mathfrak{R}(l,\bar{l},\bar{\rho},\sigma)$ and $\sigma(l)$ is a constant, then $\mathfrak{R}(\bar{l},\bar{\rho},\mathsf{dec}(l,\sigma))$.

D2 If $\mathfrak{R}(l,\bar{l},\bar{\rho},\sigma)$ and $\mathsf{refcount}(l,\sigma)\neq 1$, then $\mathfrak{R}(\bar{l},\bar{\rho},\mathsf{dec}(l,\sigma))$.

D3 If $\mathfrak{R}(l,\bar{l},\bar{\rho},\sigma)$, then $\mathfrak{R}(\bar{l},\bar{\rho},\mathsf{dec\text{-}ptrs}(l,\sigma))$.

Laws of Increment

I1 If $\mathfrak{R}(\bar{l},\bar{\rho},\sigma)$, $l\in\mathsf{dom}(\sigma)$, and $\sigma(l)$ is not a thunk, then $\mathfrak{R}(l,\bar{l},\bar{\rho},\mathsf{inc}(l,\sigma))$.

I2 If $\mathfrak{R}(\bar{l},\bar{\rho},\sigma)$, $\rho(x)\in\mathsf{dom}(\sigma)$ for all $x\in\mathsf{dom}(\rho)$, and each $\sigma(\rho(x))$ is not a thunk, then $\mathfrak{R}(\bar{l},\rho,\bar{\rho},\mathsf{inc\text{-}env}(\rho,\sigma))$.

Environment Law

E Suppose $x\notin\mathsf{dom}(\rho)$. Then $\mathfrak{R}(l,\bar{l},\rho,\bar{\rho},\sigma)$ iff $\mathfrak{R}(\bar{l},\rho[x\mapsto l],\bar{\rho},\sigma)$.

Allocation Laws

N1 If $\mathfrak{R}(\bar{l},\bar{\rho},\sigma)$ and $(l',\sigma')=\mathsf{new}(c,\sigma)$ for some constant c, then $\mathfrak{R}(l',\bar{l},\bar{\rho},\sigma')$.

N2 If N is typable, $\mathfrak{R}(\bar{l},\rho,\bar{\rho},\sigma)$, and (l',σ') is equal to $\mathsf{new}(\mathsf{closure}(N,\rho),\sigma)$ or $\mathsf{new}(\mathsf{recclosure}(N,\rho),\sigma)$ where $FV(N)=\mathsf{dom}(\rho)$, then $\mathfrak{R}(l',\bar{l},\bar{\rho},\sigma')$.

N3 If N is typable, $\mathfrak{R}(\bar{l},\rho,\bar{\rho},\sigma)$, $(l_0,\sigma_0)=\mathsf{new}(\mathsf{thunk}(N,\rho),\sigma)$ where $FV(N)=\mathsf{dom}(\rho)$, and $(l',\sigma')=\mathsf{new}(\mathsf{susp}(l_0),\sigma_0)$, then $\mathfrak{R}(l',\bar{l},\bar{\rho},\sigma')$.

Update Laws

U1 Suppose $S=(l',\bar{l},\bar{\rho},\sigma)$ and $\mathfrak{R}(S)$ and $\sigma(l)$ is a constant. If $\sigma(l')=\mathsf{recclosure}(\lambda x.\ N,\rho[f\mapsto l])$, then $\mathfrak{R}(l',\bar{l},\bar{\rho},\mathsf{inc}(l',\sigma[l\mapsto\mathsf{rec}(l',f)]))$. If l is not reachable from l' in $\mathscr{G}(S)$, then $\mathfrak{R}(l',\bar{l},\bar{\rho},\mathsf{inc}(l',\sigma[l\mapsto\mathsf{susp}(l')]))$.

U2 If $\mathfrak{R}(l,\bar{l},\bar{\rho},\sigma)$, $\mathsf{refcount}(l,\sigma)\neq 1$, $\sigma(l)=\mathsf{susp}(l')$, and $\sigma(l')=\mathsf{thunk}(N,\rho)$, then $\mathfrak{R}(\rho,\bar{l},\bar{\rho},\mathsf{dec}(l',\mathsf{dec}(l,\sigma[l\mapsto c])))$.

hypothesis. Finally, $\mathfrak{R}6$ holds because, by hypothesis, for any $x\in\mathsf{dom}(\rho)$, $\sigma(\rho(x))$ is not a thunk. Thus, $\mathfrak{R}(S')$. The property is called an 'attenuation law' because pointers previously held inside the store are drawn out to the root set.

The next goal is to define an interpreter for the LL-based programming language. To understand the interpreter it is essential to appreciate how the invariants influence its design. We therefore describe the theorem that the interpreter is expected to satisfy, and mingle the proof of the theorem with the definition of the interpreter itself. The interpreter is a function `interp` which takes as its arguments a term M, an environment ρ, and a store σ. It is assumed that the domain of ρ is the set of free variables in M and that the range of ρ is contained in the domain of σ. The result of $\texttt{interp}(M,\rho,\sigma)$ is a pair (l',σ') where σ' is a store and l' is a location in the domain of σ' such that $\sigma'(l')$ is a value, which can be viewed as the result of the computation. We use a binary infix @ for appending two lists. The theorem is stated as follows:

Theorem 2

Let $S=(\rho,\sigma,\bar{l},\bar{\rho})$ *be a state and suppose* M *is a typable term. If* $\mathfrak{R}(S)$ *and* $\texttt{interp}(M,\rho,\sigma)=(l',\sigma')$, *then* $\mathfrak{R}(l',\sigma',\bar{l},\bar{\rho})$.

Moreover, if $\bar{\rho}=\bar{\rho}_1@\bar{\rho}_2$, $\bar{l}=\bar{l}_1@\bar{l}_2$ *and* $l\in\mathsf{dom}(\sigma)$ *is not reachable from* $\rho::\bar{\rho}_1$ *or* $\bar{l}_1$ *in the memory graph induced by* S, *then the contents and reference count of* l *remain* *unchanged and l is not reachable from $\bar{\rho}_1$ or $l' :: \bar{l}_1$ in the memory graph induced by $(l', \sigma', \bar{l}, \bar{\rho})$.*

The first part of the theorem says that regularity is preserved under execution of typable terms. The second part of the theorem expresses what we will call the **reachability property.** The special case of interest says that the evaluation of a program M in environment ρ and store σ does not affect locations in $\mathsf{dom}(\sigma)$ that are not reachable from ρ. The extra complexity of the statement is required to maintain a usable inductive hypothesis in the proof of the property. A simplified version of Theorem 2 can be expressed as follows:

Corollary 3
Suppose M is a closed, typable term. If $\texttt{interp}(M, \emptyset, \emptyset) = (l', \sigma')$*, then* $\Re(l', \sigma')$.

The assumption that M is typable is crucial in the proof of the theorem, because untypable terms may not maintain reference counts correctly. For instance, the term

$$(\lambda x.\,(\mathsf{dispose}\ x\ \mathsf{before}\ x))\ (\mathsf{store}\ 1)$$

would cause a run-time error in the maintenance of reference counts – after the dispose, we would try to access a portion of memory with reference count zero and get a 'dangling pointer' error. This example shows that untypable terms may cause premature deallocations. Another untypable term

$$(\lambda x.\,(\mathsf{share}\ y, z\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{dispose}\ y\ \mathsf{before}\ 2)))\ (\mathsf{store}\ 1)$$

causes a 'space leak', i.e. the reference count of the cell holding (store 1) is still greater than zero even though it is garbage at the end of the execution.

It is also worth noting, however, that 'typability' is a sufficient condition and not a *necessary* condition. One can invent a 'syntactic linearity' condition from the typing rules by simply considering only one type U and focusing only on variables. The typing rule for application, for instance, would be

$$\frac{\Gamma \vdash M : U \qquad \Delta \vdash N : U}{\Gamma, \Delta \vdash (M\ N) : U}$$

and the term

$$\lambda x : U.\,(\mathsf{share}\ u, v\ \mathsf{as}\ x\ \mathsf{in}\ (u\ v))$$

would be typable. Of course, the resultant language would not be free from run-time type errors, but one could prove Theorem 2 using this notion of 'syntactic linearity'. The advantage of the LL-based type system over 'syntactic linearity' comes in the avoidance of run-time type errors. Nevertheless, it may interesting to study 'syntactic linearity' in its own right.

4.2 *Interpreting the linear core*

The proof of Theorem 2 is by induction on the number of calls to the interpreter. The proof proceeds by considering each case for the program to be evaluated.

The interpretation of a variable is obtained by looking up the variable in the environment:

(1) $\texttt{interp}(x,\ \rho,\ \sigma) = (\rho(x),\ \sigma)$

That the store $(\rho(x), \sigma', \bar{l}, \bar{\rho})$ is regular is a consequence of the Environment Law E because of the assumption that the domain of ρ is $\{x\}$. The reachability property is clearly satisfied, since the output store is the same as the input store.

To evaluate an abstraction we create a new closure, place it in a new cell, and return the location together with the updated store:

(2) $\texttt{interp}(\lambda x.\, P,\ \rho,\ \sigma) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P,\ \rho),\ \sigma)$

To show regularity of the new state, suppose $(l', \sigma') = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho), \sigma)$; then $\mathfrak{R}(l', \sigma', \bar{l}, \bar{\rho})$ by Allocation Law N2. The reachability property is satisfied because the output store differs from the input store only by extending it.

The evaluation of an application is given as follows:

```
(3)     interp((P Q), ρ, σ) =
          let (l₀, σ₀) = interp(P, ρ | P, σ)
              (l₁, σ₁) = interp(Q, ρ | Q, σ₀)
          in  case σ₁(l₀) of closure(λx. N, ρ′) or recclosure(λx. N, ρ′) =>
                if refcount(l₀, σ₁) = 1
                then interp(N, ρ′[x ↦ l₁], dec(l₀, σ₁))
                else interp(N, ρ′[x ↦ l₁], inc-env(ρ′, dec(l₀, σ₁)))
```

The reader may compare this rule to the rule for application given above. The key difference in the semantic clauses is the manipulation of reference counts: in the rule here, a conditional breaks the evaluation of the function body into two cases based on the reference count of the location that holds the value of the operator, and each branch of the conditional performs some reference-counting arithmetic. The resulting semantic clause looks similar to a denotational semantics such as that given in Hudak (1987), where information about reference counts is included in the semantic clauses. Note that the environment ρ has been split between the two subterms P and Q. The fact that $(P\ Q)$ is typable implies that $\rho = (\rho \mid P) \cup (\rho \mid Q)$. In various forms this sort of property will be used repeatedly in the semantic clauses.

To prove the preservation of regularity of the state for application, we start with the assumption that $\mathfrak{R}(\rho, \sigma, \bar{l}, \bar{\rho})$. This is equivalent to $\mathfrak{R}(\rho \mid P, \rho \mid Q, \sigma, \bar{l}, \bar{\rho})$. Now $\mathfrak{R}(l_0, \rho \mid Q, \sigma_0, \bar{l}, \bar{\rho})$ and $\mathfrak{R}(l_1, l_0, \sigma_1, \bar{l}, \bar{\rho})$ both hold by inductive hypothesis (let us abbreviate 'inductive hypothesis' as 'IH'). Now, if $\mathsf{refcount}(l_0, \sigma_1) = 1$, then $\mathfrak{R}(\rho', l_1, \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$ holds by A1, so $\mathfrak{R}(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$ holds by E. Hence regularity follows from IH. If, on the other hand, $\mathsf{refcount}(l_0, \sigma_1) \neq 1$, then $\mathfrak{R}(\rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1)), \bar{l}, \bar{\rho})$ follows from D2, I2, and E, so we are done by IH.

To see that the reachability property holds for the interpretation of application, suppose $l \in \mathsf{dom}(\sigma)$ is unreachable from $\rho :: \bar{\rho}_1$ where $\bar{\rho} = \bar{\rho}_1 @ \bar{\rho}_2$ and unreachable from $\bar{l}_1$ where $\bar{l} = \bar{l}_1 @ \bar{l}_2$. If l is unreachable from $(\bar{l}_1, \rho :: \bar{\rho}_1)$, then it is unreachable from $(\bar{l}_1, (\rho \mid P) :: (\rho \mid Q) :: \bar{\rho}_1)$, so, by IH, it is unreachable from $(l_0 :: \bar{l}_1, (\rho \mid Q) :: \bar{\rho}_1)$ in the memory graph induced by the state resulting from the evaluation of P. A second application of IH allows us to conclude that it is also unreachable from $(l_1 :: l_0 :: \bar{l}_1, \bar{\rho}_1)$ in the memory graph induced by $(l_1, l_0, \sigma_1, \bar{l}, \bar{\rho})$. By the definition of the memory graph, this implies that l is unreachable from ρ' as well, so it is unreachable from $(\bar{l}_1, \rho'[x \mapsto l_1], \bar{\rho}_1)$ in the memory graphs induced by the states $(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$ and $(\rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1)), \bar{l}, \bar{\rho})$. The desired conclusion therefore follows from IH. The proof of the reachability is similar for all of the remaining cases, so we will omit arguing it in the rest of the discussion.

The expression (store N where $x_1 = M_1, \ldots, x_n = M_n$) is interpreted by first evaluating the terms $M_1, \ldots, M_n$ to locations $l_1, \ldots, l_n$, building an environment that maps x_i to l_i for all i, creating a thunk out of this environment and N, and finally returning a location holding a suspension of this thunk:

```
(4)     interp((store N where $x_1 = M_1, \ldots, x_n = M_n$), $\rho$, $\sigma$) =
          let ($l_1$, $\sigma_1$) = interp($M_1$, $\rho | M_1$, $\sigma$)
                ⋮
              ($l_n$, $\sigma_n$) = interp($M_n$, $\rho | M_n$, $\sigma_{n-1}$)
              $\rho'$  =  $[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n]$
              ($l_{n+1}$, $\sigma_{n+1}$) = new(thunk($N$, $\rho'$), $\sigma_n$)
          in  new(susp($l_{n+1}$), $\sigma_{n+1}$)
```

To prove that the desired property is maintained, note that repeated application of the inductive hypothesis allows us to conclude that $\mathfrak{R}(\rho', \sigma_n, \bar{\rho}, \bar{l})$. Let $(l_{n+2}, \sigma_{n+2}) = \mathsf{new}(\mathsf{susp}(l_{n+1}), \sigma_{n+1})$. Then $\mathfrak{R}(l_{n+2}, \sigma_{n+2}, \bar{\rho}, \bar{l})$ by N3.

The fetch of a suspended object is the most complex of all the operations. It must evaluate a thunk if the suspension holds one. The code is again similar to that for the interpreter we examined earlier, but, in addition to the reference-counting arithmetic, there is a clause dealing with recursion:

```
(5)     interp((fetch P), $\rho$, $\sigma$) =
          let ($l_0$, $\sigma_0$) = interp($P$, $\rho$, $\sigma$)
          in  case $\sigma_0(l_0)$
                of susp($l_1$) =>
                   case $\sigma_0(l_1)$
                     of thunk($R$, $\rho'$) =>
                          if refcount($l_0$, $\sigma_0$) = 1
                          then interp($R$, $\rho'$, dec($l_1$, dec($l_0$, $\sigma_0$)))
                          else let ($l_2$, $\sigma_1$) =
                                     interp($R$, $\rho'$, dec($l_1$, dec($l_0$, $\sigma_0[l_0 \mapsto 0]$)))
                               in  ($l_2$, inc($l_2$, $\sigma_1[l_0 \mapsto$ susp($l_2$)])))
                      _ => if refcount($l_0$, $\sigma_0$) = 1
                           then ($l_1$, dec($l_0$, $\sigma_0$))
                           else ($l_1$, inc($l_1$, dec($l_0$, $\sigma_0$)))
                 | rec($l_1$, $f$) => ($l_1$, dec($l_0$, inc($l_1$, $\sigma_0$)))
```

By IH, we have $\mathfrak{R}(l_0, \sigma_0, \bar{l}, \bar{\rho})$. Suppose $\sigma_0(l_0) = \mathsf{susp}(l_1)$ and $\sigma_0(l_1) = \mathsf{thunk}(R, \rho')$. If $\mathsf{refcount}(l_0, \sigma_0) = 1$, then $\mathfrak{R}(\rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0)), \bar{l}, \bar{\rho})$ by A2 so we are done by IH. Otherwise, $\mathsf{refcount}(l_0, \sigma_0) \neq 1$, so by U2, $\mathfrak{R}(\rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])), \bar{l}, \bar{\rho})$ and so $\mathfrak{R}(l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]), \bar{l}, \bar{\rho})$ by IH and U1; the reachability property is used to ensure the applicability of U1. More specifically for $\mathfrak{R}4$, in σ_0 the location l_0 is not reachable from ρ' (because there are no loops of this form in a regular state); thus, it is not reachable from l_2 in σ_1 either, and so $\sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]$ does not create an illegal loop in the memory graph. The cases when $\sigma_0(l_1)$ is a value or $\sigma_0(l_0) = \mathsf{rec}(l_1, f)$ are left to the reader.

The share command increments the reference count of a location:

```
(6)     interp((share $x, y$ as $P$ in $Q$),  $\rho$,  $\sigma$) =
          let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho \mid P$,  $\sigma$)
          in   interp($Q$,  $(\rho \mid Q)[x, y \mapsto l_0]$,  inc($l_0$,  $\sigma_0$))
```

Note first that $\Re(l_0, \rho \mid Q, \sigma_0, \bar{l}, \bar{\rho})$ by IH. By $\Re 6$, $\sigma(l_0)$ is not a thunk, so it follows from I1 that $\Re(l_0, l_0, \rho \mid Q, \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$. By the Environment Law E, it follows that $\Re((\rho \mid Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$, so the result follows from IH.

The dispose command decrements the reference count of a location. This requires calculating the consequences of possibly removing a node from the memory graph if the reference count of the disposed node falls to 0.

```
(7)     interp((dispose $P$ before $Q$),  $\rho$,  $\sigma$) =
          let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho \mid P$,  $\sigma$)
          in   interp($Q$,  $\rho \mid Q$,  dec-ptrs($l_0$,  $\sigma_0$))
```

Now, $\Re(l_0, \rho \mid Q, \sigma_0, \bar{l}, \bar{\rho})$ by IH, so $\Re(\rho \mid Q, \mathsf{dec\text{-}ptrs}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ by D3. The result therefore follows from IH.

4.3 Interpreting PCF extensions

The interpreter evaluates a constant simply by creating a cell holding the value of the constant.

```
(8)     interp($n, \rho, \sigma$) = new($n, \sigma$)

(9)     interp(true, $\rho, \sigma$) = new(true, $\sigma$)

(10)    interp(false, $\rho, \sigma$) = new(false, $\sigma$)
```

That regularity is preserved for these cases follows immediately from N1.

The rules for the arithmetic and boolean operations of PCF mimic the rules of the high-level operational semantics.

```
(11)    interp((succ $P$),  $\rho$,  $\sigma$) =
          let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
          in   new($\sigma_0(l_0) + 1$,  dec($l_0$,  $\sigma_0$))

(12)    interp((pred $P$),  $\rho$,  $\sigma$) =
          let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
               $n = \sigma_0(l_0)$
          in   if $n = 0$
               then new(0,  dec($l_0$,  $\sigma_0$))
               else new($n - 1$,  dec($l_0$,  $\sigma_0$))

(13)    interp((zero? $P$),  $\rho$,  $\sigma$) =
          let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
          in if $\sigma_0(l_0) = 0$
             then new(true,  dec($l_0$,  $\sigma_0$))
             else new(false,  dec($l_0$,  $\sigma_0$))
```

To prove the desired property for the successor operation, note that $\mathfrak{R}(l_0, \sigma_0, \bar{l}, \bar{\rho})$ follows from IH so we are done by D1 and N1. Proofs for the other two cases are similar.

The conditional statement has the expected form, but the reference count of the condition must be decremented in each of the branches:

```
(14)    interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
          let ($l_0$, $\sigma_0$) = interp($N$, $\rho | N$, $\sigma$)
          in  if $\sigma_0(l_0)$ = true
              then interp($P$, $\rho | P$, dec($l_0$, $\sigma_0$))
              else interp($Q$, $\rho | Q$, dec($l_0$, $\sigma_0$))
```

The IH implies $\mathfrak{R}(l_0, \sigma_0, \bar{l}, \bar{\rho})$. Whether or not $\sigma_0(l_0)$ = true, the desired conclusion follows from D1.

Finally, to interpret recursion, we will need a rule similar to the rule for interpreting store.

```
(15)    interp((fix (store ($\lambda f. \lambda x. M$) where $x_1 = M_1, \dots, x_n = M_n$)), $\rho$, $\sigma$) =
          let ($l_1$, $\sigma_1$) = interp($M_1$, $\rho | M_1$, $\sigma$)
              ⋮
              ($l_n$, $\sigma_n$) = interp($M_n$, $\rho | M_n$, $\sigma_{n-1}$)
              $\rho'$  =  [$x_1, \dots, x_n \mapsto l_1, \dots, l_n$]
              ($l_{n+1}$, $\sigma_{n+1}$) = new(0, $\sigma_n$)
              ($l_{n+2}$, $\sigma_{n+2}$) = new(recclosure($\lambda x. M$, $\rho'[f \mapsto l_{n+1}]$), $\sigma_{n+1}$)
          in  ($l_{n+2}$, inc($l_{n+2}$, $\sigma_{n+2}[l_{n+1} \mapsto$ rec($l_{n+2}$, $f$)]))
```

As with the interpretation of store, repeated application of the IH and E implies that $\mathfrak{R}(\rho', \sigma_n, \bar{l}, \bar{\rho})$. By N1, E, and N2, we therefore also have $\mathfrak{R}(l_{n+2}, \sigma_{n+2}, \bar{l}, \bar{\rho})$. The desired conclusion now follows from U1.

5 Properties of the semantics

In order for the reference-counting interpreter to make sense, it must satisfy a number of invariants and correctness criteria. In this section we describe these precisely.

5.1 No space leaks

As a short example of the kind of property one expects the semantics to satisfy, let us consider how the idea that 'there are no space leaks' can be expressed in our formalism. Given a state $S = (\bar{l}, \bar{\rho}, \sigma)$, we say that a location l is reachable from $(\bar{l}, \bar{\rho})$ if it is reachable in $\mathscr{G}(S)$ from some $l_i \in \bar{l}$ or from some $\rho_j \in \bar{\rho}$. The desired property can now be expressed as follows:

Theorem 4
Suppose $(\rho, \sigma, \bar{l}, \bar{\rho})$ *is a regular state such that each* $l \in \mathrm{dom}(\sigma)$ *is reachable from* $(\rho, \bar{l}, \bar{\rho})$. *If* M *is typable and* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, *then every* $l \in \mathrm{dom}(\sigma')$ *is reachable from* $(l', \bar{l}, \bar{\rho})$.

The theorem is proved by induction on the number of calls to the interpreter.

5.2 Invariance under different allocation relations

If the design of the interpreter is correct, the exact memory usage pattern should be unimportant to the final answers returned by the interpreter. Since the allocation relation new completely determines memory usage (i.e. which cell (with reference count 0) will be filled next), it should not matter which allocation relation is used. We set this up formally as follows: if f is an allocation relation, let $\texttt{interp}_f$ be the partial interpreter function defined by using f in the place of new. Recall that the environment and store with empty domains are denoted by $\emptyset$. Then we would like to prove something like the following statement by induction on the number of calls to $\texttt{interp}_f$:

If $\texttt{interp}_f(M,\emptyset,\emptyset) = (l_f,\sigma_f)$, then $\texttt{interp}_g(M,\emptyset,\emptyset) = (l_g,\sigma_g)$ for some l_g,σ_g. Moreover, if $\sigma_f(l_f) = n$, true, or false, then $\sigma_f(l_f) = \sigma_g(l_g)$.

A naïve induction runs afoul, though, since the interpreter can return intermediate results that are neither numbers nor booleans. We therefore need to strengthen the inductive hypothesis. If $\texttt{interp}_f$ returns a closure or suspension, the result returned by $\texttt{interp}_g$ may not literally be the same: for instance, $\texttt{interp}_f$ may return a location holding $\mathsf{susp}(l_0)$ and $\texttt{interp}_g$ may return a location holding $\mathsf{susp}(l_1)$. Nevertheless, these values should be the same up to a renaming of the locations in the domain of the returned store σ_f.

Formalizing the notion of when two stores are 'equivalent' up to renaming of their locations can be done using the underlying graphs. Two stores are 'equivalent' if their underlying graph representations are isomorphic via some function h, and the values held at the cells are 'equivalent' under h. More formally,

Definition

Two states $S = (\bar{l},\bar{\rho},\sigma)$ *and* $S' = (\bar{l'},\bar{\rho'},\sigma')$ *are* **congruent** *if there is an isomorphism* $h : \mathscr{G}(\sigma) \rightarrow \mathscr{G}(\sigma')$ *such that for any* $l \in \mathsf{dom}(\sigma)$, $\mathsf{refcount}(l,\sigma) = \mathsf{refcount}(h(l),\sigma')$ *and for any* $l \in \mathsf{dom}(\sigma)$,

1. *If* $\bar{l} = [l_1,\ldots,l_m]$, *then* $\bar{l'} = [h(l_1),\ldots,h(l_m)]$;
2. *If* $\bar{\rho} = [\rho_1,\ldots,\rho_n]$, *then* $\bar{\rho'} = [\rho'_1,\ldots,\rho'_n]$ *and for all* $1 \leq i \leq n$, $\mathsf{dom}(\rho_i) = \mathsf{dom}(\rho'_i)$ *and for all* $x \in \mathsf{dom}(\rho_i)$, $h(\rho_i(x)) = \rho'_i(x)$;
3. *For all* i, $\mathsf{dom}(\rho_i) = \mathsf{dom}(\rho'_i)$ *and for all* $x \in \mathsf{dom}(\rho_i)$, $h(\rho_i(x)) = \rho'_i(x)$;
4. *If* $\sigma(l) = n$, true, *or* false, *then* $\sigma(l) = \sigma'(h(l))$;
5. *If* $\sigma(l) = \mathsf{susp}(l')$, *then* $\sigma'(h(l)) = \mathsf{susp}(h(l'))$;
6. *If* $\sigma(l) = \mathsf{rec}(l',f)$, *then* $\sigma'(h(l)) = \mathsf{rec}(h(l'),f)$;
7. *If* $\sigma(l) = \mathsf{closure}(\lambda x.\, P,\rho)$, *then* $\sigma'(h(l)) = \mathsf{closure}(\lambda x.\, P,\rho')$, $\mathsf{dom}(\rho)$ *is equal to* $\mathsf{dom}(\rho')$, *and for any* $x \in \mathsf{dom}(\rho)$, $\rho'(x) = h(\rho(x))$;
8. *If* $\sigma(l) = \mathsf{recclosure}(\lambda x.\, P,\rho)$, *then* $\sigma'(h(l)) = \mathsf{recclosure}(\lambda x.\, P,\rho')$, $\mathsf{dom}(\rho)$ *is equal to* $\mathsf{dom}(\rho')$, *and for any* $x \in \mathsf{dom}(\rho)$, $\rho'(x) = h(\rho(x))$; *and*
9. *If* $\sigma(l) = \mathsf{thunk}(P,\rho)$, *then* $\sigma'(h(l)) = \mathsf{thunk}(P,\rho')$, $\mathsf{dom}(\rho)$ *is equal to* $\mathsf{dom}(\rho')$, *and for any* $x \in \mathsf{dom}(\rho)$, $\rho'(x) = h(\rho(x))$.

Then one may prove

Lemma 5
Suppose $(\bar{l}', \rho_f, \bar{\rho}', \sigma_f)$ *and* $(\bar{l}'', \rho_g, \bar{\rho}'', \sigma_g)$ *are congruent. If* $\mathtt{interp}_f(M, \rho_f, \sigma_f) = (l'_f, \sigma'_f)$, *then* $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$ *and the resultant states* $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ *and* $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ *are congruent.*

The proof is deferred to the appendix. From this lemma, the following theorem follows directly:

Theorem 6
Suppose f *and* g *are allocation relations. If* $\mathtt{interp}_f(M, \emptyset, \emptyset) = (l_f, \sigma_f)$, *then* $\mathtt{interp}_g(M, \emptyset, \emptyset) = (l_g, \sigma_g)$. *Moreover, if* $\sigma_f(l_f) = n$, true, *or* false, *then* $\sigma_f(l_f) = \sigma_g(l_g)$.

5.3 Correctness of the interpreter

Finally, we need to verify that the reference-counting semantics implements the natural semantics of Tables 4 and 5, i.e. evaluating a closed term of base type yields the same result in either semantics. The proof proceeds by induction on the number of steps in the evaluation (the height of the proof tree for the ($\Rightarrow$) direction, and the number of calls to interp for the ($\Leftarrow$) direction). We again need an expanded inductive hypothesis to carry out the proof, one in which we can relate the values held in memory locations to terms. To this end, we define the extraction functions valof(M, ρ, σ) and valofcell(l, σ). Intuitively, the function valofcell extracts a term from the storable value held at location l in store σ, and the function valof replaces the free variables of M with the extracted versions of the cells bound to the free variables according to ρ. The idea is easy to understand intuitively from an example. Suppose, for instance, cell l_0 holds thunk((dispose x before y), $[x \mapsto l_1, y \mapsto l_2]$), l_1 holds susp(l_3), l_3 holds 0, and l_2 holds true in the store σ. Then valofcell$(l_0, \sigma) =$ (dispose (store 0) before true). A larger example appears in Figure 5 (where reference counts have been ignored); if σ is the store depicted there, then

$$\text{valofcell}(l, \sigma) = \\ \lambda f.\ (((\lambda h.\ \lambda y.\ (\text{share } h \text{ as } h_1, h_2 \text{ in } h_1(h_2\ y)))\ f)\ (\text{store } ((\lambda x.\ x)\ \text{true})))$$

Formal definitions for valof and valofcell are given by simultaneous induction in Table 8 in the appendix. A similar definition is given in Plotkin (1975) for unwinding a closure relative to an SECD machine state.

Since we will be interpreting terms of arbitrary type, the inductive hypothesis must relate values returned by the natural semantics to values returned by the reference-counting interpreter. The key definition missing here is the definition of 'related values'. One might attempt to prove that for closed terms M, $M \Downarrow c$ iff $\mathtt{interp}(M, \emptyset, \emptyset) = (l', \sigma')$ and valofcell$(l', \sigma') = c$. While this statement holds for basic values, it *does not* hold for values of other types. The problem arises because the reference-counting interpreter memoizes the results of evaluating under store's whereas the natural semantics does not. For instance, evaluating the term

$$(\lambda x : !\text{Nat}.\ (\text{share } y, z \text{ as } x \text{ in if } (\text{zero? } (\text{fetch } y)) \text{ then } z \text{ else } z))\ (\text{store } (\text{succ } 5))$$

l_g	closure($\lambda h.\, \lambda y.$ (share h as h_1, h_2 in $h_1(h_2\ y)$), $\emptyset$)
l	closure($\lambda f.\, ((g\ f)\ x)$, $[g \mapsto l_g,\ x \mapsto l_x]$)
l_x	susp(l'_x)
l'_x	thunk((f true), $[f \mapsto l']$)
l'	closure($\lambda x.\, x$, $\emptyset$)

Fig. 5. Store for example of the valofcell operation.

in the natural semantics returns the value (store (succ 5)), whereas evaluating the expression in the reference-counting semantics returns the value (after unwinding) (store 6). The proof thus requires relating terms that are 'less evaluated' to terms that are 'more evaluated'.

Definition
$M \geq N$, read 'N requires less evaluation than M', iff there is a context $C[\,]$ such that $M = C[M']$, $N = C[c]$, M' is closed, and $M' \Downarrow c$.

Here, $C[\,]$ denotes a term with a missing subterm and $C[M']$ the term resulting from using M' for that subterm. Let $\geq^*$ be the reflexive, transitive closure of $\geq$. This relation is necessary in order to express the desired property:

Theorem 7
Suppose M is typable, $\mathrm{dom}(\rho) = FV(M)$, M' is closed, and $M' \geq^* \mathrm{valof}(M,\rho,\sigma)$. Suppose also that $\Re(\bar{l}', \rho, \bar{\rho}', \sigma)$.

1. *If $M' \Downarrow c$, then there exist l', σ' such that $\mathtt{interp}(M,\rho,\sigma) = (l',\sigma')$ and $c \geq^*$ $\mathrm{valofcell}(l',\sigma')$.*
2. *If $\mathtt{interp}(M,\rho,\sigma) = (l',\sigma')$, then there exists a c such that $M' \Downarrow c \geq^*$ $\mathrm{valofcell}(l',\sigma')$.*

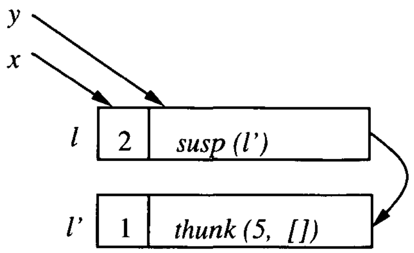
The extra assumptions about the state $(\bar{l}', \rho, \bar{\rho}', \sigma)$ – namely that it satisfies the invariants above – are used in constructing an execution in the reference-counting interpreter. The proof is deferred to the appendix.

6 Linear logic and memory

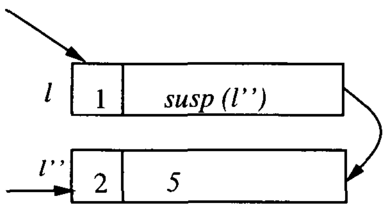
Let us now examine the question of the circumstances under which we are ensured that a location holding a value of linear type will maintain a reference count of at most one. In general, there is no guarantee that locations holding linear values will always have a reference count of one during the evaluation of a program. Consider, for example, the term

$$(\lambda w : !\mathrm{Nat}.\ (\mathrm{share}\ x, y\ \mathrm{as}\ w\ \mathrm{in\ if\ (zero?\ (fetch}\ y))\ \mathrm{then}\ x\ \mathrm{else}\ x))\ (\mathrm{store}\ 5).$$

During evaluation, a suspension is placed in a location l, which in turn holds a pointer to a location l' holding a thunk containing the value 5. This location l is then passed to w, and two pointers called x and y are then created by the share which reference l. Pictorially,



When the evaluation continues to the point of (fetch y), the contents of the location l' are evaluated to a location l'' holding 5, the suspension in l is updated to point to l'', and a pointer to l'' is then passed to the evaluation of zero?. Pictorially,



Thus, the cell containing 5 now has two pointers to it, even though it has linear type, Nat.

Clearly the issue here is whether the location holding a linear value is accessible from a location holding a non-linear one, like a susp. We would like a static condition under which we know that this does not happen. This seems difficult because, on the face of it, there are circumstances where a computation can alter the memory graph so that a linear value is brought into a location that is referenced by a non-linear value. Consider the term:

$$M \equiv \lambda x : \mathsf{Nat}.\, \lambda f : \mathsf{Nat} \multimap !\mathsf{Nat}.\, (\mathsf{store}\ y\ \mathsf{where}\ y = (f\ x)) \tag{1}$$

If N is a term of type Nat⊸!Nat, then the evaluation of $((M\ 0)\ N)$ will create a memory graph in which the location holding 0 has been brought into precisely the circumstance above, so its reference count might be increased by pointers passed through a susp. We need to know when this can happen if we are to have any way to ensure that a linear value maintains a reference count of at most one.

There is some help on this point to be found in the proof theory of linear logic. Note that the problem with term M in (1) relies on having a term N of type Nat⊸!Nat. From the standpoint of linear logic and its translation under the Curry-Howard correspondence, this is a suspicious assumption, however. The proposition $A \multimap !A$ is *not* provable in LL, and the situation illustrated by M runs contrary to proof-theoretic facts about what propositions are moved through 'boxes' in a proof net during cut elimination (Girard, 1987). This does not directly prove that a static property exists for the LL-based programming language, but it does suggest that there is hope.

To assert the desired property precisely, we will need some more terminology. Let us say that a storable object is **linear** if it is a numeral, boolean, closure, or recclosure and say that it is **non-linear** if it has the form $\mathsf{susp}(l)$, $\mathsf{rec}(l, f)$, or $\mathsf{thunk}(M, \rho)$. We say that a location l is **non-linear in store** σ if $\sigma(l)$ is a non-linear object; similarly, a location l is **linear in store** σ if $\sigma(l)$ is a linear object. The key property concerns the nature of the path in the memory graph between a location and the root set.

Definition
Suppose $S = (l, \sigma, \bar{l}, \bar{\rho})$ is a regular state and $\hat{l} \in \mathsf{dom}(\sigma)$. The location $\hat{l}$ is said to be **linear from** *l* **in** *S if there is a path p from l to $\hat{l}$ in $\mathscr{G}(S)$ such that each l' on p satisfies the following two properties:*

1. *$\sigma(l')$ is linear and*
2. $\mathsf{refcount}(l', \sigma) = 1$.

Note that the two conditions satisfied by the path p could only be satisfied by a *unique* path from l to $\hat{l}$; if there were more than one such path, condition (2) could not be satisfied. It will be convenient to say that a path satisfying these conditions is linear. Given a regular state $S = (\rho, \sigma, \bar{l}, \bar{\rho})$, we also say that $\hat{l}$ is linear from ρ in S if there is an x in the domain of ρ such that there is a (unique) linear path from $\rho(x)$ to $\hat{l}$.

To prove the desired property we will need to know some basic facts about types and evaluation. For the high-level semantics we already expressed the Subject Reduction Theorem 1 for the LL-based programming language. In conjunction with the Correctness Theorem 7 we have a version of the result for the low-level semantics as well:

Lemma 8
Suppose $S = (l, \sigma, \bar{l}, \bar{\rho})$ is a regular state, $\mathsf{dom}(\rho) = FV(M)$, $\vdash \mathsf{valof}(M, \rho, \sigma) : t$, and $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$. Then $\vdash \mathsf{valofcell}(l', \sigma') : t$.

The theorem we wish to express says that if a program is evaluated in an environment from which a location $\hat{l}$ is linear, then the value at the location is either used and deallocated or not used and linear from the location returned as the result of the evaluation. This statement is intended to formally capture the idea that a location that is linear from an environment is used once *or* left untouched with a reference count of one. Unfortunately, the assertion contains the term 'deallocate', which needs to be made precise. If we assert instead that the reference count of the location is 0 or linear from the result at the end of the computation, then there is a problem in the case where reference count falls to 0 because the allocation relation might *reallocate* the location $\hat{l}$ to hold a value that is unrelated to the one placed there originally. This would make it impossible to assert anything interesting about the outcome of the computation. To resolve this worry, we can make a restriction on the allocation relation insisting that $\hat{l}$ is not in its range. This assumption is harmless in a sense made precise by Lemma 5. The result of interest can now be asserted precisely as follows:

Theorem 9

Suppose $S = (\rho, \sigma, \bar{l}, \bar{p})$ is regular, $\mathsf{dom}(\rho) = FV(M)$, *and* $\mathsf{valof}(M, \rho, \sigma)$ *is typable. If $\hat{l}$ is linear from ρ in S, $\hat{l}$ is not in the range of* new, *and* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, *then one of the following two properties holds of the regular state $S' = (l', \sigma', \bar{l}, \bar{p})$:*

1. *Either* $\mathsf{refcount}(\hat{l}, \sigma') = 0$, *or*
2. $\mathsf{refcount}(\hat{l}, \sigma') = 1$ *and $\hat{l}$ is linear from l' in S'.*

Proof: The proof is by induction on the number of calls to interp. We exhibit only a few of the key cases here and leave the others for the reader.

1. $M = (P\ Q)$. The evaluation of M begins as follows:

$$\begin{aligned}&\mathtt{interp}(P, \rho \mid P, \sigma) = (l_0, \sigma_0)\\ &\mathtt{interp}(Q, \rho \mid Q, \sigma) = (l_1, \sigma_1)\\ &\sigma_1(l_0) = \mathsf{closure}(\lambda x.\, N, \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N, \rho').\end{aligned}$$

The fact that $\hat{l}$ is linear from ρ means that it is reachable from exactly one of $\rho \mid P$ or $\rho \mid Q$. We consider the two cases separately.

(a) $\hat{l}$ is reachable from $\rho \mid P$. By the inductive hypothesis ('IH'), one of the following two subcases applies:

i $\mathsf{refcount}(\hat{l}, \sigma_0) = 0$. By assumption, $\hat{l}$ is never reallocated by new, and hence it follows that $\mathsf{refcount}(\hat{l}, \sigma') = 0$.

ii $\mathsf{refcount}(\hat{l}, \sigma_0) = 1$ and $\hat{l}$ is linear from l_0. Then in the memory graph, there is a linear path

$$l_0 = l'_0, l'_1, \ldots, \hat{l}$$

(where we list only the locations associated with the path since the fact the reference counts are all equal to one means that the edges are uniquely determined). None of the locations l'_i can be reachable from $\rho \mid Q$ since that would imply that the reference count of at least one of them is greater than one. By Theorem 2, the contents and reference counts of the locations l'_i therefore do not change during the evaluation of Q. Now, $\hat{l}$ is linear from l_0 in $(l_1, l_0, \sigma_1, \bar{l}, \bar{p})$ and $\sigma_1(l_0)$ has the form $\mathsf{closure}(\lambda x.\, N, \rho')$ or $\mathsf{recclosure}(\lambda x.\, N, \rho')$, so $\hat{l}$ must be linear from ρ' in $(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{p})$ as well. Since we know that $\mathsf{refcount}(l_0, \sigma_1) = 1$, we conclude that

$$\mathtt{interp}(N, \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1)) = (l', \sigma')$$

and the desired conclusion follows from IH.

(b) $\hat{l}$ is reachable from $\rho \mid Q$. By assumption, there is a linear path

$$l'_0, l'_1, \ldots, \hat{l}$$

such that l_0' is in the range of $\rho \mid Q$. None of the locations on this path is reachable from $\rho \mid P$ because they all have reference count equal to one. Thus, by Theorem 2, their values are unchanged by the evaluation of P, and each l_i' is still unreachable from l_0 in σ_0. By IH, there are two possibilities regarding the regular state $(l_1, l_0, \sigma_1, \bar{l}, \bar{\rho})$ obtained after evaluating P and Q.

i refcount$(\hat{l}, \sigma_1) = 0$. By assumption $\hat{l}$ is never reallocated by **new**, so refcount$(\hat{l}, \sigma') = 0$ as needed.

ii refcount$(\hat{l}, \sigma_1) = 1$, In this case, the IH implies that there is a linear path from l_1 to $\hat{l}$. There are now two subcases to consider: either refcount$(l_0, \sigma_0) = 1$ or refcount$(l_0, \sigma_0) > 1$. We consider only the second and leave the first to the reader. By laws D2, I2, and E, we know that the state

$$S' = (\rho'[x \mapsto l_1], \text{inc-env}(\rho', \text{dec}(l_0, \sigma_1)), \bar{l}, \bar{\rho})$$

is regular and it is not hard to check that $\hat{l}$ is linear from $\rho'[x \mapsto l_1]$ in S'. Since we must have

$$\texttt{interp}(N, \rho'[x \mapsto l_1], \text{inc-env}(\rho', \text{dec}(l_0, \sigma_1))) = (l', \sigma')$$

we are done by IH.

2. $M =$ (store N where $x_1 = M_1, \ldots, x_n = M_n$). In this case, $\hat{l}$ is reachable from exactly one of the environments $\rho \mid M_i$. In the evaluation of M, we have

$$\texttt{interp}(M_1, \rho \mid M_1, \sigma) = (l_1, \sigma_1)$$
$$\vdots$$
$$\texttt{interp}(M_i, \rho \mid M_i, \sigma) = (l_i, \sigma_i)$$

By IH, there are two possibilities for the regular state

$$(l_1, \ldots, l_i, \rho \mid M_{i+1}, \ldots, \rho \mid M_n, \sigma_i, \bar{l}, \bar{\rho})$$

arising after the evaluation of M_i: either the reference count of $\hat{l}$ is zero in σ_i or it is one and there is a linear path from l_i to $\hat{l}$. If the first case holds, then we are done, since $\hat{l}$ is not reallocated in the remainder of the computation, and therefore the conclusion of the theorem is satisfied. On the other hand, the second case is impossible: by Lemma 8, valofcell(l_i, σ_i) has type $!t$ and so, since l_i must be a value by Theorem 2, l_i must be either susp(l'') or rec(l'', f). This contradicts the assumption that $\hat{l}$ is linear from l_i. Therefore reference count of $\hat{l}$ must be 0 in σ_i and hence we are done, since **new** never reallocates $\hat{l}$.

3. $M =$ (share x, y as P in Q). In the evaluation of M we compute

$$\texttt{interp}(P, \rho \mid P, \sigma) = (l_0, \sigma_0)$$
$$\texttt{interp}(Q, (\rho \mid Q)[x, y \mapsto l_0], \text{inc}(l_0, \sigma_0)) = (l', \sigma')$$

Now $\hat{l}$ is reachable for exactly one of the environments $\rho \mid P$ or $\rho \mid Q$. We consider the two cases separately.

(a) $\hat{l}$ is reachable from $\rho \mid P$. For the same reasons discussed in the case for store above, IH implies that $\mathsf{refcount}(\hat{l}, \sigma_0) = 0$, and thus we are done since new never reallocates $\hat{l}$.

(b) $\hat{l}$ is reachable from $\rho \mid Q$. Then there is a linear path from $\rho \mid Q$ to $\hat{l}$ which, by Theorem 2, is unaffected by the evaluation of P. In particular, $\hat{l}$ is not reachable from l_0, so it is linear from $\rho \mid Q$ in the regular state $((\rho \mid Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ so we are done by IH.

The remaining cases are treated similarly. □

To see an example of how the theorem can be applied to reasoning about properties that depend on the memory graph, suppose we want to evaluate *add* (store 2) 3 (where *add* is shorthand for the term in § 3) in the empty environment and empty store. The key steps are

- *add* (store 2) evaluates to (l_0, σ_0) with $\sigma_0(l_0) = \mathsf{closure}(\lambda x.\, N, \rho)$.
- 3 in σ_0 evaluates to (l_1, σ_1) such that $\sigma_1(l_1) = 3$.
- The body of *add* is evaluated with y mapped to l_1.

At this point the conditions required for Theorem 9 are true. Hence we know that the reference count of l_1 does not exceed one (so long as it is not deallocated and then reallocated). This implies that it is safe to update y in place during the recursive call. Similar analysis applies to definitions of multiplication and other recursive functions where we use a variable as an accumulator to store the result. This technique of proof allows us to achieve goals like those for which Hudak (1987) defined a collecting interpretation for reference counts.

7 Discussion

For this paper we have focused on a particular natural deduction presentation of linear logic, the proposal of Benton, Bierman, de Paiva, and Hyland (1993). Nevertheless, there is relatively little consensus on which proof system is the right system for constructing a typed programming language. As we pointed out before, substitutivity is one of the key technical properties for a proof system. Wadler (1993) proposes a different approach than the one taken here, using a syntax of patterns that yields a syntax with the substitutivity property. His approach also extends to a presentation of LL using judgements of the form $\Gamma; \Delta \vdash s$ where Γ is a set of 'intuitionistic assumptions' (types of non-linear variables) and Δ is a multi-set of 'linear assumptions' (types of linear variables). Another approach to the substitutivity problem has been to modify linear logic by adding new assumptions. For instance, Filinski (1992) and O'Hearn (1991) propose taking $!!A$ to be isomorphic to $!A$. (From the perspective of this paper, such an identification would collapse two levels of indirection and suspension into one and hence fundamentally change the character of the language.) Finally, even if substitutivity is achieved, automatic type checking or inference – an issue of clear importance in the design of a programming language – may not be achievable. For instance, Abramsky's (1993) system, which uses the sequent formulation of linear logic, satisfies substitutivity (the *cut rule* gives it automatically), but there is no clear means of doing type checking for his language. Others (Mackie, 1991; Lincoln and Mitchell, 1992) have attempted to reconcile substitutivity and type inference by proposing restricted forms of these properties. For each of these proposals – particularly the one of splitting assumptions into linear and intuitionistic assumptions – it would be interesting to see whether our interpretation of LL as reference counting is robust; our preliminary investigations suggest that it is.

It is also possible to fold reference-counting operations into the interpretation of an ordinary functional language (that is, one based on intuitionistic logic). The ways in which the result differs from the semantics we have given for an LL-based language are illuminating. First of all, there are several choices about how to do this. One approach is to maintain the invariant that interp is evaluated on triples (M, ρ, σ) where the domain of ρ is exactly the set of free variables of M. When evaluating an application $M \equiv (P\ Q)$, for example, it is essential to account for the possibility that some of the free variables of M are shared between P and Q. This means that when P is interpreted, the reference counts of variables they have in common must be incremented (otherwise they may be deallocated before the evaluation of Q begins):

```
interp($(P\ \ Q)$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($P$, $\rho \mid P$, inc-env($\rho \mid P \cap \rho \mid Q$, $\sigma$))
      ($l_1$, $\sigma_1$) = interp($Q$, $\rho \mid Q$, $\sigma_0$)
  in  case $\sigma_1(l_0)$ of closure($\lambda x. N$, $\rho'$) or recclosure($\lambda x. N$, $\rho'$) =>
        if refcount($l_0$, $\sigma_1$) = 1
        then interp($N$, $\rho'[x \mapsto l_1]$, dec($l_0$, $\sigma_1$))
        else interp($N$, $\rho'[x \mapsto l_1]$, inc-env($\rho'$, dec($l_0$, $\sigma_1$))).
```

The deallocation of variables is driven by the requirement that only the free variables of M can lie in the domain of ρ; this arises particularly in the semantics for the conditional:

```
interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($N$, $\rho \mid N$, inc-env($\rho|N \cap (\rho|P \cup \rho|Q)$, $\sigma$))
  in  if $\sigma_0(l_0)$ = true
      then interp($P$, $\rho \mid P$, dec($l_0$, dec-ptrs-env($(\rho|P) - (\rho|Q)$, $\sigma_0$)))
      else interp($Q$, $\rho \mid Q$, dec($l_0$, dec-ptrs-env($(\rho|Q) - (\rho|P)$, $\sigma_0$))).
```

An alternative approach to providing a reference-counting semantics for an intuitionistic language would be to delay the deallocation of variables until 'the last minute' and permit the application of interp to triples (M, ρ, σ) where the domain of ρ includes the free variables of M but may also include other variables. This makes it possible to simplify the interpretation of the conditional:

```
interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($N$, $\rho$, $\sigma$)
  in  if $\sigma_0(l_0)$ = true
      then interp($P$, $\rho$, dec($l_0$, $\sigma_0$))
      else interp($Q$, $\rho$, dec($l_0$, $\sigma_0$))
```

but the burden of disposal then shifts to the evaluation of constants:

interp(n, ρ, σ) = new(n, dec-ptrs-env(ρ, σ)).

The basic difference between a 'reference-counting interpretation of intuitionistic logic' following one of the approaches just described versus reference counting and linear logic is the way in which the LL primitives make many distinctions *explicit* in the code. The LL primitives make it possible to describe certain kinds of 'code motion' that concern when memory is deallocated. For example, the program

$$\lambda x : s.\ \mathsf{if}\ B\ \mathsf{then}\ (\mathsf{dispose}\ x\ \mathsf{before}\ P)\ \mathsf{else}\ (\mathsf{dispose}\ x\ \mathsf{before}\ Q)$$

can be shown to be equivalent *in the higher-level semantics* to

$$\lambda x : s.\ (\mathsf{dispose}\ x\ \mathsf{before\ if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q)$$

but the latter program can be viewed as preferable in our reference-counting semantics because it may deallocate the locations referenced by x sooner. (In fact, the LL proofs corresponding to these terms are equivalent by commutative conversions.) As another example, the program

$$\lambda x : s.\ (\mathsf{dispose}\ y\ \mathsf{before}\ M)$$

is equivalent to

$$(\mathsf{dispose}\ y\ \mathsf{before}\ \lambda x : s.\ M)$$

if x and y are different variables. The transformation may be significant if the value of y would be deallocated rather than needlessly held in a closure.

The question of whether an LL-based language could be useful as an intermediate language for compiler analysis for intuitionistic programming languages is certainly related to the techniques for translating between them. By analogy, there have been various studies of the subtleties of transformation to continuation-passing style (Lawall and Danvy, 1993, is a one recent example). A closer analogy is the translation of a language meant to be executed in call-by-name into a call-by-value language with primitives for delaying (store'ing) and forcing (fetch'ing). There is a standard translation for this purpose and many of the issues that arise for that translation also arise in the translation from intuitionistic to linear logic. For instance, a pair of programs that are strongly reminiscent of those in Table 1 appears in the discussion of the ALFL compiler in Bloss *et al.* (1988) based on a sample from the test suite in Gabriel (1985). This problem is addressed by the technique of *strictness analysis* (Abramsky and Hankin, 1987): with strictness analysis the translation can be made more efficient or the translated program can be optimized. There are several techniques known for translating intuitionistic logic into linear logic. To illustrate, consider the combinator S (here written in SML syntax):

```
fn x:(s -> t -> u) => fn y:(s -> t) => fn z:s => (x z) (y z).
```

When we apply one of Girard's translations (Girard, 1987), the result (using a syntax similar to the one in Table 1) is the following program:

```
fn x:!(!s -o !t -o u) => fn y:!(!s -o t) => fn z:!s =>
  share z1,z2 as z in
```

```
        ((fetch x) (store (fetch z1)))
        (store ((fetch y) (store (fetch z2))))
```

However, another program having S as its 'erasure' is

```
    fn x:(!s -o t -o u) => fn y:(!s -o t) => fn z:!s =>
      share z1,z2 as z in (x z1) (y z2)
```

which is evidently a much simpler and more efficient program. (This example also appears in Abramsky, 1993.) An analog of strictness analysis that applies to the LL translation is clearly needed if an LL intermediate language is to be of practical significance in analyzing 'intuitionistic' programs.

Our reference-counting interpreter and the associated invariance properties can easily be extended to the linear connectives &, ⊗, and ⊕ (although it is unclear, in the present framework, how to handle the 'classical' connectives). Extending the results to dynamic allocation of references and arrays is not difficult if such structures do not create cycles. For instance, it can be assumed that only integers and booleans are assignable to mutable reference cells. To see this in a little more detail, if we assume that o is Nat or Bool, then typing rules can be given as follows:

$$\frac{\Gamma \vdash M : o}{\Gamma \vdash \mathsf{ref}(M) : \mathsf{ref}(o)} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(o) \qquad \Delta \vdash N : o}{\Gamma, \Delta \vdash M := N : \mathsf{ref}(o)} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(o)}{\Gamma \vdash !M : o}$$

To create a reference cell initialized with the value of a term M, the term M is evaluated and its value is *copied* into a new cell:

(16) $\mathtt{interp}(\mathsf{ref}(M), \rho, \sigma) =$
  $\mathtt{let}\ (l_0, \sigma_0) = \mathtt{interp}(M, \rho, \sigma)$
  $\mathtt{in}\ \ \mathsf{new}(\sigma_0(l_0), \mathsf{dec}(l_0, \sigma_0))$

The location l_0 holds the *immutable* value of M; a new *mutable* cell must be created with the value of M as its initial value. Assignment mutates the value associated with such a cell:

(17) $\mathtt{interp}(M := N, \rho, \sigma) =$
  $\mathtt{let}\ (l_0, \sigma_0) = \mathtt{interp}(M, \rho \mid M, \sigma)$
    $(l_1, \sigma_1) = \mathtt{interp}(N, \rho \mid N, \sigma_0)$
  $\mathtt{in}\ \ (l_0, \mathsf{dec}(l_1, \sigma_1[l_0 \mapsto \sigma_1(l_1)]))$

To obtain the value held in a mutable cell denoted by M, the contents of the cell must be copied to a new immutable cell:

(18) $\mathtt{interp}(!M, \rho, \sigma) =$
  $\mathtt{let}\ (l_0, \sigma_0) = \mathtt{interp}(M, \rho, \sigma)$
  $\mathtt{in}\ \ \mathsf{new}(\sigma_0(l_0), \mathsf{dec}(l_0, \sigma_0))$

Although the code for creating a reference cell and the code for dereferencing look the same, they are dual to one another in the sense that cell creation, $\mathsf{ref}(M)$, copies the contents of an immutable cell to a mutable one while dereferencing, $!M$ copies the contents of a mutable cell to an immutable one. The language designed in this way is similar to Scheme with force and delay primitives, but with restrictions like those SML places on the types of values that are mutable. The restriction on the types of elements held in reference cells is similar to those made for block-structured languages, which do not permit higher-order procedures to be assigned to variables (reference cells).

We have demonstrated that a language whose design is guided by an analog of the Curry–Howard correspondence applied to linear logic can be interpreted as providing fine-grained information about reference counts in the memory graphs produced by the program during run-time. As such, the LL-based language may be useful for detecting or proving the correctness of forms of program analysis that rely on reference counts of nodes of memory graphs. As a secondary theme we have illustrated an approach to expressing and proving properties of programs at a level of abstraction in which properties of memory graphs are significant but some lower-level properties, such as memory layout, are abstracted away. Isolating this level of abstraction could be useful for correctness proofs of lower levels, such as the correctness of a memory allocation scheme.

Acknowledgements

This research was partially supported by ONR Young Investigator Award number N00014-88-K-0557, NSF grant CCR-8912778, NRL grant N00014-91-J-2022, and NOSC grant 19-920123-31. A preliminary version of the paper appeared in the Proceedings of the 1992 ACM Conference on LISP and Functional Programming. For discussions that contributed to the investigation and exposition in this paper we thank Samson Abramsky, Andrew Appel, Val Breazu-Tannen, Amy Felty, Elsa Gunter, James Hicks, Dave MacQueen, Diana Meadows, Andre Scedrov, Phil Wadler, David Wise, and the anonymous referees.

A Proofs of the main theorems

A.1 Verification of the basic laws in Table 7

Proposition 10
Each of the laws A1, A2, D1, D2 *given in* § *2 hold.*

Proof: The proof of A1 may be found in § 4, and the proof of A2 is similar. We need to verify D1 and D2.

D1 Suppose $S = (l, \bar{l}, \bar{\rho}, \sigma)$, $\mathfrak{R}(S)$ holds, $\sigma(l)$ is a numeral or boolean, and $S' = (\bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$. Note that there are no outgoing edges from l in the memory graph induced by S; thus, even if $l \notin \mathsf{dom}(\mathsf{dec}(l, \sigma))$, the state S' is count-correct. Since $\mathsf{dom}(\sigma) \supseteq \mathsf{dom}(\mathsf{dec}(l, \sigma))$, each of the properties $\mathfrak{R}2$–$\mathfrak{R}5$ follow directly from the hypothesis. It is obvious also that $\mathfrak{R}6$ holds. Thus, $\mathfrak{R}(S')$.

D2 Suppose $\mathfrak{R}(l, \bar{l}, \bar{\rho}, \sigma)$ and $\mathsf{refcount}(l, \sigma) \neq 1$, and let $S' = (\bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$. By hypothesis, it follows that $\mathsf{refcount}(l, \sigma) > 1$ since l is in the root set. Thus, $\mathsf{refcount}(l, \mathsf{dec}(l, \sigma)) \geq 1$ and hence S' is count-correct, satisfying $\mathfrak{R}1$. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\mathsf{dec}(l, \sigma))$, each of the properties $\mathfrak{R}2$–$\mathfrak{R}5$ follow directly from the hypothesis. It is obvious also that $\mathfrak{R}6$ holds. Thus, $\mathfrak{R}(S')$.

This completes the verification of each part. $\square$

Proposition 11
Law D3 *holds; more generally,*

1. If $\Re(l, \bar{l}, \bar{\rho}, \sigma)$, then $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$.
2. If $\Re(\bar{l}, \rho, \bar{\rho}, \sigma)$, then $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs-env}(\rho, \sigma))$.

Proof: By induction on the total number of calls to dec-ptrs and dec-ptrs-env. In the basis, suppose the number of calls is one; there are two cases:

1. dec-ptrs is called. Then there are three subcases:
   (a) $\sigma(l) = n$, true, or false. Then $\text{dec-ptrs}(l, \sigma) = \text{dec}(l, \sigma)$. By D1, it follows that $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$.
   (b) $\sigma(l) = \text{susp}(l')$, $\text{thunk}(M, \rho)$, or $\text{closure}(\lambda x.\, M, \rho)$, and $\text{refcount}(l, \sigma) > 1$. Then $\text{dec-ptrs}(l, \sigma) = \text{dec}(l, \sigma)$, and hence by D2, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$.
   (c) $\sigma(l) = \text{rec}(l', f)$ or $\text{recclosure}(\lambda x.\, N, \rho)$, and $\text{refcount}(l, \sigma) > 2$. Then it follows that $\text{dec-ptrs}(l, \sigma) = \text{dec}(l, \sigma)$, and hence by D2, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$.
2. dec-ptrs-env is called. Then since dec-ptrs is not called, $\text{dom}(\rho)$ must be the empty set. Thus, $\text{dec-ptrs-env}(\rho, \sigma) = \sigma$ and hence $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs-env}(\rho, \sigma))$.

For the induction case, suppose the total number of calls to dec-ptrs and dec-ptrs-env is greater than one. There are again two main cases:

1. dec-ptrs is called. There are five subcases depending on the reference count and the value stored at l.
   (a) $\sigma(l) = \text{susp}(l')$, $\text{refcount}(l, \sigma) = 1$. Then
   $$\text{dec-ptrs}(l, \sigma) = \text{dec-ptrs}(l', \text{dec}(l, \sigma)).$$
   If $\sigma(l') = \text{thunk}(N, \rho)$, then by A2 $\Re(\rho, \bar{l}, \bar{\rho}, \text{dec}(l', \text{dec}(l, \sigma)))$, so by induction, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs-env}(\rho, \text{dec}(l', \text{dec}(l, \sigma))))$. Thus, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$. The other case, when $\sigma(l')$ is not a thunk, is similar and hence omitted.
   (b) $\sigma(l) = \text{thunk}(M, \rho)$, $\text{refcount}(l, \sigma) = 1$. Then
   $$\text{dec-ptrs}(l, \sigma) = \text{dec-ptrs-env}(\rho, \text{dec}(l, \sigma)).$$
   By A1, $\Re(\bar{l}, \rho, \bar{\rho}, \text{dec}(l, \sigma))$ so by induction, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs-env}(\rho, \text{dec}(l, \sigma)))$. Thus, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$.
   (c) $\sigma(l) = \text{closure}(\lambda x.\, M, \rho)$, $\text{refcount}(l, \sigma) = 1$. Similar to the previous case.
   (d) $\sigma(l) = \text{recclosure}(\lambda x.\, N, \rho)$, $\sigma(l') = \text{rec}(l, f)$, $\text{refcount}(l', \sigma) = 1$, and $\text{refcount}(l, \sigma) = 2$. Then
   $$\text{dec-ptrs}(l, \sigma) = \text{dec-ptrs-env}(\rho - f, \text{dec}(l', \text{dec}(l, \text{dec}(l, \sigma)))).$$
   Let $\sigma_0 = \text{dec}(l', \text{dec}(l, \text{dec}(l, \sigma)))$; then the state $S = (\bar{l}, \rho - f, \bar{\rho}, \sigma_0)$ is count-correct, since both l and l' have disappeared from the memory graph. Also, S satisfies properties $\Re2$–$\Re5$, since $\text{dom}(\sigma_0) \subseteq \text{dom}(\sigma)$, and $\Re6$ holds by hypothesis. Thus, $\Re(S)$, and so by induction $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs-env}(\rho - f, \sigma_0))$. Thus, $\Re(\bar{l}, \bar{\rho}, \text{dec-ptrs}(l, \sigma))$.

(e) $\sigma(l) = \mathsf{rec}(l',f)$, $\sigma(l') = \mathsf{recclosure}(\lambda x.\, N, \rho)$, $\mathsf{refcount}(l,\sigma) = 2$, and $\mathsf{refcount}(l',\sigma) = 1$. Then

$$\mathsf{dec\text{-}ptrs}(l,\sigma) = \mathsf{dec\text{-}ptrs\text{-}env}(\rho - f, \mathsf{dec}(l', \mathsf{dec}(l, \mathsf{dec}(l,\sigma))))$$

and one may proceed in a manner similar to the previous case.

2. dec-ptrs-env is called. Since the number of calls is greater than 1, $\mathsf{dom}(\rho) = \{x_1, \ldots, x_n\}$ for $n \geq 0$. Since $\mathfrak{R}(\rho(x_1), \ldots, \rho(x_n), \bar{l}, \bar{\rho}, \sigma)$ and

$$\mathsf{dec\text{-}ptrs\text{-}env}(\rho,\sigma) = \mathsf{dec\text{-}ptrs}(\rho(x_n), \mathsf{dec\text{-}ptrs}(\ldots \mathsf{dec\text{-}ptrs}(\rho(x_1),\sigma)\ldots)),$$

by repeated applications of the inductive hypothesis, $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(\rho,\sigma))$.

This completes the induction case and hence the proof. □

Proposition 12
Each of the laws I1, I2, E, N1, N2, N3, U1, *and* U2 *in § 4 hold.*

Proof: We verify each law individually.

I1 Suppose $\mathfrak{R}(\bar{l}, \bar{\rho}, \sigma)$, $l \in \mathsf{dom}(\sigma)$, and $\sigma(l)$ is not a thunk. Let S' be the state $(l, \bar{l}, \bar{\rho}, \mathsf{inc}(l,\sigma))$. Since there is one more pointer to l from the root set of S' and the reference count has been incremented, S' is count-correct. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\mathsf{inc}(l,\sigma))$, each of the properties $\mathfrak{R}2$–$\mathfrak{R}5$ follow directly from the hypothesis; property $\mathfrak{R}6$ is also obvious from the hypothesis. Thus, $\mathfrak{R}(S')$.

I2 Suppose $\mathfrak{R}(\bar{l}, \bar{\rho}, \sigma)$, $\rho(x) \in \mathsf{dom}(\sigma)$ for all $x \in \mathsf{dom}(\rho)$, and each $\sigma(\rho(x))$ is not a thunk. Then by an easy induction on the size of $\mathsf{dom}(\rho)$ using arguments similar to the last case, $\mathfrak{R}(\bar{l}, \rho, \bar{\rho}, \mathsf{inc\text{-}env}(\rho,\sigma))$. Note that it is important that each $\sigma(\rho(x))$ is not a thunk, for otherwise $\mathfrak{R}3$ would be violated.

E Suppose $\mathfrak{R}(l, \bar{l}, \rho, \bar{\rho}, \sigma)$ and $x \notin \mathsf{dom}(\rho)$, and let $S' = (\bar{l}, \rho[x \longmapsto l], \bar{\rho}, \sigma)$. Then the root set points of S and S' are identical, and the memory graph induced by S and S' are hence identical. Thus, $\mathfrak{R}(S')$. The converse is similar.

N1 Suppose $S = (\bar{l}, \bar{\rho}, \sigma)$, $\mathfrak{R}(S)$, and $(l',\sigma') = \mathsf{new}(c,\sigma)$ for some constant c, and let $S' = (l', \bar{l}, \bar{\rho}, \sigma')$. Since new is an allocation relation, it follows that $\mathsf{refcount}(l',\sigma) = 0$, $\mathsf{refcount}(l',\sigma') = 1$, and for any $l \neq l'$, $\sigma(l) = \sigma(l')$ and $\mathsf{refcount}(l,\sigma) = \mathsf{refcount}(l,\sigma')$. First, note that S' is count-correct, since the only location in σ' that is different from σ is l', and that location has a pointer in the root set. This verifies property $\mathfrak{R}1$. Since $\mathsf{dom}(\sigma)$ is finite, $\mathsf{dom}(\sigma')$ is also finite and so property $\mathfrak{R}2$ holds of S. Finally, since new does not create any additional cycles in the memory graph or thunks or closures, properties $\mathfrak{R}3$–$\mathfrak{R}6$ hold in S'. Thus, $\mathfrak{R}(S')$.

N2 Suppose $\mathfrak{R}(\bar{l}, \rho, \bar{\rho}, \sigma)$, $FV(N) = \mathsf{dom}(\rho)$, $(l',\sigma') = \mathsf{new}(\mathsf{closure}(N,\rho),\sigma)$ or $\mathsf{new}(\mathsf{recclosure}(N,\rho),\sigma)$, and N is typable, and let $S' = (l', \bar{l}, \bar{\rho}, \sigma')$. Since new is an allocation relation, $\mathsf{refcount}(l',\sigma) = 0$, $\mathsf{refcount}(l',\sigma') = 1$, and for any location $l \neq l'$, $\sigma(l) = \sigma(l')$ and $\mathsf{refcount}(l,\sigma) = \mathsf{refcount}(l,\sigma')$. To see that property $\mathfrak{R}1$—namely count-correctness—holds of S', note that all of the pointers from ρ are accounted for in the closure stored in l', and that l' only has reference count 1. To see $\mathfrak{R}2$, $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$ is finite because $\mathsf{dom}(\sigma)$ is. No cycles are created in the induced memory graph by new, so $\mathfrak{R}4$ holds. Finally, $\mathfrak{R}5$ and $\mathfrak{R}6$ hold by hypothesis. Thus, $\mathfrak{R}(S')$.

N3 Suppose $\mathfrak{R}(\bar{l}, \rho, \bar{\rho}, \sigma)$, $(l_0, \sigma_0) = \mathsf{new}(\mathsf{thunk}(N, \rho), \sigma)$ where $FV(N) = \rho$, and $(l', \sigma') = \mathsf{new}(\mathsf{susp}(l_0), \sigma_0)$. Then $\mathfrak{R}(l', \bar{l}, \bar{\rho}, \sigma')$ follows in a manner similar to the previous case.

U1 Suppose $S = (l', \bar{l}, \bar{\rho}, \sigma)$ and $\mathfrak{R}(S)$, $\sigma(l)$ is a constant. We prove the second statement of U1 only; the first follows similarly. So suppose l is not reachable from l' in the graph induced by S, and let $S' = (l', \bar{l}, \bar{\rho}, \mathsf{inc}(l', \sigma[l \mapsto \mathsf{susp}(l')]))$. In S' the in-degree of l' is now one greater than in S; the in-degree of all other nodes remains the same. Thus, S' satisfies property $\mathfrak{R}1$. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$, the domain of σ' is finite, satisfying property $\mathfrak{R}2$. No new thunks are created, so property $\mathfrak{R}3$ holds of S'. Since l is not reachable from l' in S, there is no cycle through l in S'. Thus, S' satisfies property $\mathfrak{R}4$. Finally, properties $\mathfrak{R}5$ and $\mathfrak{R}6$ hold since no thunks or closures are added to σ. Thus, $\mathfrak{R}(S')$.

U2 Suppose $S = (l, \bar{l}, \bar{\rho}, \sigma)$ and $\mathfrak{R}(S)$, $\mathsf{refcount}(l, \sigma) \neq 1$, $\sigma(l) = \mathsf{susp}(l')$, and $\sigma(l') = \mathsf{thunk}(N, \rho)$, and let $S' = (\rho, \bar{l}, \bar{\rho}, \mathsf{dec}(l', \mathsf{dec}(l, \sigma[l \mapsto c])))$. To verify property $\mathfrak{R}1$, note first that $\mathsf{refcount}(l', \sigma) = 1$ by hypothesis. Thus, since the pointers from $\sigma(l')$ are mentioned in the root set of S', it follows that S' is count-correct. It is also clear that each of the properties $\mathfrak{R}2$–$\mathfrak{R}6$ hold of S'. Thus, $\mathfrak{R}(S')$.

This completes the verification of each part. □

A.2 Proof of Lemma 5

Lemma 5

Suppose $(\bar{l}', \rho_f, \bar{\rho}', \sigma_f)$ and $(\bar{l}'', \rho_g, \bar{\rho}'', \sigma_g)$ are congruent. If $\mathtt{interp}_f(M, \rho_f, \sigma_f) = (l'_f, \sigma'_f)$, *then* $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$ *and the resultant states $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent.*

Proof: By induction on the number of calls to `interp`. We cover the four cases in the core language and leave the other cases to the reader. Let h be an isomorphism from $\mathscr{G}(\sigma_f)$ to $\mathscr{G}(\sigma_g)$ that makes the above states congruent.

1. $M = x$. Then

$$\mathtt{interp}_f(M, \rho_f, \sigma_f) = (\rho_f(x), \sigma_f)$$
$$\mathtt{interp}_g(M, \rho_g, \sigma_g) = (\rho_g(x), \sigma_g),$$

and the resultant states $(\rho_f(x), \bar{l}', \bar{\rho}', \sigma'_f)$ and $(\rho_g(x), \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent via h.

2. $M = (\lambda x.\, P)$. Then

$$\mathtt{interp}_f(M, \rho_f, \sigma_f) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho_f), \sigma_f) = (l'_f, \sigma'_f).$$

Since f is an allocation relation,

- $l'_f \notin \mathsf{dom}(\sigma_f)$ and $\mathsf{dom}(\sigma'_f) = \mathsf{dom}(\sigma_f) \cup \{l'_f\}$;
- for all $l \in \mathsf{dom}(\sigma_f)$, $\sigma_f(l) = \sigma'_f(l)$ and $\mathsf{refcount}(l, \sigma_f) = \mathsf{refcount}(l', \sigma'_f)$; and
- $\sigma'_f(l'_f) = \mathsf{closure}(\lambda x.\, P, \rho_f)$ and $\mathsf{refcount}(l'_f, \sigma'_f) = 1$.

Note that $\mathtt{interp}_g(M,\rho_g,\sigma_g) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P,\rho_g),\sigma_g) = (l'_g,\sigma'_g)$. Again, since g is an allocation relation,

- $l'_g \notin \mathsf{dom}(\sigma_g)$ and $\mathsf{dom}(\sigma'_g) = \mathsf{dom}(\sigma_g) \cup \{l'_g\}$;
- for all $l \in \mathsf{dom}(\sigma_g)$, $\sigma_g(l) = \sigma'_g(l)$ and $\mathsf{refcount}(l,\sigma_g) = \mathsf{refcount}(l',\sigma'_g)$; and
- $\sigma'_g(l'_g) = \mathsf{closure}(\lambda x.\, P,\rho_g)$ and $\mathsf{refcount}(l'_g,\sigma'_g) = 1$.

Let $h' = h[l'_f \mapsto l'_g]$. It is clear that h' is an isomorphism from $\mathscr{G}(\sigma'_f)$ to $\mathscr{G}(\sigma'_g)$. Now consider the resultant states $(l'_f,\bar{l}',\bar{\rho}',\sigma'_f)$ and $(l'_g,\bar{l}'',\bar{\rho}'',\sigma'_g)$. Using the isomorphism h, the first two conditions for congruence of states are satisfied, and so we just need to show that the last six properties, stating the relationship between the values stored at locations, is satisfied. But the contents of the cells in σ_f and σ_g do not change, and for the new locations, $\sigma'_f(l'_f) = \mathsf{closure}(\lambda x.\, N,\rho_f)$, $\sigma'_g(h'(l'_f)) = \sigma'_g(l'_g) = \mathsf{closure}(\lambda x.\, N,\rho_g)$, $\mathsf{dom}(\rho_f) = \mathsf{dom}(\rho_g)$, and for all $x \in \mathsf{dom}(\rho_f)$, $\rho_g(x) = h'(\rho_f(x))$; the last two facts follow from the hypothesis. Thus, the resultant states are congruent.

3. $M = (P\ Q)$. Since $\mathtt{interp}_f(M,\rho_f,\sigma_f) = (l'_f,\sigma'_f)$,

   - $\mathtt{interp}_f(P,\rho_f \mid P,\sigma_f) = (l_{f,0},\sigma_{f,0})$;
   - $\mathtt{interp}_f(Q,\rho_f \mid Q,\sigma_{f,0}) = (l_{f,1},\sigma_{f,1})$;
   - $\sigma_{f,1}(l_{f,0}) = \mathsf{closure}(\lambda x.\, N,\rho'_f)$ or $\mathsf{recclosure}(\lambda x.\, N,\rho'_f)$.

   By hypothesis the environments ρ_f and ρ_g have the same domain, so they can also be divided into $\rho_g \mid P$ and $\rho_g \mid Q$. By two applications of the inductive hypothesis,

   - $\mathtt{interp}_g(P,\rho_g \mid P,\sigma_g) = (l_{g,0},\sigma_{g,0})$ and
   - $\mathtt{interp}_g(Q,\rho_g \mid Q,\sigma_{g,0}) = (l_{g,1},\sigma_{g,1})$,

   and the states $(l_{f,0},l_{f,1},\bar{l}',\bar{\rho}',\sigma_{f,1})$ and $(l_{g,0},l_{g,1},\bar{l}'',\bar{\rho}'',\sigma_{g,1})$ are congruent. In particular, note that $\sigma_{g,0}(l_{g,0}) = \mathsf{closure}(\lambda x.\, N,\rho'_g)$ or $\mathsf{recclosure}(\lambda x.\, N,\rho'_g)$. There are now two cases:

   (a) $\mathsf{refcount}(l_{f,0},\sigma_{f,1}) = 1$. Then $\mathsf{refcount}(l_{g,0},\sigma_{g,1})$ is also 1, since the two reference counts must be the same. Since the states $(\bar{l}',\rho'_f[x \mapsto l_{f,1}],\bar{\rho}',\sigma_{f,1})$ and $(\bar{l}'',\rho'_g[x \mapsto l_{g,1}],\bar{\rho}'',\sigma_{g,1})$ are congruent, it follows from the inductive hypothesis that $\mathtt{interp}_g(N,\rho'_g[x \mapsto l_{g,1}],\mathsf{dec}(l_0,\sigma_{g,1})) = (l'_g,\sigma'_g)$ and $(l'_f,\bar{l}',\bar{\rho}',\sigma'_f)$ and $(l'_g,\bar{l}',\bar{\rho}',\sigma'_g)$ are congruent. Putting the pieces together, $\mathtt{interp}_g(M,\rho_g,\sigma_g) = (l'_g,\sigma'_g)$ as desired.

   (b) $\mathsf{refcount}(l_{f,0},\sigma_{f,2}) \neq 1$. Then $\mathsf{refcount}(l_{g,0},\sigma_{g,2}) \neq 1$ also, since the two reference counts must be the same. Since $(\bar{l}',\rho'_f[x \mapsto l_{f,1}],\bar{\rho}',\sigma_{f,1})$ and $(\bar{l}''\rho'_g[x \mapsto l_{g,1}],\bar{\rho}'',\sigma_{g,1})$ are congruent, by the inductive hypothesis it follows that $\mathtt{interp}_g(N,\rho'_g[x \mapsto l_{g,1}],\mathsf{inc\text{-}env}(\rho'_g,\mathsf{dec}(l_{g,0},\sigma_{g,1}))) = (l'_g,\sigma'_g)$, and the states $(l'_f,\bar{l}',\bar{\rho}',\sigma'_f)$ and $(l'_g,\bar{l}'',\bar{\rho}'',\sigma'_g)$ are congruent. Putting the pieces together, $\mathtt{interp}_g(M,\rho_g,\sigma_g) = (l'_g,\sigma'_g)$ as desired.

This completes the induction and hence the proof. □

A.3 Proof of Theorem 7

Recall from § 5 that, in order to prove a correctness theorem, we needed a definition of how to unwind a term from a store. The definition of two mutually-recursive

Table 8. *Definitions of* valof *and* valofcell.

$$
\begin{aligned}
\mathsf{valof}(x,\rho,\sigma) &= \mathsf{valofcell}(\rho(x),\sigma)\\
\mathsf{valof}(\lambda x.\,P,\rho,\sigma) &= \lambda x.\,\mathsf{valof}(P,\rho,\sigma), \quad x \notin \mathsf{dom}(\rho)\\
\mathsf{valof}((P\ Q),\rho,\sigma) &= (\mathsf{valof}(P,\rho,\sigma)\ \mathsf{valof}(Q,\rho,\sigma))\\
\mathsf{valof}((\mathsf{fetch}\ P),\rho,\sigma) &= (\mathsf{fetch}\ \mathsf{valof}(P,\rho,\sigma))\\
\mathsf{valof}((\mathsf{share}\ x,y\ \mathsf{as}\ P\ \mathsf{in}\ Q),\rho,\sigma) &= (\mathsf{share}\ x,y\ \mathsf{as}\ \mathsf{valof}(P,\rho,\sigma)\ \mathsf{in}\ \mathsf{valof}(Q,\rho,\sigma)),\\
&\quad\ \mathsf{where}\ x,y \notin \mathsf{dom}(\rho)\\
\mathsf{valof}((\mathsf{dispose}\ P\ \mathsf{before}\ Q),\rho,\sigma) &= (\mathsf{dispose}\ \mathsf{valof}(P,\rho,\sigma)\ \mathsf{before}\ \mathsf{valof}(Q,\rho,\sigma))\\
\mathsf{valof}(n,\rho,\sigma) &= n\\
\mathsf{valof}(\mathsf{true},\rho,\sigma) &= \mathsf{true}\\
\mathsf{valof}(\mathsf{false},\rho,\sigma) &= \mathsf{false}\\
\mathsf{valof}((\mathsf{succ}\ P),\rho,\sigma) &= (\mathsf{succ}\ \mathsf{valof}(P,\rho,\sigma))\\
\mathsf{valof}((\mathsf{pred}\ P),\rho,\sigma) &= (\mathsf{pred}\ \mathsf{valof}(P,\rho,\sigma))\\
\mathsf{valof}((\mathsf{zero?}\ P),\rho,\sigma) &= (\mathsf{zero?}\ \mathsf{valof}(P,\rho,\sigma))\\
\mathsf{valof}((\mathsf{fix}\ P),\rho,\sigma) &= (\mathsf{fix}\ \mathsf{valof}(P,\rho,\sigma))\\
\mathsf{valof}((\mathsf{if}\ N\ \mathsf{then}\ P\ \mathsf{else}\ Q),\rho,\sigma) &= \mathsf{if}\ \mathsf{valof}(N,\rho,\sigma)\ \mathsf{then}\ \mathsf{valof}(P,\rho,\sigma)\ \mathsf{else}\ \mathsf{valof}(Q,\rho,\sigma)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{valof}((\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1,\dots,x_n = M_n),\rho,\sigma)\\
&\quad = (\mathsf{store}\ \mathsf{valof}(N,\rho,\sigma)\ \mathsf{where}\ x_1 = \mathsf{valof}(M_1,\rho,\sigma),\dots,x_n = \mathsf{valof}(M_n,\rho,\sigma))\\
&\quad\ \mathsf{where}\ x_i \notin \mathsf{dom}(\rho)
\end{aligned}
$$

$$
\mathsf{valofcell}(l,\sigma) = \begin{cases}
n & \text{if } \sigma(l) = n\\
\mathsf{true} & \text{if } \sigma(l) = \mathsf{true}\\
\mathsf{false} & \text{if } \sigma(l) = \mathsf{false}\\
\lambda x.\,\mathsf{valof}(M,\rho,\sigma) & \text{if } \sigma(l) = \mathsf{closure}(\lambda x.\,M,\rho) \text{ or}\\
 & \quad \sigma(l) = \mathsf{recclosure}(\lambda x.\,M,\rho)\\
 & \quad \text{where } x \notin \mathsf{dom}(\rho)\\
(\mathsf{store}\ \mathsf{valofcell}(l',\sigma)) & \text{if } \sigma(l) = \mathsf{susp}(l')\\
\mathsf{valof}(M,\rho,\sigma) & \text{if } \sigma(l) = \mathsf{thunk}(M,\rho)\\
\mathsf{valof}((\mathsf{fix}\ (\mathsf{store}\ (\lambda f.\,\lambda x.\,M))),\rho - f,\sigma) & \text{if } \sigma(l) = \mathsf{rec}(l',f),\\
 & \quad \sigma(l') = \mathsf{recclosure}(\lambda x.M,\rho)\\
 & \quad \text{where } x \notin \mathsf{dom}(\rho), \rho(f) = l
\end{cases}
$$

functions for performing this task, valof and valofcell, appears in Table 8. It is obvious from the definitions that only the reachable cells affect the value returned by valof and valofcell. For instance, if l' is not reachable from l in store σ and $\sigma' = \mathsf{dec}(l',\sigma)$, then $\mathsf{valofcell}(l,\sigma) = \mathsf{valofcell}(l,\sigma')$. We will use this fact throughout the arguments that follow.

Also essential to the proof of Theorem 7 is a notion of when one term is 'more evaluated' than another. A relation $\geq^*$ between terms which expresses this relationship is defined in § 5. We can prove three lemmas about the relationship of $\geq$ and canonical forms.

Lemma 13

If $c \geq P$ and c is a canonical form, then P is a canonical form. Moreover, c and P have the same shape, i.e. if c is a numeral or boolean, then $c = P$; if $c = \lambda x.\,Q$, then $P = \lambda x.\,Q'$; and if $c =$ (store Q)*, then $P =$* (store Q').

Proof: There are two cases to consider: either $c \Downarrow P$, or $c = C[M]$, $P = C[N]$, $C[\cdot]$ is nontrivial, and $M \Downarrow N$. In the first case, since c is canonical, $c = P$, and hence

P is canonical. In the second case, for c to be canonical it must be the case that $C[\cdot] = n$, true, false, $\lambda x.\, D[\cdot]$, or (store $C[\cdot]$). Thus, P must be canonical as well, and must have the same shape as c. □

Lemma 14
If c is a canonical form and $M \geq c$, then $M \Downarrow d \geq c$.

Proof: By the definition of $M \geq c$, we know that $M = C[M']$, $c = C[d]$, and $M' \Downarrow d$. In order for c to be canonical, it must be the case that either $C[\cdot] = [\cdot]$, n, true, false, $\lambda x.\, D[\cdot]$, or (store $D[\cdot]$). In the first case, $M' = M$ and $d = c$, so $M \Downarrow c \geq c$. For the other cases, $M \Downarrow M \geq c$. □

Lemma 15
If c is a canonical form and $M \geq^* c$, then $M \Downarrow d \geq^* c$.

Proof: An easy induction on the length of $M = M_1 \geq \ldots \geq M_k \geq c$ using Lemma 14. □

We now have enough machinery to prove the main correctness theorem.

Definition
Suppose $\Re(l'_1, \ldots, l'_k, \rho'_1, \ldots, \rho'_n, \sigma)$. Suppose $c_1, \ldots, c_k$ are closed canonical terms, $M_1, \ldots, M_n$ are closed terms, and $M'_1, \ldots, M'_n$ are typable terms. Suppose also that for all i, $\mathrm{dom}(\rho_i) = FV(M'_i)$. Then we say that

$$(c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma)$$

if the following conditions hold:

1. *For all $1 \leq j \leq k$, $c_j \geq^*$ $\mathsf{valofcell}(l'_j, \sigma)$.*
2. *For all $1 \leq j \leq n$, $M_j \geq^*$ $\mathsf{valof}(M'_j, \rho'_j, \sigma)$.*

Theorem 7
Suppose M is typable, $\mathrm{dom}(\rho) = FV(M)$, M' is closed, and $M' \geq^*$ $\mathsf{valof}(M, \rho, \sigma)$. Suppose also that $\Re(\bar{l}', \rho, \bar{\rho}', \sigma)$.

1. *If $M' \Downarrow c$, then there exist l', σ' such that $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$ and $c \geq^*$ $\mathsf{valofcell}(l', \sigma')$.*
2. *If $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, then there exists a c such that $M' \Downarrow c \geq^*$ $\mathsf{valofcell}(l', \sigma')$.*

Proof: We prove the following more general property: suppose

$$(c_1, \ldots, c_k, M, M_1, \ldots, M_n) \geq^* (l'_1, \ldots, l'_k, (M', \rho), (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma).$$

Then

1. If $M \Downarrow c$, then $\mathtt{interp}(M', \rho, \sigma) = (l', \sigma')$ and
$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma').$$
2. If $\mathtt{interp}(M', \rho, \sigma) = (l', \sigma')$, then $M \Downarrow c$ and
$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma').$$

The first part is proven by induction on the height of the proof of $M \Downarrow c$. We consider the cases for the core language and leave the cases for the PCF extensions to the reader. To ease the readability of the various cases, we can separate each induction case into two cases based on whether or not M' is a variable or a canonical form. The first of these cases can be seen immediately. If M' is a canonical form or variable, then the form of the rules guarantees that $\mathtt{interp}(M', \rho, \sigma)$ returns a result (l', σ') and $\mathsf{valofcell}(l', \sigma') = \mathsf{valof}(M', \rho, \sigma)$. Thus, by Lemma 15, it follows that $c \geq^* \mathsf{valofcell}(l', \sigma')$. For instance, if $M' = (\lambda x.\, P')$, then $\mathtt{interp}(M', \rho, \sigma) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P', \rho), \sigma) = (l', \sigma')$. Since $\Re(\bar{l'}, \rho, \bar{\rho'}, \sigma)$ and $l' \notin \mathsf{dom}(\sigma)$, the new cell l' in σ' cannot be reached from σ. Thus,

$$\mathsf{valofcell}(l', \sigma') = \mathsf{valof}(\lambda x.\, P', \rho, \sigma') = \mathsf{valof}(\lambda x.\, P', \rho, \sigma)$$

and the required property holds.

If, on the other hand, M' is not a variable or canonical form, then there is some interpretation required in the reference-counting interpreter. Now we divide into cases depending on the last rule used in the proof of $M \Downarrow c$.

1. $M = (P\ Q)$, where $P \Downarrow (\lambda x.\, N)$, $Q \Downarrow d$, and $N[x := d] \Downarrow c$. The only case to consider is $M' = (P'\ Q')$, where $P \geq^* \mathsf{valof}(P', \rho, \sigma)$ and $Q \geq^* \mathsf{valof}(Q', \rho, \sigma)$. Since M is typable, the free variables of P and Q are disjoint. The first step is to evaluate the operator and operand. By induction, it follows that $\mathtt{interp}(P', \rho \mid P', \sigma) = (l_0, \sigma_0)$ and

$$\begin{aligned}&((\lambda x.\, N), c_1, \ldots, c_k, Q, M_1, \ldots M_n)\\ &\quad \geq^* (l_0, l'_1, \ldots, l'_k, (Q', \rho \mid Q'), (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma_0)\end{aligned}$$

Next evaluate the operand. By the induction, $\mathtt{interp}(Q', \rho \mid Q', \sigma) = (l_1, \sigma_1)$ and

$$\begin{aligned}&((\lambda x.\, N), d, c_1, \ldots, c_k, M_1, \ldots M_n)\\ &\quad \geq^* (l_0, l_1, l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma_1)\end{aligned}$$

We need to show that l_0 really holds a closure. Note that

$$(\lambda x.\, N) \geq^* \mathsf{valofcell}(l_0, \sigma_1)$$

and hence, by Lemma 13, $(\lambda x.\, N)$ and $\mathsf{valofcell}(l_0, \sigma_1)$ must have the same shape. By Theorem 2, $\sigma_1(l_0)$ is not a thunk (condition $\Re 6$). Thus,

$$\sigma_1(l_0) = \mathsf{closure}(\lambda x.\, N', \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N', \rho').$$

To evaluate the application, there are two subcases: either $\mathsf{refcount}(l_0, \sigma_1) = 1$ or $\mathsf{refcount}(l_0, \sigma_1) > 1$. We do the first case and leave the other case to the reader. If $\mathsf{refcount}(l_0, \sigma_1) = 1$, then $\Re(\bar{l'}, \rho'[x \mapsto l_1], \bar{\rho'}, \mathsf{dec}(l_0, \sigma_1))$ by laws A1 and E. It is easy to see that

$$\begin{aligned}&(c_1, \ldots, c_k, N[x := d], M_1, \ldots M_n)\\ &\quad \geq^* (l'_1, \ldots, l'_k, (N', \rho'[x \mapsto l_1]), (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \mathsf{dec}(l_0, \sigma_1)).\end{aligned}$$

Thus, by induction,

$$\mathtt{interp}(N', \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1)) = (l', \sigma')$$
$$(c, c_1, \dots, c_k, M_1, \dots M_n) \geq^* (l', l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma').$$

2. $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = N_1, \dots, x_m = N_m)$, $N_i \Downarrow d_i$, and the canonical form $c = (\mathsf{store}\ N[x_1, \dots, x_m := d_1, \dots, d_m])$. The only case to consider is $M' = (\mathsf{store}\ N'\ \mathsf{where}\ x_1 = N'_1, \dots, x_m = N'_m)$, where

$$\begin{aligned}&(c_1, \dots, c_k, N_1, \dots, N_m, M_1, \dots M_n)\\ &\quad \geq^* (l'_1, \dots, l'_k, (N'_1, \rho \mid N'_1), \dots, (N'_m, \rho \mid N'_m), (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma).\end{aligned}$$

(This makes sense because the free variables of each N'_i are disjoint.) Thus, by induction, $\mathtt{interp}(N'_1, \rho \mid N'_1, \sigma) = (l_1, \sigma_1)$ and

$$\begin{aligned}&(d_1, c_1, \dots, c_k, N_2, \dots, N_m, M_1, \dots M_n)\\ &\quad \geq^* (l_1, l'_1, \dots, l'_k, (N'_2, \rho \mid N'_2), \dots, (N'_m, \rho \mid N'_m), (M'_1, \rho'_1), \dots, \sigma_1)\end{aligned}$$

Using similar repeated applications of the inductive hypothesis, we end up with

$$\begin{aligned}&(d_1, \dots, d_m, c_1, \dots, c_k, M_1, \dots M_n)\\ &\quad \geq^* (l_1, \dots, l_m, l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma_m).\end{aligned}$$

Finally, let

$$\begin{aligned}&\rho' = \emptyset[x_1, \dots, x_m \mapsto l_1, \dots, l_m]\\ &\mathsf{new}(\mathsf{thunk}(N', \rho'), \sigma_m) = (l_{m+1}, \sigma_{m+1})\\ &\mathsf{new}(\mathsf{susp}(l_{m+1}), \sigma_{m+1}) = (l', \sigma')\end{aligned}$$

It is easy to see that

$$\mathtt{interp}(M', \rho, \sigma) = (l', \sigma')$$
$$(c, c_1, \dots, c_k, M_1, \dots M_n) \geq^* (l', l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma')$$

as desired.

3. $M = (\mathsf{fetch}\ N)$, where $N \Downarrow (\mathsf{store}\ Q)$ and $Q \Downarrow c$. The only case to consider is $M' = (\mathsf{fetch}\ N')$. By induction,

$$\mathtt{interp}(N', \rho, \sigma) = (l_0, \sigma_0)$$
$$((\mathsf{store}\ Q), c_1, \dots, c_k, M_1, \dots M_n) \geq^* (l_0, l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma_0)$$

By Theorem 2, $\mathfrak{R}(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Since $(\mathsf{store}\ Q) \geq^* \mathsf{valofcell}(l_0, \sigma_0)$ and $\sigma_0(l_0)$ must be a non-thunk by $\mathfrak{R}6$, it follows from Lemma 13 that $\sigma_0(l_0) = \mathsf{susp}(l_1)$ or $\mathsf{rec}(l_1, f)$ and $Q \geq^* \mathsf{valofcell}(l_1, \sigma_0)$. We consider only the case when $\sigma_0(l_0)$ is $\mathsf{susp}(l_1)$ and leave the other case to the reader. There are two subcases:

(a) $\sigma_0(l_1) = \mathsf{thunk}(Q', \rho')$. There are two subcases:

i $\mathsf{refcount}(l_0, \sigma_0) = 1$. First, note that neither l_0 nor l_1 is reachable from ρ'—if either were, the state $S = (l_0, \bar{l}', \bar{\rho}', \sigma_0)$ would have a cycle that was not composed solely of a rec and a recclosure, contradicting the regularity of the state S. By law A2, $\mathfrak{R}(\bar{l}', \rho', \bar{\rho}', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0)))$. Thus,

$$\begin{aligned}&(c_1, \dots, c_k, Q, M_1, \dots M_n)\\ &\quad \geq^* (l'_1, \dots, l'_k, (Q', \rho'), (M'_1, \rho'_1), \dots, \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))).\end{aligned}$$

It then follows by induction that

$$\begin{gathered}\mathtt{interp}(Q', \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))) = (l', \sigma') \\ (c, c_1, \dots, c_k, M_1, \dots M_n) \geq^* (l', l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma')\end{gathered}$$

as desired.

ii $\mathsf{refcount}(l_0, \sigma_0) > 1$. Let $\sigma'_0 = \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0]))$. First, by law U2, $\Re(\bar{l}', \rho', \bar{\rho}', \sigma'_0)$. Second, note that neither l_0 nor l_1 is reachable from ρ'—if either were, there would be an illegal cycle in the memory graph induced by $(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. It follows that

$$Q \geq^* \mathsf{valof}(Q', \rho', \sigma'_0).$$

Third, note that there exist contexts $C_1, \dots, C_n$ and closed terms $N_{i,j}$ such that $M_i = C_i[N_{i,1}, \dots, N_{i,l_i}]$, $N_{i,l_i} \geq^* \mathsf{valof}(Q', \rho', \sigma'_0)$, and $P_i = C_i[0, \dots, 0] \geq^* \mathsf{valof}(M'_i, \rho_i, \sigma'_0)$. Thus, it follows that

$$\begin{aligned}&(c_1, \dots, c_k, Q, P_1, \dots P_n) \\ &\qquad \geq^* (l'_1, \dots, l'_k, (Q', \rho'), (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma'_0)\end{aligned}$$

and hence by induction

$$\begin{gathered}\mathtt{interp}(Q', \rho', \sigma'_0) = (l_2, \sigma_1) \\ (c, c_1, \dots, c_k, P_1, \dots P_n) \geq^* (l_2, l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma_1)\end{gathered}$$

Let $(l', \sigma') = (l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]))$. Since l_0 is not reachable from ρ' in the store $\mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])$, by Theorem 2 it follows that l_0 is not reachable from l_2 in σ_1. Thus, $c \geq^* \mathsf{valofcell}(l', \sigma')$. Also, note that $M_i = C_i[N_{i,1}, \dots, N_{i,l_i}] \geq^* \mathsf{valof}(M'_i, \rho'_i, \sigma')$. Thus,

$$\begin{gathered}\mathtt{interp}(M', \rho, \sigma) = (l', \sigma') \\ (c, c_1, \dots, c_k, M_1, \dots M_n) \geq^* (l', l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma')\end{gathered}$$

as desired.

(b) $\sigma_0(l_1) \neq \mathsf{thunk}(R, \rho')$. This case is straightforward and left to the reader.

4. $M = (\mathsf{share}\ x, y\ \mathsf{as}\ P\ \mathsf{in}\ Q)$, where $P \Downarrow d$ and $Q[x, y := d] \Downarrow c$. Then the only case to consider is $M' = (\mathsf{share}\ x, y\ \mathsf{as}\ P'\ \mathsf{in}\ Q')$, where $P \geq^* \mathsf{valof}(P', \rho \mid P', \sigma)$ and $Q \geq^* \mathsf{valof}(Q', \rho \mid Q', \sigma)$. Since M' is typable, the free variables of P' and Q' are disjoint. It follows by induction that $\mathtt{interp}(P', \rho \mid P', \sigma) = (l_0, \sigma_0)$ and

$$\begin{aligned}&(d, c_1, \dots, c_k, Q, M_1, \dots M_n) \\ &\qquad \geq^* (l_0, l'_1, \dots, l'_k, (Q', \rho \mid Q'), (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma_0)\end{aligned}$$

By laws I1 and E, $\Re(\bar{l}', (\rho \mid Q')[x, y \mapsto l_0], \bar{\rho}', \mathsf{inc}(l_0, \sigma_0))$. Thus, since

$$\begin{aligned}&(c_1, \dots, c_k, Q[x, y := d], M_1, \dots M_n) \\ &\qquad \geq^* (l'_1, \dots, l'_k, (Q', (\rho \mid Q')[x, y \mapsto l_0]), (M'_1, \rho'_1), \dots, \mathsf{inc}(l_0, \sigma_0))\end{aligned}$$

it follows by induction that

$$\begin{gathered}\mathtt{interp}(Q', (\rho \mid Q')[x, y \mapsto l_0]) = (l', \sigma') \\ (c, c_1, \dots, c_k, Q, M_1, \dots M_n) \geq^* (l', l'_1, \dots, l'_k, (M'_1, \rho'_1), \dots, (M'_n, \rho'_n), \sigma')\end{gathered}$$

as desired.

5. $M = (\textsf{dispose } P \textsf{ before } Q)$, where $Q \Downarrow c$. This case is similar to the previous case and hence omitted.

This completes the proof of the first part. The second part is proven by induction on the number of calls to `interp`. We consider the cases for the core of the language and leave the cases for the PCF extensions to the reader.

1. $M' = x$. Then $\texttt{interp}(M', \rho, \sigma) = (\rho(x), \sigma) = (l', \sigma')$. Note that
$$\textsf{valofcell}(l', \sigma') = \textsf{valof}(M', \rho, \sigma),$$
and hence $M \geq^* \textsf{valofcell}(l', \sigma')$. Since $\Re(\bar{l}', \rho, \bar{\rho}', \sigma)$, $\sigma'(l') = \sigma(\rho(x))$ must be a non-thunk by $\Re 6$, and hence $\textsf{valofcell}(l', \sigma') = d$ where d is a canonical form. Thus, by Lemma 15, $M \Downarrow c \geq^* d$ and
$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma').$$
2. $M' = (\lambda x.\, N')$. Similar to the previous case.
3. $M' = (P'\ Q')$. Since $\texttt{interp}(M', \rho, \sigma) = (l', \sigma')$, it follows that
$$\begin{array}{l} \texttt{interp}(P', \rho \mid P', \sigma) = (l_0, \sigma_0) \\ \texttt{interp}(Q', \rho \mid Q', \sigma_0) = (l_1, \sigma_1) \\ \sigma_1(l_0) = \textsf{closure}(\lambda x.\, N', \rho') \text{ or } \textsf{recclosure}(\lambda x.\, N', \rho') \end{array}$$
Since $M \geq^* \textsf{valof}(M', \rho, \sigma)$, it must be that $M = (P\ Q)$ for some closed P and Q, where $P \geq^* \textsf{valof}(P', \rho, \sigma)$ and $Q \geq^* \textsf{valof}(Q', \rho, \sigma)$. By induction, $P \Downarrow d'$, $Q \Downarrow d$, and
$$(d', d, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l_0, l_1, l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma_1)$$
Since $\sigma_1(l_0)$ is a closure, it follows that $\textsf{valofcell}(l_0, \sigma_1)$ must be a λ-abstraction, and so by Lemma 13 it follows that $d' = (\lambda x.\, N)$. If $\textsf{refcount}(l_0, \sigma_1) = 1$, then
$$N[x := d] \geq^* \textsf{valof}(N', \rho'[x \mapsto l_1], \textsf{dec}(l_0, \sigma_1)).$$
If, on the other hand, $\textsf{refcount}(l_0, \sigma_1) > 1$, then
$$N[x := d] \geq^* \textsf{valof}(N', \rho'[x \mapsto l_1], \textsf{inc-env}(\rho', \textsf{dec}(l_0, \sigma_1))).$$
In either case, it follows by induction that $N[x := d] \Downarrow c$ and
$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l'_1, \ldots, l'_k, (M'_1, \rho'_1), \ldots, (M'_n, \rho'_n), \sigma'),$$
and hence also $M \Downarrow c$ as desired.
4. $M' = (\textsf{store } N' \textsf{ where } x_1 = N'_1, \ldots, x_m = N'_m)$. Since M' evaluates,
$$\begin{array}{l} \texttt{interp}(N'_1, \rho \mid N'_1, \sigma) = (l_1, \sigma_1) \\ \qquad\vdots \\ \texttt{interp}(N'_m, \rho \mid N'_m, \sigma_{m-1}) = (l_m, \sigma_m) \\ \rho' = \emptyset[x_1, \ldots, x_m \mapsto l_1, \ldots, l_m] \\ (l_{m+1}, \sigma_{m+1}) = \textsf{new}(\textsf{thunk}(N', \rho'), \sigma_m) \\ (l', \sigma') = \textsf{new}(\textsf{susp}(l_{m+1}), \sigma_{m+1}) \end{array}$$

Since $M \geq^* \mathsf{valof}(M', \rho, \sigma)$, it follows that

$$M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = N_1, \ldots, x_m = N_m)$$

and $N \geq^* \mathsf{valof}(N', \emptyset, \sigma)$ and $N_i \geq^* \mathsf{valof}(N_i', \rho \mid N_i', \sigma)$. By induction, $N_1 \Downarrow d_1$ and

$$\begin{aligned}&(d_1, c_1, \ldots, c_k, N_2, \ldots, M_1, \ldots M_n)\\ &\quad \geq^* (l_1, l_1', \ldots, l_k', (N_2', \rho \mid N_2'), \ldots, (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma_1).\end{aligned}$$

Extending the inductive hypothesis, $N_i \Downarrow d_i$ for all i and

$$\begin{aligned}&(d_1, \ldots, d_m, c_1, \ldots, c_k, M_1, \ldots M_n)\\ &\quad \geq^* (l_1, \ldots, l_m, l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma_1).\end{aligned}$$

Let $c = (\mathsf{store}\ N[x_1, \ldots, x_m := d_1, \ldots, d_m])$; then $M \Downarrow c$ and

$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma').$$

5. $M' = (\mathsf{fetch}\ P')$. Since M' evaluates, $\mathtt{interp}(P', \rho, \sigma) = (l_0, \sigma_0)$; by Theorem 2, $\mathfrak{R}(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Since $M \geq^* \mathsf{valof}(M', \rho, \sigma)$, it follows that $M = (\mathsf{fetch}\ P)$ for some P and $P \geq^* \mathsf{valof}(P', \rho, \sigma)$. By induction, $P \Downarrow d$ and

$$(d, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l_0, l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma_0).$$

Note that $\sigma_0(l_0) = \mathsf{susp}(l_1)$ or $\mathsf{rec}(l_1, f)$; we consider the first case here and leave the other to the reader. Since $\sigma_0(l_0)$ is a suspension, $\mathsf{valofcell}(l_0, \sigma_0) = (\mathsf{store}\ Q')$ for some Q'. It follows from Lemma 13 that $d = (\mathsf{store}\ Q)$ for some Q. There are now two subcases:

(a) $\sigma_0(l_1) = \mathsf{thunk}(R', \rho')$. There are two subcases depending on the reference count of l_0.

i $\mathsf{refcount}(l_0, \sigma_0) = 1$. Then $\mathtt{interp}(R', \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))) = (l', \sigma')$. Since the state $S = (l_0, \bar{l}', \bar{\rho}', \sigma_0)$ is regular, it follows that l_0 is not reachable from ρ'—otherwise, there would be an illegal cycle in the memory graph induced by S. Thus,

$$\begin{aligned}&(c_1, \ldots, c_k, Q, M_1, \ldots M_n)\\ &\quad \geq^* (l_1', \ldots, l_k', (R', \rho'), (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))).\end{aligned}$$

By induction, $Q \Downarrow c$ and

$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma').$$

ii $\mathsf{refcount}(l_0, \sigma_0) > 1$. Let $\sigma_0' = \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0]))$; then

$$\begin{aligned}\mathtt{interp}(R', \rho', \sigma_0') &= (l_2, \sigma_1)\\ (l', \sigma') &= (l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]))\end{aligned}$$

First, by law U2, $\mathfrak{R}(\bar{l}', \rho', \bar{\rho}', \sigma_0')$. Second, neither l_0 nor l_1 is reachable from ρ'—otherwise, there would be an illegal cycle in the memory graph induced by $(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Third, note that there exist contexts $C_1, \ldots, C_n$ and closed terms $N_{i,j}$ such that $M_i = C_i[N_{i,1}, \ldots, N_{i,l_i}]$,

$N_{i,l_i} \geq^* \mathsf{valof}(Q', \rho', \sigma_0')$, and $P_i = C_i[0, \ldots, 0] \geq^* \mathsf{valof}(M_i', \rho_i, \sigma_0')$. Thus, it follows that

$$\begin{aligned}&(c_1, \ldots, c_k, Q, P_1, \ldots P_n)\\ &\qquad \geq^* (l_1', \ldots, l_k', (R', \rho'), (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma_0').\end{aligned}$$

By induction, $Q \Downarrow c$ and

$$(c, c_1, \ldots, c_k, P_1, \ldots P_n) \geq^* (l_2, l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma_1).$$

Since l_0 is not reachable from ρ' in the store $\mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])$, by Theorem 2 it follows that l_0 is not reachable from l_2 in σ_1. Thus, $c \geq^* \mathsf{valofcell}(l', \sigma')$. Note that $M_i = C_i[N_{i,1}, \ldots, N_{i,l_i}] \geq^* \mathsf{valof}(M_i', \rho_i', \sigma')$. Thus, $M \Downarrow c$ and

$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma').$$

(b) $\sigma_0(l_1) \neq \mathsf{thunk}(R, \rho')$. This case is straightforward and left to the reader.

6. $M' = (\mathsf{share}\ x, y\ \mathsf{as}\ P'\ \mathsf{in}\ Q')$. Since M' evaluates,

$$\begin{aligned}&\mathtt{interp}(P', \rho \mid P', \sigma) = (l_0, \sigma_0)\\ &\mathtt{interp}(Q', (\rho \mid Q')[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0)) = (l', \sigma')\end{aligned}$$

Since $M \geq^* \mathsf{valof}(M', \rho, \sigma)$, it follows that $M = (\mathsf{share}\ x, y\ \mathsf{as}\ P\ \mathsf{in}\ Q)$ for some terms $P \geq^* \mathsf{valof}(P', \rho \mid P', \sigma)$ and $Q \geq^* \mathsf{valof}(Q', \rho \mid Q', \sigma)$. By induction, $P \Downarrow d$ and

$$\begin{aligned}&(d, c_1, \ldots, c_k, Q, M_1, \ldots M_n)\\ &\qquad \geq^* (l_0, l_1', \ldots, l_k', (Q', \rho \mid Q'), (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma_0).\end{aligned}$$

Thus,

$$\begin{aligned}&(c_1, \ldots, c_k, Q[x, y := d], M_1, \ldots M_n)\\ &\qquad \geq^* (l_1', \ldots, l_k', (Q', (\rho \mid Q')[x, y \mapsto l_0]), (M_1', \rho_1'), \ldots, \mathsf{inc}(l_0, \sigma_0)).\end{aligned}$$

and so by induction, $Q[x, y := d] \Downarrow c$ and

$$(c, c_1, \ldots, c_k, M_1, \ldots M_n) \geq^* (l', l_1', \ldots, l_k', (M_1', \rho_1'), \ldots, (M_n', \rho_n'), \sigma').$$

7. $M' = (\mathsf{dispose}\ P\ \mathsf{before}\ Q)$. Similar to the previous case and hence omitted.

This completes the proof of the second claim and the proof of the theorem. □

References

Abramsky, S. (1993) Computational interpretations of linear logic. *Theoretical Computer Science* **111**:3–57.

Abramsky, S. and Hankin, C. (eds.) (1987) *Abstract Interpretation of Declarative Languages.* Ellis Horwood.

Appel, A. (1992) *Compiling with Continuations.* Cambridge University Press.

Baker, H. G. (1978) List processing in real time on a serial computer. *Communications of the ACM* **21**(7):11–20.

Benton, N., Bierman, G., de Paiva, V. and Hyland, M. (1992) Term assignment for intuitionistic linear logic. Technical Report 262, University of Cambridge Computer Laboratory.

Benton, N., Bierman, G., de Paiva, V. and Hyland, M. (1993) A term calculus for intuitionistic linear logic. In *Typed Lambda Calculi and Applications: Lecture Notes in Computer Science vol 664*, pp. 75–90. Springer-Verlag.

Bloss, A., Hudak, P. and Young, J. (1988) An optimizing compiler for a modern functional programming language. *Computer Journal* **31**(6).

Breazu-Tannen, V., Gunter, C. and Scedrov, A. (1990) Computing with coercions. *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 44–60.

Chirimar, J., Gunter, C. A. and Riecke, J. G. (1992) Proving memory management invariants for a language based on linear logic. *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 139–150.

Collins, G. E. (1960) A method for overlapping and erasure of lists. *Communications of the ACM* **3**(12): 655–657.

Despeyroux, J. (1986) Proof of translation in natural semantics. *Proceedings, Symposium on Logic in Computer Science*. IEEE.

Deutsch, L. P. and Bobrow, D. G. (1976) An efficient, incremental, automatic garbage collector. *Communications of the ACM* **19**(9): 522–526.

Filinski, A. (1992) Linear continuations. *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 27–38. ACM.

Gabriel, R. P. (1985) *Performance and Evaluation of Lisp Systems*. MIT Press.

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50**: 1–102.

Goldberg, B. and Gloger, M. (1992) Polymorphic type reconstruction for garbage collection without tags. *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 53–65. ACM.

Guzmán, J. C. and Hudak, P. (1990) Single-threaded polymorphic lambda calculus. *Proceedings 5th Annual IEEE Symposium on Logic in Computer Science*, pp. 333–343.

Holmström, S. (1988) Linear functional programming. In *Implementation of Lazy Functional Languages*, T. Johnsson, S. Peyton-Jones and K. Karlsson (eds.), pp. 13–32.

Howard, W. A. (1980) The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Hindley and J. Seldin (eds.), pp. 479–490. Academic Press.

Hudak, P. (1987) A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*, pp. 45–62. Chichester: Ellis Horwood. (Preliminary version appeared in *Proceedings 1986 ACM Conference on LISP and Functional Programming*, August 1986, pp. 351–363).

Kahn, G. (1987) Natural semantics. *Proceedings Symposium on Theoretical Aspects of Computer Science: Lecture Notes in Computer Science vol 247*. Springer-Verlag.

Lafont, Y. (1988) The linear abstract machine. *Theoretical Computer Science* **59**: 157–180.

Launchbury, J. (1993) A natural semantics for lazy evaluation. *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 144–154. ACM.

Lawall, J. L. and Danvy, O. (1993) Separating stages in the continuation-passing style transformation. *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 124–136. ACM.

Lincoln, P. and Mitchell, J. C. (1992) Operational aspects of linear lambda calculus. *Proceedings 7th Annual IEEE Symposium on Logic in Computer Science*, pp. 235–247.

Mackie, I. (1991) Lilac: A functional programming language based on linear logic. Master's thesis, Imperial College, University of London.

Milner, R. and Tofte, M. (1991) *Commentary on Standard ML.* MIT Press.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* MIT Press.

O'Hearn, P. W. (1991) Linear logic and interference control (preliminary report). In *Category Theory and Computer Science: Lecture Notes in Computer Science vol 530*, D. H. Pitt (ed.), pp. 74–93. Springer-Verlag.

Plotkin, G. D. (1975) Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* **1**:125–159.

Plotkin, G. D. (1977) LCF considered as a programming language. *Theoretical Computer Science* **5**: 223–257.

Purushothaman, S. and Seaman, J. (1991) An adequate operational semantics of sharing in lazy evaluation. *Technical Report PSU-CS-91-18*, Pennsylvania State University.

Scott, D. S. (1993) A type theoretical alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Science* **121**: 411–440. (Published version of unpublished manuscript, Oxford University, 1969.)

Wadler, P. (1990) Linear types can change the world! In *Programming Concepts and Methods*, M. Broy and C. B. Jones (eds.). North Holland.

Wadler, P. (1991) Is there a use for linear logic? *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 255–273. ACM.

Wadler, P. (1993) A syntax for linear logic. *Workshop on Mathematical Foundations of Programming Language Semantics*, New Orleans, LA.

Wand, M. and Oliva, D. P. (1992) Proving the correctness of storage representations. In *Lisp and Functional Programming*, W. Clinger (ed.), pp. 151–160. ACM.

Wise, D. S., Hess, C., Hunt, W. and Ost, E. (1992) Uniprocessor performance of reference-counting hardware heap. Unpublished manuscript.