A Semantic Model of Reference Counting and its Abstraction

Paul Hudak

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520

1 Introduction

Most interpreters for functional languages (as well as Lisp) employ at some level in the implementation a notion of *sharing*, whether manifested indirectly through an environment or directly via pointers. Sharing is essential to an efficient implementation, saving time by not recomputing values, and saving space by having only one copy of each value. From the perspective of lambda calculus, sharing arises whenever a beta-reduction results in substitution for more than one occurrence of a particular bound variable.

Although sharing is pervasive at the implementation level, it is rarely seen expressed in any formal context. This is unfortunate, since knowing certain sharing properties of a program *at compile time* allows one to perform a variety of useful optimizations. Perhaps the most important of these arises in languages supporting aggregate data structures that are updated "functionally," where knowing that an aggregate has only *one reference* allows the update to be done *destructively* rather than by copying [6, 8, 11]. A special case of this situation arises in the construction of semantics-directed compilers, where knowing sharing properties of store arguments is crucial to generating efficient code[10]. Other optimizations include eliminating reference count operations determined to be extraneous, performing "compile-time garbage collection" of objects whose extent can be determined statically[1], stack-allocating instead of heap-allocating activation records, and reusing activation records in tail-recursive calls.

In this paper we present a precise semantic model of reference counting for an applicative-order interpreter of a first-order functional language. Although a reference count is an operational concept, its semantics is expressed in a conventional denotational style. We also present an abstraction of (or approximation to) the model over finite domains, with a decideable inferencing algorithm. We demonstrate the usefulness of the abstraction by applying it to some non-trivial programs that can be optimized based on the inferred reference counts.

The methodology that we use to model reference counts is also interesting in its own right, since it demonstrates some useful ideas about semantic program analysis. In particular, it demonstrates how a "non-standard" semantics may be "lifted" denotationally and expressed just as precisely as a

This research was supported in part by the National Science Foundation under Grants DCR-8403304 and DCR-8451415, and a Faculty Development Award from IBM. A preliminary version of this paper appeared in the Proceedings of the ACM Symposium on Lisp and Functional Programming, August 1986, pp. 351-363.

standard semantics. It also represents a classical use of *abstract interpretation* [3, 8], demonstrating the utility of that approach even on a non-standard semantics. Finally, perhaps the most interesting aspect of the methodology, we introduce the notion of a *collecting interpretation of expressions* which allows one to infer properties of "program points" rather than functions as a whole.

2 Preliminaries

We adopt the following conventional notation. Double brackets are used to surround syntactic objects, as in $\mathcal{E}[\![exp]\!]$. Square brackets are used for environment update, as in $env[e/x]$. The notation $env[e_i/x_i]$ is shorthand for $env[e_1/x_1 \ldots e_n/x_n]$, where the subscript bounds are inferred from context. Similarly, "new" environments are created by $[e_1/x_1 \ldots e_n/x_n]$, being shorthand for $\perp[e_1/x_1 \ldots e_n/x_n]$. Angle brackets are used for tupling, as in $\langle e_1, e_2, e_3\rangle$. The notation $A* \to B$ denotes the domain $B + (A \to B) + (A \to A \to B) + \cdots$. $P(S)$ denotes the powerset of S. When necessary, domains are assumed to be chain-complete partial orders with a unique least element (i.e., "pointed" cpos). We write "$d \in D = Exp$" to define the domain (or set) D with "canonical" element d, and whose bottom element is denoted $\perp_D$.

Our language takes the form of mutually-recursive first-order recursion equations with constants. Its abstract syntax is given by:

$$
\begin{array}{rcll}
c, p & \in & Con & \text{(constants)} \\
x & \in & Bv & \text{(bound variables)} \\
f & \in & Fv & \text{(function variables)} \\
body, e & \in & Exp \text{ where} & \\
 & & e ::= c \mid x \mid p(e_1 \ldots e_n) \mid f(e_1 \ldots e_n) & \text{(expressions)} \\
pr & \in & Prog \text{ where} & \\
 & & pr ::= \{ \quad f_1(x_{11} \ldots x_{1k_1}) = body_1 & \text{(programs)} \\
 & & \qquad\qquad f_2(x_{21} \ldots x_{2k_2}) = body_2 & \\
 & & \qquad\qquad \cdots & \\
 & & \qquad\qquad f_n(x_{n1} \ldots x_{nk_n}) = body_n\} &
\end{array}
$$

For the remainder of the paper we generally ignore the differing numbers of arguments to functions, and refer to a "generic" function as having n arguments.

A standard semantics for this language can be given as follows: Let D be some suitable domain of basic values, and define two environments, one for bound variables, the other for function variables:

$$
\begin{array}{rcl}
fve \in Fve & = & Fv \to D* \to D \\
bve \in Bve & = & Bv \to D
\end{array}
$$

Now define a semantic function $\mathcal{E}_p$ that gives meaning to programs, and a function $\mathcal{E}$ that gives

meaning to expressions in a given bound and function variable environment:

$$\begin{array}{l}\mathcal{E}_p : \ Prog \to Fve \\ \mathcal{E} : \ Exp \to Fve \to Bve \to D \\ \mathcal{K} : \ Con \to D* \to D \quad \text{(assumed given)}\end{array}$$

$$\begin{array}{rcl}\mathcal{E}_p[\![\ \{f_i(x_1...x_n) = body_i\}\ ]\!] = fve \text{ whererec} & & \\ fve & = & [\ strict(\lambda y_1...y_n.\ \mathcal{E}[\![body_i]\!]\ fve\ [y_j/x_j])\ /\ f_i\ ]\end{array}$$

$$\begin{array}{rcl}\mathcal{E}[\![c]\!]\ fve\ bve & = & \mathcal{K}[\![c]\!] \\ \mathcal{E}[\![x]\!]\ fve\ bve & = & bve[\![x]\!] \\ \mathcal{E}[\![p(e_1...e_n)]\!]\ fve\ bve & = & \mathcal{K}[\![p]\!]\ (\mathcal{E}[\![e_1]\!]\ fve\ bve)\ldots(\mathcal{E}[\![e_n]\!]\ fve\ bve) \\ \mathcal{E}[\![f_i(e_1...e_n)]\!]\ fve\ bve & = & fve[\![f_i]\!](\mathcal{E}[\![e_1]\!]\ fve\ bve)\ldots(\mathcal{E}[\![e_n]\!]\ fve\ bve)\end{array}$$

where $\lambda y_1...y_n.exp$ is assumed to be just exp when $n = 0$, and *strict* is a function that "strictifies" its functional argument.[1]

Note that the meaning of a program is the meaning of its top-level functions, which we capture through $\mathcal{E}_p$ in the environment fve. For simplicity we assume that the first function f_1 takes no arguments, and a program is "run" by calling f_1. More specifically, to run a program $pr = [\![\ \{f_i(x_1...x_n) = body_i\}\ ]\!]$ is to evaluate $\mathcal{E}_p\ pr\ [\![f_1]\!]$.

3 A Formal Semantics of Reference Counting

There are two operational notions whose understanding helps clarify the semantics of reference counting.

First, an object needed in more than one place is typically handled by an interpreter in one of two ways: either the *object itself* is copied, or a *pointer* to the object is copied. We refer to these two methods as *pass-by-value* and *pass-by-reference*, respectively.[2] It is our goal to keep track of how many copies of a given pointer are "active" at any given time during program execution – that number is referred to as the *reference count* of the object they point to.

We shall ignore the rationale for using either pass-by-value or pass-by-reference, except to point out that some values can be represented as compactly as a pointer, and can thus be copied as easily as copying a pointer. Other values may occupy several words of memory and are thus better shared using pointers.

The second operational notion is that *a reference count operation is in essence a form of side-effect*, and thus we should expect our semantics to require techniques similar to those for dealing with side-effects in a conventional language. In particular, we should expect to need a *store*, whereas in a standard interpretation, such as that given earlier, a simple bound variable environment with no store suffices.

[1] That is, $(strictf)$ is a function just like f but that returns $\perp$ if applied to any argument that is $\perp$. It is similar to Stoy's use of the same function [12].

[2] These terms have traditionally been used to describe parameter-passing mechanisms in procedure calls, but here we attribute the term to the objects themselves.

To see that a reference count operation is a form of side-effect, consider the expression $E =$ "if *pred* then *con* else *alt*", where each subexpression has one occurrence of x. If there are no other references to x, its reference count can be thought of as 3 prior to the evaluation of E. But once x is used in *pred* its reference count will be 2, and once *pred* is completely evaluated either *con* or *alt* can be discarded, so the reference count will be 1. Eventually x will be used in evaluating either *con* or *alt*, at which point the count drops to 0. This behavior is clearly not "referentially transparent," and resembles a variable being side-effected.

Our model distinguishes only between pass-by-value and pass-by-reference objects; typical examples of the former include booleans and small integers, and of the latter include lists and arrays. To make our presentation more concrete we shall assume one particular type of each, and their associated operators: the domain of *integers* with a standard set of operators, and the domain of *arrays* with the operators *new_array* and *update*. The call $new\_array(n)$ returns a one-dimensional array of length n whose elements are all *nil*, and $update(a, i, x)$ returns an array just like a but whose ith element is x. For simplicty we restrict each element x to be pass-by-value.

3.1 Semantic Domains

Int			non − negative integers
Loc	$=$	$Int + \{none\}$	locations
Rc	$=$	Int	reference counts
Bve	$=$	$Bv \rightarrow Loc$	bound variable environments
St	$=$	$Loc \rightarrow Rc$	stores
Fve	$=$	$Fv \rightarrow Loc* \rightarrow St \rightarrow (Loc \times St)$	function variable environments

Locations and reference counts are modelled using the standard flat domain of non-negative integers. A bound variable environment maps bound variables to locations in the store. In a standard semantics a store would map locations to values in some standard domain, but for our purposes a store maps locations to reference counts, since that is all we are interested in. It is our intent that a store $st \in St$ "emulate" the real store of an interpreter, in that no cells are allocated in st unless they would have been in the interpreter's store. Unused, or "free" elements in the store can be recognized as those whose reference count is zero.

Elements of Fve give meaning to top-level functions in the program, analogous to Fve in the standard semantics.

3.2 Semantic Functions

The operational semantics that we wish to capture can be summarized as follows: A program begins with an empty store. As the program executes, certain operators (such as *new_array* and *update*) cause the allocation of new cells in the store whose initial reference count is one. Sharing results when an object is passed as an argument in a function call, since there may be more than one occurrence of the corresponding formal parameter. Upon a function call, the arguments are evaluated from left to right, and then the body of the function is evaluated with a new bound variable environment and updated store. The updated store is obtained by increasing the reference

count of each actual parameter that is a location by the number of occurrences of the corresponding formal parameter in the body of the function. Reference counts are ultimately decremented as occurrences of formal parameters are encountered and "used." This includes a special dereferencing mechanism for unevaluated arms of conditionals.

Let us now make all this precise. We introduce the following semantic functions:

$$\begin{array}{rl} \mathcal{R}_p: & Prog \to Fve \\ \mathcal{R}: & Exp \to Fve \to Bve \to St \to (Loc \times St) \\ \mathcal{K}: & Con \to Exp* \to Fve \to Bve \to St \to (Loc \times St) \\ \mathcal{D}: & Exp \to Bve \to St \to St \end{array}$$

$\mathcal{R}[\![exp]\!]\ fve\ bve\ st$ returns a pair, $\langle loc, st' \rangle$, which is interpreted as follows: st' is the modified store that results from evaluating exp in bve and st. If exp evaluates to a pass-by-reference object, then $st'(loc)$ contains the value (i.e., reference count) of exp; otherwise $loc = none$, meaning exp is pass-by-value. Note that the evaluation of a single expression may induce an arbitrary number of changes to the store as each subexpression is evaluated.

$\mathcal{K}$ is used in the definition of $\mathcal{R}$, and describes the behavior of primitive functions, which return in this semantics the same kind of object as $\mathcal{R}$. $\mathcal{K}$ uses $\mathcal{D}$ to give meaning to the conditional if; i.e., to "dereference" the unevaluated arm. Whereas $\mathcal{R}$ mimics the *evaluation* of an expression, $\mathcal{D}$ mimics the *dereferencing* of an expression, which in turn might dereference other (sub)expressions. Thus $\mathcal{D}[\![exp]\!]\ bve\ st$ returns the store st' that results from dereferencing exp.

3.3 Auxiliary Functions

We define the following auxiliary functions to simplify the semantics: *dec* and *inc* are used to decrement and increment, respectively, the reference count of a location in a store, *alloc* is used to allocate a new location in a store, and *inc_st* is used to create the new store in which a function body is to be evaluated. More formally:

$$\begin{array}{rcl} dec\ st\ loc & = & \text{if } (loc = none) \text{ then } st \text{ else } st[(st(loc) - 1)/loc] \\ inc\ st\ loc & = & \text{if } (loc = none) \text{ then } st \text{ else } st[(st(loc) + 1)/loc] \\ alloc\ st & = & \text{let } loc = \text{first location in } st \text{ such that } st(loc) = 0 \\ & & \text{in } \langle loc, st[1/loc] \rangle \\ inc\_st\ st\ i\ loc_1 ... loc_n & = & \text{let } \#x_j = \text{number of occurrences of } x_j \text{ in } body_i \\ & & \quad rc_j = st(loc_j) + \#x_j - 1, \quad j = 1...n \\ & & \text{in } st[rc_j/loc_j] \end{array}$$

Note that *inc_st* increments the reference count of each actual location by *one less* than the number of occurrences of the corresponding formal parameter. This is an optimization based on the way *inc_st* is used, which should become clear shortly.

3.4 Semantic Equations

The intuitive ideas described in Section 3.2 can now be expressed formally through $\mathcal{R}_p$ and $\mathcal{R}$:

$$\begin{aligned}
&\mathcal{R}_p[\![\ \{f_i(x_1...x_n) = body_i\}\ ]\!] = fve \text{ whererec} \\
&\quad fve = [\ strict(\lambda\ loc_1...loc_n\ st.\ \text{let } st' = inc_st\ st\ i\ loc_1...loc_n \\
&\qquad\qquad \text{in } \mathcal{R}[\![body_i]\!]\ fve\ [loc_j/x_j]\ st')\ /\ f_i\ ]
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}[\![c]\!]\ fve\ bve\ st &= \langle none, st\rangle \\
\mathcal{R}[\![x]\!]\ fve\ bve\ st &= \langle bve[\![x]\!], st\rangle \\
\mathcal{R}[\![p(e_1...e_n)]\!]\ fve\ bve\ st &= \mathcal{K}[\![p]\!][\![e_1]\!]...[\![e_n]\!]\ fve\ bve\ st \\
\mathcal{R}[\![f(e_1...e_n)]\!]\ fve\ bve\ st &= \text{let } \langle loc_1, st_1\rangle = \mathcal{R}[\![e_1]\!]\ fve\ bve\ st \\
&\qquad\qquad \langle loc_2, st_2\rangle = \mathcal{R}[\![e_2]\!]\ fve\ bve\ st_1 \\
&\qquad\qquad \ldots \\
&\qquad\qquad \langle loc_n, st_n\rangle = \mathcal{R}[\![e_n]\!]\ fve\ bve\ st_{n-1} \\
&\qquad \text{in } fve[\![f]\!]\ loc_1...loc_n\ st_n
\end{aligned}$$

Note that after all of the arguments in a call are evaluated, each has a reference count of at least one – this is because the call itself still has a "handle" on them. This handle is released once the function body begins execution, which is why *inc_st* increments the reference counts by *one less* than the number of bound variable occurrences. Thus if there are *no* occurrences of x_j then the object it is bound to will have its reference count decremented (since $\#x_j - 1 = -1$), which is precisely the intent.

What remains to be defined is the behavior of each primitive function. Throughout this paper we will express, for each semantics introduced, the meaning of three "canonical" primitive functions: the conditional if (which normally only evaluates one of its arms), arithmetic + (the canonical strict operator with left-to-right evaluation of its arguments), and *update* (the canonical "location generator"). Thus $\mathcal{K}$ is defined by:

$$\begin{aligned}
\mathcal{K}[\![if]\!]\ p\ c\ a\ fve\ bve\ st &= \text{let } \langle loc, st'\rangle = \mathcal{R}\ p\ fve\ bve\ st \\
&\qquad \text{in if } Oracle(p) \text{ then } (\mathcal{R}\ c\ fve\ bve\ (\mathcal{D}\ a\ bve\ st')) \\
&\qquad\qquad\qquad \text{else } (\mathcal{R}\ a\ fve\ bve\ (\mathcal{D}\ c\ bve\ st')) \\
\mathcal{D}[\![c]\!]\ bve\ st &= st \\
\mathcal{D}[\![x]\!]\ bve\ st &= dec\ st\ bve[\![x]\!] \\
\mathcal{D}[\![p(e_1...e_n)]\!]\ bve\ st &= \mathcal{D}[\![e_n]\!]\ bve\ (...\ (\mathcal{D}[\![e_2]\!]\ bve\ (\mathcal{D}[\![e_1]\!]\ bve\ st))) \\
\mathcal{D}[\![f(e_1...e_n)]\!]\ bve\ st &= \mathcal{D}[\![e_n]\!]\ bve\ (...\ (\mathcal{D}[\![e_2]\!]\ bve\ (\mathcal{D}[\![e_1]\!]\ bve\ st)))
\end{aligned}$$

The predicate is evaluated; if it is true, the alternate is dereferenced and the consequent is evaluated; otherwise the consequent is dereferenced and the alternate is evaluated. Note that the call to *Oracle* could be replaced with a call to $\mathcal{E}$ and suitable environments in the standard semantics. The oracle itself is used only for convenience, to avoid cluttering up this semantics with the standard one.

$$\begin{aligned}
\mathcal{K}[\![+]\!]\ e_1\ e_2\ fve\ bve\ st = &\ \text{let } \langle loc_1, st_1\rangle = \mathcal{R}\ e_1\ fve\ bve\ st \\
&\ \text{in } \mathcal{R}\ e_2\ fve\ bve\ st_1
\end{aligned}$$

The arguments are evaluated left-to-right. For simplicity we assume programs to be "well-typed," and thus the arguments to and the result of the call to + are pass-by-value, so no dereferencing of

the store is required.

$$\begin{aligned}\mathcal{K}[\![update]\!]\ a\ i\ x\ fve\ bve\ st = \ & \text{let } \langle loc_1, st_1\rangle = \mathcal{R}\ a\ fve\ bve\ st \\ & \quad \langle loc_2, st_2\rangle = \mathcal{R}\ i\ fve\ bve\ st_1 \\ & \quad \langle loc_3, st_3\rangle = \mathcal{R}\ x\ fve\ bve\ st_2 \\ & \text{in } alloc\ (dec\ st_3\ loc_1)\end{aligned}$$

The to-be-updated array a, index i, and new element x are evaluated, in that order. Then a new, updated array with reference count one is created using *alloc*. Note that loc_1's reference count is not decremented until both i and x are evaluated, because it is only then that the update to a can be made.

4 Abstract Interpretation of Reference Counting

The semantics presented thus far is *exact*, and thus evaluating a particular reference count may not terminate, any more than a program in the standard semantics would. For use by a compiler we must choose a suitable abstraction (i.e., approximation) that will guarantee termination yet still provide useful information about the true reference counts. For us a suitable abstraction is one in which the inferred reference count is *at least as great as the true one*; i.e., we wish to err on the side of thinking there are more pointers to an object than there actually are. In this section we methodically develop such an abstraction in which: (1) the base domains are abstracted to powersets of finite approximations, (2) the primitive functions are abstracted similarly, (3) an abstract interpretation is thus induced on $\mathcal{R}_p$ and $\mathcal{R}$, and finally (4) a collecting interpretation of expressions is developed.

4.1 Preliminaries

To set up things to come, we first give alternative versions of $\mathcal{R}_p$ and $\mathcal{R}$ in which:

- We assume that every expression in a program has a unique *label*. An expression *exp* with label *lab* is written *lab.exp*, and the syntactic functions *expr* and *label* are defined by: $expr[\![lab.exp]\!] = exp$, and $label[\![lab.exp]\!] = lab$. Using labels allows us to distinguish different occurrences of the same bound variable (or common subexpression), which will be required in Section 5.
- We use *powersets* to allow an expression to have multiple values (i.e., locations). This will be needed to express the abstract behavior of the conditional.

With these two changes in mind, the new functions, which we call $\hat{\mathcal{R}}_p$, $\hat{\mathcal{R}}$, $\hat{\mathcal{K}}$ and $\hat{\mathcal{D}}$, are defined by:

$$\begin{aligned}\hat{\mathcal{R}}_p :&\ Prog \to Fve \\ \hat{\mathcal{R}} :&\ Label \to Fve \to Bve \to St \to P(Aloc \times St) \\ \hat{\mathcal{K}} :&\ Con \to Label* \to Fve \to Bve \to St \to P(Aloc \times St) \\ \hat{\mathcal{D}} :&\ Label \to Bve \to St \to St\end{aligned}$$

where, for the moment, we leave the domains *Aloc* and *Src* undefined.

$$\hat{\mathcal{R}}_p[\![\ \{f_i(x_1...x_n) = body_i\}\ ]\!] = fve \text{ whererec}$$
$$fve = [\ strict(\lambda\ loc_1...loc_n\ st.\ \text{let } st' = inc\_st\ st\ i\ loc_1...loc_n$$
$$\text{in } \hat{\mathcal{R}}\ label[\![body_i]\!]\ fve\ [loc_j/x_j]\ st')\ /\ f_i\ ]$$

$$\begin{aligned}
\hat{\mathcal{R}}\ lab\ fve\ bve\ st = \text{case } expr(lab) \text{ of}&\\
[\![c]\!]: &\ \{\langle none, st\rangle\}\\
[\![x]\!]: &\ \{\langle bve[\![x]\!], st\rangle\}\\
[\![p(e_1...e_n)]\!]: &\ \hat{\mathcal{K}}[\![p]\!]\ label[\![e_1]\!]...label[\![e_n]\!]\ bve\ st\\
[\![f(e_1...e_n)]\!]: &\ \bigcup\{fve[\![f]\!]\ loc_1...loc_n\ st_n \mid \langle loc_1, st_1\rangle \in \hat{\mathcal{R}}\ label[\![e_1]\!]\ fve\ bve\ st,\\
&\qquad \langle loc_2, st_2\rangle \in \hat{\mathcal{R}}\ label[\![e_2]\!]\ fve\ bve\ st_1,\\
&\qquad \ldots\\
&\qquad \langle loc_n, st_n\rangle \in \hat{\mathcal{R}}\ label[\![e_n]\!]\ fve\ bve\ st_{n-1}\}
\end{aligned}$$

$$\begin{aligned}
\hat{\mathcal{D}}\ lab\ bve\ st = \text{case } expr(lab) \text{ of}&\\
[\![c]\!]: &\ st\\
[\![x]\!]: &\ dec\ st\ bve[\![x]\!]\\
[\![p(e_1...e_n)]\!]: &\ \hat{\mathcal{D}}\ label[\![e_n]\!]\ bve\ (...\ (\hat{\mathcal{D}}\ label[\![e_2]\!]\ bve\ (\hat{\mathcal{D}}\ label[\![e_1]\!]\ bve\ st)))\\
[\![f(e_1...e_n)]\!]: &\ \hat{\mathcal{D}}\ label[\![e_n]\!]\ bve\ (...\ (\hat{\mathcal{D}}\ label[\![e_2]\!]\ bve\ (\hat{\mathcal{D}}\ label[\![e_1]\!]\ bve\ st)))
\end{aligned}$$

If *Aloc* and *Src* are interpreted as just *Loc* and *Rc*, respectively (as defined earlier), and $\hat{\mathcal{K}}$ is imagined to be a function that suitably "mimics" $\mathcal{K}$, then these equations yield a semantics functionally *equivalent* to that given earlier, since $\hat{\mathcal{K}}$ would be deterministic and thus $\hat{\mathcal{R}}$ and $\hat{\mathcal{K}}$ would always return singleton sets. Our goal, of course, is to interpret the equations for $\hat{\mathcal{R}}_p$ and $\hat{\mathcal{R}}$ over *abstract* domains and likewise define an *abstract* version of $\hat{\mathcal{K}}$. This is done in the next two sections.

4.2 Abstract Domains

The domain of *sticky reference counts* is defined by:

$$Src = \{0, 1, \ldots, maxrc, \infty\}$$

where *maxrc* is an arbitrary positive integer. The idea is that if a reference count ever exceeds *maxrc*, it "jumps" to infinity and "sticks" there, never to decrease again.[3] Justification for using this domain comes from empirical studies (at least of Lisp programs [2]) that indicate that objects are generally not shared very much, so that choosing a suitably high value for *maxrc* will give fairly accurate results most of the time. As with *Rc*, *Src* is a flat domain. We define the following primitive operations on the elements:

$$\begin{aligned}
x \oplus n &= \text{if } (x + n > maxrc) \text{ then } \infty \text{ else } (x + n)\\
x \ominus n &= \text{if } (x = \infty) \text{ then } \infty \text{ else } (x - n)
\end{aligned}$$

[3]This same idea is often used in implementations that try to save space by using a small, fixed-size reference count field.

which capture the "stickiness" property described above. We assume that the auxiliary functions *dec*, *inc*, and *inc_st* are redefined to use these new operations.

Similarly, the domain of *bounded locations* is defined by:

$$ALoc = \{none, 1, 2, \ldots, maxloc\}$$

where *maxloc* is the total number of occurrences of calls to primitive functions that generate new locations (such as *update* and *new_array*). The idea here is that each such occurrence of a "location generator" can be approximated by an operator that generates the *same* location every time it is called. For convenience we assume that each occurrence of such a primitive operator *op* carries with it its unique location as a subscript, as in op_{loc}. *ALoc* is a flat domain, where *none* is the bottom element and the others are pairwise incomparable.

Although we have found *ALoc* to be suitable for most needs, better approximations to the domain of locations are possible. For example, one might give a unique location for each operator occurrence and call to the function containing that occurrence. That is, suppose op_i is an occurrence of *op* in the body of f. Then the location returned by a call to op_i depends on which occurrence of a call to f gave rise to it. For example, if *maxloc* is as defined above, and f_j is the jth occurrence of a call to f, then one way to compute such a location is simply $i + j * maxloc$.[4]

The above two abstract domains induce the following remaining ones:

$$\begin{array}{rcl} Bve & = & Bv \to Aloc \\ St & = & Aloc \to Src \\ Fve & = & Fv \to Aloc* \to St \to P(Aloc \times St) \end{array}$$

and these induce the following types on $\hat{\mathcal{R}}_p$, $\hat{\mathcal{R}}$, $\hat{\mathcal{K}}$, and $\hat{\mathcal{D}}$, as stated earlier:

$$\begin{array}{rl} \hat{\mathcal{R}}_p: & Prog \to Fve \\ \hat{\mathcal{R}}: & Label \to Fve \to Bve \to St \to P(Aloc \times St) \\ \hat{\mathcal{K}}: & Con \to Label* \to Fve \to Bve \to St \to P(Aloc \times St) \\ \hat{\mathcal{D}}: & Label \to Bve \to St \to St \end{array}$$

Note that a *powerset* is used to model the multiple outcomes in $(Aloc \times St)$. Thus if one of the possible outcomes is $\perp$ this will not be detected (unless it is the *only* possible outcome, in which case the result will be $\{\}$). If reasoning about possible termination is important, then a suitable *powerdomain* construction could be used, but this is an unnecessary complication for our purposes.

4.3 Abstract Primitives

$\hat{\mathcal{R}}_p$, $\hat{\mathcal{R}}$ and $\hat{\mathcal{D}}$ as defined in Section 4.1 now capture our desired abstraction, but we have the remaining task of giving an abstract interpretation of primitive functions, captured in the definition

[4]Note that this can be viewed as a "second-order" approximation, and that even higher order approximations can be imagined, such as calls to functions which call functions which call primitive operators (which would be "third-order"), and so on. At the other extreme, a "zero-order" approximation would be one in which the store had just *one* location that *every* operator shared!

of $\hat{\mathcal{K}}$:

$$\begin{array}{rcl}
\hat{\mathcal{K}}[\![if]\!]\ lp\ lc\ la\ fve\ bve\ st & = & \text{let } S = \hat{\mathcal{R}}\ lp\ fve\ bve\ st \\
& & \text{in } \bigcup\{\ (\hat{\mathcal{R}}\ lc\ fve\ bve\ (\hat{\mathcal{D}}\ la\ bve\ st'))\cup \\
& & \qquad (\hat{\mathcal{R}}\ la\ fve\ bve\ (\hat{\mathcal{D}}\ lc\ bve\ st')) \mid \langle loc, st'\rangle \in S\} \\
\hat{\mathcal{K}}[\![+]\!]\ l_1\ l_2\ fve\ bve\ st & = & \bigcup\{\hat{\mathcal{R}}\ l_2\ fve\ bve\ st_1 \mid \langle loc_1, st_1\rangle \in \hat{\mathcal{R}}\ l_1\ fve\ bve\ st\} \\
\hat{\mathcal{K}}[\![update_{loc}]\!]\ la\ li\ lx\ fve\ bve\ st & = & \{\langle loc, st_3'\rangle \mid \langle loc_1, st_1\rangle \in \hat{\mathcal{R}}\ la\ fve\ bve\ st, \\
& & \qquad \langle loc_2, st_2\rangle \in \hat{\mathcal{R}}\ li\ fve\ bve\ st_1, \\
& & \qquad \langle loc_3, st_3\rangle \in \hat{\mathcal{R}}\ lx\ fve\ bve\ st_2, \\
& & \qquad st_3' = inc\ (dec\ st_3\ loc_1)\ loc\ \}
\end{array}$$

Note that (1) the conditional no longer makes an appeal to the oracle – instead, both outcomes of the predicate are considered equally likely, and thus the values returned from the alternate and consequent are joined together; (2) + remains essentially unchanged; and (3) instead of a call to *alloc*, the new location generated by *update* is simply its subscript, as described earlier, whose reference count is increased by one.

5 A Collecting Interpretation of Reference Counts

5.1 Introduction

Unfortunately, $\hat{\mathcal{R}}_p$ and $\hat{\mathcal{R}}$ are not exactly what we want! We'd rather point to an expression *exp* in a program *pr* and answer the following question: "What are the reference counts of all possible values that *exp* could have during program execution?" The most obvious way to gather this data would be to completely construct $(\hat{\mathcal{R}}_p\ pr)$ using fixpoint iteration as described in the proof of Theorem 1, and then explicitly build a caching function to remember the values returned by all calls to $\hat{\mathcal{R}}$ when the program *pr* is "run." Although this technique is feasible, it may be intractable in that $\hat{\mathcal{R}}$ is a very large function for typical programs.

A better approach is to directly write a recursive description of the desired result. More precisely, we want a function $\mathcal{RC}$ such that $\mathcal{RC}(lab)$ returns the set of all possible reference counts that $expr(lab)$ could possibly evaluate to. In actuality the $\mathcal{RC}$ that will be defined below can be described by:

$$\begin{array}{rl}
\mathcal{RC}(lab) = \{\ \langle bve, st, loc, st'\rangle \mid & (\mathcal{R}\ lab\ fve\ bve\ st) \text{ was called during program execution,} \\
& \text{returning } \langle loc, st'\rangle\ \}
\end{array}$$

from which the aforementioned information is easily recovered. We refer to $\mathcal{RC}$ as a *collecting interpretation of expressions*, since it "collects" all possible values that each expression might have during program execution.[5] The general theory of such an interpretation is beyond the scope of this paper; more details may be found in [5]. For our purposes the following discussion suffices: We will write a recursive set equation for $\mathcal{RC}(lab)$ whose least fixpoint (i.e., smallest set) has the general

[5] Our use of the term collecting interpretation is somewhat non-standard, in that the normal use refers to collecting values in the *standard* semantics. Reference counts are clearly a *non-standard* semantics, and furthermore our primary interest is collecting values in an *abstraction of* the non-standard semantics.

property that if values resulting from the left-to-right evaluation of arguments in a call are in the corresponding sets of results for those arguments, then the result of calling the function on those values must be in the set of results of the call. The least fixpoint of such an equation will be the empty set, of course, unless it is "primed" by requiring the set to contain the result of evaluating the whole program.

5.2 Details of the Collecting Interpretation

To simplify the equations defining $\mathcal{RC}$ we make extensive use of a "pattern-matching" convention that is best explained by example:

$$f\ x = \{c \mid \begin{array}{l} \langle x, a\rangle \in S1 \\ \langle a, -, c\rangle \in S2\} \end{array}$$

is shorthand for:

$$f\ x = \{c \mid (\exists x', a, a', b, c) \text{ such that } \begin{array}{l} \langle x', a\rangle \in S1, \\ \langle a', b, c\rangle \in S2, \\ (x' = x),\ (a' = a)\} \end{array}$$

The symbol "$-$" essentially means "don't care."

A second convention is necessary to speak about the syntactic "context" in which a labelled expression lies. Given a label lab, $context(lab)$ is defined as the immediately surrounding expression containing $expr(lab)$, or $[\![]\!]$ if lab is the label of the body of some function. For example, if $exp = [\![f(l.e)]\!]$ then $context(l) = exp$, and if f_i is defined by $[\![f_i = l.body_i]\!]$ then $context(l) = [\![]\!]$.

Now for the details of the collecting interpretation. The functions have types:

$$\begin{array}{l} Cache = P(Bve \times St \times Aloc \times St) \\ \mathcal{RC}_p : Prog \to Label \to Cache \\ \mathcal{RC} : Label \to Cache \\ \hat{\mathcal{R}}' : Label \to Bve \to St \to Cache \\ \hat{\mathcal{K}}' : Con \to Label* \to Bve \to St \to Cache \end{array}$$

and the equations defining them are:

$$\mathcal{RC}_p[\![\ \{f_i(x_1...x_n) = body_i\}\ ]\!] = \mathcal{RC} \text{ whererec}$$

$$\begin{aligned}
\mathcal{RC}\ lab = (primer\ lab)\ \cup\ &\text{case } context(lab) \text{ of} \\
[\![l.f(lab.e_1 \cdots)]\!]: &\ \bigcup\{\hat{\mathcal{R}}'\ lab\ bve\ st \mid \langle bve, st, -, -\rangle \in \mathcal{RC}\ l\} \\
[\![f(\cdots l.e_{i-1}\ lab.e_i \cdots)]\!]: &\ \bigcup\{\hat{\mathcal{R}}'\ lab\ bve\ st \mid \langle bve, -, -, st\rangle \in \mathcal{RC}\ l,\ st \neq \bot\} \\
[\![]\!]: &\ \bigcup\{\hat{\mathcal{R}}'\ lab\ bve\ st \mid \exists\ [\![f_i(l_1.e_1 \ldots l_n.e_n)]\!] \\
&\qquad \text{such that } lab = label[\![body_i]\!], \\
&\qquad\qquad \langle -, -, bve, st\rangle \in linkargs\ i\ l_1...l_n\} \\
[\![l.p(lab.e_1 \cdots)]\!]: &\ \bigcup\{\hat{\mathcal{R}}'\ lab\ bve\ st \mid \langle bve, st, -, -\rangle \in \mathcal{RC}\ l\} \\
[\![p(\cdots l.e_{i-1}\ lab.e_i \cdots)]\!], \text{ where } [\![p]\!] \neq [\![if]\!]: &\ \bigcup\{\hat{\mathcal{R}}'\ lab\ bve\ st \mid \langle bve, -, -, st\rangle \in \mathcal{RC}\ l,\ st \neq \bot\} \\
[\![if(l_1.p, lab.c, l_2.a)]\!] \text{ or } [\![if(l_1.p, l_2.c, lab.a)]\!]: &\ \bigcup\{\hat{\mathcal{R}}'\ lab\ bve\ (\hat{\mathcal{D}}\ l_2\ bve\ st) \mid \\
&\qquad \langle bve, -, -, st\rangle \in \mathcal{RC}\ l_1,\ st \neq \bot\}
\end{aligned}$$

$$\begin{aligned}
\hat{\mathcal{R}}'\ lab\ bve\ st = &\text{ case } expr(lab) \text{ of} \\
[\![c]\!]: &\ \{\langle bve, st, none, st\rangle\} \\
[\![x]\!]: &\ \{\langle bve, st, bve[\![x]\!], st\rangle\} \\
[\![p(l_1.e_1 \cdots l_n.e_n)]\!]: &\ \{\langle bve, st, \bot, \bot\rangle\} \cup (\hat{\mathcal{K}}'[\![p]\!]\ l_1...l_n\ bve\ st) \\
[\![f_i(l_1.e_1 \cdots l_n.e_n)]\!]: &\ \{\langle bve, st, \bot, \bot\rangle\} \cup \{\ \langle bve, st, loc, st''\rangle \\
&\qquad \mid \langle bve, st, bve', st'\rangle \in linkargs\ i\ l_1...l_n, \\
&\qquad\ \ \langle bve', st', loc, st''\rangle \in \mathcal{RC}\ label[\![body_i]\!]\}
\end{aligned}$$

$$\begin{aligned}
linkargs\ i\ l_1...l_n = \{\langle bve, st, [loc_j/x_j], st'\rangle \mid\ &\langle bve, st, loc_1, st_1\rangle \in \mathcal{RC}\ l_1, \\
&\langle bve, st_1, loc_2, st_2\rangle \in \mathcal{RC}\ l_2, \\
&\ldots \\
&\langle bve, st_{n-1}, loc_n, st_n\rangle \in \mathcal{RC}\ l_n, \\
&st_i \neq \bot,\ i = 1, ..., n \\
&st' = inc_st\ st_n\ i\ loc_1...loc_n\}
\end{aligned}$$

$$primer\ lab = \text{if } (lab = label[\![body_1]\!]) \text{ then } (\hat{\mathcal{R}}'\ lab\ \bot_{Bve}\ (\lambda loc.0)) \text{ else } \{\}$$

The fifth case in the equation for $\mathcal{RC}$ assumes that all primitives except for $[\![if]\!]$ evaluate their arguments left-to-right.

To understand these equations it helps to note that, although the definition of $\mathcal{RC}$ was arrived at in a direct manner, its construction using fixpoint iteration models precisely the behavior of a cache! We begin with the first approximation $\mathcal{RC}^0 = \lambda\ lab.\{\}$. The first iteration yields $\mathcal{RC}^1 = primer$, indicating that the program has been "started." Further iterations "simulate" the top-down execution of the (abstract) program, which eventually terminates because of monotonicity over finite domains.

Computing contributions to the "cache" for bound variables and constants is easy (representing the "leaves" in the top-down evaluation). However for function calls we must first "record" the fact that the call, say $\hat{\mathcal{R}}\ lab\ bve\ st$, was made by adding a tuple $\langle bve, st, \bot_{loc}, \bot_{st}\rangle$ to the cache, and then continuing the evaluation top-down by evaluating the arguments and eventually the body of the function. $\langle \bot_{loc}, \bot_{st}\rangle$ can be viewed as a first approximation to the value of $\hat{\mathcal{R}}\ lab\ bve\ st$. As in a conventional semantics this value might never be improved upon, thus reflecting non-termination.

But as with $\hat{\mathcal{R}}$, this semantics is able to indicate non-termination as a possible outcome (in a given *bve* and *st*) only if that is the only possible outcome – powerdomains are needed for a finer distinction of termination.

Most of the details in this semantics is concerend with "linking together" the sequential evaluation of arguments, and returning the ultimate value of a call back to the calling location. To aid this, $linkargs\ i\ l_1...l_n$ returns all $\langle bve, st, bve', st' \rangle$ such that a call to $[\![f_i(l_1.e_1 \cdots l_n.e_n)]\!]$ in $\langle bve, st \rangle$ resulted in the evaluation of $body_i$ in $\langle bve', st' \rangle$. In other words, *linkargs* links up the stores that result from the evaluation of the arguments in a call.

Note the similarity of $\hat{\mathcal{R}}'$ to $\hat{\mathcal{R}}$ defined earlier – the primary difference is that $\hat{\mathcal{R}}$ consulted fve to determine the result of a call, whereas $\hat{\mathcal{R}}'$ simply looks up results in the cache function $\mathcal{RC}$.

There is of course something missing from the above equations – the definition of $\hat{\mathcal{K}}'$, given below:

$$
\begin{array}{rcl}
\hat{\mathcal{K}}'[\![if]\!]\ lp\ lc\ la\ bve\ st & = & \{\langle bve, st, loc_2, st_2\rangle \mid \langle bve, st, loc_1, st_1\rangle \in \mathcal{RC}\ lp, \text{ where} \\
 & & \quad ((st_1' = \mathcal{D}\ lc\ bve\ st_1) \wedge (\langle bve, st_1', loc_2, st_2\rangle \in \mathcal{RC}\ la)) \vee \\
 & & \quad ((st_1' = \mathcal{D}\ la\ bve\ st_1) \wedge (\langle bve, st_1', loc_2, st_2\rangle \in \mathcal{RC}\ lc)), \\
 & & \quad st_i \neq \bot,\ i = 1, 2\} \\
\hat{\mathcal{K}}'[\![+]\!]\ l_1\ l_2\ bve\ st & = & \{\langle bve, st, none, st_2\rangle \mid \langle bve, st, loc_1, st_1\rangle \in \mathcal{RC}\ l_1,\ st_1 \neq \bot \\
 & & \quad \langle bve, st_1, loc_2, st_2\rangle \in \mathcal{RC}\ l_2,\ st_2 \neq \bot\} \\
\hat{\mathcal{K}}'[\![update_{loc}]\!]\ la\ li\ lx\ bve\ st & = & \{\langle bve, st, loc, st'\rangle \mid \langle bve, st, loc_1, st_1\rangle \in \mathcal{RC}\ la, \\
 & & \quad \langle bve, st_1, loc_2, st_2\rangle \in \mathcal{RC}\ li, \\
 & & \quad \langle bve, st_2, loc_3, st_3\rangle \in \mathcal{RC}\ lx, \\
 & & \quad st_i \neq \bot,\ i = 1, 2, 3 \\
 & & \quad st' = inc\ (dec\ st_3\ loc_1)\ loc\}
\end{array}
$$

6 Correctness

Theorem 1 (Liveness) *For any (finite) program $pr \in Prog$, $(\hat{\mathcal{R}}_p\ pr)$ is computable.*

Proof: Let $fve = (\hat{\mathcal{R}}_p\ pr)$. *Label*, *Bve*, *St*, and *Aloc* are finite domains. Furthermore, $\hat{\mathcal{R}}_p$, $\hat{\mathcal{R}}$, $\hat{\mathcal{K}}$ and $\hat{\mathcal{D}}$ are monotonic functions since they are constructed solely from monotonic operators. Therefore fve can be effectively computed in the standard iterative manner: start with the bottom element $fve^0 = \lambda\ fv\ loc_1...loc_n\ st.\ \{\}$ and iterate until the least upper bound of the chain is reached, which is guaranteed because the domains are finite and the chain is monotonically increasing. □

Theorem 2 (Liveness) *For any (finite) program $pr \in Prog$, $(\mathcal{RC}_p\ pr)$ is computable.*

Proof: By an argument analogous to that for Theorem 1. □

We would also like to show that $\hat{\mathcal{R}}_p$ and $\hat{\mathcal{R}}$ form an *abstraction* of the standard reference count semantics. For this purpose we wish elements of Src to be ordered by arithmetic "less-than-or-equal-to," thus forming a chain. More precisely, define $\sqsubseteq_{rc}$, read "is an abstraction of," as follows:

$$
\begin{array}{rcl}
\langle loc, st\rangle \sqsubseteq_{rc} \langle loc', st'\rangle & \text{iff} & st(loc) \geq st'(loc') \\
\langle bve, st\rangle \sqsubseteq_{rc} \langle bve', st'\rangle & \text{iff} & (\forall x \in Bv)\ \langle bve[\![x]\!], st\rangle \sqsubseteq_{rc} \langle bve'[\![x]\!], st'\rangle \\
S \sqsubseteq_{rc} t & \text{iff} & (\exists s \in S)\ s \sqsubseteq_{rc} t
\end{array}
$$

Theorem 3 (Safety) If $\langle bve, st\rangle \sqsubseteq_{rc} \langle bve', st'\rangle$ then $(\hat{\mathcal{R}}[\![exp]\!]\ fve\ bve\ st) \sqsubseteq_{rc} (\mathcal{R}[\![exp]\!]\ fve\ bve'\ st')$

Proof: (Outline) $\oplus$ and $\ominus$ are abstractions of $+$ and $-$, respectively, since $(x \oplus y) \sqsubseteq_{rc} (x + y)$ (and similarly for $\ominus$). From this it can be shown that $\hat{\mathcal{K}}$ is an abstraction of $\mathcal{K}$. Then by an argument similar to the proof of Mycroft's correctness theorem for abstract interpretation [8], it can be shown that $\hat{\mathcal{R}}_p$ and $\hat{\mathcal{R}}$ are abstractions of $\mathcal{R}_p$ and $\mathcal{R}$, respectively. □

Corollary: Let $\langle loc, st\rangle = \mathcal{R}[\![exp]\!]\ fve\ bve\ st$ and $S = \hat{\mathcal{R}}[\![exp]\!]\ fve\ bve\ st$. Then there exists $\langle loc', st'\rangle \in S$ such that $st'(loc') \geq st(loc)$. That is, given the same bound variable environment and store, the abstraction yields at least one reference count whose value is greater than the true one.

As mentioned earlier, the inferred reference count information could be used in a variety of ways. One of the more important uses (and the one that originally motivated this research) is to determine when it is safe to perform destructive updates on aggregate data structures. Such an optimization is possible if the reference count of the aggregate is always 1 when the update is about to be performed. With respect to locations and reference counts, this interpreter optimization can be formalized as follows:

$$\begin{aligned}\mathcal{K}[\![update]\!]\ a\ i\ x\ bve\ st = \ & \text{let } \langle loc_1, st_1\rangle = \mathcal{R}\ a\ bve\ st \\ & \quad \langle loc_2, st_2\rangle = \mathcal{R}\ i\ bve\ st_1 \\ & \quad \langle loc_3, st_3\rangle = \mathcal{R}\ x\ bve\ st_2 \\ & \text{in if } st_3(loc_1) = 1 \text{ then } \langle loc_1, st_3\rangle \text{ else } alloc\ (dec\ st_3\ loc_1)\end{aligned}$$

which should be contrasted against the original definition of $\mathcal{K}[\![update]\!]$ given in Section 3.4. As stated this is a run-time optimization; but if one could infer at *compile time* that the test "$st_3(loc_1) = 1$" is true, then the optimization could be done at compile-time by essentially "constant-folding" the above conditional expression into $\langle loc_1, st_3\rangle$. This process can be formalized using the collecting interpretation as follows:

Theorem 4 (Copy Avoidance) *Consider a particular update operation $u = [\![update(l_1.a, l_2.i, l_3.x)]\!]$ in program pr, and let $RC = \mathcal{RC}_p\ pr$. Then u can be done destructively if the existence of bve, loc, and st_i, $i = 1, 2, 3$ such that:*

$$\begin{aligned}&\langle bve, st_0, loc, st_1\rangle \in RC\ l_1 \\ &\langle bve, st_1, -, st_2\rangle \in RC\ l_2 \\ &\langle bve, st_2, -, st_3\rangle \in RC\ l_3\end{aligned}$$

always implies that $st_3(loc) = 1$.

7 Examples of Copy Avoidance

We give three examples of applying the copy avoidance optimization described in the last section. The first is very simple:

$$\begin{aligned}\{\ & result() = init(new\_array(100), 1) \\ & init(a, i) = \text{if } i > 100 \text{ then } a \text{ else } init(update(a, i, 0), i + 1)\ \}\end{aligned}$$

which creates an array whose elements are all zero. A completely naive implementation of *update* would create a *new copy* of the array upon each update, thus consuming 100^2 locations! Using the collecting interpretation, however, it can be determined through Theorem 4 that the update can always be done destructively. It is worth stepping through the fixpoint computation of the cache for this example to see how this is accomplished, thus allowing the reader to see how the program's execution is "simulated." To do this, let us first rewrite the above program to conform to the formal syntax, so that there is a label on each expression, the functions are renamed, and the "location generators" are subscripted with their fixed locations:

$$\begin{aligned} pr = [\![\{\ & f_1() = 1.f_2(2.new\_array_1(3.100), 4.1) \\ & f_2(a,i) = 5.if(6.>(7.1, 8.100), 9.a, 10.f_2(11.update_2(12.a, 13.i, 14.0), 15.+(16.i, 17.1)\}]\!] \end{aligned}$$

In what follows, $g_1 \sqcup g_2$, where $g_1, g_2 \in (Label \to Cache)$, indicates the standard least upper bound by the subset ordering; i.e., $\lambda lab.\ (g_1\ lab) \cup (g_2\ lab)$. Also, $x \mapsto val$ is just shorthand for $\bot[val/x]$. Finally, in "anticipation" of the needed stores and environments, let:

$$\begin{array}{rcl} st_0 & = & \lambda loc.0 \\ st_1 & = & \lambda loc.\ \text{if } loc = 1 \text{ then } 2 \text{ else } 0 \\ st_2 & = & \lambda loc.\ \text{if } loc = 2 \text{ then } 2 \text{ else } 0 \\ st'_1 & = & \lambda loc.\ \text{if } loc = 1 \text{ then } 1 \text{ else } 0 \\ st'_2 & = & \lambda loc.\ \text{if } loc = 2 \text{ then } 1 \text{ else } 0 \\ bve_1 & = & [1/a, none/i] \\ bve_2 & = & [2/a, none/i] \end{array}$$

We wish to compute $RC = \mathcal{RC}_p\ pr$. To do so, the following chain is constructed, whose least upper bound is RC:

$$\begin{array}{rcl} RC^0 & = & \bot_{Label \to Cache} = \lambda lab.\{\} \\ RC^i & = & RC^{i-1} \sqcup exp_i \end{array}$$

where the exp_i are defined in Figure 1, each representing one "step" in the simulation. For example, exp_7 through exp_{10} represent first the call to $>(i, 100)$ (exp_7), followed by the evaluation of i and 100 (exp_8 and exp_9), and finally the return from the call (exp_{10}). Note that $RC^{37} = RC^{38}$ and is thus the least upper bound.

According to Theorem 4, we now need only examine the cache at labels 12, 13, and 14. From Figure 1 we see that these can be summarized by:

$$\begin{array}{c} 12 \mapsto \{\langle bve_1, st'_1, 1, st'_1\rangle, \langle bve_2, st'_2, 2, st'_2\rangle\} \\ 13 \mapsto \{\langle bve_1, st'_1, none, st'_1\rangle, \langle bve_2, st'_2, none, st'_2\rangle\} \\ 14 \mapsto \{\langle bve_1, st'_1, none, st'_1\rangle, \langle bve_2, st'_2, none, st'_2\rangle\}) \end{array}$$

Since $st'_1(1) = 1$ and $st'_2(2) = 1$, then according to Theorem 4 the update can be done destructively.

The next example is somewhat more complex. It is the quicksort algorithm as it was initially described by Hoare [4], in which a vector of keys is recursively "side-effected" in a divide-and-

$$
\begin{array}{rcl}
exp_1 & = & 1 \mapsto \hat{\mathcal{R}}'\ 1\ \bot_{Bve}\ st_0 = \{\langle \bot, st_0, \bot, \bot\rangle\} \\
exp_2 & = & 2 \mapsto \hat{\mathcal{R}}'\ 2\ \bot_{Bve}\ st_0 = \{\langle \bot, st_0, \bot, \bot\rangle\} \\
exp_3 & = & 3 \mapsto \hat{\mathcal{R}}'\ 3\ \bot_{Bve}\ st_0 = \{\langle \bot, st_0, none, st_0\rangle\} \\
exp_4 & = & 2 \mapsto \{\langle \bot, st_0, 1, st_1'\rangle\} \\
exp_5 & = & 4 \mapsto \hat{\mathcal{R}}'\ 4\ \bot_{Bve}\ st_1' = \{\langle \bot, st_1', none, st_1'\rangle\} \\
exp_6 & = & 5 \mapsto \hat{\mathcal{R}}'\ 5\ bve_1\ st_1 \text{ where} \\
 & & st_1 = inc\_st\ st_1'\ 2\ 1\ none \\
 & & \{\langle \bot, st_0, bve_1, st_1\rangle\} = linkargs\ 2\ 2\ 4 \\
 & = & 5 \mapsto \{\langle bve', st_1, \bot, \bot\rangle\} \\
exp_7 & = & 6 \mapsto \{\langle bve_1, st_1, \bot, \bot\rangle\} \\
exp_8 & = & 7 \mapsto \hat{\mathcal{R}}'\ 7\ bve_1\ st_1 = \{\langle bve_1, st_1, none, st_1\rangle\} \\
exp_9 & = & 8 \mapsto \{\langle bve_1, st_1, none, st_1\rangle\} \\
exp_{10} & = & 6 \mapsto \{\langle bve_1, st_1, none, st_1\rangle\} \\
exp_{11} & = & 9 \mapsto \hat{\mathcal{R}}'\ 9\ bve_1\ (\hat{\mathcal{D}}\ 10\ bve_1\ st_1) = \{\langle bve_1, st_1', 1, st_1'\rangle\} \\
 & & \sqcup\ 10 \mapsto \hat{\mathcal{R}}'\ 10\ bve_1 (\hat{\mathcal{D}}\ 9\ bve_1\ st_1) = \{\langle bve_1, st_1', \bot, \bot\rangle\} \\
exp_{12} & = & (5 \mapsto \{\langle bve_1, st_1, 1, st_1'\rangle\}) \sqcup (11 \mapsto \{\langle bve_1, st_1', \bot, \bot\rangle\}) \\
exp_{13} & = & (1 \mapsto \{\langle \bot, st_0, 1, st_1'\rangle\}) \sqcup (12 \mapsto \{\langle bve_1, st_1', 1, st_1'\rangle\}) \\
exp_{14} & = & 13 \mapsto \{\langle bve_1, st_1', none, st_1'\rangle\} \\
exp_{15} & = & 14 \mapsto \{\langle bve_1, st_1', none, st_1'\rangle\} \\
exp_{16} & = & 11 \mapsto \hat{\mathcal{K}}'[\![update_2]\!]\ 12\ 13\ 14\ bve_1\ st_1' = \{\langle bve_1, st_1', 2, st_2'\rangle\} \\
exp_{17} & = & 15 \mapsto \{\langle bve_1, st_2', \bot, \bot\rangle\} \\
exp_{18} & = & 16 \mapsto \{\langle bve_1, st_2', none, st_2'\rangle\} \\
exp_{19} & = & 17 \mapsto \{\langle bve_1, st_2', none, st_2'\rangle\} \\
exp_{20} & = & 15 \mapsto \{\langle bve_1, st_2', none, st_2'\rangle\} \\
exp_{21} & = & 5 \mapsto \{\langle bve_2, st_2, \bot, \bot\rangle\} \text{ where } st_2 = inc\_st\ st_2'\ 2\ 2\ none \\
exp_{22} & = & 6 \mapsto \{\langle bve_2, st_2, \bot, \bot\rangle\} \\
exp_{23} & = & 7 \mapsto \{\langle bve_2, st_2, none, st_2\rangle\} \\
exp_{24} & = & 8 \mapsto \{\langle bve_2, st_2, none, st_2\rangle\} \\
exp_{25} & = & 6 \mapsto \{\langle bve_2, st_2, none, st_2\rangle\} \\
exp_{26} & = & (9 \mapsto \{\langle bve_2, st_2', 2, st_2'\rangle\}) \sqcup (10 \mapsto \{\langle bve_2, st_2', \bot, \bot\rangle\}) \\
exp_{27} & = & (5 \mapsto \{\langle bve_2, st_2, 2, st_2'\rangle\}) \sqcup (11 \mapsto \{\langle bve_2, st_2', \bot, \bot\rangle\}) \\
exp_{28} & = & (10 \mapsto \{\langle bve_1, st_1', 2, st_2'\rangle\}) \sqcup (12 \mapsto \{\langle bve_2, st_2', 2, st_2'\rangle\}) \\
exp_{29} & = & (5 \mapsto \{\langle bve_1, st_1, 2, st_2'\rangle\}) \sqcup (13 \mapsto \{\langle bve_2, st_2', none, st_2'\rangle\}) \\
exp_{30} & = & (1 \mapsto \{\langle \bot, st_0, 2, st_2'\rangle\}) \sqcup (14 \mapsto \{\langle bve_2, st_2', none, st_2'\rangle\}) \\
exp_{31} & = & 11 \mapsto \{\langle bve_2, st_2', 2, st_2'\rangle\} \\
exp_{32} & = & 15 \mapsto \{\langle bve_2, st_2', \bot, \bot\rangle\} \\
exp_{33} & = & 16 \mapsto \{\langle bve_2, st_2', none, st_2'\rangle\} \\
exp_{34} & = & 17 \mapsto \{\langle bve_2, st_2', none, st_2'\rangle\} \\
exp_{35} & = & 15 \mapsto \{\langle bve_2, st_2', none, st_2'\rangle\} \\
exp_{36} & = & (5 \mapsto \{\langle bve_2, st_2, \bot, \bot\rangle\}) \sqcup (10 \mapsto \{\langle bve_2, st_2', 2, st_2'\rangle\}) \\
exp_{37} & = & 5 \mapsto \{\langle bve_2, st_2, 2, st_2'\rangle\}
\end{array}
$$

Figure 1: Sample Fixpoint Computation of $\mathcal{RC}$

conquer manner.[6] We can express this algorithm functionally as:

$$
\begin{array}{rcl}
result() & = & quicksort(vector) \\
quicksort(v) & = & qsort(v, 1, n) \\
qsort(v, left, right) & = & \text{if } left \geq right \text{ then } v \\
 & & \text{else } scanright(v, left + 1, right, v[left], left, right) \\
scanright(v, l, r, pivot, left, right) & = & \text{if } l = r \text{ then } finish(update(v, l, pivot), l, left, right) \\
 & & \text{else if } v[r] \geq pivot \text{ then } scanright(v, l, r - 1, pivot, left, right) \\
 & & \text{else } scanleft(update(v, l, v[r]), l + 1, r, pivot, left, right) \\
scanleft(v, l, r, pivot, left, right) & = & \text{if } l = r \text{ then } finish(update(v, l, pivot), l, left, right) \\
 & & \text{else if } v[l] \leq pivot \text{ then } scanleft(v, l + 1, r, pivot, left, right) \\
 & & \text{else } scanright(update(v, r, v[l]), l, r - 1, pivot, left, right) \\
finish(v, mid, left, right) & = & qsort(qsort(v, left, mid - 1), mid + 1, right)
\end{array}
$$

Except for the unaesthetic appearance of our bare-bones syntax (whose first-order nature requires that everything be "flattened"), this is actually a rather elegant formulation of the algorithm, in which mutual recursion is used to express the alternation between scanning the vector from the left and right.

Using $\mathcal{RC}_p$ we can infer that *all* of the updates in this program may be done destructively. This is quite interesting, because (1) most functional versions of quicksort use lists and consume space proportional to $n \lg n$, (2) the "naive" implementation of the above program consumes space proportional to $n^2 \lg n$, and (3) the optimized version consumes space proportional to n. A compiler using our optimization would thus allow the first (to our knowledge) functional version of quicksort to match the linear space complexity of Hoare's original algorithm!

The third example is the standard reference count semantics given in Section 3.4 considered as a functional program. More specifically, those equations can be rewritten to conform to our first-order syntax, and the store updates of form $st[rc/loc]$ can be replaced with $update(st, loc, rc)$. Once that is done the collecting interpretation is able to infer that *all* updates can be done destructively, just as the true interpreter would. So in some sense we have "boot-strapped" the analysis upon itself! This is a very encouraging result for semantics-directed compilers. A store that can be updated destructively in this manner is said to be *single-threaded*, a property that can be inferred via abstract reference counting as described here, or by a more direct (but perhaps less general) analysis as described in [10]. It is also related to the pebbling games described in [9].

8 Extensions

There are several extensions to the analysis that should be straightforward, such as adding constants that may be pass-by-reference (i.e., that are initially allocated in the store), and allowing nested equation groups or LET expressions. It should also be possible to allow arrays to contain arrays (thus generalizing lists) if one used the abstraction that any element selected from the array must be considered as a likely candidate for the subarray. Reasoning about the *structure* of such composite objects is more difficult; Jones and Muchnick [7] provide such a treatment for lists.

[6] The original "challenge" to solve this problem functionally and efficiently was posed in January 1985 by Daniel Friedman.

It is interesting to note that since our model has a store component, local side-effects may be easily handled. However, a continuation semantics would probably be necessary to handle global side-effects. A much harder extension would be the proper handling of higher-order functions and lazy evaluation. Actually it is not difficult to give a proper reference count semantics for these (indeed we have done so for lazy evaluation), but it is difficult to decide what a suitable (finite) abstraction should be. We are continuing work in this area as well as in the general area of collecting interpretations of expressions.

9 Acknowledgements

Thanks to Jonathan Young, Adrienne Bloss, Rich Kelsey and other members of the "Wrestling Team" at Yale for their many helpful comments.

References

[1] J.M. Barth. Shifting garbage collection overhead to compile time. *CACM*, 20(7):513–518, 1977.

[2] D.W. Clark. An empirical study of list structure in lisp. *CACM*, 20(2):78–87, February 1977.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Sym. on Prin. of Prog. Lang.*, pages 238–252, ACM, 1977.

[4] C.A.R. Hoare. Quicksort. *Computing J.*, 5(4):10–15, April 1962.

[5] P. Hudak. *Collecting interpretations of expressions.* Research Report 497, Yale University, Department of Computer Science, 1986.

[6] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 300–314, ACM, 1985.

[7] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Sym. on Prin. of Prog. Lang.*, pages 66–74, ACM, January 1982.

[8] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[9] J-C. Raoult and R. Sethi. The global storage needs of a subcomputation. In *11th ACM Sym. on Prin. of Prog. Lang.*, pages 148–157, ACM, January 1984.

[10] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Trans. on Prog. Lang. and Systems*, 7(2):299–310, 1985.

[11] J. Schwarz. Verifying the safe use of destructive operations in applicative programs. In B. Robinet, editor, *Program Transformations – Proc. of the 3rd Int'l Sym. on Programming*, pages 395–411, Dunod Informatique, 1978.

[12] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Mass., 1977.