Reference Escape Analysis: Optimizing Reference Counting based on the Lifetime of References*

Young Gil Park and Benjamin Goldberg
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University†

Abstract

In reference counting schemes for automatically reclaiming storage, each time a reference to an object is created or destroyed, the reference count of the object needs to be updated. This may involve expensive inter-processor message exchanges in distributed environments. This overhead can be reduced by analyzing the lifetimes of references to avoid unnecessary updatings.

This paper describes a technique for reducing the run-time reference counting overhead through compile-time optimization. We present a compile-time analysis called *reference escape analysis* for higher-order functional languages that determines whether the *lifetime* of a reference *exceeds* the lifetime of the environment in which the reference was created. Using this statically inferred information, a method for optimizing reference counting schemes is described. Our method can be applied to reference counting schemes in both uniprocessor and multiprocessor environments.

1 Introduction

An implementation of a programming language with dynamic (indefinite extent) storage allocation requires some kind of storage reclamation mechanism. Automatic storage reclamation is especially important for functional languages, which have no notion of explicit storage control and tend to use storage extensively. There are three basic approaches, with a number of variants, to automatically detect and reclaim storage: reference counting, mark-and-sweep garbage collection, and copying garbage collection.

Reference counting is a storage reclamation method in which each object contains a count, called the reference count, of the number of references (pointers) pointing to it. When an object is first allocated, its reference count is set to one. The reference count is updated during execution as follows: Each time a new reference to an object is created, its reference count is incremented by one. Each time a reference to an object is destroyed, its reference count is decremented by one. When an object's reference count becomes zero, it can be reclaimed and the reference count of each object that it points to is decremented.

Though the reference counting strategy has disadvantages, such as storage fragmentation and the inability to reclaim cyclic structures, its major advantage is that storage reclamation occurs incremently throughout program execution; storage can be reclaimed as soon as it has become garbage. It is also especially suitable in multiprocessor architectures with distributed memory, since reference counting is an inherently real-time and localized activity.

The major overhead that is incurred in reference counting schemes are as follows:

- Space overhead for maintaining a reference count in each object.
- Time and code overhead for updating reference counts when references are created or destroyed.
- Communication overhead for manipulating a remote reference and for synchronizing the operations on reference counts in distributed memory environments.

In this paper, we describe a method for reducing the time, code, and communication overhead of reference counting in both uniprocessor and multiprocessor environments by compile-time program analysis. Our approach is based on the observation that such overheads can be reduced by avoiding unnecessary reference count updates using statically inferred information about the

*This research was funded in part by the National Science Foundation (#CCR-8909634) and by DARPA/ONR (#N00014-90-1110).

†Authors' address: 251 Mercer Street, New York, NY 10012. Email: park@cs.nyu.edu, goldberg@cs.nyu.edu

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-433-3/91/0006/0178...$1.50

lifetime of each reference. The lifetime of a reference to an object is the period from when the reference is created until its last use. Suppose O is an object that is active at some time t_0 during execution. Since this object is active, there is at least one reference pointing to it. If its reference count O_{rc} is n then there are exactly n references pointing to it. Suppose A is one of those references. Now, suppose that a new reference B to the object is created. The current reference count of the object is incremented, i.e. $O_{rc} := O_{rc} + 1$. At some later time t_1, suppose that B is discarded. Then, the current reference count of the object is decremented, i.e. $O_{rc} := O_{rc} - 1$.

If we can determine, at compile-time, that A will still be active at time t_1, then no reference count operations are required when B is created or destroyed. Since the reference count of O always remains greater than or equal to one from time t_0 to time t_1, O will *not* to be reclaimed between time t_0 and time t_1. Thus, the reference count updating operation for the reference B can be avoided. This avoidance optimization is safe because any object which is still active will not be reclaimed.

1.1 Related Work

[DB76] proposed a method for reducing the overhead of updating reference counts in which reference counting activities are *deferred* by being stored into a file called a transaction file instead of being immediately performed. Reference counts are then adjusted at suitable intervals. [Bar77] showed that this particular reference counting scheme could benefit from compile time optimization by generating fewer transactions (reference counting activities) based on compile time analysis of *first-order* programs.

Using the idea of weighted references, i.e. each reference carries a weight such that the sum of the weights of all references to an object is equal to the reference count of the object, there have been a variety of works [Bev87,WW87], in which, when a new reference is created to an object, no access to the object is needed.

[Gol89] presented a generation-based approach for distributed systems that also avoids reference count operations when a reference is created, and also described the applicability of escape information among references to reference counting schemes, but did not present the analysis.

[Hud86] presented a semantic model for describing the number of active pointers to objects for an applicative-order interpreter of a first-order function language, and a variety of its abstractions based on abstract and collecting interpretations.

Jones and Le Metayer [JL89] presented a compile-time optimization to replace the allocation of new cells by the reuse of collectible cells based on sharing analysis of objects in first-order strict functional languages.

$c \in Con$	Constants(including primitive functions) $= \{\ldots, -1, 0, 1, \ldots, \texttt{true}, \texttt{false}, \texttt{+}, \texttt{-}, \texttt{=}, \texttt{nil}, \texttt{cons}, \texttt{car}, \texttt{cdr}\}$
$x \in Id$	Identifiers
$e \in Exp$	Expressions, defined by $e ::= c \mid x \mid e_1 e_2 \mid \lambda x.e \mid$ $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e3 \mid$ $\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n \texttt{ in } e$
$pr \in Pgm$	Programs, defined by $pr ::= \texttt{letrec } x_1 = e_1; \ldots; x_n = e_n \texttt{ in } e$

Figure 1: Syntax of Functional Language

This approach considers sharing as a compile-time abstraction of reference counting of objects and is based on a forward analysis and backward analyses (for transmission and necessity information).

Deutsch [Deu90] described a static analysis for determining information about lifetime and sharing of objects in higher-order, strict, polymorphic languages. This method is based on a low-level operational model for higher-order functional languages that explicits details such as storage allocation and sharing, and thus the analysis is performed on the program that is translated from an original program into a sequence of operations in this model. In our method, the analysis is performed at the source level.

The goal of our reference escape analysis is to statically determine the lifetime of dynamically created references, and to apply the information to lessen the run-time overhead of reference counting scheme. Even though our analysis is used to improve reference counting, it is not based explicitly on an abstraction of reference counting (as [Hud86] and [JL89] are).

2 The Functional Language

We define a simple *higher-order* functional language based on the *typed* lambda calculus, augmented with constants (including primitive functions). The syntax is defined in Figure 1. For convenience, we will use the form $f\ x_1\ \ldots\ x_n\ =\ e$ as syntactic sugar for $f = \lambda x_1. \ldots \lambda x_n. e$. Since we are not concerned here with the problems of type checking or inference, we will assume that the compiler has already performed type checking before our analysis is attempted. Thus, we assume that functions will always be applied to arguments of the correct type.

We use the following notations; the double bracket, $[\![\ ]\!]$, is used to surround syntactic objects, the square bracket and map arrow, $[\ \mapsto\ ]$, are used for environment updates, the angle bracket, $\langle\ \rangle$, is used for tupling,

and the subscripts (1) and (2) are used to denote the first and second elements of a pair, respectively.

3 Reference Escape Analysis

In this section we present a compile-time semantic analysis, called *reference escape analysis*, which is based on abstract interpretation ([AH87],[BHA86],[Myc81]) and provides information about the lifetimes of dynamically created references. Initially, we will assume that the language is strict and monomorphically typed. The analysis is extended in sections 5 and 6 to handle polymorphically typed and non-strict versions of the language, respectively.

Before describing the analysis, we need to describe the operational model in which references are created and destroyed. We choose a model that is commonly implemented in LISP and functional language systems, that of a call by value language with pointer semantics for heap allocated structures. References are created in three ways:

1. When a heap allocated object is created, a reference to that object is created and returned by the allocation procedure (e.g. `cons`).
2. When a heap allocated object is passed as a parameter in a function call, a reference to the object is copied into the activation record of the called function.
3. When an assignment occurs (in a `letrec`, for example) and the value of the right hand side is a heap-allocated object, a reference to the object is copied into the variable (or record field) on the left hand side.

Consider the following function definition:

```
f x y =  letrec
            g a b = cons a b
         in
            cons x (g x y)
```

When f is called, its activation record contains two references to lists, corresponding to the parameters x and y. Thus, when `(g x y)` is evaluated, the references corresponding to x and y are copied into the activation record for g. Likewise, any argument to cons that is represented by a reference is also copied.

We analyze the lifetimes of a reference by determining its *escapement*, that is, whether or not a reference is returned out of the scope in which it was created. When does a reference escape? Intuitively, a reference can escape when it is placed in a structure or closure that escapes. In a previous paper [GP90], we described escape analysis for structures and closures. As we discuss later in this paper, if a reference does not escape the scope of its creation, no reference counting operations are necessary when the reference is created or destroyed.

In the above example, when the references corresponding to `x` and `y` are copied into `g`'s activation record, no reference count increment operation is required. Likewise, when `g` returns, no decrement operation is required. However, when a cons cell, corresponding to `cons a b`, is created, the reference counts of the objects pointed to by `a` and `b` must be incremented. This is because the lifetime of the cons cell exceeds that of `g` and `f`, and it cannot be determined, at compile time, when the references contained in the cons cell will be destroyed.

Definition 1 (Reference Escapement)
Given a function f with n formal parameters, $x_1 \ldots x_n$, and m local variables $l_1 \ldots l_m$, the jth occurrence of the ith parameter x_i (or local variable l_i), which we shall refer to as x_{ij} (or l_{ij}), in the body of f, is said to:

- *reference-escape* the function definition of f *globally* if, in *some* possible application of f, the reference associated with x_{ij}(or l_{ij}) is contained in the result of the function application.
- *reference-escape* *locally* in a function call (f e_1 ... e_n) if the reference associated with x_{ij} (or l_{ij}) is contained in the result of a *particular* function application of (f e_1 ... e_n).

From the escapement of a reference, we can deduce its lifetime: If an occurrence of a parameter or local object does not reference-escape the function call *globally* then we can conclude that the lifetime of the reference associated with the occurrence is confined to the lifetime of *any* possible call to the function. Similarly, if it does not reference-escape the function call *locally* in a particular function application then we can conclude that the lifetime of the reference associated with the occurrence is confined to the lifetime of *that* particular function call.

3.1 A Reference Escape Semantics

We introduce an exact but uncomputable non-standard denotational semantics, called *reference escape semantics*, which describes completely the escaping behaviors of references in a program. As we discuss later, each reference in an expression is analyzed separately to determine its escape behavior. Thus, our semantics is defined in terms of a single reference whose escape behavior we are trying to determine. We say that a reference is *interesting* if it is the one whose escape whose escape behavior we are trying to determine.

For each expression, its corresponding value in the reference escape semantic domain should indicate whether interesting references are returned by that expression

or not. In the non-standard reference escape semantics, the meaning of an expression is a pair, called *a reference escape pair*, whose first element denotes the presence or absence of interesting references and whose second element denotes the functional behavior of the expression when the expression itself is applied to another expression (as in [HY86]). Thus, for a non-list type expression, the corresponding value in the non-standard reference escape semantic domain D_r has two components: The first component is an element of a domain called a *basic reference escape domain*, B_r which is a two-element domain whose elements are 0 and 1 ordered by $0 \sqsubseteq 1$, and interpreted as follows:

- 1 : An interesting reference *is* contained in the value of the expression.

- 0 : *No* interesting reference is contained in the value of the expression.

The second component is a function over D_r. The second component of the value of an expression which has no higher-order behavior is *err*, which denotes a function that can never be applied. The reference escape value of a list L is a list of the reference escape values of the elements of L.

The reference escape semantic domain D_r and the domain of reference escape environments Env_r are defined as follows (in the style of [BHA86]):

$$\begin{array}{lll} D_r^{int} & = & B_r \times \{err\} \\ D_r^{bool} & = & B_r \times \{err\} \\ D_r^{\tau_1 \to \tau_2} & = & B_r \times (D_r^{\tau_1} \to D_r^{\tau_2}) \\ D_r^{\tau\ list} & = & (B_r \times \{err\}) + (D_r^{\tau} \times D_r^{\tau\ list}) \\ D_r = \sum_{\tau} D_r^{\tau} & & \text{Reference escape domain} \\ Env_r = Id \to D_r & & \text{Reference escape environment} \end{array}$$

We introduce the following reference escape semantic functions:

$$\begin{array}{ll} K_r : & Con \to D_r \\ E_r : & Exp \to Env_r \to D_r \\ P_r : & Pgm \to D_r \end{array}$$

K_r gives reference escape meaning to constants, E_r gives reference escape meaning to expressions, and P_r gives reference escape meaning to programs. The semantic equations for K_r, E_r and P_r are defined in Figure 2. Note that an oracle is used, for convenience, to resolve the exact behavior of the conditional `if`. This can be otherwise be accomplished by having the exact escape semantics directly compute the standard meaning as well as escape meaning of expressions. $nullenv_r$ denotes the empty environment.

3.2 An Abstract Reference Escape Semantics

Even though the reference escape semantics described in the last section can provide exact reference escape information, it cannot be used as a basis of compile-time analysis because it must rely on the standard semantics and thus is not guaranteed to terminate at compile time. For a compile-time analysis, we need a computable approximation of the exact reference escape semantics that provides safe and useful, but less complete, reference escape information. We safely approximate the exact reference escape semantics by abstracting the reference escape semantic subdomains for list type expressions, and by approximating the reference escape semantic functions.

In order to approximate the exact reference escape domains for lists, we treat a list to be a collection of *spines* instead of an individual elements. The spines of a list are pictured in Figure 3. This allows us to use type information (which tells us the number of spines of a list) to assist our analysis.

Definition 2 (Spines of a list)
Given a list L and some $i \geq 1$, the *top* i^{th} spine of L is defined as the set of cons cells accessible by a sequence of operations consisting of `car` and `cdr` where the number of occurrence of `car` is $(i-1)$. Similarly, given a list L with d spines and some $j \geq 1$, the *bottom* j^{th} spine of L is defined as the *top* $(d-j+1)^{th}$ spine of L.

The *abstract* basic reference escape domain, $\hat{B}_r^d$, for some fixed d is defined as a $(d+2)$-element domain whose ordering on elements is defined by

$$\langle 0,0\rangle \sqsubseteq \langle 1,0\rangle \sqsubseteq \langle 1,1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,d-1\rangle \sqsubseteq \langle 1,d\rangle.$$

The interpretation of elements of $\hat{B}_r^d$ is defined as follows:

- $\langle 1,j\rangle$: An interesting reference *may* be contained in the value of the expression, and, if $j \geq 1$, it is a reference to the cons cell at the bottom j^{th} spine of a list, for $0 \leq j \leq d$. (If $j = 0$ then the object pointed by the interesting reference is not a cons cell.)

- $\langle 0,0\rangle$: *No* interesting reference is contained in the value of the expression.

Abstracting the reference escape semantic subdomains for list type expressions is done by representing lists as finite objects, i.e. by combining the reference escape pairs of all its elements into a single reference escape pair. The domain $\hat{D}_r$ is an abstraction of D_r.

$$
\begin{array}{lcl}
K_r[\![c]\!] & = & \langle 0, err\rangle, \quad c \in \{\ldots,-1,0,1,\ldots,\mathtt{true},\mathtt{false},\ \mathtt{nil}^{\tau\ list}\} \\
K_r[\![c]\!] & = & \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\langle 0, err\rangle\rangle\rangle, \quad c \in \{\mathtt{+},\mathtt{-},\mathtt{=}\} \\
K_r[\![\mathtt{cons}]\!] & = & \langle 0, \lambda x.\langle x_{(1)}, \lambda y.\langle x, y\rangle\rangle\rangle \\
K_r[\![\mathtt{car}]\!] & = & \langle 0, \lambda x.x_{(1)}\rangle \\
K_r[\![\mathtt{cdr}]\!] & = & \langle 0, \lambda x.x_{(2)}\rangle
\end{array}
$$

$$
\begin{array}{lcl}
E_r[\![c]\!]env_r & = & K_r[\![c]\!] \\
E_r[\![x]\!]env_r & = & env_r[\![x]\!] \\
E_r[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]env_r & = & \text{if } \mathbf{Oracle}(e_1) \text{ then } E_r[\![e_2]\!]env_r \text{ else } E_r[\![e_2]\!]env_r \\
E_r[\![e_1 e_2]\!]env_r & = & (E_r[\![e_1]\!]env_r)_{(2)}\ (E_r[\![e_2]\!]env_r) \\
E_r[\![\lambda x.e]\!]env_r & = & \langle O_r, \lambda y.E_r[\![e]\!]env_r[x \mapsto y]\rangle
\end{array}
$$

where

$$O_r = (\bigsqcup_{v \in F} (env_r[\![v]\!])_{(1)})) \sqcup (\bigsqcup_{v \in F^{list}} (\bigsqcup_{p\ \mathbf{in}\ env_r[v]} p_{(1)}))$$

p **in** l denotes each pair element in l

F and F^{list} are the set of non-list type and list type free variables in $(\lambda x.e)$.

$$E_r[\![\mathtt{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathtt{in}\ e]\!]env_r = E_r[\![e]\!]env_r'$$

where $env_r' = env_r[x_1 \mapsto E_r[\![e_1]\!]env_r', \ldots, x_n \mapsto E_r[\![e_n]\!]env_r']$

$$P_r[\![pr]\!] = E_r[\![pr]\!]nullenv_r$$

Figure 2: Reference Escape Semantic Functions

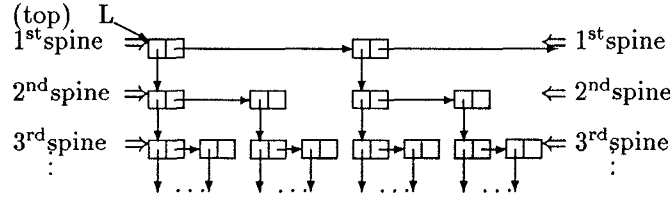

Figure 3: Spines of a List

$$
\begin{array}{lcl}
\hat{D}_r^{int} & = & \hat{B}_r^d \times \{err\} \\
\hat{D}_r^{bool} & = & \hat{B}_r^d \times \{err\} \\
\hat{D}_r^{\tau_1 \to \tau_2} & = & \hat{B}_r^d \times (\hat{D}_r^{\tau_1} \to \hat{D}_r^{\tau_2}) \\
\hat{D}_r^{\tau\ list} & = & \hat{D}_r^{\tau}
\end{array}
$$

$\hat{D}_r = \sum_{\tau} \hat{D}_r^{\tau}$ Abstract escape domain

$\hat{Env}_r = Id \to \hat{D}_r$ Abstract escape environments

The abstract reference escape semantic functions

$$
\begin{array}{ll}
\hat{K}_r: & Con \to \hat{D}_r \\
\hat{E}_r: & Exp \to \hat{Env}_r \to \hat{D}_r \\
\hat{P}_r: & Pgm \to \hat{D}_r
\end{array}
$$

are defined in Figure 4. Note that the conditional **if** no longer makes an appeal to the oracle, but rather takes the least upper bound of the escapement of both branches. $\mathbf{car}^{\mathbf{s}}$ is the application of **car** to a list with **s** spines. For each **car** in a program, **s** can be determined statically by type checking. It may be arbitrary large, but is fixed at compile-time. $\mathbf{car}^{\mathbf{s}}$ takes a list with **s** spines as an argument, and returns a list with $(\mathbf{s}-\mathbf{1})$ spines when $\mathbf{s} > \mathbf{1}$ or a non-list object when $\mathbf{s} = \mathbf{1}$. In any case, the result cannot contain a reference pointing to the cons cell at the bottom s^{th} spine of a list. Note that **cons** returns a single reference escape pair that approximates a list of reference escape pairs.

Theorem 1 (Termination)
For any program $pr \in Pgm$, $P_r[\![pr]\!]$ can be found in a finite number of steps.

Proof : Every functional under the abstract reference escape semantics is monotonic, because it is composed of operators, such as the least upper bound operator $\sqcup$ and cut^s, that are monotonic. The abstract reference semantic domain for each type is finite. Recall that the language is monomorphically typed. Thus, any least fixpoint iteration is guaranteed to terminate in a finite number of steps. □

$$
\begin{array}{lll}
\hat{K}_r[\![c]\!] & = & \langle\langle 0,0\rangle, err\rangle, \quad c \in \{\ldots,-1,0,1,\ldots,\texttt{true},\texttt{false}\} \\
\hat{K}_r[\![c]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0\rangle, err\rangle\rangle\rangle, \quad c \in \{\texttt{+},\texttt{-},\texttt{=}\} \\
\hat{K}_r[\![\texttt{nil}^{\tau\ list}]\!] & = & \bot_\tau \text{ (bottom element in } \hat{D}_r^\tau) \\
\hat{K}_r[\![\texttt{cons}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle \\
\hat{K}_r[\![\texttt{car}^{\texttt{s}}]\!] & = & \langle\langle 0,0\rangle, \lambda x.\langle cut^s(x_{(1)}), x_{(2)}\rangle\rangle \text{ /* s is the number of spines of the argument of car */} \\
 & & \text{where} \\
 & & cut^s(z) = \text{if } (s = z_{(2)}) \text{ then } \langle 0,0\rangle \text{ else } z \\
\hat{K}_r[\![\texttt{cdr}]\!] & = & \langle\langle 0,0\rangle, \lambda x.x\rangle
\end{array}
$$

$$
\begin{array}{lll}
\hat{E}_r[\![c]\!]\hat{env}_r & = & \hat{K}_r[\![c]\!] \\
\hat{E}_r[\![x]\!]\hat{env}_r & = & \hat{env}_r[\![x]\!] \\
\hat{E}_r[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\hat{env}_r & = & \hat{E}_r[\![e_2]\!]\hat{env}_r \sqcup \hat{E}_r[\![e_3]\!]\hat{env}_r \\
\hat{E}_r[\![e_1 e_2]\!]\hat{env}_r & = & (\hat{E}_r[\![e_1]\!]\hat{env}_r)_{(2)}\,(\hat{E}_r[\![e_2]\!]\hat{env}_r) \\
\hat{E}_r[\![\lambda x.e]\!]\hat{env}_r & = & \langle \hat{O}_r, \lambda y.\hat{E}_r[\![e]\!]\hat{env}_r[x \mapsto y]\rangle \\
 & & \text{where} \\
 & & \hat{O}_r = (\bigsqcup_{u \in F} (env_r[\![u]\!])_{(1)})) \text{ and } F \text{ is the set of free variables in } (\lambda x.e).
\end{array}
$$

$$
\hat{E}_r[\![\texttt{letrec } x_1 = e_1; \ldots; x_n = e_n \texttt{ in } e]\!]\hat{env}_r = \hat{E}_r[\![e]\!]\hat{env}'_r
$$
$$
\text{where } \hat{env}'_r = \hat{env}_r[x_1 \mapsto \hat{E}_r[\![e_1]\!]\hat{env}'_r, \ldots, x_n \mapsto \hat{E}_r[\![e_n]\!]\hat{env}'_r]
$$

$$
\hat{P}_r[\![pr]\!] = \hat{E}_r[\![pr]\!]null\hat{env}_r
$$

Figure 4: Abstract Reference Escape Semantic Functions

The safety of interpretation under the abstract reference escape semantics can be proved as follows. Let NAP_n be an apply operator for elements in the nonstandard reference escape semantic domains, defined by: For $ep\ ep_1\ \ldots ep_n \in D_r(\hat{D}_r)$,

$$
\text{NAP}_n(ep, ep_1, \ldots, ep_n) \stackrel{\text{def}}{=} \begin{cases} ep & n = 0 \\ \text{NAP}_{n-1}(ep_{(2)} ep_1, ep_2, \ldots, ep_n) & n > 0 \end{cases}
$$

Let u and v be values of an expression e of type τ or τ *list* in $\hat{D}_r$ and D_r, respectively. Let n be the number of arguments that the type τ can take before returning a non-function value. We say that u is a *safe* aproximation (with respect to exact reference escape information) for v iff

$$
(\bigsqcup_{p \text{ in NAP}_k(v, t_1 \ldots t_k)} p_{(1)}) \sqsubseteq (\text{NAP}_k(u, s_1 \ldots s_k))_{(1)(1)}
$$

where $\forall k \leq n$, $\forall i \leq k$, s_i is a safe approximation of t_i.

Theorem 2 (Safety) *For all expression e and environments env_r and $\hat{env}_r$ such that $\hat{env}_r[\![y]\!]$ is safe for $env_r[\![y]\!]$ for all y, $\hat{E}_r[\![e]\!]\hat{env}_r$ is safe (with respect to reference escape information) for $E_r[\![e]\!]env_r$.*

Proof : This can be proved by structural and fixpoint induction[Par91]. □

3.3 Testing for Reference Escapement

Reference escape testing is performed separately on the reference associated with each occurrence of a foraml parameter of a function. Thus, at any time we are only interested in whether or not a particular reference escapes. Other references may escape in the result of a function call, but are ignored by our analysis. If a parameter has n occurrences, then reference escape analysis will be performed n times (and a different reference is considered interesting each time).

Consider the example from before:

```
f x y =  letrec g a b = cons a b;
         in cons x (g x y)
```

As we discussed previously, each occurrence of x in the body of f denotes the creation of a new reference. To differentiate between the occurrences of x, we label each occurrence differently. In fact, the different occurrences can be considered different parameters to an auxiliary function derived from `f`:

```
f' x1 x2 y =  letrec g a b = cons a b;
              in cons x1 (g x2 y)
```

Our abstract reference escape semantics will give the escapement of the parameters to `f'`, and thus of the references corresponding to `x1` and `x2`. We describe below how this analysis proceeds.

3.3.1 Auxiliary Functions

In order to make each occurrence of each parameter of a function distinct, we introduce an auxiliary function f' for each function f. Then, we perform reference escape testing on f' to determine reference escape property of each occurrence of each parameter of f. Given a function

$$f\ x_1\ \ldots\ x_n\ =\ e,$$

the auxiliary function f' is given as follows:

$$f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots\ x_{n1}\ \ldots\ x_{no(n)}\ =\ e'$$

where $o(i)$ is the number of occurrences of x_i in e and e' is derived from e by replacing the jth occurrence of x_i by x_{ij} for all i and j. Note that each parameter of f' will now have only one occurrence, and f' will be never called from anywhere and thus is not recursive. To determine the escape behavior of references associated with occurrences of parameters of f, we perform the test on its auxiliary function f'.

3.3.2 Global Reference Escape Test

Using a global reference escape test, we find reference escape information about each function in a program that holds true for any possible application of the function.

Definition 3 (Worst-Case Escape Function) For each non-list type τ, we define the abstract function W^τ that corresponds to a function from which every argument escapes.

$$W^\tau \stackrel{\text{def}}{=} \lambda x_1.\langle x_{1(1)}, \lambda x_2.\langle x_{1(1)} \sqcup x_{2(1)}, \cdots, \lambda x_m.\langle \bigsqcup_{i=1,m} x_{i(1)}, err\rangle \ldots\rangle\rangle$$

where m is the number of arguments that a function of type τ can take before returning a non-function value. For each type τ *list*, $W^{\tau\ list}$ is defined to be W^τ.

Given an identifier f bound to a function of n parameters and an abstract reference escape environment env_r in which f is defined, the global reference escape test function G_r determines whether the jth occurrence of the ith parameter could possibly escape. It is defined as follows:

$$G_r(f, i, j, env_r) = (\hat{E}_r[\![f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots\ x_{n1}\ \ldots\ x_{no(n)}]\!]\ env_r[f' \mapsto \hat{f}', x_{io(i)} \mapsto y_{io(i)}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\hat{f}' = \hat{E}_r[\![f']\!]env_r,$$

$$y_{ij} = \langle\langle 1, s_i\rangle, W^{\tau_i}\rangle,$$

τ_i is the type of the ith parameter of f, and s_i is the number of total spines of type τ_i (when τ_i is not list type, s_i is 0). For all $k \leq o(i)$, $k \neq j$,

$$y_{ik} = \langle\langle 0, 0\rangle, W^{\tau_i}\rangle,$$

and for all $h \leq n$, $h \neq i$, and for all $k \leq o(h)$,

$$y_{hk} = \langle\langle 0, 0\rangle, W^{\tau_h}\rangle,$$

where $o(i)$ is the number of occurrences of ith parameter of f. From the result of the global reference escape test, we can conclude the following:

- If $G_r(f, i, j, env_r) = 0$ then in *any* possible application of f to n arguments, the reference associated with the jth occurrence of the ith argument *does not* escape the function call.
- If $G_r(f, i, j, env_r) = 1$ then in *some* possible application of f to n arguments, the reference associated with the jth occurrence of the ith argument *could* escape the function call.

3.3.3 Local Reference Escape Test

Using a local reference escape test, we find reference escape information about a function in a particular call to the function, which depends on the values of arguments of that call. Given an identifier f bound to a function of n parameters, an application $f\ e_1\ \ldots\ e_n$, and an abstract reference escape environment env_r in which f and all identifiers in e_1 through e_n are defined, the local reference escape test function L_r determines the escapement of the jth occurrence of the f's i^{th} parameter during the evaluation of $f\ e_1\ \ldots\ e_n$ as follows:

$$L_r(f, i, j, e_1, \ldots, e_n, env_r) = (\hat{E}_r[\![f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots\ x_{n1}\ \ldots\ x_{no(n)}]\!]\ env_r[f' \mapsto \hat{f}', x_{io(i)} \mapsto y_{io(i)}])_{(1)(1)}$$

where where f' is the auxiliary function for f,

$$\hat{f}' = \hat{E}_r[\![f']\!]cnv_r,$$

$$y_{ij} = \langle\langle 1, s_i\rangle, (\hat{E}_r[\![e_i]\!]env_r)_{(2)}\rangle,$$

and s_i is the number of total spines of the type of the ith parameter of f (when τ_i is not list type, s_i is 0). For all $k \leq o(i)$, $k \neq j$,

$$y_{ik} = \langle\langle 0, 0\rangle, (\hat{E}_r[\![e_i]\!]env_r)_{(2)}\rangle,$$

and for all $h \leq n$, $h \neq i$, and for all $k \leq o(h)$,

$$y_{hk} = \langle\langle 0, 0\rangle, (\hat{E}_r[\![e_h]\!]env_r)_{(2)}\rangle$$

where $o(i)$ is the number of occurrences of ith parameter of f. From the result of the local reference escape test, we can conclude the following:

- If $L_r(f, i, j, e_1, \ldots, e_n, env_r) = 0$ then, in the particular application of f to e_1 through e_n, the reference associated with the jth occurrence of the ith argument *does not* escape the function call.
- If $L_r(f, i, j, e_1, \ldots, e_n, env_r) = 1$ then, in the particular application of f to e_1 through e_n, the reference associated with the jth occurrence of the ith argument *may* escape the function call.

Notice that we only consider those applications of f to n arguments. If f is applied to fewer than n arguments, no references corresponding to occurrences of the formal parameters of f escape. This is because the body of f isn't executed in a partial application of f.

3.4 Examples

Consider the following program:

```
letrec
   map f l = if (l=nil) then nil
             else cons (f (car l))
                       (map f (cdr l));

   sum l = if (l=nil) then 0
           else 1 + sum (cdr l);

   addsum x y = cons x (cons y (cons
            (map (lambda(z). (sum y) + z)
                 x) nil));
in ...
```

We assume that

map : $(int \to int) \to int\ list \to int\ list$
sum : $int\ list \to int$
addsum : $int\ list \to int\ list \to (int\ list)list$

The definitions of reference escape semantic values of **map**, **sum**, and **addsum** are:

$$\begin{aligned} map\ f\ l &= \langle\langle 0,0\rangle, err\rangle \sqcup \\ &\quad (f\ \langle cut^1(l_{(1)}), l_{(2)}\rangle) \sqcup (map\ f\ l) \\ sum\ l &= \langle\langle 0,0\rangle, err\rangle \sqcup (sum\ l) \\ addsum\ x\ y &= x \sqcup y \sqcup \\ &\quad (map\ \langle y_{(1)}, \lambda z.\langle\langle 0,0\rangle, err\rangle\rangle\ x) \end{aligned}$$

Since *map* and *sum* are defined recursively, the meanings of *map* and *sum* are found by a fixpoint iteration:

$$\begin{aligned} map^{(0)}\ f\ l &= \langle\langle 0,0\rangle, err\rangle \\ map^{(1)}\ f\ l &= \langle\langle 0,0\rangle, err\rangle \sqcup (f\ \langle cut^1(l_{(1)}), l_{(2)}\rangle) \\ &\quad \sqcup (map^{(0)}\ f\ l) \\ &= f\ \langle cut^1(l_{(1)}), l_{(2)}\rangle \\ map^{(2)}\ f\ l &= \langle\langle 0,0\rangle, err\rangle \sqcup (f\ \langle cut^1(l_{(1)}), l_{(2)}\rangle) \\ &\quad \sqcup (map^{(1)}\ f\ l) \\ &= f\ \langle cut^1(l_{(1)}), l_{(2)}\rangle \end{aligned}$$

Since $map^{(1)} = map^{(2)}$, we have that $map = map^{(2)}$.

$$\begin{aligned} sum^{(0)}\ l &= \langle\langle 0,0\rangle, err\rangle \\ sum^{(1)}\ l &= \langle\langle 0,0\rangle, err\rangle \sqcup (sum\ l) \\ &= \langle\langle 0,0\rangle, err\rangle \\ sum^{(2)}\ l &= \langle\langle 0,0\rangle, err\rangle \end{aligned}$$

Similarly, we have that $sum = sum^{(2)}$.

$$\begin{aligned} addsum\ x\ y &= x \sqcup y \sqcup \\ &\quad (map\ \langle y_{(1)}, \lambda z.\langle\langle 0,0\rangle, err\rangle\rangle\ x) \\ &= x \sqcup y \sqcup \\ &\quad (\lambda z.\langle\langle 0,0\rangle, err\rangle)\ \langle cut^1(x_{(1)}), x_{(2)}\rangle \\ &= x \sqcup y\ \sqcup\ \langle\langle 0,0\rangle, err\rangle \\ &= x \sqcup y \end{aligned}$$

The auxiliary functions **map'** and **addsum'** for **map** and **addsum** are defined as follows:

```
map' f1 f2 l1 l2 l3 = if (l1=nil) then nil
                      else cons (f1 (car l2))
                                (map f2 (cdr l3));

addsum' x1 x2 y1 y2 = cons x1 (cons y1
          (cons (map (lambda(z).(sum y2)+z)
                x2) nil ));
```

Note that **maps'** is not recursively defined. Then, the definitions of reference escape semantic values of **map'**, and **addsum'** are given as follows (without a fixpoint iteration):

$$\begin{aligned} map'\ f1\ f2\ l1\ l2\ l3 &= \langle\langle 0,0\rangle, err\rangle \sqcup \\ &\quad (f1\ \langle cut^1(l2_{(1)}), l2_{(2)}\rangle) \\ &\quad \sqcup (map\ f2\ l3) \\ &= \langle\langle 0,0\rangle, err\rangle \sqcup \\ &\quad (f1\ \langle cut^1(l2_{(1)}), l2_{(2)}\rangle) \\ &\quad (f2\ \langle cut^1(l3_{(1)}), l3_{(2)}\rangle) \\ addsum'\ x1\ x2\ y1\ y2 &= x1 \sqcup y1 \end{aligned}$$

Let env_r= [**map** $\mapsto$ *map*, **add** $\mapsto$ *add*, **addsum** $\mapsto$ *addsum*]. Then,

$$\begin{aligned} &G_r(\texttt{addsum}, 1, 1, env_r) = \\ &\quad (\hat{E}_r[\![\texttt{addsum' x1 x2 y1 y2}]\!] \\ &\quad env_r[\texttt{addsum'} \mapsto addsum', \\ &\quad x_1 \mapsto \langle\langle 1,1\rangle, err\rangle, \\ &\quad x_2, y_1, y_2 \mapsto \langle\langle 0,0\rangle, err\rangle])_{(1)(1)} = 1 \end{aligned}$$

Thus, the reference **x1** associated with the first occurrence of the first parameter **x** of **addsum** escapes. And,

$$\begin{aligned} &G_r(\texttt{addsum}, 1, 2, env_r) = \\ &\quad (\hat{E}_r[\![\texttt{addsum' x1 x2 y1 y2}]\!] \\ &\quad env_r[\texttt{addsum'} \mapsto addsum', \\ &\quad x_2 \mapsto \langle\langle 1,1\rangle, err\rangle, \\ &\quad x_1, y_1, y_2 \mapsto \langle\langle 0,0\rangle, err\rangle])_{(1)(1)} = 0 \end{aligned}$$

Thus, the reference **x2** associated with the second occurrence of the first parameter **x** of **addsum** does not escape. Similarly, we can conclude that the reference **y1** associated with the first occurrence of the second parameter **y** of **addsum** escapes, but the reference **y2** associated with the second occurrence of the second parameter **y** of **addsum** does not escape.

In the same way, we also can conclude that both the reference **f1** and **f2** associated with occurrences of the first parameter **f** of **map** does not escape.

3.5 Complexity of Reference Escape Analysis

The abstract interpretation framework for our reference escape analysis is very similar to the framework for strictness analysis in [BHA86,HY86,Wad87] except for the size of the basic abstract domain. Like any other analysis based on abstract interpretation, the major complexity of our analysis comes from finding the fixpoints of recursive functions in the abstract semantic domains. In our analysis, the reference escape testing is performed on each reference associated with each occurrence of a parameter of a function separately using its auxiliary function. However, since the auxiliary function for a function is never recursive even though the original function is a recursive function, the process of finding fixpoint is needed only for an original function, but is never needed for its auxiliary function. Thus, the order of complexity of reference escape analysis that deals with higher-order functional languages with non-flat domains is similar to and comparable with that of strictness analysis for higher-order languages with non-flat domains, which is exponential in the number of arguments to the function being analyzed.

4 Improving Reference Escape Analysis

Up to now, our reference escape analysis has treated each list as follows: Once an interesting reference is put in a list, no matter how many times **cdr** is applied to the list, we assume that the interesting reference remains in the portion of the result list, which can be returned when **car** is applied to the result list. In this section, we present a method of improving the accuracy of the reference escape analysis described so far by keeping track of a reference's position within a list.

Definition 4 (Positions of a list)
A reference is said to be at *position* k of a list L if it *only* resides somewhere in the sublist of L whose root cell is specified by $\mathtt{cdr}^k\ L$ for some $k \geq 0$. In other words, the reference is not contained in the first $(k-1)$ positions of L(See Figure 5).

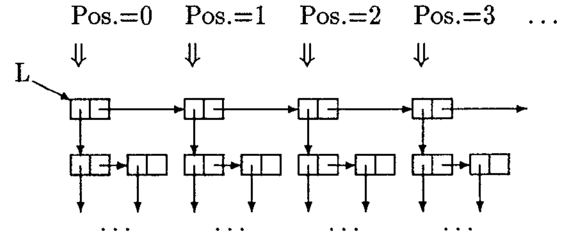

Figure 5: Positions of a List

Since no finite bound on the length of lists can be computed at compile time, we impose a bound on the position in the list that we are willing to keep track of. Beyond this position, we assume that the reference remains in the list. We extend the basic abstract reference escape domain by including position information of references in a list. The improved basic abstract reference escape domain, $\tilde{B}_r^{d,p}$, for some fixed d and p is a $((d+1)(p+1)+1)$-element domain whose ordering on elements is defined by

$$\begin{array}{l} \langle 0,0,0\rangle \sqsubseteq \\ \langle 1,0,p\rangle \sqsubseteq \langle 1,0,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,0,0\rangle \sqsubseteq \\ \ldots \sqsubseteq \langle 1,d,p\rangle \sqsubseteq \langle 1,d,p-1\rangle \sqsubseteq \ldots \sqsubseteq \langle 1,d,0\rangle \end{array}$$

The elements of $\tilde{B}_r^{d,p}$ are interpreted as follows:

- $\langle 1,j,k\rangle$: An interesting reference *may* be contained in the value of the expression, and, it is a reference to the cons cell at the bottom j^{th} spine of a list and it *may* occur only at $\geq k$ positions in the result, for $0 \leq j \leq d$ and $0 \leq k \leq p$. (If $j = 0$ then the object pointed by the interesting reference is not a cons cell, and if the expression does not denote a list then k is always 0.)
- $\langle 0,0,0\rangle$: No interesting reference *is* contained in the value of the expression.

The improved abstract reference escape semantic domain $\tilde{D}_r$ is defined as follows:

$$\begin{array}{lll} \tilde{D}_r^{int} & = & \tilde{B}_r^{d,p} \times \{err\} \\ \tilde{D}_r^{bool} & = & \tilde{B}_r^{d,p} \times \{err\} \\ \tilde{D}_r^{\tau_1 \to \tau_2} & = & \tilde{B}_r^{d,p} \times (\tilde{D}_r^{\tau_1} \to \tilde{D}_r^{\tau_2}) \\ \tilde{D}_r^{\tau\ list} & = & \tilde{D}_r^{\tau} \\ \tilde{D}_r = \sum_\tau \tilde{D}_r^\tau & & \text{Improved abstract domain} \\ \tilde{Env}_r = Id \to \tilde{D}_r & & \text{Improved abstract environments} \end{array}$$

where p is some fixed constant that is a bound on lists that we keep track of. The improved abstract reference escape semantic function

$$\tilde{K}_r : Con \to \tilde{D}_r$$

$$\tilde{K}_r[\![c]\!] = \langle\langle 0,0,0\rangle, err\rangle, c \in \{\ldots,-1,0,1,\ldots,\mathbf{true},\mathbf{false}\}$$
$$\tilde{K}_r[\![c]\!] = \langle\langle 0,0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0,0\rangle, err\rangle\rangle\rangle, c \in \{+,-,=\}$$
$$\tilde{K}_r[\![\mathbf{nil}^{\tau\ list}]\!] = \bot_\tau \text{ (bottom element in } \tilde{D}_r^\tau)$$
$$\tilde{K}_r[\![\mathbf{cons}]\!] = \langle\langle 0,0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle push(x_{(1)}, y_{(1)}), x_{(2)} \sqcup y_{(2)}\rangle\rangle\rangle$$
where
$$push(u,v) = \text{if } (u_{(1)} = 1) \text{ then } \langle 1, u_{(2)}, 0\rangle \text{ elseif } (v_{(1)} = 1) \text{ then } \langle 1, v_{(2)}, min[v_{(3)}+1, p]\rangle \text{ else } \langle 0,0,0\rangle$$
$$\tilde{K}_r[\![\mathbf{car^s}]\!] = \langle\langle 0,0,0\rangle, \lambda x.\langle icut^s(x_{(1)}), x_{(2)}\rangle\rangle$$
where
$$icut^s(z) = \text{if } (s = z_{(2)}) \text{ or } (z_{(3)} > 0) \text{ then } \langle 0,0,0\rangle \text{ else } z$$
$$\tilde{K}_r[\![\mathbf{cdr}]\!] = \langle\langle 0,0,0\rangle, \lambda x.\langle rest(x_{(1)}), x_{(2)}\rangle\rangle$$
where
$$rest(z) = \langle z_{(1)}, z_{(2)}, max[z_{(3)} - 1, 0]\rangle$$

Figure 6: Improved Abstract Reference Escape Semantic Function

is defined in Figure 6. $\mathbf{car^s}$ takes a list with **s** spines and returns the element in the 0th position, having **s** − 1 spines. Thus, the result cannot contain a reference pointing to a cons cell in the bottom $\mathbf{s^{th}}$ spine of a list or a reference that occurred at a position ≥ 1 in the original list. Note that both **cons** and **cdr** update the position information appropriately. Notice that the semantic function $\tilde{K}_r$ for **cons**, **car** and **cdr** provides more precise information than $\hat{K}_r$.

The improved abstract reference escape semantic functions

$$\tilde{E}_r : Exp \rightarrow \tilde{Env}_r \rightarrow \tilde{D}_r$$
$$\tilde{P}_r : Pgm \rightarrow \tilde{D}_r$$

are defined the same as $\hat{E}_r$ and $\hat{P}_r$, respectively.

Termination is still guaranteed, since *push*, $icut^s$, and *rest* are all monotonic functions. The abstract reference escape semantics described in last section is equivalent to the improved abstract reference escape semantics when p is 0. Thus, any improved abstract reference escape semantics with some $p > 0$ provides more precise reference escape information than the abstract reference escape semantics.

The *improved* global reference escape test function G'_r is defined as follows:

$$G'_r(f, i, j, env_r) = (\tilde{E}_r[\![f'\ x_{11}\ \ldots\ x_{1o(1)}\ \ldots\ x_{n1}\ \ldots\ x_{no(n)}]\!]\ env_r[f' \mapsto \hat{f}', x_{io(i)} \mapsto y_{io(i)}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\hat{f}' = \hat{E}_r[\![f']\!]env_r,$$
$$y_{ij} = \langle\langle 1, s_i, 0\rangle, W^{\tau_i}\rangle,$$

τ_i is the type of the ith parameter of f, and s_i is the number of total spines of type τ_i. For all $k \leq o(i)$, $k \neq j$,

$$y_{ik} = \langle\langle 0,0,0\rangle, W^{\tau_i}\rangle,$$

and for all $h \leq n$, $h \neq i$, and for all $k \leq o(h)$,

$$y_{hk} = \langle\langle 0,0,0\rangle, W^{\tau_h}\rangle$$

where $o(i)$ is the number of occurrences of ith parameter of f. Similarly, the *improved* local reference escape test function L'_r is defined as follows:

$$L'_r(f, i, j, e_1, \ldots, e_n, env_r) = (\tilde{E}_r[\![f', x_{11}, \ldots, x_{1o(1)}, \ldots\ x_{n1}, \ldots, x_{no(n)}]\!]\ env_r[f' \mapsto \hat{f}', x_{io(i)} \mapsto y_{io(i)}])_{(1)(1)}$$

where f' is the auxiliary function for f,

$$\hat{f}' = \hat{E}_r[\![f']\!]env_r,$$
$$y_{ij} = \langle\langle 1, s_i, 0\rangle, (\tilde{E}_r[\![e_i]\!]env_r)_{(2)}\rangle,$$

and s_i is the number of total spines of the type of the ith parameter of f. For all $k \leq o(i)$, $k \neq j$,

$$y_{ik} = \langle\langle 0,0,0\rangle, (\tilde{E}_r[\![e_i]\!]env_r)_{(2)}\rangle,$$

and for all $h \leq n$, $h \neq i$, and for all $k \leq o(h)$,

$$y_{hk} = \langle\langle 0,0,0\rangle, (\tilde{E}_r[\![e_h]\!]env_r)_{(2)}\rangle$$

where $o(i)$ is the number of occurrences of ith parameter of f.

5 Polymorphic Invariance

So far, we have described the reference escape analysis for a monomorphically-typed language. Most modern functional programming languages have a polymorphic type system. In this section, through the notion of *polymorphic invariance*[Abr86], we describe the reference

escape analysis for a higher-order functional language with a *polymorphic* type system.

The polymorphic invariance of reference escape analysis says that whether a reference escapes a call to a polymorphic function or not is independent of the type of the reference. This means that given a polymorphic function, reference escape analysis will provide the same result on any two monotyped instances of that function. Consider two monotyped instances e' of type τ' and e'' of type of τ'' of a polymorphically-typed expression e. Let n be $\mathbf{min}[n_{\tau'}, n_{\tau''}]$ where $n_{\tau'}$ and $n_{\tau''}$ are the number of arguments that τ' and τ'' can take before returning a non-function value. Let u' and u'' be the values in $\hat{D}_r$ of e' and e'', respectively. We define the equivalence of u' and u'' (written $u' \sim u''$) as follows:

$$u' \sim u'' \textbf{iff}$$
$$(\mathrm{NAP}_k(u', s_1 \ldots s_k))_{(1)} = (\mathrm{NAP}_k(u'', t_1 \ldots t_k))_{(1)},$$
$$\text{where } \forall k \leq n, \forall i \leq k, s_i \sim t_i.$$

Lemma 1 *Let f be a polymorphic recursive function defined as $f = \lambda x_1 \ldots . \lambda x_n.e$ where $\lambda x_1 \ldots . \lambda x_n.e$ contains free variables $v_1 \ldots v_m$. Let f' and f'' be two monotyped instances of f, typed as follows:*

$$[v_1 : \sigma'_1, \ldots, v_m : \sigma'_m] f' : \tau'_1 \rightarrow \ldots \rightarrow \tau'_n \rightarrow \tau'$$
$$[v_1 : \sigma''_1, \ldots, v_m : \sigma''_m] f'' : \tau''_1 \rightarrow \ldots \rightarrow \tau''_n \rightarrow \tau''$$

where each σ and τ is a monotype. For monotyped abstract reference escape environments env'_r and env''_r that map each v_i to an element in $\hat{D}_r$ and for each v_i, $env'_r[\![v_i]\!] \sim env''_r[\![v_i]\!]$, respectively,

$$\hat{f}' \sim \hat{f}''$$

where

$$\hat{f}' = \hat{E}_r[\![\lambda x_1 \ldots . \lambda x_n.e]\!] env'_r$$
$$\hat{f}'' = \hat{E}_r[\![\lambda x_1 \ldots . \lambda x_n.e]\!] env''_r.$$

Proof : This can be proved by structural and fixpoint induction[Par91]. □

Theorem 3 (Polymorphic Invariance) *Let f be a polymorphic function of arity n, and let f' and f'' be any two monotyped instances of f. Assume that env' and env'' are abstract reference escape semantic environments that map f' and f'' to elements of $\hat{D}_r$, respectively. Then, for $1 \leq i \leq n$ and for $1 \leq j \leq o(i)$,*

$$G_r(f', i, j, env') = G_r(f'', i, j, env'').$$

Proof : Let f'_a and f''_a be the auxiliary functions for f' and f'', respectively. Then,

$$\hat{f}'_a = \hat{E}_r[\![f'_a]\!] env'[f' \mapsto \hat{f}']$$
$$\hat{f}''_a = \hat{E}_r[\![f''_a]\!] env''[f'' \mapsto \hat{f}'']$$
$$G_r(f', i, j, env') =$$
$$(\mathrm{NAP}_n(\hat{f}'_a, y'_{11}, \ldots, y'_{no(n)}))_{(1)(1)}$$
$$G_r(f'', i, j, env'') =$$
$$(\mathrm{NAP}_n(\hat{f}''_a, y''_{11}, \ldots, y''_{no(n)}))_{(1)(1)}$$

By Lemma 1, $\hat{f}' \sim \hat{f}''$ and $\hat{f}'_a \sim \hat{f}''_a$. By the definitions of the worst-case escape function W and y'_{ij} and y''_{ij}, $y'_{ij} \sim y''_{ij}$ for all $1 \leq i \leq n$ and $1 \leq j \leq o(i)$. Thus,

$$\mathrm{NAP}_n(\hat{f}'_a, y'_{11}, \ldots, y'_{no(n)})_{(1)} =$$
$$\mathrm{NAP}_n(\hat{f}''_a, y''_{11}, \ldots, y''_{no(n)})_{(1)}.$$

We conclude that

$$G_r(f', i, j, env') = G_r(f'', i, j, env'')$$

□

As a consequence of this fact, the reference escape analysis problem for polymorphic functions can be reduced to the problem for monomorphic functions. The reference escape analysis algorithm need only be applied to the simplest monotyped instance of a function. Smaller types implies fewer elements of that type, and the efficiency of reference escape analysis and similar analyses requiring fixpoint finding is dependent on the number of elements in the domain.

6 Extension to Nonstrict Languages

We have described the reference escape analysis for strict functional languages. In this section, we describe a method for extending the reference escape analysis to non-strict functional languages. In non-strict languages each argument in a function call is not evaluatea unless and until its value is required. One way to implement non-strict semantics is to *delay* the evaluation of expressions that are not immediately needed using closures, and to *force* the evaluation when their values is needed by applying closures to a dummy argument.

Thus, the reference escape analysis for non-strict languages can be achieved by first transforming nonstrict programs into equivalent strict programs, and then by applying the reference escape analysis technique for strict languages to them. Of course, this does not handle *lazy evaluation*, in which delayed expressions are represented by closures that are modified when the expression is evaluated. In order to provide an analysis that works for lazy evaluation, order of evaluation information is required (such information is provided by[Blo89,Par91]).

7 Optimizations using Reference Escape Information

In section 3, we described the operational model for creating references. Reference escape information, from the analyses described here, can be used as follows: If a reference, created within a function by the occurrence of

a formal parameter does not escape from the function, then no reference count operations need be performed when the reference is created or destroyed.

Furthermore, given two references A and B to a heap allocated object, the relative lifetimes of A and B can be computed by determining if there is a scope from which one of them escapes but not the other. If so, when the shorter-lived reference is created and destroyed, no reference count operations are necessary. In some programs, in fact, our analysis can determine if some reference R to an object outlives all others. Thus, the object can be reclaimed as soon as R is destroyed. No other reference counting operations are needed. Notice, however, that it may not be possible to determine lifetime of R (if it is embedded in a structure, for instance), and thus of the object, at compile time.

8 Conclusion

We have presented a compile-time analysis called *reference escape analysis* for higher-order functional languages that provides safe static information about the run-time lifetime of references. Based on the statically inferred lifetime of references, a method for optimizing reference counting schemes has been presented. This method should significantly reduce the time, code, and communication overhead that is incurred in reference counting schemes for uniprocessor and multiprocessor implementations.

References

[Abr86] S. Abramsky. Strictness analysis and polymorphic invariance. In *Proceedings of Workshop on Programs as Data Objects*, pages 1-24, LNCS 217 Springer-Verlag, 1986.

[AH87] S. Abramsky and C.L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[Bar77] J.M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM,* 20(7), pages 513-518, 1977.

[Bev87] D.I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of PARLE Parallel Architectures and Languages Europe II,* pages 176-187, LNCS 259 Springer-Verlag, 1987.

[BHA86] G.L. Burn, C.L. Hankin and S. Abramsky. The theory of strictness analysis for higher order functions. In *Proceedings of Workshop on Programs as Data Objects*, pages 42-62, LNCS 217 Springer-Verlag, 1986.

[Blo89] A. Bloss. *Path Analysis: Using Order-of-Evaluation Information to Optimize Lazy Functional Languages.* Ph.D. Thesis, Yale University, 1989.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages,* pages 157-168, 1990.

[DB76] L.P. Deutsch and D.G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM,* 19(9), pages 522-526, 1976.

[Gol89] B. Goldberg. Generational reference counting: a reduced-communication distributed storage reclamation scheme. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* pages 313-321, 1989.

[GP90] B. Goldberg and Y.G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *Proceedings of the 3rd European Symposium on Programming,* pages 152-160, LNCS 432 Springer-Verlag, 1990.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming,* pages 351-363, Auguat 1986.

[HY86] P. Hudak and J. Young. Higher-order strictness for untyped lambda calculus. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages,* pages 97-100, January 1986.

[JL89] S.B. Jones and D. Le Metayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the 1989 Functional Programming Languages and Computer Architecture Conference,* pages 54-74, 1989.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[Par91] Y.G. Park. *Semantic Program Analyses for Storage Optimizations in Functional Language Implementations.* Ph.D. Thesis, New York University, 1991.(To appear)

[Wad87] P. Wadler. Strictness analysis on non-flat domains. In *Abstract Interpretation of Declarative Languages.* C.L. Hankin and S. Abramsky, editors. Ellis Horwood, 1987.

[WW87] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architecture. In *Proceedings of PARLE Parallel Architectures and Languages Europe II,* pages 432-443, LNCS 259 Springer-Verlag, 1987.