Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations[1]

Benjamin Goldberg and Young Gil Park

Department of Computer Science[2]
Courant Institute of Mathematical Sciences
New York University

Abstract.

In this paper, we present a method for optimizing the allocation of closures in memory. This method is based on *escape analysis*, an application of abstraction interpretation to higher order functional languages. Escape analysis determines, at compile time, if any arguments to a function have a greater lifetime than the function call itself. Such arguments, especially if they are closures, must be allocated in the heap rather than in the stack. In most implementations, however, stack allocation of closures is preferable due to the lower cost of allocation and reclamation. Therefore, we use escape analysis to determine when arguments can be stack allocated safely.

In the past, first order escape analysis has been used in optimizing LISP compilers, and has been described in various data-flow analysis frameworks for a language with complex types. The analysis described here, being higher order, provides more accurate escape information, although for a very simple higher order functional language.

1. Introduction

Higher order functions are an important part of functional languages. They have generally been seen, however, as having a high implementation overhead. Two reasons for this view are that 1) they force an implementation to use a heap to store closures, and 2) programs using higher order functions are particularly difficult to analyze for optimization purposes.

The need for heap allocation arises when parameters and locally defined objects within a function outlive a call to that function. For example, in the following program fragment

```
let  f x y z = x + y + z
     g a b = f b a
in   g 1 2
```

a closure representing the partial application of **f** during the execution of **g** will outlive the call to **g**. Thus, the closure containing the parameters **a** and **b** will have to be heap allocated. In this case, the partial application of **f**, along with **g**'s parameters **a** and **b**, are said to *escape* from the call to **g**.

Notice that in the following program fragment

```
let  f x y z = x + y + z
     h x = x 3
     g a b = h (f b a)
in   g 1 2
```

the closure representing the partial application of **f** does not escape from **g**. In this case, the closure and pa-

1. This research was supported in part by a National Science Foundation Research Initiation Award, CCR-8909634.
2. Address: 251 Mercer St., New York, NY 10012. Email: goldberg@cs.nyu.edu, park@cs.nyu.edu.

$c \in Con$	constants
$x \in Id$	identifiers
$e \in Exp$	expressions, defined by
	$e ::= c \mid x \mid e_1\ e_2 \mid e_1 \rightarrow e_2, e_3 \mid e_1 + e_2 \mid e_1 = e_2 \mid \lambda x.\ e \mid$
	$\textbf{let}\ x_1 = e_1; \ldots ; x_n = e_n\ \textbf{in}\ e$

Figure 1. The syntax of nml

rameters can be allocated in g's activation record on the stack. *Escape analysis* is a compile time analysis that determines whether an object, such as a closure or a parameter, needs to be heap allocated.

In this paper we assume that stack allocation is less expensive than heap allocation, although we recognize that there are safety issues (such as stack overflow, etc.) that are important (see [Chase88]). There are also persuasive arguments in favor of heap allocation. In the Standard ML compiler [AM87], all closures are allocated in the heap. It appears that with a large amount of memory and a sophisticated garbage collection strategy, the overhead of garbage collection is quite small. However, on most current systems (and especially in distributed systems where garbage collection is more expensive), stack allocation of closures is preferable.

A simple escape analysis was used in the Orbit compiler for Scheme [Kranz88]. It is a first order escape analysis in which the following program could not be analyzed accurately.

```
let f a b = a + b
    g h a b = h b a
in  g f 1 2
```

because **h** is an unknown function (i.e. a function bound to a formal parameter) in the body of **g**. Orbit's escape analysis assumes that any argument to an unknown function will escape from that function. Thus, it will assume that both **a** and **b** escape from the call to **h** inside of **g**, and therefore escape from **g**. In fact, neither **a** nor **b** escape from **g**.

Another analysis, called lifetime analysis [RM88], was developed to compute, if possible, the relative lifetimes of dynamically allocated objects in a first order language with structures and recursive types (such as trees). Escape analysis is a particular instance of lifetime analysis in which the lifetime of a function's activation record is compared to the objects defined inside the function. Other analyses for optimizing storage allocation were proposed in [JM76], [MJ81], [Schwartz75], [Barth77], and [Chase87].

In the following sections, we present an analysis that, using abstract interpretation ([CC77],[Mycroft81]), gives escape information in the presence of higher order functions (although we do not deal with structures and recursive types). Higher order abstract interpretation has been mainly used for strictness analysis ([BHA85],[HY86]), although other higher order analyses have been developed (such as sharing analysis [Goldberg87]). Our use of abstraction interpretation differs from that of Mycroft, and is similar to that of Hudak and Young, because we form an abstraction of a nonstandard semantics, rather than the standard semantics, of our programming language.

2. A Simple Higher Order Functional Language

For this discussion a very simple higher order monomorphically typed strict functional language, *nml* (for *n*ot *m*uch of a *l*anguage), will suffice. The syntax of the language is given in figure 1, although we omit the type declarations. The standard semantic domains of nml are as follows:

D, the standard domain of values,

$Env: Id \rightarrow D$, the domain of environments,

$$E[\![c]\!]\ Env = c, \text{ for each constant } c \in Con$$
$$E[\![x]\!]\ Env = Env[\![x]\!], \text{ for each identifier } x \in Id$$
$$E[\![e_1 + e_2]\!]\ Env = (E[\![e_1]\!]\ Env) + (E[\![e_2]\!]\ Env)$$
$$E[\![e_1 = e_2]\!]\ Env = (E[\![e_1]\!]\ Env = E[\![e_2]\!]\ Env)$$
$$E[\![e_1\ e_2]\!]\ Env = (E[\![e_1]\!]\ Env)\ (E[\![e_2]\!]\ Env)$$
$$E[\![e_1 \rightarrow e_2, e_3]\!]\ Env = E[\![e_1]\!]\ Env \rightarrow E[\![e_2]\!]\ Env, E[\![e_3]\!]\ Env$$
$$E[\![\lambda x.e]\!]\ Env = \lambda y.E[\![e]\!]Env[y/x]$$
$$E[\![\textbf{let } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e]\!]\ Env = E[\![e]\!]\ Env'$$
$$\text{where } Env' = [(E[\![e_1]\!]\ Env')/x_1, \ldots, (E[\![e_n]\!]\ Env')/x_n]$$

Figure 2. The standard semantics of nml

$E: Exp \rightarrow Env \rightarrow D$, the semantic function for expressions.

The standard semantic function E is defined in figure 2. For notational convenience, all syntactic objects are printed in boldface type, including all nml identifiers. Variables referring to syntactic objects are printed in boldface italic type. Semantic variables are printed in non-bold italic type.

3. An Exact Escape Semantics

In this section, we describe a nonstandard semantics for nml such that the result of a function call indicates whether a particular parameter escapes. The escape semantic domains are defined as follows:

$D_e^{int} = D_e^{bool} = \ldots = 2 \times \{err\}$, where 2 is the two element domain ordered by $0 \leq 1$,

$D_e^{T_1 \rightarrow T_2} = 2 \times (D_e^{T_1} \rightarrow D_e^{T_2})$, for any types T_1 and T_2,

$$D_e = \sum_T D_e^T,$$

$Env_e = Id \rightarrow D_e$, the domain of escape environments.

The semantic function is

$E_e: Exp \rightarrow Env_e \rightarrow D_e$

and is defined in figure 3.

Under these semantics, the value of an expression is a pair whose first element is a boolean (0 or 1) that

$$E_e[\![c]\!]\ Env = \langle 0, err \rangle, \text{ for any constant } c$$
$$E_e[\![x]\!]\ Env = Env[\![x]\!]$$
$$E_e[\![e_1 + e_2]\!]\ Env = \langle 0, err \rangle, \text{ likewise for the other arithmetic operators}$$
$$E_e[\![e_1\ e_2]\!]\ Env = (E_e[\![e_1]\!]\ Env)_{(2)}\ (E_e[\![e_2]\!]\ Env)$$
$$E_e[\![e_1 \rightarrow e_2, e_3]\!]\ Env = Oracle[\![e_1]\!] \rightarrow E_e[\![e_2]\!]\ Env, E_e[\![e_3]\!]\ Env$$
$$E_e[\![\lambda x.e]\!]\ Env = \langle v, \lambda y.\ E_e[\![e]\!]\ Env[y/x] \rangle$$
$$\text{where } v = \bigvee_{z \in F} (Env[\![z]\!])_{(1)}, \text{ and } F \text{ is the set of free variables in } (\lambda x.e)$$
$$E_e[\![\textbf{let } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e]\!]\ Env = E_e[\![e]\!]\ Env'$$
$$\text{where } Env' = [(E_e[\![e_1]\!]\ Env')/x_1, \ldots, (E_e[\![e_n]\!]\ Env')/x_n]$$

Figure 3. The exact escape semantics of nml

indicates whether a particular parameter escapes, and whose second element is a function that captures the higher order behavior of the expression. *err* denotes a non-function value.

Given an $x \in D_e$ we use the notation $x_{(1)}$ and $x_{(2)}$ to refer to the first and second elements of x, respectively. The domain D_e is partially ordered in the standard way:

$$\forall\, x,y \in D_e, x \leq y \text{ iff } x_{(1)} \leq y_{(1)} \text{ and } x_{(2)} \leq y_{(2)}$$

For each type $T = T_1 \rightarrow T_2$, there is a bottom element $\perp_T$ of D_e defined as follows:

$$\perp_T = < 0,\ \lambda x.\ \perp_{T_2} >$$

For each primitive type T, $\perp_T = < 0, err >$.

In order to return the actual escape value of each expression, we must be able to determine which branch of a conditional would be evaluated at run-time. The only way to do this would be to embed the standard semantics within the escape semantics. For convenience, we instead resort to an oracle to choose the appropriate branch of the conditional.

Given an nml function **f**, its meaning under the escape semantics will be a pair $<f_{(1)} f_{(2)}>$. We then use $f_{(2)}$ to determine if a particular argument in a call to **f** escapes. Suppose, for example, we want to know, given the function application (**f x**), if **x** escapes. To do so, we let $<x_{(1)}, x_{(2)}>$ be the value of **x** under that escape semantics and let $y = f_{(2)} <1, x_{(2)}>$. If $y_{(1)} = 1$ then **x** escapes in the standard semantics, otherwise **x** does not. Section 6 gives a detailed description of how the escape semantics is used.

4. The Abstract Escape Semantics

We now present an abstraction of the exact escape semantics that allows an approximation of the exact escape behavior to be found at compile time. The semantic domains are essentially identical to those of the exact escape semantics:

$$D_{ae}^{int} = D_{ae}^{bool} = \ldots = 2 \times \{err\},$$

$$D_{ae}^{T_1 \rightarrow T_2} = 2 \times (D_{ae}^{T_1} \rightarrow D_{ae}^{T_2}),$$

$$D_{ae} = \sum_T D_{ae}^{T},$$

$$Env_{ae} = Id \rightarrow D_{ae}.$$

The semantic function

$$E_{ae}: Exp \rightarrow Env_{ae} \rightarrow D_{ae}$$

is defined in figure 4. The difference between the exact and the abstract semantics lies in the handling of the conditional. Rather than referring to the standard semantics (as denoted by the oracle) the conditional is handled by taking the least upper bound of the escape values of the two branches.

5. Termination

In our abstract escape semantics, a function may be expressed recursively and is thus defined as the least fixpoint of the corresponding functional. That is, for the function

$$f = F(f)$$

where f is of type T and F is a functional (corresponding to the body of f), the meaning of f is defined to be the least function satisfying the above equation. Domain theory tells us that the least fixpoint f can be found as follows:

$$f = \lim_{i \rightarrow \infty} F^i(\perp_T)$$

where $F^0(x) = x$ and $F^i(x) = F(F^{i-1}(x))$.

$$E_{ae}[\![c]\!]\ Env = \langle 0, err \rangle$$

$$E_{ae}[\![x]\!]\ Env = Env[\![x]\!]$$

$$E_{ae}[\![e_1 + e_2]\!]\ Env = \langle 0, err \rangle, \text{ likewise for the other arithmetic operators}$$

$$E_{ae}[\![e_1\ e_2]\!]\ Env = (E_{ae}[\![e_1]\!]\ Env)_{(2)}\ (E_{ae}[\![e_2]\!]\ Env)$$

$$E_{ae}[\![e_1 \rightarrow e_2, e_3]\!]\ Env = (E_{ae}[\![e_2]\!]\ Env) \sqcup (E_{ae}[\![e_3]\!]\ Env)$$

$$E_{ae}[\![\lambda x.e]\!]\ Env = \langle v, \lambda y.\ E_{ae}[\![e]\!]\ Env[y/x] \rangle$$

$$\text{where } v = \bigvee_{z \in F} (Env[\![z]\!])_{(1)}, \text{ and } F \text{ is the set of free variables in } (\lambda x.e)$$

$$E_{ae}[\![\textbf{let } x_1 = e_1; \ldots; x_n = e_n \textbf{ in } e]\!]\ Env = E_{ae}[\![e]\!]\ Env'$$

$$\text{where } Env' = [(E_{ae}[\![e_1]\!]\ Env')/x_1, \ldots, (E_e[\![e_n]\!]\ Env')/x_n]$$

Figure 4. The abstract escape semantics of nml

To ensure that our analysis terminates, we must show that a fixpoint is reached in finite time. That is, for all functions defined according to the above equation, there must exist some j such that

$$F^k(\perp_T) = F^j(\perp_T)$$

for all $k > j$.

A simple way of showing that there exists such a j is to show that every functional F must be monotonic and that the fixpoint iteration is performed over a finite domain (using the technique described in [BHA85]).

Every functional is composed of the monotonic operation $\vee$ (logical *or*) and the least upper bound operator (as defined in figure 4) and is thus monotonic. Furthermore, each subdomain $D_{ae}{}^T$ is finite since $D_{ae}{}^{T_0}$ is finite for each primitive type T_0 (**int**, **bool**, etc.) and $D_{ae}{}^{T_1 \rightarrow T_2}$ is finite whenever $D_{ae}{}^{T_1}$ and $D_{ae}{}^{T_2}$ are finite. When finding the least fixpoint of a function of type T we need only search over the subdomain $D_{ae}{}^T$. Thus, the least fixpoint can be computed in a finite number of iterations. Figure 5 contains an example of fixpoint finding.

6. Using the Abstract Functions

In this section, we describe how the abstract functions are used to detect the escape properties of the corresponding functions in an nml program.

6.1. Global Escape Analysis

Using a global escape analysis, we find escape information about each nml function f that holds true for every possible application of f. To do so, we apply the corresponding abstract function to arguments that cause the greatest escapement possible. For each type T, we define the abstract function R^T that corresponds to an nml function from which every argument escapes.

$$R^T = \lambda z_1.\ \langle z_{1(1)}, \lambda z_2.\ \langle z_{1(1)} \vee z_{2(1)}, \ldots, \lambda z_m.\ \langle \bigvee_{p=1}^{m} z_{p(1)}, err \rangle \ldots \rangle \rangle$$

where m is the number of arguments that a function of type T can take (before returning a primitive value).

Given an identifier f bound to a function of n arguments in some environment Env, $G_i(f, Env)$ returns *1* if the ith parameter could escape and *0* otherwise. G_i is defined as follows:

$$G_i(f, Env) = (E_{ae}[\![(f\ x_1 \ldots\ x_n)]\!]\ Env[y_1/x_1, \ldots, y_n/x_n])_{(1)}$$

where for all $j \leq n, j \neq i$,

Consider the following nml function definition (of type *int→int→ int*, for example):

f = λx. λy. (x=0) → y, f (x-1) y

Using the definitions in figure 4, the corresponding function in D_{ae} is described by:

$$f = <0, \lambda x.\ <x_{(1)}, \lambda y.\ E_{ae}[\![\ \textbf{(x=0)} \rightarrow \textbf{y, f (x-1) y}\ ]\!]\ [x/x, y/y, f/f]>>$$

$$= <0, \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup (f_{(2)}\ <0,err>)_{(2)}\ y)>>$$

Therefore f is the least fixpoint of the functional F defined by

$$F = \lambda f.\ <0,\ \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup (f_{(2)}\ <0,err>)_{(2)}\ y)>>$$

The least fixpoint is found by the following fixpoint iteration:

$$f^0 = F(\bot_{int\rightarrow int\rightarrow int}) = <0, \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup (\bot_{int\rightarrow int\rightarrow int(2)}\ <0,err>)_{(2)}\ y)>>$$

$$= <0, \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup \bot_{int\rightarrow int(2)}\ y>>$$

$$= <0, \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup \bot_{int})>>$$

$$= <0, \lambda x.\ <x_{(1)}, \lambda y.y>>$$

$$f^1 = F(f^0) = <0, \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup ((\lambda x.\ <x_{(1)}, \lambda y.y>)\ <0,err>)_{(2)}\ y))>>$$

$$= <0, \lambda x.\ <x_{(1)}, \lambda y.\ y \sqcup (\lambda y.y)\ y))>>$$

$$= <0, \lambda x.\ <x_{(1)}, \lambda y.\ y)>>$$

Since $f^0 = f^1 = F(f^0)$, a fixpoint has been found, thus $f = <0, \lambda x.\ <x_{(1)}, \lambda y.y)>>$

This means that when **f** is applied to two arguments, only the second argument may escape.

Figure 5. An example of fixpoint finding

$$y_j = <0, R^{T_j}>$$

where T_j is the type of x_j and

$$y_i = <1, R^{T_i}>.$$

Since each $y_{j(2)}$ is a function from which every argument escapes, and since the abstract function for f is monotonic, $G_i(f, Env)$ provides the worst case behavior with respect to the escapement of f's ith argument.

6.2. Local Escape Analysis

Generally, we would like to know if an argument escapes from a particular call to a function f. This depends on the values of the arguments of that call. We define the function L_i such that $L_i(f, e_1, ..., e_n, Env)$ returns *1* if the ith argument of $(f\ e_1 \ldots e_n)$ might escape, *0* otherwise. The environment Env must be an environment mapping the free identifiers within e_1 through e_n to elements of D_{ae}. The function L_i is defined as follows

$$L_i(f, e_1, ..., e_n, Env) = (E_{ae}[\![\ (f\ x_1 \ldots\ x_n)\ ]\!]\ Env[y_j/x_j])_{(1)}$$

where, for all $j \le n, j \neq i$,

$$y_j = <0,\ (E_{ae}[\![\ e_j\ ]\!]\ Env)_{(2)}>$$

and

$$y_i = <1,\ (E_{ae}[\![\ e_i\ ]\!]\ Env)_{(2)}>.$$

Env = [f/**f**, h/**h**, p/**p**, q/**q**, g/**g**], where

f = $\langle 0, \lambda x.\ \langle x_{(1)},\ \lambda y.\ E_{ae}\ [\![\ \mathbf{x+y}\ ]\!]\ Env[x/\mathbf{x}, y/\mathbf{y}]\rangle\rangle$

= $\langle 0, \lambda x.\ \langle x_{(1)},\ \lambda y.\ \langle 0, err\rangle\rangle\rangle$

p = q = $\langle 0,\ \lambda b.\ \langle 0, err\rangle\rangle$

h = $\langle 0, \lambda a.\ E_{ae}\ [\![\ \mathbf{(a=0) \rightarrow p, q}\ ]\!]\ Env[a/\mathbf{a}]\rangle$

= $\langle 0, \lambda a.\ p \sqcup q\rangle$

= $\langle 0, \lambda a.\ \langle 0,\ \lambda b.\ \langle 0, err\rangle\rangle\rangle$

g = $\langle 0, \lambda m.\ \langle m_{(1)},\ \lambda n.\ E_{ae}\ [\![\ \mathbf{m\ n}\ ]\!]\ Env[m/\mathbf{m}, n/\mathbf{n}]\rangle\rangle$

= $\langle 0, \lambda m.\ \langle m_{(1)},\ \lambda n.\ m_{(2)}\ n\rangle\rangle$

Figure 6. Abstract Escape Functions

7. Examples

Consider the following nml program:

```
let  f = λx. λy. x+y;
     h = λa. (a=0) → p, q;
     p = λb. b+1;
     q = λb. b-1;
     g = λm. λn. m n;
in   ... (g f 4) ... (g h 4) ...
```

Figure 6 shows the corresponding abstract escape functions. Our aim to is analyze the escape properties of g, both globally and locally.

7.1. A Global Escape Analysis Example

To find the global (i.e. worst case) escape property of **g**, we apply the global analysis function G_i described in section 6.1 to the abstract function g and the environment Env shown in figure 6. We assume **g** is of type $(int \rightarrow int \rightarrow int) \rightarrow int \rightarrow int \rightarrow int$.

$$G_1(\mathbf{g}, Env) = (E_{ae}\ [\![\ (\mathbf{g}\ x_1\ \mathbf{x2})\ ]\!]\ \text{Env}[y_1/x_1, y_2/x_2])_{(1)}$$

where $y_1 = \langle 1, \lambda z_1.\ \langle z_{1(1)},\ \lambda z_2.\ \langle z_{1(1)} \vee z_{2(1)}, \text{err}\rangle\rangle\rangle$ and $y_2 = \langle 0, err\rangle$.

Thus

$$\begin{aligned} G_1(\mathbf{g}, Env) &= ((g_{(2)}\ y_1)_{(2)}\ \ y_2)_{(1)} \\ &= (((\lambda m.\ \langle m_{(1)},\ \lambda n.\ m_{(2)}\ n\rangle)\ y_1)_{(2)}\ \ y_2)_{(1)} \\ &= ((\lambda z_1.\ \langle z_{1(1)},\ \lambda z_2.\ \langle z_{1(1)} \vee z_{2(1)}, \text{err}\rangle\rangle)\ \ \langle 0, err\rangle)_{(1)} \\ &= 0 \end{aligned}$$

indicating that **g**'s first parameter can never escape.

$$G_2(\mathbf{g}, Env) = (E_{ae}\ [\![\ (\mathbf{g}\ x_1\ \mathbf{x2})\ ]\!]\ \text{Env}[y_1/x_1, y_2/x_2])_{(1)}$$

where $y_1 = \langle 0, \lambda z_1.\ \langle z_{1(1)},\ \lambda z_2.\ \langle z_{1(1)} \vee z_{2(1)}, \text{err}\rangle\rangle\rangle$ and $y_2 = \langle 1, err\rangle$.

Thus

$$\begin{aligned} G_2(\mathbf{g}, Env) &= ((g_{(2)}\ y_1)_{(2)}\ \ y_2)_{(1)} \\ &= (((\lambda m.\ \langle m_{(1)},\ \lambda n.\ m_{(2)}\ n\rangle)\ y_1)_{(2)}\ \ y_2)_{(1)} \\ &= ((\lambda z_1.\ \langle z_{1(1)},\ \lambda z_2.\ \langle z_{1(1)} \vee z_{2(1)}, \text{err}\rangle\rangle)\ \ \langle 1, err\rangle)_{(1)} \\ &= 1 \end{aligned}$$

indicating that the first parameter to **g** might escape in some situations.

7.2. A Local Escape Analysis Example

To find what arguments escape from the each application of **g** in the program, we use the local analysis function L_i described in section 6.2. Since we know from the global analysis of **g** that its first parameter can never escape, we only need to test if the second parameter escapes (using L_2). For the expression **(g f 4)**,

$$\begin{aligned} L_2(\mathbf{g}, \mathbf{f}, \mathbf{4}, Env) &= (E_{ae}[\![(g\ x_1\ x_2)]\!]\ \text{Env}[\langle 0, f_{(2)}\rangle/x_1, \langle 1, err\rangle/x_2])_{(1)} \\ &= ((g_{(2)}\ \langle 0, f_{(2)}\rangle)_{(2)}\ \langle 1, err\rangle)_{(1)} \\ &= (((\lambda m.\ \langle m_{(1)},\ \lambda n.\ m_{(2)}\ n\rangle)\ \langle 0, \lambda x.\ \langle x_{(1)},\ \lambda y.\ \langle 0, err\rangle\rangle\rangle)_{(2)}\ \langle 1, err\rangle)_{(1)} \\ &= ((\lambda x.\ \langle x_{(1)},\ \lambda y.\ \langle 0, err\rangle\rangle\rangle)_{(2)}\ \langle 1, err\rangle)_{(1)} \\ &= 1. \end{aligned}$$

This indicates that the second argument to **g** escapes. For the expression **(g h 4)**,

$$\begin{aligned} L_2(\mathbf{g}, \mathbf{h}, \mathbf{4}, Env) &= (E_{ae}[\![(g\ x_1\ x_2)]\!]\ \text{Env}[\langle 0, h_{(2)}\rangle/x_1, \langle 1, err\rangle/x_2])_{(1)} \\ &= ((g_{(2)}\ \langle 0, h_{(2)}\rangle)_{(2)}\ \langle 1, err\rangle)_{(1)} \\ &= (((\lambda m.\ \langle m_{(1)},\ \lambda n.\ m_{(2)}\ n\rangle)\ \langle 0, \lambda a.\ \langle 0,\ \lambda b.\ \langle 0, err\rangle\rangle\rangle)_{(2)}\ \langle 1, err\rangle)_{(1)} \\ &= ((\ \lambda a.\ \langle 0,\ \lambda b.\ \langle 0, err\rangle\rangle)_{(2)}\ \langle 1, err\rangle)_{(1)} \\ &= 0. \end{aligned}$$

This indicates that no argument to **g** escapes (even though the result is a partial application).

8. Escape Analysis on Lists

We have not yet discussed escape analysis in the presence of the list operators **cons**, **car**, and **cdr**. We extend the abstract semantic function E_{ae} as follows:

$$\begin{aligned} E_{ae}[\![\ \mathbf{cons}\ ]\!]\ Env &= \langle 0, \lambda a.\ \langle a_{(1)},\ \lambda b. \langle a_{(1)} \vee b_{(1)},\ err\rangle\rangle\rangle \\ E_{ae}[\![\ \mathbf{cdr}\ ]\!]\ Env &= \langle 0, \lambda x.\ \langle x_{(1)}, err\rangle\rangle \\ E_{ae}[\![\ \mathbf{car}\ ]\!]\ Env &= \langle 0, R^T\rangle \end{aligned}$$

where T is the type of the elements of the list to which **car** is being applied and R^T is defined in section 6.1. In other words, once an object has been placed on a list, we are unable to determine when it is removed. This means that if some head or tail of a list escapes from a function then all of the elements of the list are seen as escaping. In addition, the function value of the car of a list is seen as the maximally escaping function of that type. Admittedly, this is an unsatisfactory analysis on lists. We are working on an escape analysis that could be applied to lists in a manner similar to the way that strictness analysis was extended to lists [Wadler87].

9. Conclusions

We have taken an existing, useful optimization and used denotational semantics and abstract interpretation to apply it to higher order programming languages. We have yet to implement the analysis in a real compiler and thus it remains to be seen if the benefit of the analysis outweighs its cost (mainly fixpoint finding).

10. Acknowledgments

We would like to thank the National Science Foundation for funding this work. We would also like to thank our wives, Wendy Goldberg and Jihwa Park, for their support and encouragement. We also thank our children, Jonathan Goldberg and Grace Park, for providing pleasant distractions.

References

[AM87]
A. Appel and D.B. MacQueen. A standard ML compiler. In *Proceedings of the 1987 Conference on Functional Programming and Computer Architecture*. September, 1987.

[Barth77]
J.M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7), July 1977.

[BHA85]
G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects*, LNCS 217. Springer-Verlag. 1985

[CC77]
P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*. January 1977.

[Chase87]
D.R. Chase. *Garbage Collection and Other Optimizations*. Ph.D. Thesis, Rice University. 1987

[Chase88]
D.R. Chase. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. June, 1988.

[Goldberg87]
B. Goldberg. Detecting sharing of partial applications in functional programs. In *Proceedings of the 1987 Conference on Functional Programming and Computer Architecture*. September, 1987.

[HY86]
P. Hudak and J. Young. Higher-order strictness analysis for the untyped lambda calculus. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*. January, 1986.

[JM76]
N. Jones and S. Muchnick. Binding time optimization in programming languages: An approach to the design of an ideal language. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Programming Languages*. January 1976.

[Kranz88]
D. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. Ph.D. Thesis, Yale University, Department of Computer Science. May 1988.

[MJ81]
S. Muchnick and N. Jones, editors. *Flow Analysis and Optimization of LISP-like Structures*. Prentice-Hall, 1981.

[Mycroft81]
A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. Ph.D. Thesis, University of Edinburgh. 1981.

[RM88]
C. Ruggieri and T.P. Murtagh. Lifetime analysis of dynamically allocated objects. *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*. January, 1988.

[Schwartz75]
J.T. Schwartz. Optimization of very high level languages - I. Value transmission and its corollaries. *Journal of Computer Languages*, 1:161-194, 1975.

[Wadler87]
P. Wadler, Strictness analysis on non-flat domains. In *Abstract Interpretation of Declarative Languages*. C. L. Hankin and S. Abramsky, editors. Ellis Horwood, 1987.