Escape Analysis on Lists*

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

Abstract

Higher order functional programs constantly allocate objects dynamically. These objects are typically cons cells, closures, and records and are generally allocated in the heap and reclaimed later by some garbage collection process. This paper describes a compile time analysis, called escape analysis, for determining the lifetime of dynamically created objects in higher order functional programs, and describes optimizations that can be performed, based on the analysis, to improve storage allocation and reclamation of such objects. In particular, our analysis can be applied to programs manipulating lists, in which case optimizations can be performed to allow whole cons cells in spines of lists to be either reclaimed at once or reused without incurring any garbage collection overhead. In a previous paper on escape analysis [10], we had left open the problem of performing escape analysis on lists.

Escape analysis simply determines when the argument (or some part of the argument) to a function call is returned by that call. This simple piece of information turns out to be sufficiently powerful to allow stack allocation of objects, compile-time garbage collection, reduction of run-time storage reclamation overhead, and other optimizations that are possible when the lifetimes of objects can be computed statically.

Our approach is to define a high-level non-standard semantics that, in many ways, is similar to the standard semantics and captures the escape behavior caused by the constructs in a functional language. The advantage of our analysis lies in its conceptual simplicity and portability (i.e. no assumption is made about an underlying abstract machine).

1 Introduction

Higher order functional programs constantly allocate objects dynamically. These objects are typically cons cells, closures, and records and are generally allocated in the heap and reclaimed later by some garbage collection process. Garbage collection overhead can be reduced by performing compile time analyses that allow the following optimizations to be performed:

- *Stack allocation*: Objects that would otherwise be allocated in the heap and then reclaimed using garbage collection are allocated in an activation record on the stack and automatically (and cheaply) reclaimed when the activation record is popped off the stack.
- *In-place Reuse*: When objects (such as cons cells) are no longer needed, they can be reused directly by the program without invoking the garbage collector. A typical example of this is when a function constructs a new list by destructively modifying the cons cells that were contained in an argument to that function.
- *Block Allocation/Reclamation*: A number of objects (such as the cons cells of a list) are allocated together in a contiguous block of memory. Eventually, the whole block is put on the free list, rather than the individual objects. This allows reclamation of larger segments of memory,

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

†Present address: School of Computer Science, University of Windsor, 401 Sunset Ave, Windsor, Ontario, Canada N9B 3P4, Email: ypark@cs.uwindsor.ca

‡Authors' address: 251 Mercer Street, New York, N.Y. 10012, Email: park@cs.nyu.edu, goldberg@cs.nyu.edu.

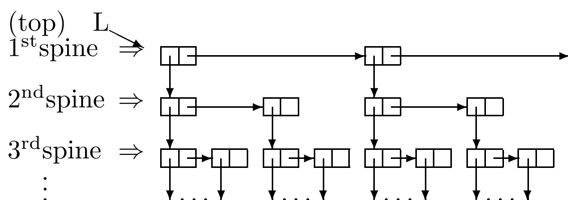

Figure 1: Spines of a List

and reduces run-time overhead by avoiding the traversal of the individual objects (in a mark-sweep collection, for instance).

In this paper, we present a compile time analysis that provides sufficient information to perform all of these optimizations on higher order functional programs. The analysis is called *escape analysis*. In a previous paper [10] we described an escape analysis for non-list objects, such as closures, and left open the problem of performing the analysis in the presence of lists. This paper solves that problem. Escape analysis answers a simple question: Given a function application, does a parameter (or some part of a parameter) get returned in the result of the application? If so, the parameter is said to *escape* from the application. If the escaping parameter is a list, we would particularly like to know which spines of the parameter escape. The spines of a list are a way of describing substructures of a list and are defined as follows:

Definition 1 (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 occurrences 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 *$(d-j+1)$ spine of L.*

In Figure 1 we illustrate what we mean by the spines of a list. Naturally, an empty list (`nil`) and any non-list objects have no (or zero) spine.

We have chosen to analyze the escape properties of lists in terms of their spines for two reasons:

- It is an approximation to the run-time behavior that allows a compile-time analysis.
- It reflects the programming style commonly used for strongly typed languages, such as ML, in which lists are homogeneous (all elements have the same type) and functions (such as `append`, `map`, etc.) often operate over complete spines of lists.

The first point reflects our inability to determine precisely, without actually running the program, which individual cells of a list might escape. To form a terminating compile-time escape analysis, one must choose an approximation of program behavior. The second point reflects our belief that the spines are a good choice of approximation, since the cells of each spine of a list tend to be treated identically. Many functions, such as `append`, `reduce`, `map`, `length`, etc, operate on all cells of a spine. Many other functions have the form:

```
f L = if predicate(car L) then ....
      else f (cdr L)
```

or

```
f L x = if x = n then ...
        else f (cdr L) (arith-op x)
```

In general, it is impossible to determine at compile time when the recursion will bottom out. One simply has to assume that all cells in the spine of the list `L` will be visited.

Consider, for example, the following program:

```
let map f l = if (l=nil) then nil
              else cons (f (car l))
                        (map f (cdr l)) ;
    pair x = [car x, car (cdr x)] ;
in map pair [[1,2],[3,4],[5,6]]
```

An escape analysis would allow us to determine the following properties of the program at compile time:

1. The top spine of `pair`'s parameter does not escape from `pair`, only some elements do.
2. The top spine of `map`'s parameter `l` does not escape from `map`, and the elements of `l` escape from `map` to the extent they escape from the unknown function `f`.
3. In the call `(map pair [[1,2],[3,4],[5,6]])`, the top two spines of the second argument of `map` do not escape.

This means that (at least) the following optimizations could be performed:

- *Stack Allocation*: The spine of the list `[[1,2],[3,4],[5,6]]` and the spine of each element of the list could be allocated in the activation record for `map`. Thus, when `map` returns, the cons cells of those spines would disappear. It seems strange to allocate a list in a stack, but there is really no reason not to (as long as we keep in mind the safety considerations outlined in [6]).

- *In-place Reuse*: If the parameter `l` of `map` is unshared (sharing information can also be obtained from escape information), we can recycle the cells of the spine of `l` within `map` to be used in the spine of the result (since `l`'s spine does not escape). The call to `cons` inside `map` can reuse the first cell in the spine of `l` for the result. `Pair` can reuse the spine of `x` in its result.

- *Block Allocation/Reclamation*: If neither of the above optimizations are used, then the top two spines of the list `[[1,2],[3,4],[5,6]]` can be allocated in some block in the heap. When `map` finishes, that whole block can be placed on the free list, freeing all the cons cells without traversing the list.

This paper particularly describes an escape analysis on lists in higher-order, polymorphically-typed functional programs, and describes optimizations and other analysis (sharing analysis) that can be performed based on this analysis to improve storage allocation and reclamation of such objects. Our approach for lists could be applied to other data structures such as tuples, trees, etc.

2 Related Work

There has been a number of papers describing analyses for optimizing storage of lists and other structures. Most of these analyses have been first-order (i.e. not accounting for higher order functions) or have analyzed first-order languages. Brooks, Gabriel, and Steele [3] describe an escape analysis for numbers, but do not extend it for arbitrary objects. Ruggieri and Murtagh [18] describe a lifetime analysis for a language with side-effects and complex data structures, but, again, it is first order. Jones and Le Metayer [13] describe an algorithm, based on forward and backward analysis, for detecting sharing of objects in first order functional languages, and describe a method for reusing of cells based on the sharing analysis. We use only forward analysis providing, perhaps, a simpler conceptual framework for higher-order languages. Inoue, Seki, and Yagi [12] describe an analysis for functional languages to detect, and reclaim, run-time garbage cells based on the formal language theory and grammars. The paper focuses only on the explicit reclamation of cons cells and it is unclear that this approach could be extended for higher-order languages. Besides being higher order, the escape analysis described here is a more general lifetime analysis that can be applied to objects other than lists, and other optimizations are supported. Chase, Wegman, and Zadeck [7] describe an first-order analysis for LISP that constructs graphs representing possible list structures and analyzes the graphs for possible storage optimizations. Our analysis, in contrast, concentrates on typed languages such as ML and benefits from a type system that restricts the ways that lists can be created (for example, one cannot say `f = cons(x,x)`) and the sharing that can occur within a list. Orbit, an optimizing compiler for Scheme, uses a simple first-order escape analysis to stack allocate closures [15]. Other analyses for optimizing storage allocation were proposed in [14, 5].

Deutsch [8] presents a lifetime and sharing analysis for higher order languages. While the goals of the paper seem to be similar to ours, the approach is very different. The analysis consisted of defining a low-level operational model for a higher order functional language, translating a program into a sequence of operations in this model, and then performing an analysis to determine the lifetimes of dynamically created objects. The approach is also one of collecting interpretation, in that it analyzes a whole program to infer properties of program points. Our approach is to define a high-level non-standard semantics that in many ways is similar to the standard semantics and captures the precise escape behavior caused by the constructs in a functional language. We then define an abstraction of these semantics which provides less precise information but which allows the analysis to be performed at compile time. The advantage of our analysis lies in its conceptual simplicity and lower computational cost (compared to a collecting interpretation).

Baker [2] describes an interesting approach to higher-order escape analysis of functional languages based on the type inference (unification) technique. The analysis sketched there provides escape information of lists only, and might be extended to give information comparable to what our analysis gives. Our analysis, based on abstract interpretation, provides escapement of other objects (closures) as well as lists. We also describe how to use escape information for various optimizations.

The notion of spines of a list is used in [17] as a way of modeling the location of dynamically created references in a list structure.

3 Escape Semantics

3.1 The Language - nml

We define a simple, strict, monomorphically typed, higher order functional language. For lack of a better name, we will call this language **nml**, for **n**ot **m**uch of a **l**anguage. The syntax of **nml** is defined as follows:

$c \in Con$	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 $e ::= c \mid x \mid e_1 e_2 \mid \texttt{lambda}(x).e \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e3 \mid \texttt{letrec}\ x_1 = e_1; \ldots x_n = e_n\ \texttt{in}\ e$
$pr \in Pgm$	Programs, defined by $pr ::= \texttt{letrec}\ x_1 = e_1; \ldots x_n = e_n\ \texttt{in}\ e$

For convenience, we omit type declarations. Our analysis assumes that monomorphic type inference (perhaps requiring declarations) has already been performed. Subsequently, we will relax the monomorphic typing restriction and show that our analysis works for a polymorphic version of **nml**. Also for convenience, we will allow function definitions of the form $f\ x_1 \ldots x_n\ =\ e$ and assume that this is just syntactic sugar for $f\ = \texttt{lambda}(x_1). \ldots \texttt{lambda}(x_n).e$.

3.2 An Exact Escape Semantics

We first define a non-standard semantics of **nml**, called the *escape semantics*, such that the meaning of an expression is exact information about what escapes in the result of the expression. Since **nml** is a higher order language, the result of an expression may be a function. Such a function, represented by a closure, has two important characteristics with respect to the escape semantics:

1. The closure is an object itself. We may be interested in whether the closure escapes or not, or we may be interested in another object that is captured (bound) within the closure. Thus, the value of an expression returning a function must indicate whether an interesting object has escaped.

2. A function value may be applied to arguments (which may themselves escape from the application). Therefore, in our non-standard semantics, the escape value of a function must include its behavior as a function.

Thus, the value of an expression in our non-standard escape semantics is an element of a non-standard escape domain and must have two components. First, it must contain information about what is contained within the result of the expression. Second it must contain a function over the values in the escape domain. This approach (a two component value) was used by Hudak and Young for performing higher order strictness analysis [11].

We perform escape analysis on each argument of a function call separately. Thus, at any time we are only interested in whether or not a single object escapes. Other objects may escape in the result of a function call, but are ignored by our analysis. An object is *interesting* if it is the one whose escape behavior we are trying to determine. If a function has n parameters, then we perform escape analysis n times, each time treating a different parameter as interesting. Thus, our escape semantics is defined in terms of interesting objects; The value of an expression in our escape semantics will indicate if some portion of, or all of, an interesting object is contained in the result.

Given an expression, we want its corresponding value in the escape semantic domain to tell us how many spines (if any) of an interesting object are returned by that expression. Values in our semantic domain D_e have two components; The first component is an element of *basic escape domain* B_e, which is a domain of pairs ordered as follows:

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

where d is some integer constant, i.e. for $a = \langle a_1, a_2\rangle \in B_e$ and $b = \langle b_1, b_2\rangle \in B_e$, $a \sqsubseteq b$ iff $a_1 \leq b_1$ and $a_2 \leq b_2$. The constant d is fixed for each program on which our escape analysis is performed. Each expression in the program returns a value whose first component has the following meaning:

- $\langle 1, i\rangle$: The bottom i spines of an interesting object *is* contained in the value of the expression. (If an interesting object is not a list then i will always be 0, which means that an indivisible interesting object is contained in the value of the expression.)

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

The second component is a function in $(D_e \to D_e)$.

The escape semantic domains are defined as follows (in the style of [4]):

$$\begin{aligned} D_e^{int} &= B_e \times \{err\} \\ D_e^{bool} &= B_e \times \{err\} \\ D_e^{\tau_1 \to \tau_2} &= B_e \times (D_e^{\tau_1} \to D_e^{\tau_2}) \\ D_e^{\tau\ list} &= (B_e \times \{err\}) + (D_e^{\tau} \times D_e^{\tau\ list}) \end{aligned}$$

where err is a function (weaker than all others) that can never be applied.

$$\begin{aligned} D_e &= \sum_{\tau} D_e^{\tau} && \text{escape semantic domain} \\ Env_e &= Id \to D_e && \text{escape environment} \end{aligned}$$

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 two components of the first element of x are referred to as $x_{(1)(1)}$ and $x_{(1)(2)}$, respectively (i.e., x has the form $\langle\langle x_{(1)(1)}, x_{(1)(2)}\rangle, x_{(2)}\rangle$.

The escape semantic functions are:

$$
\begin{array}{lcl}
C & : & Con \to D_e \\
E & : & Exp \to Env_e \to D_e \\
P & : & Pgm \to D_e
\end{array}
$$

The semantic function C for constants is defined as follows:

$$
\begin{array}{l}
C[\![c]\!] = \langle\langle 0,0\rangle, err\rangle, \text{ where} \\
\quad c \in \{\ldots, \mathtt{0}, \mathtt{1}, \ldots, \mathtt{true}, \mathtt{false}, \mathtt{nil}\} \\
C[\![c]\!] = \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\langle\langle 0,0\rangle, err\rangle\rangle\rangle, \\
\quad \text{where} c \in \{+, -, =\} \\
C[\![\mathtt{cons}]\!] = \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.\ \text{pair}(x,y)\rangle\rangle \\
C[\![\mathtt{car}]\!] = \langle\langle 0,0\rangle, \lambda x.\ \text{fst}(x)\rangle \\
C[\![\mathtt{cdr}]\!] = \langle\langle 0,0\rangle, \lambda x.\ \text{snd}(x)\rangle \\
\quad \text{where } \ \text{pair}(x,y) = \langle x, y\rangle,\ \text{fst}(x) = x_{(1)}, \\
\qquad\quad \text{and } \text{snd}(x) = x_{(2)}. \\
C[\![\mathtt{null}]\!] = \langle\langle 0,0\rangle, \lambda x.\langle\langle 0,0\rangle, err\rangle\rangle
\end{array}
$$

The escape semantic function E for expressions and P for programs are defined as follows:

$$
\begin{array}{l}
E[\![c]\!]env_e = C[\![c]\!] \\
E[\![x]\!]env_e = env_e[\![x]\!] \\
E[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!]env_e = \\
\quad \text{if } \mathbf{Oracle}(e_1) \text{ then } E[\![e_2]\!]env_e \text{ else } E[\![e_3]\!]env_e \\
E[\![e_1 e_2]\!]env_e = (E[\![e_1]\!]env_e)_{(2)}\ (E[\![e_2]\!]env_e) \\
E[\![\mathtt{lambda}(x).e]\!]env_e = \\
\quad \langle V, \lambda y.E[\![e]\!]env_e[x \mapsto y]\rangle \\
\quad \text{where} \\
\quad V = \langle 0,0\rangle \sqcup (\bigsqcup_{z \in F^{non_list}} (env_e[\![z]\!])_{(1)}) \\
\qquad\quad \sqcup (\bigsqcup_{z \in F^{list}} (\bigsqcup_{p \ \mathbf{in}\ (env_e[\![z]\!])} p_{(1)}))
\end{array}
$$

Here, p **in** $(env_e[\![z]\!])$ denotes that p is an escape pair in $env_e[\![z]\!]$, F^{non_list} is the set of non-list type free identifiers in $\mathtt{lambda}(x).e$, and F^{list} is the set of list type free identifiers in $\mathtt{lambda}(x).e$).

$$
\begin{array}{l}
E[\![\mathtt{letrec}\ x_1 = e_1; \ldots x_n = e_n; \mathtt{in}\ e]\!]env_e = \\
\\
\quad E[\![e]\!]env'_e \\
\quad \text{where} \\
\quad env'_e = env_e[x_1 \mapsto E[\![e_1]\!]env'_e, \\
\qquad\qquad\qquad \ldots, x_n \mapsto E[\![e_n]\!]env'_e] \\
P[\![pr]\!] = E[\![pr]\!]nullenv_e
\end{array}
$$

In order to return the actual escape value of each expression, we must be able to determine which branch of the conditional primitive `if-then-else` would be evaluated at run-time. Here, for convenience, we instead resort to an *oracle* to choose the appropriate branch of the `if`. Note that free identifiers are treated separately according to whether they are of a list type or a non-list type. $nullenv_e$ is a escape environment that maps every identifier to the least element of its escape semantic domain.

3.3 Correctness

One naturally wonders what the relationship between the standard semantics and our non-standard escape semantics is. This is especially important if one wants to prove that the information provided by escape analysis is correct. However, it should be noticed that our escape analysis strives to gain information about the run-time behavior of a certain implementation that uses a stack and a heap and uses aliasing, rather than copying, of aggregate objects. Therefore, it is an operational semantics (perhaps couched in denotational semantic terms) of which our escape semantics can be considered an abstraction. Although we do not have space in this extended abstract to provide the operational definition of our abstract machine, we can give such a definition and prove correctness properties of our analysis with respect to that definition. To be complete, of course, we would have to prove that the abstract machine implements the standard semantic definition of the language.

3.4 Abstraction of the Escape Semantics

The escape semantics presented in the last section specifies exact escape information about functions. But, it is not suitable as a basis for compile time analysis because conditionals cannot be evaluated at compile time and the subdomain of escape values for lists is infinite.

In this section, we present a safe and computable but less complete abstraction of the exact escape semantics that allows an approximation of the exact escape behavior to be found at compile time. We modified the meanings of elements in the basic escape domain B_e. We represent lists as finite objects by combining the escape values of all the elements into a single value. We then modified the escape semantic functions for constants and expressions to approximate the escape behavior of expressions by assuming that both branches of a conditional could be taken.

The interpretation of elements of the basic escape domain B_e is modified as follows:

- $\langle 1, i\rangle$: The bottom i spines of an interesting object *may be* contained in the value of the expression. (If an interesting object is not a list then i will always be 0.)

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

The escape semantic subdomain $D_e^{\tau\ list}$ for lists of type $\tau\ list$ is modified as follows:

$$D_e^{\tau\ list} \quad = \quad D_e^{\tau}$$

The escape semantic function C for constants is modified as follows:

$$C[\![\texttt{nil}^{\tau\ \texttt{list}}]\!] = \bot_\tau$$
$$(\bot_\tau \text{ is the bottom element in } D_e^\tau)$$
$$C[\![\texttt{cons}]\!] = \langle\langle 0,0\rangle, \lambda x.\langle x_{(1)}, \lambda y.x \sqcup y\rangle\rangle$$
$$C[\![\texttt{car}^{\texttt{s}}]\!] = \langle\langle 0,0\rangle, \lambda x.\text{sub}^{\text{s}}(x)\rangle$$
$$\text{where } \text{sub}^{\text{s}}(z) = \text{if } (z_{(1)(2)} = \text{s}) \text{ then } \langle z_{(1)(1)}, z_{(1)(2)} - 1, z_{(2)}\rangle \text{ else } z$$

The typed constant function $\texttt{car}^{\texttt{s}}$ is applied to a list that has `s` spines. (For each `car` in a program, `s` can be statically determined by type inference.) If that list contains the bottom n spines of an interesting object then the first component of the list's escape value will be $\langle 1, n\rangle$. There are two possible results when $\texttt{car}^{\texttt{s}}$ is applied:

- If $\texttt{s} = n$, then the nth spine (from the bottom) of the interesting object is part of the top spine of the list. Thus, taking the car of the list returns an object containing at most $n-1$ spines of the interesting object. Thus the result should have the value $\langle 1, n-1\rangle$.
- If $\texttt{s} > n$, then the nth spine (from the bottom) of the interesting object is not part of the top spine of the list. Thus, applying car to the list returns a list that could contain the n spines of the interesting object.

Notice that `s` cannot be less than n, since a list with s spines cannot contain a list with more than s spines.

The escape semantic function E for expressions of `if-then-else` and `lambda(x).e` is modified as follows:

$$E[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]env_e = (E[\![e_2]\!]env_e)_{(2)} \sqcup (E[\![e_3]\!]env_e)$$
$$E[\![\texttt{lambda}(x).e]\!]env_e = \langle V, \lambda y.E[\![e]\!]env_e[x \mapsto y]\rangle$$
$$\text{where } V = \langle 0,0\rangle \sqcup (\bigsqcup_{z \in F} (env_e[\![z]\!])_{(1)})$$

Here, F is the set of all free identifiers in $\texttt{lambda}(x).e$.

3.5 Safety and Termination

The safety of interpretation under the abstract escape semantics with respect to the exact escape semantics means that whenever an object escapes under the exact escape semantics it escapes in the abstract escape semantics. This can be proved by showing that, using the principle of abstract interpretation, the abstract escape semantics is a safe abstraction of the exact escape semantics [16].

The termination of interpretation under the abstract escape semantics can be proved as follows: Since **nml** functions are recursive, their values in the escape domain may be computed by the usual fixpoint iteration. The body of the functional corresponding to a recursive function is composed of the monotonic least upper bound operator and other monotonic operators. Thus each functional is monotonic and its fixpoint can be found in finite time if the domain is finite. For each type τ, we defined a finite domain D_e^τ. When computing the fixpoint of a functional of type $\tau \to \tau$, we need only iterate over values in the finite domain D_e^τ. Thus, the fixpoint iteration terminates.

4 Escape Test on Lists

We use the abstract escape semantics to infer escape properties of programs. Below we show how it is used to determine both the global and local escape properties of programs. Global escape analysis is performed on a function definition, and thus gives general information about the escape properties of that function in any possible application. Local escape analysis determines the escape behavior of a particular function call, and yields more specific results.

4.1 Global Escape Test

In global escape analysis, we find escape information about a function `f` in an **nml** program that holds true for every possible application of `f`. To do so, we apply the abstract escape semantic value of `f` to arguments that cause the greatest escapement possible.

Definition 2 (Worst-case escape function) *For each non-list type τ, we define the function W^τ that corresponds to an* **nml** *function from which every argument escapes:* $W^\tau = \lambda x_1.\langle x_{1(1)}, \lambda x_2.\langle x_{1(1)} \sqcup x_{2(1)}, \ldots, \lambda x_m.\langle \bigsqcup_{i=1}^{m} x_{i(1)}, err\rangle \ldots\rangle\rangle$ *for $m \geq 1$ and $W^\tau = err$ for $m = 0$, where m is the number of arguments that a function of type τ can take before returning a primitive value. For each list type of τ list, $W^{\tau\ list}$ is defined to be W^τ.*

Given a function f of n arguments, the position i of an interesting parameter, and the abstract escape semantic environment env_e mapping f to an element of the abstract escape semantic domain D_e, the global escape test function $G(f, i, env_e)$ determines how much of the ith parameter of f could possibly escape f globally. It is defined as follows:

$$G(f, i, env_e) = (E_e[\![f\ x_1\ \ldots\ x_n]\!]\ env_e[x_i \mapsto y_i])_{(1)}$$

where $y_i = \langle\langle 1, s_i\rangle, W^{\tau_i}\rangle$, s_i is the number of spines of the ith parameter of f (if it is a list type, otherwise s_i is 0), τ_i is the type of the ith parameter of f, and for all $j \leq n$ and $j \neq i$, $y_j = \langle\langle 0, 0\rangle, W^{\tau_j}\rangle$, τ_j is the type of the jth parameter of f. Note that the whole i^{th} argument with s_i spines is interesting, and any other argument is not interesting. Then, ¿from the result of the global escape test function, we can conclude as follows:

- If $G(f, i, env_e) = \langle 0, 0\rangle$ then we conclude that none of the ith argument escapes f in any possible application of f to n arguments.
- If $G(f, i, env_e) = \langle 1, k\rangle$ then we conclude that, if $s_i \geq 1$ then, the top $(s_i - k)$ spines of the ith argument *do not* escape f in any possible application of f to n arguments, but the bottom k spines of the ith argument *could* escape f in some application of f to n arguments. (If $s_i = 0$ then the ith argument, which is not a list type, *could* escape in some application of f to n arguments.)

4.2 Local Escape Test

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. Given a function of n arguments in an application $f\ e_1\ \ldots\ e_n$, the position i of an interesting parameter, and the abstract escape semantic environment env_e mapping f and the free identifiers within e_1 through e_n to elements of the abstract escape semantic domain D_e, the local escape test function $L(f, i, e_1, \ldots, e_n, env_e)$ determines how much of the ith parameter of f could escape f in the evaluation of $f\ e_1\ \ldots\ e_n$. It is defined as follows:

$$L(f, i, e_1, \ldots, e_n, env_e) = (E_e[\![f\ x_1\ \ldots\ x_n]\!]\ env_e[x_i \mapsto z_i])_{(1)}$$

where $z_i = \langle\langle 1, s_i\rangle, (E_e[\![e_i]\!]\ env_e)_{(2)}\rangle$, s_i is the number of spines of e_i (if it is a list type, otherwise s_i is 0), and for all $j \leq n$ and $j \neq i$, $z_j = \langle\langle 0, 0\rangle, (E_e[\![e_j]\!]\ env_e)_{(2)}\rangle$. Similarly, we can determine local escape information ¿from the result of the local escape test function.

5 Polymorphic Invariance of Escape Analysis

Up to this point, we have assumed that **nml** is a monomorphically typed language. It remains to be shown that escape analysis works on polymorphically typed **nml** as well. We particular concentrate on parametric polymorphism. To show this, we prove that escape analysis exhibits the property of polymorphic invariance [1]. This means that given a polymorphic function, escape analysis will return the same result on any two monotyped instances of that function. Actually, escape analysis is *polymorphically invariant* when it is stated the following way (essentially the converse of the way it was stated previously): *Given a function application, how many spines of an argument are not returned in the result of the application.* This is important because it is the portions of an argument that do not escape from the application that can be stack allocated or reused (if unshared).

Theorem 1 (Polymorphic Invariance) *Let f be a polymorphic function of arity n, and let f' and f'' be any two monomorphic instances of f. Assume that env_e' and env_e'' are escape semantic environments that map f' and f'' to elements of D_e, respectively. Then, for $1 \leq i \leq n$,*
$G(f', i, env_e') = \langle 0, 0\rangle \Longleftrightarrow G(f'', i, env_e'') = \langle 0, 0\rangle$ or
$G(f', i, env_e') = \langle 1, k'\rangle \Longleftrightarrow G(f'', i, env_e'') = \langle 1, k''\rangle$
such that $s_i' - k' = s_i'' - k''$ where s_i' and s_i'' are the number of spines of the ith parameter of f' and f'', respectively.

Proof : This is shown by structural and fixpoint induction on the expression e (the body of f) [16]. □

Polymorphic invariance indicates that to analyze a function definition in a polymorphic language, we need only perform the analysis on the simplest monotyped instance of that function. The result is then applicable to all possible instances of that function.

6 Application of Escape Analysis

As mentioned in the introduction, escape information can be used in a number of other analyses and optimizations. Due to space limitations, we will only sketch them.

Several papers [8, 9] have been published on sharing analysis of objects in higher order functional languages (and many papers for first-order languages). It turns out that for strict languages (in which the evaluation order is obvious), sharing analysis of lists becomes easy in the presence of escape information.

Theorem 2 (Sharing Information) *Let f be a function which takes n arguments such that d_i is the number of spines of the ith parameter of f for $i = 1 \ldots n$, and let f return a list with d_f spines. If esc_i is the number of escaping spines of the ith parameter of f, for $i = 1 \ldots n$ (statically inferred by escape analysis), then*

1. *all cons cells in the top* $(d_f - max\{min\{esc_1, (d_1 - u_1)\}, \ldots, min\{esc_n, (d_n - u_n)\}\})$ *spines of the result of* $(f\ e_1 \ldots e_n)$ *are unshared when* u_i *is the number of unshared spines of* e_i *for* $1 \leq i \leq n$.

2. *all cons cells in the top* $(d_f - max\{esc_1, \ldots, esc_n\})$ *spines of the result of* $(f\ e_1 \ldots e_n)$ *are unshared for any set of arguments* $e_1, \ldots . e_n$.

Proof : 1. The number of spines of e_i that are shared is $(d_i - u_i)$. The number of shared spines of e_i that could escape f is $min\{esc_i, (d_i - u_i)\}$. In the result of $(f\ e_1 \ldots e_n)$, the bottom $max\{min\{esc_i, (d_i - u_i)\}\}$ spines will be shared. Thus, all cells in the top $(d_f - max\{min\{esc_1, (d_1 - u_1)\}, \ldots, min\{esc_n, (d_n - u_n)\}\})$ spines of the result are not shared.
2. Since we consider any set of arguments $e_1, \ldots . e_n$, and we have no sharing information of e_i, we assume that $u_i = 0$ as the worst-case. Then, $min\{esc_i, (d_i - 0)\} = esc_i$ because $esc_i \leq d_i$. Thus, all cells at top $(d_f - max\{esc_1, \ldots, esc_n\})$ spines of the result are not shared. □

Both escape information and sharing information that are determined by escape analysis can be used for the in-place reuse optimization. Consider an expression of the form of $(f\ e_1 \ldots\ e_i\ \ldots e_n)$ where f is a function with n parameters, the ith parameter of f is a list type with d_i spines, and there occurs some `cons` in the body of f. Let all cons cells at the top u_i spines of the result of e_i be unshared. Using the *global* escape information of parameters of f, the in-place reuse of cons cells can be performed as follows:

- If the bottom esc_i spines of the ith parameter of f escapes f globally then the expression can safely be transformed into $(f'\ e_1 \ldots\ e_i\ \ldots e_n)$ where f' is a new version of f which directly reuses cons cells in the top $s = min\{u_i, (d_i - esc_i)\}$, spines of the ith argument of f for new cons cells needed in the body of f.

- Let f be defined as $f\ x_1\ \ldots\ x_n\ =\ \ldots$ (`cons` $e_1\ e_2$) $\ldots$. If there is no further use of the ith parameter x_i of f after the evaluation of the subexpression (`cons` $e_1\ e_2$) then a new version f' of f which uses the in-place reuse optimization can be defined as follows:
  $f'\ x_1\ \ldots\ x_n\ =\ \ldots$ (`DCONS` $x_i\ e_1\ e_2$) $\ldots$ where `DCONS` is a destructive version of `cons` defined by `DCONS` $a\ b\ c\ = \{p := a; car.a := b; cdr.a := c; return(p)\}$

7 Conclusions

We have presented an analysis that answers a simple question, but in doing so, subsumes a number of other program analyses for storage optimization described in the literature. It works on higher order functional languages in the presence of lists, is relatively simple and (hopefully) easy to understand, and makes no assumptions about an underlying abstract machine. It remains to be seen if it is useful in practice due to the computational complexity of finding fixpoints of higher order functions. However, the fact that the analysis is polymorphically invariant allows one to analyze only the simplest instance of each polymorphic function, and we hope that in practice this makes the analysis useful at compile-time.

References

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

[2] H. Baker. Unifying and conquer (garbage, updating, aliasing ...) in functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 218-226, 1990.

[3] R.A. Brooks, R.P. Gabriel and G.L. Steele. An optimizing compiler for lexically scoped LISP. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp. 261-275, 1982.

[4] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis for higher order functions. *Science of Computer Programming*, 7:249-278, 1986.

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

[6] D.R. Chase. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-9, 1988.

[7] D.R. Chase, M. Wegman, F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 296-310, 1990.

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

[9] B. Goldberg. Detecting sharing of partial applications in functional programs. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, LNCS 274, Springer-Verlag, pp. 408-425, 1987.

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

[11] P. Hudak and J. Young. Higher-order strictness analysis for the untyped lambda calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 107-118, 1986.

[12] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages*, 10(4):555-578, October 1988.

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

[14] N. Jones and S. Muchnick. Binding time optimization in programming languages: An approach to the design of an ideal language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 77-94, 1976.

[15] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* Ph.D. Thesis, Yale University, May 1988.

[16] Y.G. Park. *Semantic Analyses for Storage Management Optimizations in Functional Language Implementations.* Ph.D. Thesis, New York University, 1991.

[17] Y.G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pp. 178-189, 1991.

[18] C. Ruggieri and T.P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 285-293, 1988.

A Examples

Consider the following partition sort (often mistakenly called quicksort) program:

```
letrec PS x = if (null x) then nil
              else letrec y = SPLIT (car x) (cdr x) nil nil;
                   in APPEND(PS (car y))
                             (cons (car x) (PS (car (cdr y))));

       SPLIT p x l h = if (null x) then (cons l (cons h nil))
                       elseif (car x)<p then
                              SPLIT p (cdr x) (cons (car x) l) h
                       else SPLIT p (cdr x) l (cons (car x) h);

       APPEND x y = if (null x) then y
                    else cons (car x) (APPEND(cdr x) y);
in PS [5,2,7,1,3,4]
```

We assume that the type of each function is given by `PS` : $int\ list \rightarrow int\ list$, `SPLIT` : $int \rightarrow int\ list \rightarrow int\ list \rightarrow int\ list \rightarrow int\ list\ list$, and `APPEND` : $int\ list \rightarrow int\ list \rightarrow int\ list$ and that each `car` in the program is annotated as $\texttt{car}^{\texttt{s}}$ which denotes the `car` takes as its argument a list with `s` spines.

A.1 Escape Analysis

The definitions of the escape semantic values *append*, *split*, and *ps* of `APPEND`, `SPLIT`, and `PS` (shown uncurried for convenience) are:

$$\begin{aligned}
&append\ x\ y\ =\ y \sqcup (\mathrm{sub}^1(x) \sqcup append\ x\ y)\\
&split\ p\ x\ l\ h\ =\ l \sqcup h \sqcup (split\ p\ x\ (\mathrm{sub}^1(x) \sqcup l)\ h) \sqcup (split\ p\ x\ \ (\mathrm{sub}^1(x) \sqcup h))\\
&ps\ x\ =\ append\ (ps\ \mathrm{sub}^2(split\ \mathrm{sub}^1(x)\ x\ \langle\langle 0,0\rangle err\rangle\ \langle\langle 0,0\rangle err\rangle))\\
&\qquad\qquad (\mathrm{sub}^1(x) \sqcup (ps\ \mathrm{sub}^2(split\ \mathrm{sub}^1(x)\ x\ \langle\langle 0,0\rangle err\rangle\ \langle\langle 0,0\rangle err\rangle)))
\end{aligned}$$

The meaning of each function in the escape semantic domain is found by fixpoint iteration. Here is the fixpoint iteration for `APPEND`:

$$\begin{aligned}
append^{(0)}\ x\ y &= \bot_{int\ list}\\
append^{(1)}\ x\ y &= y \sqcup (\mathrm{sub}^1(x) \sqcup append^{(0)}\ x\ y)\\
&= y \sqcup \mathrm{sub}^1(x)\\
append^{(2)}\ x\ y &= y \sqcup (\mathrm{sub}^1(x) \sqcup append^{(1)}\ x\ y)\\
&= y \sqcup (\mathrm{sub}^1(x) \sqcup (y \sqcup \mathrm{sub}^1(x)))\\
&= y \sqcup \mathrm{sub}^1(x)
\end{aligned}$$

and for `SPLIT`:

$$\begin{aligned}
split^{(0)}\ p\ x\ l\ h &= \bot_{(int\ list)\ list}\\
split^{(1)}\ p\ x\ l\ h &= l \sqcup h \sqcup \bot_{(int\ list)\ list} \sqcup \bot_{(int\ list)\ list}\\
&= l \sqcup h\\
split^{(2)}\ p\ x\ l\ h &= l \sqcup h \sqcup (split^{(1)}\ p\ x\ (\mathrm{sub}^1(x) \sqcup l)\ h) \sqcup (split^{(1)}\ p\ x\ \ (\mathrm{sub}^1(x) \sqcup h))\\
&= l \sqcup h \sqcup ((\mathrm{sub}^1(x) \sqcup l) \sqcup h) \sqcup ((\mathrm{sub}^1(x) \sqcup h))\\
&= l \sqcup h \sqcup \mathrm{sub}^1(x)\\
split^{(3)}\ p\ x\ l\ h &= l \sqcup h \sqcup (split^{(2)}\ p\ x\ (\mathrm{sub}^1(x) \sqcup l)\ h) \sqcup (split^{(2)}\ p\ x\ \ (\mathrm{sub}^1(x) \sqcup h))\\
&= l \sqcup h \sqcup ((\mathrm{sub}^1(x) \sqcup l) \sqcup h) \sqcup (l \sqcup (\mathrm{sub}^1(x) \sqcup h) \sqcup \mathrm{sub}^1(x))\\
&= l \sqcup h \sqcup \mathrm{sub}^1(x)
\end{aligned}$$

and for PS (using the values computed for APPEND and SPLIT):

$$
\begin{array}{rcl}
ps^{(0)}\ x & = & \bot_{int\ list} \\
ps^{(1)}\ x & = & append(ps^{(0)}\ \mathrm{sub}^2(\mathrm{sub}^1(x)))\ (\mathrm{sub}^1(x) \sqcup (ps^{(0)}\ \mathrm{sub}^2(\mathrm{sub}^1(x))) \\
 & = & append\ \bot_{int\ list}\ (\mathrm{sub}^1(\mathrm{sub}^1(x)) \sqcup \bot_{int\ list}) \\
 & = & \mathrm{sub}^1(x) \\
ps^{(2)}\ x & = & append(ps^{(1)}\ \mathrm{sub}^2(\mathrm{sub}^1(x)))\ (\mathrm{sub}^1(x) \sqcup (ps^{(1)}\ \mathrm{sub}^2(\mathrm{sub}^1(x))) \\
 & = & append\ \mathrm{sub}^1(\mathrm{sub}^2(\mathrm{sub}^1(x)))\ (\mathrm{sub}^1(x) \sqcup \mathrm{sub}^1(\mathrm{sub}^2(\mathrm{sub}^1(x))) \\
 & = & \mathrm{sub}^1(x)
\end{array}
$$

Let $env_e = [\mathtt{APPEND} \mapsto append, \mathtt{SPLIT} \mapsto split, \mathtt{PS} \mapsto ps]$. Then,

$$
\begin{array}{rcl}
G(\mathtt{APPEND}, 1, env_e) & = & (E_e[\![\mathtt{APPEND\ x\ y}]\!] env_e[\mathtt{x} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{y} \mapsto \langle\langle 0,0\rangle, err\rangle])_{(1)} \\
 & = & (\langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,0\rangle \\
G(\mathtt{APPEND}, 2, env_e) & = & (E_e[\![\mathtt{APPEND\ x\ y}]\!] env_e[\mathtt{x} \mapsto \langle\langle 0,0\rangle, err\rangle, \mathtt{y} \mapsto \langle\langle 1,1\rangle, err\rangle].)_{(1)} \\
 & = & (\langle\langle 1,1\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,1\rangle
\end{array}
$$

Thus, we conclude that APPEND returns all of its second argument y, and all but the top spine of the first argument x.

$$
\begin{array}{rcl}
G(\mathtt{SPLIT}, 1, env_e) & = & (E_e[\![\mathtt{SPLIT\ p\ x\ l\ h}]\!] env_e[\mathtt{p} \mapsto \langle\langle 1,0\rangle, err\rangle, \mathtt{x}, \mathtt{l}, \mathtt{h} \mapsto \langle\langle 0,0\rangle, err\rangle])_{(1)} \\
 & = & (\langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
 & = & \langle 0,0\rangle \\
G(\mathtt{SPLIT}, 2, env_e) & = & (E_e[\![\mathtt{SPLIT\ p\ x\ l\ h}]\!] env_e[\mathtt{x} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{p}, \mathtt{l}, \mathtt{h} \mapsto \langle\langle 0,0\rangle, err\rangle].)_{(1)} \\
 & = & (\langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,0\rangle \\
G(\mathtt{SPLIT}, 3, env_e) & = & (E_e[\![\mathtt{SPLIT\ p\ x\ l\ h}]\!] env_e[\mathtt{l} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{p}, \mathtt{x}, \mathtt{h} \mapsto \langle\langle 0,0\rangle, err\rangle].)_{(1)} \\
 & = & (\langle\langle 1,1\rangle, err\rangle \sqcup \langle\langle 0,0\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,1\rangle \\
G(\mathtt{SPLIT}, 4, env_e) & = & (E_e[\![\mathtt{SPLIT\ p\ x\ l\ h}]\!] env_e[\mathtt{h} \mapsto \langle\langle 1,1\rangle, err\rangle, \mathtt{p}, \mathtt{x}, \mathtt{l} \mapsto \langle\langle 0,0\rangle, err\rangle].)_{(1)} \\
 & = & (\langle\langle 0,0\rangle, err\rangle \sqcup \langle\langle 1,1\rangle, err\rangle \sqcup \mathrm{sub}^1(\langle\langle 0,0\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,1\rangle
\end{array}
$$

¿From above, we conclude that SPLIT returns all of its third and fourth arguments l and h, none of the first argument p, and all but the top spine of the second argument x.

$$
\begin{array}{rcl}
G(\mathtt{PS}, 1, env_e) & = & (E_e[\![\mathtt{PS\ x}]\!]\ env_e[\mathtt{x} \mapsto \langle\langle 1,1\rangle, err\rangle])_{(1)} \\
 & = & (\mathrm{sub}^1(\langle\langle 1,1\rangle, err\rangle))_{(1)} \\
 & = & \langle 1,0\rangle
\end{array}
$$

So, we conclude that PS returns all but the top spine of its argument x.

A.2 Sharing Information from Escape Analysis

PS takes a list with one spine as its argument, and returns a list with one spine. ¿From the global escape analysis, we know that no spine of the argument escapes PS globally. SPLIT takes four arguments p, x, l and h where p is an integer, and x, l and h are lists with one spine, respectively, and returns a list with two spines. ¿From the global escape analysis, we know that none of the first parameter p, all but the top spine of the second parameter x, and all of the third and fourth parameters l and h escape SPLIT globally. Thus, we can determine the following sharing properties (among others) of the program:

- For the expression (PS e) where e is any list with one spine, the top spine of the result list of (PS e) is *not* shared.
- For the expression of (SPLIT e_1 e_2 e_3 e_4) where each e_i is any possible expression, the top spine of the result list of (SPLIT e_1 e_2 e_3 e_4) is *not* shared.

A.3 Optimizations based on Escape Analysis

A.3.1 Stack Allocation

Our escape analysis has determined that the spine of the original list [5,2,7,1,3,4] does not escape from PS. Thus the spine of that list can be allocated in PS activation record. All the cells of the spine will disappear when PS's activation is removed from the stack.

A.3.2 In-place Reuse

¿From the global escape analysis, we know that APPEND returns all of its second argument y, and all but the top spine of the first argument x. We also know that, for any expression (PS e) where e is a list with one spine, the top spine of the result of (PS e) is *unshared*. Thus, the definition of PS can be transformed into PS' as follows:

```
PS' x = if (null x) then nil
        else letrec y = SPLIT (car x) (cdr x) nil nil;
             in APPEND' (PS' (car y))
                (cons (car x) (PS' (car (cdr y))));
```

where APPEND' is a version of APPEND in which cons cells in the top spine of its first argument x are directly reused. It is defined by

```
APPEND' x y = if (null x) then y
             else DCONS x (car x) (APPEND' (cdr x) y);
```

Furthermore, if we know that the top spine of the argument of PS is unshared, then the definition of PS can be transformed into PS'' in which cons cells in the top spine of its argument x are reused as follows:

```
PS'' x = if (null x) then nil
         else letrec y = SPLIT (car x) (cdr x) nil nil;
            in APPEND' (PS'' (car y))
               (DCONS x (car x) (PS'' (car (cdr y))));
```

Consider, as another example, a naive reverse function REV:

```
REV l = if (null l) then nil
        else APPEND (REV (cdr l)) (cons (car l) nil);
```

¿From our analysis, REV can be transformed into REV' which reuses cons cells in the top spine of its argument l, if unshared, as follows:

```
REV' l = if (null l) then nil
         else APPEND' (REV' (cdr l)) (DCONS l (car l) nil);
```

A.3.3 Block Allocation/Reclamation

Suppose we are given a program identical to the partition sort program, except that the result expression was

```
PS (create_list i)
```

where create_list is some recursive function creating a list and i is some variable. The list that create_list returns cannot be allocated in PS's activation record because the activation record doesn't exist when the list is created. An optimization that can be performed is as follows: create_list should allocate the spine of the list in some block of memory. The spine of the list does not escape from PS, so when PS is finished, the whole block of memory can be put back on the free list. This block of memory is the "local heap" described by Ruggieri and Murtagh [18].