Optimizing Closures in O(0) time

Andrew W. Keep
Cisco Systems, Inc.
Indiana Univeristy
akeep@cisco.com

Alex Hearn
Indiana University
adhearn@cs.indiana.edu

R. Kent Dybvig
Cisco Systems, Inc.
Indiana University
dyb@cisco.com

Abstract

The flat-closure model for the representation of first-class procedures is simple, safe-for-space, and efficient, allowing the values or locations of free variables to be accessed with a single memory indirect. It is a straightforward model for programmers to understand, allowing programmers to predict the worst-case behavior of their programs. This paper presents a set of optimizations that improve upon the flat-closure model along with an algorithm that implements them, and it shows that the optimizations together eliminate over 50% of run-time closure-creation and free-variable access overhead in practice, with insignificant compile-time overhead. The optimizations never add overhead and remain safe-for-space, thus preserving the benefits of the flat-closure model.

1. Introduction

First-class procedures, i.e., indefinite extent procedural objects that retain the values of lexically scoped variables, were incorporated into the design of the Scheme programming language in 1975 and within a few years started appearing in functional languages such as ML. It has taken many years, but they are fast becoming commonplace, with their inclusion in contemporary languages such as JavaScript and newer versions of other languages such as C# and Perl.

First-class procedures are typically represented at run time as *closures*. A closure is a first-class object encapsulating some representation of a procedure's code (e.g., the starting address of its machine code) along with some representation of the lexical environment. In 1983, Cardelli [**?**] introduced the notion of *flat closures*. A flat closure resembles a vector, with a code slot plus one slot for each free variable[1]. The code slot holds a code pointer, which might be the address of a block of machine code implementing the procedure, or it might be some other representation of code, such as byte code in a virtual machine. The free-variable slots each hold the value of one free variable. Because the same variable's value might be stored simultaneously in one or more closures and also in the original location in a register or stack, mutable variables are not directly supported by the flat-closure model. In 1987, Dybvig [**?**] addressed this for languages, like Scheme, with mutable variables by adding a separate assignment conversion step that converts the locations of assigned variables (but not unassigned variables) into explicit heap-allocated boxes, thereby avoiding problems with duplication of values.

Flat closures have the useful property that each free variable (or location, for assigned variables) is accessible with a single indirect. This compares favorably with any mechanism that requires traversal of a nested environment structure. The cost of creating a flat closure is proportional to the number of free variables, which is often small. When not, the cost is more than compensated for by the lower cost of free-variable reference, in the likely case that each free variable is accessed at least once and possibly many times. Flat closures also hold onto no more of the environment than the procedure might require and so are "safe for space" [**?**]. This is important because it allows the storage manager to reclaim storage from the values of variables that are visible in the environment but not used by the procedure.

This paper describes a set of optimizations of the flat-closure model that reduce closure-creation costs and eliminate memory operations without losing the useful features of flat closures. It also sketches an algorithm that performs the optimizations. These optimizations never do any harm, i.e., they never add allocation overhead or memory operations relative to a naive implementation of flat closures. Thus, a programmer can count on at least the performance of the straight flat-closure model, and most likely better. The algorithm adds a small amount of compile-time overhead during closure conversion, but since it produces less code, the overhead is more than made up for by the reduced overhead in later passes of the compiler, hence the facetious title of this paper.

A key contribution of this work is the detailed description of the optimizations and their relationships. While a few of the optimizations have long been performed by our compiler, descriptions of them have never been published. Various closure optimizations have been described by others [**? ? ? ? ? ? ? ? ?**], but most of the optimizations described here have not been described previously in the literature, and many are likely novel.

The remainder of this paper is organized as follows. Section **??** describes the optimizations, and Section **??** sketches an algorithm that implements them. Section **??** provides a brief, preliminary empirical analysis of the optimizations. Section **??** describes related work, and Section **??** presents our conclusions.

2. The Optimizations

The closure optimizations described in this section collectively act to eliminate some closures and reduce the sizes of others. When closures are eliminated in one section of the program, the

[1] In this context, free variables are those referenced within the body of a procedure but not bound within the procedure.

optimizations can cascade to further optimizations that allow other closures to be eliminated or reduced in size. They also sometimes result in the selection of alternate representations that occupy fewer memory locations. In most cases, they also reduce the number of indirects required to access free variables. The remainder of this section presents each optimization in turn, grouped by direct effect:

- avoiding unnecessary closures (Section **??**),
- eliminating unnecessary free variables (Section **??**), and
- sharing closures (Section **??**).

A single algorithm that implements all of the optimizations described in this section is provided in Section **??**.

2.1 Avoiding unnecessary closures

A flat closure contains a code pointer and a set of free-variable values. Depending on the number of free variables and whether the code pointer is used, we can sometimes eliminate the closure, sometimes allocate it statically, and sometimes represent it more efficiently. We consider first the case of well-known procedures.

Case 1: Well-known procedures

A procedure is *known* at a call site if the call site provably invokes that procedure's λ-expression and only that λ-expression. A *well-known* procedure is one whose value is never used except at call sites where it is known. The code pointer of a closure for a well-known procedure need never be used because, at each point where the procedure is called, the call can jump directly to the entry point for the procedure via a direct-call label associated with the λ-expression.

Depending on the number of free variables, we can take advantage of this as follows.

Case 1a: Well-known with no free variables

If the procedure has no free variables, and its code pointer is never used, the closure itself is entirely useless and can be eliminated.

Case 1b: Well-known with one free variable x

If the procedure has one free variable, and its code pointer is never used, the only useful part of the closure is the free variable. In this case, the closure can be replaced with the free variable everywhere that it is used.

Case 1c: Well-known with two free variables x and y

If the procedure has two free variables, and its code pointer is never used, it contains only two useful pieces of information, the values of the two free variables. In this case, the closure can be replaced with a pair. In our implementation, pairs occupy just two words of memory, while a closure with two free variables occupies three words.

Case 1d: Well-known with three or more free variables x ...

If the procedure has three or more free variables, but its code pointer is never used, we can choose to represent it as a closure or as a vector. The size in both cases is the same: one word for each free variable plus one additional word. The additional word for the closure is a code pointer, while the additional word for the vector is an integer length. This choice is a virtual toss-up, although storing a small constant length is slightly cheaper than storing a full-word code pointer, especially on 64-bit machines. We choose the vector representation for this reason and because it helps us share closures, as described in Section **??**.

We now turn to the case where the procedure is not well known.

Case 2: Not-well-known procedures

In this case, the procedure's value might be used at a call site where the procedure is not known. That call site must jump indirectly through the closure's code pointer, as it does not know the direct-call label or labels of the closures that it might call. In this case, the code pointer is needed, and a closure must be allocated.

We consider two subcases:

Case 2a: Not well-known with no free variables

In this case, the closure is the same each time the procedure's λ-expression is evaluated, as it contains only a static code pointer. The closure can thus be allocated statically and treated as a constant.

Case 2b: Not well-known with one or more free variables x ...

In this case, a closure must be created at run time.

2.2 Eliminating unnecessary free variables

On the surface, it seems that a closure needs to hold the values of all of its free variables. After all, if a variable occurs free in a procedure's λ-expression, it might be referenced, barring dead code that should have been eliminated by some earlier pass of the compiler. Several cases do arise, however, in which a free variable is not needed.

Case 1: Unreferenced free variables

Under normal circumstances, a variable cannot be free in a λ-expression if it is not referenced there (or assigned, prior to assignment conversion). This case can arise after free-variable analysis has been performed, however, by the elimination of a closure under Case 1a of Section **??**. Call sites that originally passed the closure to the procedure do not do so when the closure is eliminated, and because no other references to a well-known procedure's name appear in the code, the variable should be removed from any closures in which it appears.

Case 2: Global variables

The locations of global variables, i.e., variables whose locations are fixed for an entire program run, need not be included in a closure, as the address of the location can be incorporated directly in the code stream, with appropriate support from the linker.

Case 3: Variables bound to constants

If a variable is bound to a constant, references to it can be replaced with the constant (via constant propagation), and the binding can be eliminated, e.g.:

```
(let ([x 3])
  (letrec ([f (lambda () x)])
    —))
```

can be rewritten as:

```
(letrec ([f (lambda () 3)])
  —)
```

If this transformation is performed in concert with the other optimizations described in this section, a variable bound to a constant can be removed from the sets of free variables in which it appears.

Our compiler performs this sort of transformation prior to closure optimization, but this situation can also arise when a closure is allocated statically and treated as a constant by Case 2a of Section **??**. For structured data, such as closures, care should also be taken to avoid replicating the actual structure when the variable is referenced at multiple points within its scope. Downstream passes of

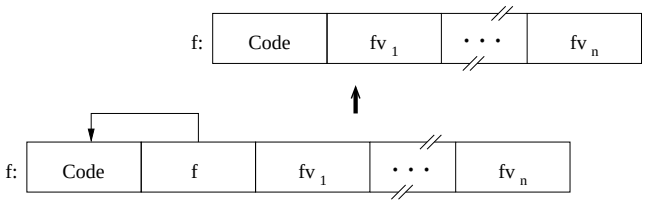

Figure 1. Function *f* with a self-reference in its closure

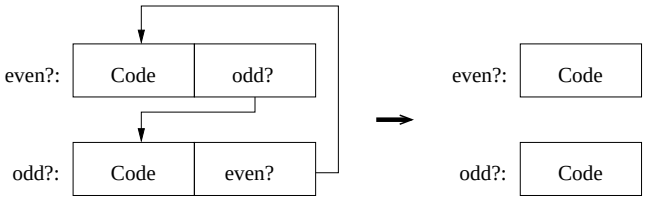

Figure 2. Mutual references for `even?` and `odd?`

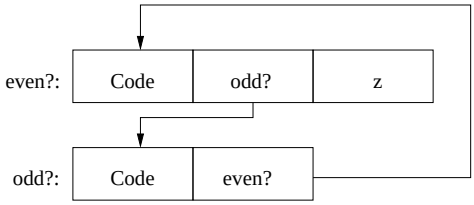

Figure 3. Mutual references for the `even?` and `odd?` closures, with `z` free in `even?`.

our compiler guarantee that this is the case, in cooperation with the linker, effectively turning the closure into a constant.

Case 4: Aliases

A similar transformation can take place when a variable *x* is bound directly to the value of another variable *y*, e.g.:

```
(let ([x y])
  (letrec ([f (lambda () x)])
    —))
```

can be rewritten (via copy propagation) as:

```
(letrec ([f (lambda () y)])
  —)
```

This transformation would not necessarily be valid if either *x* or *y* were assigned, but we assume that assignment conversion has already been performed.

In cases where both *x* and *y* are free within the same λ-expression, we can remove *x* and leave just *y*. For example, *x* and *y* both appear free in the λ-expression bound to *f*:

```
(let ([x y])
  (letrec ([f (lambda () (x y))])
    —))
```

Yet, if references to *x* are replaced with references to *y*, only *y* should be retained in the set of free variables.

Again, our compiler eliminates aliases such as this in a pass that runs before closure optimization. Nevertheless, this situation can arise as a result of Case 1b of Section **??**, in which a closure for a well-known procedure with one free variable is replaced by its single free variable. It can also arise as the result of closure sharing, as discussed in Section **??**

Case 5: Self-references

A procedure that recurs directly to itself through the name of the procedure has its own name as a free variable. For example, the λ-expression in the code for *f* below has *f* as a free variable:

```
(define append
  (lambda (ls1 ls2)
    (letrec ([f (lambda (ls1)
                  (if (null? ls1)
                      ls2
                      (cons (car ls1)
                            (f (cdr ls1) ls2))))])
      (f ls1))))
```

From the illustration of the closure in Figure **??**, it is clear that this self-reference is unnecessary. If we already have *f*'s closure in hand, there is no need to follow the indirect to find it. In general, a link at a known offset from the front of any data structure that always points back to itself is unnecessary and can be eliminated. Thus, a procedure's name need not appear in its own list of free variables.

Case 6: Unnecessary mutual references

A similar situation arises when two or more procedures are mutually recursive and have only the variables of one or more of the others as free variables. For example, in:

```
(letrec ([even? (lambda (x)
                  (or (= x 0)
                      (odd? (- x 1))))]
         [odd? (lambda (x) (not (even? x)))])
  —)
```

`even?` has `odd?` as a free variable only to provide `odd?` its closure and vice versa. Neither is necessary. This situation is illustrated in Figure **??**.

In contrast, in the modified version below:

```
(lambda (z)
  (letrec ([even? (lambda (x)
                    (or (= x z)
                        (odd? (- x 1))))]
           [odd? (lambda (x) (not (even? x)))])
    —))
```

`z` is free in `even?`, so `even?` does need its closure to hold `z`, and `odd?` needs its closure to hold `even?`. This situation is illustrated in Figure **??**.

2.3 Sharing closures

If a set of closures cannot be eliminated, they possibly can be shared. For example, in the second `even?` and `odd?` example of Section **??**, we could use a single closure for both `even?` and `odd?`. The combined closure would have just one free variable, `z`, as the pointer from `odd?` to `even?` would become a self-reference and, thus, be unnecessary. Further, when `even?` calls `odd?`, it would just pass along the shared closure rather than indirecting its own to obtain `odd?`'s closure. The same savings would occur when `odd?` calls `even?`.

There are three challenges, however. First, our representation of closures does not have space for multiple code pointers. This can be addressed with support from the storage manager, although not without some difficulty.

Second, more subtly, if two procedures have different lifetimes, some of the free-variable values might be retained longer than they should be. In other words, the representation is no longer "safe for space" [**?**]. This problem does not arise if either (a) the procedures have the same lifetime, or (b) the set of free variables (after removing mutually recursive references) is the same for all of the procedures.

Third, even more subtly, if two procedures have different lifetimes, but the same set of free variables, and one or more are not-well-known, one of the code pointers might be retained longer than necessary. In systems where all code is static, this is not a problem, but our compiler generates code on the fly, e.g., when the `eval` procedure is used; and anything that can be dynamically allocated must be subject to garbage collection, including code. This is not a problem when each of the procedures is well-known, assuming that we choose the vector representation over the closure representation in Case 1d of Section **??**.

Thus, we can share closures in the following two cases:

Case 1: Same lifetime, single code pointer

Without extending our existing representation to handle multiple code pointers, we can use one closure for any set of procedures that have the same lifetime, as long as, at most, one of them requires its code pointer. Proving that two or more procedures have the same lifetime is difficult in general, but it is always the case for sets of procedures where a call from one can lead directly or indirectly to a call to each of the others, i.e., sets that are strongly connected [**?**] in a graph of bindings linked by free-variable relationships.

Case 2: Same free variables, no code pointers

If a set of well-known procedures all have the same set of free variables, the procedures can share the same closure, even when they are not part of the same strongly connected group of procedures. No harm is done if one outlasts the others, as the shared closure directly retains no more than what each of the original closures would have indirectly retained. In determining this, we can ignore variables that name members of the set, as these will be eliminated as self-references in the shared closure.

In either case, sharing can result in aliases that can lead to reductions in the sizes of other closures (Case 4 of Section **??**).

2.4 Example

Consider the `letrec` expression in the following program:

```
(lambda (x)
  (letrec ([f (lambda (a) (a x))]
           [g (lambda () (f (h x)))]
           [h (lambda (z) (g))]
           [q (lambda (y) (+ (length y) 1))])
    (q (g))))
```

As the first step in the optimization process, we identify the free variables for the procedures defined in the `letrec`: x is free in f; x, f, and h are free in g; and g is free in h. q contains no free variables. We do not consider `+` or `length` to be free in q, as the locations of global variables are stored directly in the code stream, as discussed in Case 2 of Section **??**. Additionally, we note that f, g, h, and q are all well-known.

Next, we partition the bindings into strongly connected components, producing one `letrec` expression for each [**? ?**]. g and h are mutually recursive, and, thus, must be bound by the same `letrec` expression, while f and q each get their own. Since f appears in g, the `letrec` that binds f must appear outside the `letrec` that binds g and h. Since q neither depends on nor appears in the other procedures, we can place its `letrec` expression anywhere among the others. We arbitrarily choose to make it the outermost `letrec`.

After these partitions we have the following program:

```
(lambda (x)
  (letrec ([q (lambda (y) (+ (length y) 1))])
    (letrec ([f (lambda (a) (a x))])
      (letrec ([g (lambda () (f (h x)))]
               [h (lambda (z) (g))])
        (q (g))))))
```

We can now begin the process of applying optimizations. Since q is both well-known and has no free variables, its closure can be completely eliminated (Case 1a of Section **??**). f is a well-known procedure and has only one free variable, x, so its closure is just x (Case 1b of Section **??**). g and h are mutually recursive, so it is tempting to eliminate both closures, as described by Case 6 of Section **??**. However, g still has x as a free variable, and, therefore, needs its closure. h also needs its closure so that it can hold g. Because g and h are well-known and are part of the same strongly connected component, they can share a closure (Case 1 of Section **??**). Additionally, since f's closure has been replaced by x, there is only a single free variable, x, so the closures for g and h are also just x (Case 1b of Section **??**). If another variable, y, were free in one of g or h, the result would be a shared closure represented by a pair of x and y (Case 1c of Section **??**). If, further, g were not-well-known, a shared closure for g and h would have to be allocated with the code pointer for g and x and y as its free variables (Case 1 of Section **??**).

3. The Algorithm

We perform all of the closure optimizations described in Section **??** using a single algorithm, sketched below:

1. Gather information about the input program, including the free variables of each λ-expression and whether each λ-expression is well-known.
2. Partition the bindings of each input `letrec` expression into separate sets of bindings known to have the same lifetimes, i.e., sets of strongly connected bindings.
3. When one or more bindings of a strongly connected set of bindings is well-known (i.e., they are bindings for well-known procedures), decide which should share a single closure.
4. Determine the required free variables for each closure, leaving out those that are unnecessary.
5. Select the appropriate representation for each closure and whether it can share space with a closure from some outer strongly connected set of bindings.
6. Rebuild the code based on these selections.

The algorithm handles only *pure* `letrec` expressions, i.e., those whose left-hand sides are unassigned variables and whose right-hand sides are λ-expressions. Thus, we require that some form of `letrec` purification [**? ?**] has already been performed.

4. Results

Our implementation extends the algorithm described in Section **??** to support the ful R6RS Scheme Language [**?**]. To determine the

effectiveness of closure optimization, we ran the optimization over a starndard set of 67 R6RS benchmarks [**?**].

Overall the optimziation performs well, on average statically eliminating 56.94% of closures and 44.89% of the total free variables and dynamically eliminating, on average, 58.25% of the allocation and 58.58% of the memory references attributable to closure access. We hope to provide a full break down of these numbers, along with a break down of the effects of the individual parts of this in a future version of this paper.

5. Related Work

Our replacement of a well-known closure with a single free variable is a degenerate form of lambda lifting [**?**], in which each of the free variables of a procedure are converted into separate arguments. Increasing the number of arguments can lead to additional stack traffic, particularly for non-tail-recursive routines, and it can increase register pressure whenever two or more variables are live in place of the original single package (closure) with two or more slots. Limiting our algorithm to doing this replacement only in the single-variable case never does any harm, as we are replacing a single package of values with just one value.

Serrano:cfa describes a closure optimization based on control-flow analysis [**?**]. His optimization eliminates the code part of a closure when the closure is well-known; in this, our optimizations overlap, although our benefit is less, as the code part of a closure in his implementation occupies four words, while ours occupies just one. He also performs lambda lifting when the closure is well-known and its binding is in scope wherever it is called.

steckler:lightweight describe a closure-conversion algorithm that creates "light-weight closures" that do not contain free variables that are available at the call site. This is a limited form of lambda lifting and, as with full lambda lifting, can sometimes do harm relative to the straight flat-closure model.

kranz:orbit describes various mechanisms for reducing closure allocation and access costs, including allocating closures on the stack and allocating closures in registers. The former is useful for closures created to represent continuations in an implementation that uses a continuation-passing style [**?**] and achieves part of the benefit of the natural reuse of stack frames in a direct-style implementation. The latter is useful for procedures that act as loops and reduces the need to handle loops explicitly in the compiler. Our optimizations are orthogonal to these optimizations, but they do overlap somewhat in their benefits.

Shao:2000 describe a nested representation of closures that can reduce the amount of storage required for a set of closures that share some but not all free variables, while maintaining space safety. The sharing never results in more than one level of indirection to obtain the value of a free variable. Because a substantial portion of the savings reported resulted from global variables [**?**], which we omit entirely, and we operate under the assumption that free-variable references are typically far more common than closure creation, we have chosen to stick with the flat-closure model and focus instead on optimizing that model.

Fradet:1991:CFL:114005.102805 describe various optimizations for implementations of lazy languages. They discuss reducing the size of a closure by omitting portions of the environment not needed by a procedure, which is an inherent feature of the flat-closure model preserved by our mechanism. They also discuss avoiding the creation of multiple closures when expressions are deferred by the lazy-evaluation mechanism in cases where a closure's environment, or portions of it, can be reused when the evaluation of one expression provably precedes another, i.e., when the lifetime of one closure ends before the lifetime of another begins.

LAMP-CONF-2008-004 describes a set of optimizations aimed at reducing the overhead of higher-order functions in Scala. A closure elimination optimization is included that attempts to determine when free variables are available at the call site or on the stack to avoid creating a larger class structure around the function. The optimization also looks for heap-allocated free variables that are reachable from local variables or the stack to avoid adding them to the closure. The optimization helps eliminate the closures for well-known calls by lambda lifting, if possible.

appelCompilingWithContinuationsCh10 describes eliminating self-references and allowing mutually recursive functions (strongly connected sets of `letrec` bindings) to share a single closure with multiple code pointers. These optimizations are similar to our elimination of self-references and sharing of well-known closures, although in our optimization we allow only one not-well-known closure in a shared closure.

A few of the optimizations described in this paper have been performed by Chez Scheme since 1992: elimination of self-references, elimination of mutual references where legitimate, and allocation of constant closures (though without the propagation of those constants). Additionally, we have seen references, in various newsgroups and blogs, to the existence of similar optimizations. While other systems may implement some of the optimizations that we describe, there is no mention of them, or an algorithm to implement them, in the literature.

6. Conclusion

The flat-closure model is a simple and efficient representation for procedures that allows the values or locations of free variables to be accessed with a single memory reference. This paper presented a set of flat-closure compiler optimizations and an algorithm for implementing them. Together, the optimizations result in an average reduction in run-time closure-creation and free-variable access overhead on a set of standard benchmarks by over 50%, with insignificant compile-time overhead. The optimizations never add overhead, so a programmer can safely assume that a program will perform at least as well with the optimizations as with a naive implementation of flat closures.