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 [5] 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 [8] 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" [16]. 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 presents, in detail, an algorithm that performs the optimizations and shows that the optimizations reduce run-time closure-creation and free-variable access overhead on a set of standard benchmarks by over 50%. 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 been performed by our compiler since 1992, descriptions of them have never been published. Various closure optimizations have been described by others [3, 7, 9, 12, 13, 15, 16, 19], but most of the optimizations described here have not been described previously in the literature, and many are likely novel. A second key contribution is the algorithm to implement them, which is novel.

The remainder of this paper is organized as follows. Section 2 describes the optimizations, and Section 3 describes an algorithm that implements them. Section 4 presents an empirical analysis demonstrating the effectiveness of the optimizations. Section 5 describes related work, and Section 6 presents our conclusions.

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

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 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 describes each optimization in turn, grouped by direct effect:

- avoiding unnecessary closures (Section 2.1),
- eliminating unnecessary free variables (Section 2.2), and
- sharing closures (Section 2.3).

A single algorithm that implements all of the optimizations described in this section is given in Section 3.

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 actually 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 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 also because it helps us share closures, as described in Section 2.3.

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, since 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, since 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 actually be created at run time.

2.2 Eliminating unnecessary free variables

On the surface, it seems like 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 2.1. Call sites that originally passed the closure to the procedure do not do so when the closure is eliminated, and since 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, since 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.

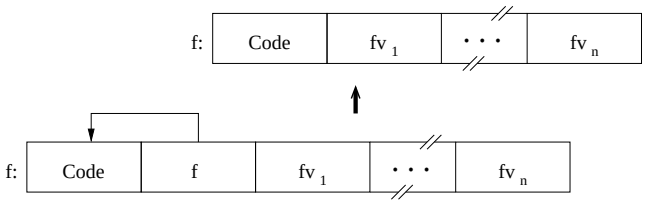

Figure 1. Function *f* with a self-reference in its closure

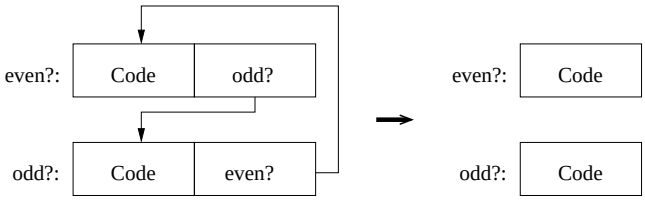

Figure 2. Mutual references for `even?` and `odd?`

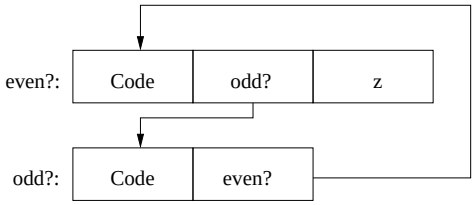

Figure 3. Mutual references for `even?` and `odd?`

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 2.1. 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 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 are assuming 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* will be replaced with references to *y*, only *y* should be retained in the set of free variables.

Again, our compiler eliminates aliases like this in a pass that runs before closure optimization, but this situation can arise as a result of Case 1b of Section 2.1, 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 2.3

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 1, 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 actually necessary. This situation is illustrated in Figure 2.

On the other hand, 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?` really does need its closure to hold `z`, and `odd?` needs its closure to hold `even?`. This situation is illustrated in Figure 3.

2.3 Sharing closures

If a set of closures cannot be eliminated, perhaps they can be shared. For example, in the second `even?` and `odd?` example of Section 2.2, perhaps we can use a single closure for both `even?` and `odd?`. The combined closure would have just one free variable, `z`, since the pointer from `odd?` to `even?` would become a self reference and thus be unnecessary. Furthermore, 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" [16]. 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 we choose the vector representation over the closure representation in Case 1d of Section 2.1.

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 [20] 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, since 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 naming members of the set, since 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 (Section 2.2 Case 4).

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, since the locations of global variables are stored directly in the code stream, as discussed in Case 2 of Section 2.2. 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 [10, 22]. g and h are mutually recursive, and so 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 2.1). f is a well-known procedure and has only one free variable, x, so its closure is just x (Case 1b of Section 2.1). g and h are mutually recursive, so it is tempting to eliminate both closures as described by Case 6 of Section 2.2. 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 2.3). 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 2.1). 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 2.1). 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 2.3).

3. The Algorithm

We now turn to a description of an algorithm that can be used to perform the optimizations described above. To simplify the presentation, we describe the algorithm using the small core language defined in Figure 4. The grammar enforces a few preconditions on the input:

- variables are not assigned,
- letrec expressions are *pure*, i.e., bind (unassigned) variables to λ-expressions, and
- λ-expressions appear nowhere else,

The first precondition can be arranged via standard assignment conversion [8, 14], while the second requires some form of letrec purification [10, 22]. The third precondition can be arranged trivially via the following local transformation on λ-expressions.

$$\texttt{(lambda (}x\texttt{) }e\texttt{)} \rightarrow \texttt{(letrec ([}f'\texttt{ (lambda (}x\texttt{) }e\texttt{)]) }f'\texttt{)}$$

where f' is a fresh variable.

The algorithm requires one more precondition not enforced by the grammar:

- variables are uniquely named.

This precondition can be arranged via a simple alpha renaming.

In broad strokes, the closure optimization algorithm is as follows, with details provided in the referenced sections:

1. Gather information about the input program, including the free variables of each λ-expression and whether each λ-expression is well-known (Section 3.1).
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 (Section 3.2).
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 (Section 3.3).
4. Determine the required free variables for each closure, leaving out those that are unnecessary (Section 3.4).
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 (Section 3.5).
6. Rebuild the code based on this selection (Section 3.6).

The final output of the algorithm is in the intermediate language shown in Figure 9 on page 6.

3.1 Gathering information

Before it can proceed, the main part of the algorithm requires a few pieces of information to be teased out of the program via static analysis:

- each λ-expression's free variables,
- call sites where the callee is known, and
- whether each λ-expression is well-known.

The set of free variables can be determined for each λ-expression via a straightforward recursive scan of the input program in which two values are returned at each step: (1) a new expression that records free variables at each λ-expression, and (2) the set of variables free in the expression. The base cases are variables and constants. The set of variables free in a variable reference includes just the variable itself, while the set of variables free in a constant is empty. The set of variables free in a `lambda`, `let`, or `letrec` is the union of the sets of variables free in each subform minus those bound by the form[2]. The set of variables free in a `call` or primitive application is the union of the sets of variables free in the subforms. The result of this scan is the intermediate language shown in Figure 5, which differs from the core language only in the appearance of a free variable set in the syntax for `lambda`. The free variable set, *fvs*, is the set of variables free in the `lambda`.

Determining the known-call sites and whether each λ-expression is well known is a bit more tricky. The desired result is the intermediate language shown in Figure 6, which differs from the preceding language in the addition of a label, l, and a well-known flag, *wk*, to each `lambda`, along with the addition of a label or bottom, denoted by $l?$, to each call. Each label represents the entry point of one λ-expression. A label is recorded in a `call` only if we can prove that the corresponding `lambda` is the only one ever called (directly) by that call. Similarly, the well-known flag, *wk*, on a `lambda` is true only if we can prove that its name is used only at call sites where it is known, i.e., at call sites where the label is recorded.

[2] For `(let (`x e_1`)` e_2`)`, x cannot appear free in e_1, since variables are uniquely named.

$$
\begin{array}{rcl}
e & ::= & c \\
 & | & x \\
 & | & \texttt{(let (}x\ e_1\texttt{)}\ e_2\texttt{)} \\
 & | & \texttt{(letrec ([}x_1\ f_1\texttt{] ...)}\ e\texttt{)} \\
 & | & \texttt{(call}\ e_0\ e_1\texttt{)} \\
 & | & \texttt{(}prim\ e\ \ldots\texttt{)} \\
f & ::= & \texttt{(lambda (}x\texttt{)}\ e\texttt{)}
\end{array}
$$

$c \in \mathit{Const},\ x \in \mathit{Var},\ \mathit{prim} \in \mathit{Prim},\ e \in \mathit{Exp},\ f \in \mathit{Fun}$

Figure 4. The core intermediate language.

$$
\begin{array}{rcl}
e & ::= & c \\
 & | & x \\
 & | & \texttt{(let (}x\ e_1\texttt{)}\ e_2\texttt{)} \\
 & | & \texttt{(letrec ([}x_1\ f_1\texttt{] ...)}\ e\texttt{)} \\
 & | & \texttt{(call}\ e_0\ e_1\texttt{)} \\
 & | & \texttt{(}prim\ e\ \ldots\texttt{)} \\
f & ::= & \texttt{(lambda}\ \underline{\mathit{fvs}}\ \texttt{(}x\texttt{)}\ e\texttt{)}
\end{array}
$$

$\mathit{fvs} \in \mathcal{P}(\mathit{Var})$

Figure 5. After uncovering free variables.

A completely accurate determination is generally impossible, but it is straightforward to compute a conservative approximation efficiently as follows. First, create an environment mapping variables to labels and a separate store mapping variables to well-known flags, both initially empty. Then, for each `letrec`, create a fresh label for each of its λ-expressions; mark each variable well-known in the store; process its body and the bodies of each of its `lambda` subforms in an extended environment that maps each LHS variable to the corresponding label; and rebuild using the processed subforms. For each of the rebuilt `lambda` subforms, record the corresponding label and the store's final value of the corresponding well-known flag. To process a `call` whose first subexpression is a variable that the environment maps to a label, process the second subexpression and rebuild the call with the label in the first position, followed by the variable and the processed second subexpression. To process any other call, process the subforms and rebuild with $\perp$ in the first position. To process a variable outside of the first position of a call, mark the variable not-well-known in the store. To process a `let` or primitive application, process the subforms and rebuild using the processed subforms. No processing is needed for constants. This process is linear in the size of the code.

The analysis above uncovers known calls only when the name of a λ-expression is in scope at the point of call. A more precise approximation can be made using a more elaborate form of control-flow analysis, as described by Serrano [15], or a type recovery that differentiates between individual procedures, as described by Adams et al. [1].

3.2 Partitioning bindings into strongly connected sets

The next step in the algorithm is to determine sets of bindings we can prove have the same lifetimes, since these are the ones that can potentially share closures without leading to space safety issues. As discussed in Section 2.3, proving that two or more procedures have the same lifetime is difficult in general, so we use a conserva-

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{(let (}x\ e_1\texttt{)}\ e_2\texttt{)} \\
& | & \texttt{(letrec ([}x_1\ f_1\texttt{] ...)}\ e\texttt{)} \\
& | & \texttt{(call}\ \underline{l?}\ e_0\ e_1\texttt{)} \\
& | & \texttt{(}prim\ e\ \ldots\texttt{)} \\
f & ::= & \texttt{(lambda}\ \underline{l}\ \underline{wk}\ fvs\ \texttt{(}x\texttt{)}\ e\texttt{)} \\
l? & ::= & l \mid \bot
\end{array}
$$

$$l \in Label,\ wk \in Bool$$

Figure 6. After uncovering known calls.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{(let (}x\ e_1\texttt{)}\ e_2\texttt{)} \\
& | & \texttt{(}\underline{\texttt{scletrec}}\ \texttt{([}x_1\ f_1\texttt{] ...)}\ e\texttt{)} \\
& | & \texttt{(call}\ l?\ e_0\ e_1\texttt{)} \\
& | & \texttt{(}prim\ e\ \ldots\texttt{)} \\
f & ::= & \texttt{(lambda}\ l\ wk\ fvs\ \texttt{(}x\texttt{)}\ e\texttt{)} \\
l? & ::= & l \mid \bot
\end{array}
$$

Figure 7. After computing strongly connected sets

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{(let (}x\ e_1\texttt{)}\ e_2\texttt{)} \\
& | & \texttt{(scletrec (}\underline{\texttt{(}}\texttt{[}x_1\ f_1\texttt{] ...}\ \underline{\texttt{)}}\ \texttt{...)}\ e\texttt{)} \\
& | & \texttt{(call}\ l?\ e_0\ e_1\texttt{)} \\
& | & \texttt{(}prim\ e\ \ldots\texttt{)} \\
f & ::= & \texttt{(lambda}\ l\ wk\ fvs\ \texttt{(}x\texttt{)}\ e\texttt{)} \\
l? & ::= & l \mid \bot
\end{array}
$$

Figure 8. After choosing subsets for sharing

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & l \\
& | & \texttt{(let (}x\ e_1\texttt{)}\ e_2\texttt{)} \\
& | & \underline{\texttt{(labels ([}l_1\ f_1\texttt{] ...)}\ e\texttt{)}} \\
& | & \texttt{(call}\ l?\ e_0\underline{?}\ e_1\texttt{)} \\
& | & \texttt{(}prim\ e\ \ldots\texttt{)} \\
f & ::= & \texttt{(lambda}\ \underline{cp?}\texttt{(}x\texttt{)}\ e\texttt{)} \\
l? & ::= & l \mid \bot \\
e? & ::= & e \mid \bot \\
cp? & ::= & cp \mid \bot
\end{array}
$$

$$cp \in Var$$

Figure 9. Final output language.

```
(make-closure code length)
(closure-set! closure index value)
(closure-ref closure index)
(vector e ...)
(vector-ref vector index)
(cons e1 e2)
(car pair)
(cdr pair)
```

Figure 10. Closure-related primitives.

tive approximation, which is to assume the same lifetime only for members of each set of bindings that are strongly connected [20] in a graph of bindings linked by free-variable relationships. In determining strongly connected sets of `letrec` bindings, it is sufficient to consider each `letrec` form individually, since the bindings of two separate `letrec` forms can be connected via free-variable links in at most one direction.

Sets of strongly connected `letrec` bindings also happen to be the minimal sets that must be allocated and initialized together so that links can be established among them, as described in Section 3.6.

The result of this step of the algorithm is a program in the new intermediate language shown in Figure 7, which differs only in that `letrec` forms have been replaced with `scletrec` forms. In general, each `letrec` form is replaced by one or more `scletrec` forms nested so that if the λ-expressions of one `scletrec` reference bindings of another `scletrec`, the first is nested within the second.

If `letrec` purification has been performed via the algorithm described by Ghuloum and Dybvig [10], the `letrec` expressions have already been split into strongly connected sets of bindings, and this step need not be repeated.

3.3 Combining bindings

Once our `letrec` forms have been split into strongly connected sets represented by `scletrec` forms, we are ready to determine the subsets of each set that will share a single closure.

If our representation allowed closures to have multiple code slots, we could use a single closure for all of the bindings of the set. Since the representation permits us just one code slot, however, we must arrange that at most a single code pointer is needed by the members of each subset.

A code pointer is needed only for not-well-known bindings, since a well-known λ-expression is invoked via its direct-call label. Thus, if there are no not-well-known bindings in the set, no code pointer is required, and we can group all of the bindings together. This leads eventually to a run-time representation without any code pointer, as covered by Case 1 of Section 2.1.

If there are not-well-known bindings in the set, however, we must have at least one closure for each. There is no advantage to creating a separate closure for the well-known bindings, so we arrange for each of the well-known bindings to share a closure with one of the not-well-known bindings. Thus, we end up with exactly as many subsets as there are not-well-known bindings in the set.

When more than one not-well-known binding exists in a set, it is unclear how to distribute the well-known bindings among the resulting subsets. If possible, we would like to combine the not-well-known bindings with well-known bindings in a way that leads to minimizing the total number of free variables in each closure. However, this situation arises infrequently enough in the programs we tested that we were not able to determine a generally useful heuristic, so our algorithm presently groups all of the well-known bindings in with an arbitrary pick of the not-well-known bindings. This transformation never does any harm, but it might not be as beneficial as some other combination in some circumstances.

The output of this step is in the intermediate language shown in Figure 8, in which the bindings of the `scletrec` are grouped into subsets that will share a single closure.

3.4 Determining required free variables

As discussed in Section 2.2, we need not include in our free variable sets all of the variables that actually occur free within our λ-expressions. In fact, we must eliminate any `letrec`-bound variable that is neither bound nor referenced in the final output because its closure has been eliminated under Case 1a of Section 2.1. We can also eliminate global variables, variables bound to constants, aliases for other variables already included, self references, and unnecessary mutual references.

To eliminate aliases and variables bound to constants, we use an environment that maps variables to expressions or $\perp$:

$$\rho \in \mathit{Env} = \mathit{Var} \rightarrow \mathit{Exp} \cup \{\perp\}$$

For the purposes of determining required free variables, the environment need map variables only to constants, other variables, and $\perp$, but we need other expressions when rebuilding the code, as described in Section 3.6.

If a local variable x is an alias for another variable y, ρ maps x to y. Similarly, if x is bound to a constant c, ρ maps x to c. If x is unbound in the final output, ρ maps x to $\perp$. Otherwise, ρ maps x to itself.

In our compiler, global variables are distinct from local variables and are referenced and assigned via static locations embedded in the code stream. Thus, they never appear in our free variable lists and hence need not be eliminated. If this were not the case, we could recognize the case where ρ does not contain a mapping for a variable and treat it as global.

We construct the environment and determine the required free variables via an outermost to innermost traversal of the input program, starting with an empty environment and augmenting it when we encounter a `let` or `scletrec`. For `(let (`$x\ e_1$`)` e_2`)`, the environment ρ' used while processing e_2 is the result of augmenting ρ as follows:

- if e_1 is a variable y, $\rho' x = \rho y$;
- if e_1 is a constant c, $\rho' x = c$;
- otherwise, $\rho' x = x$.

In the first case, ρ' maps x to ρy rather than y because y might itself be an alias, bound to a constant, or unbound.

For `scletrec`, we select representations as described in the following section and augment the environment used while processing the `lambda` and `letrec` bodies based on the representations selected, as follows.

- if a `letrec`-bound variable x is not needed because its closure has been eliminated, $\rho' x = \perp$;
- if the closure for x is a constant closure c, i.e., one containing just a code pointer, $\rho' x = c$;
- if the closure for x is just the value of the free variable y, $\rho' x = y$;
- similarly, if the closure (or other data structure) for x is shared with a closure to which another variable y has been bound, $\rho' x = y$;
- otherwise, $\rho' x = x$.

Because the traversal proceeds from outermost to innermost, by the time it is ready to process a given `scletrec` form, information about the free variables of its λ-expressions, except those bound by the `scletrec` form itself, are available in the environment. We optimistically assume that none of the variables bound by the `scletrec` form is needed until we have proved otherwise, and so compute initial free-variable sets for each group of bindings based only on free variables that are not bound by the current `scletrec` form, using the environment to eliminate unbound variables, constants, and aliases.

To be precise, if a variable $x \in$ *fvs* for any *fvs* among the λ-expressions of one group of bindings of an `scletrec`, ρx is included in the initial free variable set of the group if and only if ρx is a variable. This effectively omits unbound variables and variables bound to constants. It also eliminates aliases, since if the environment maps x and y to z (where z might be x, y, or some variable distinct from x and y), only z appears in the resulting set.

We must still decide which of the variables bound by the current `scletrec` must be included in the free variable sets of each group. At most one variable bound by each group needs to be included, since each of the variables will be bound to the same closure (if any). Thus, we pick an arbitrary representative variable *rep* from the set of variables bound by each group, and decide whether to include it in the free-variable sets of the others.

We never include *rep* in the free-variable set of its own group, effectively eliminating self references (Case 5 of Section 2.2). If all of the initial free variable sets are empty, no representatives are added, and the final free variable sets are also empty. This effectively eliminates unnecessary mutual references. (Case 5 of Section 2.2).

If, however, the free-variable set of any group is non-empty, we must add the representative *rep* of each group G to the free-variable set of each other group H if G has a nonempty set of free variables and *rep* $\in$ *fvs* for some *fvs* among the λ-expressions of H. Each group must have at least one of the other representatives in one or more *fvs*, so in this case, each of the final free-variable sets is nonempty.

The products of this step are the final free variable sets, one for each group of bindings of a single `scletrec` form. The selected representative *rep* for each group must also be communicated to the next step.

3.5 Selecting representations

Once we have the final set of free variables for each group, we are ready to decide how each group's closure is represented. This step is performed for each `scletrec` during the outermost to innermost traversal of the program described in Section 3.4. For each group in an `scletrec`, we select the representation determined by the cases in Section 2.1 as follows:

Case 1a: Well-known with no free variables

Because we have opted to combine all well-known bindings with a not-well-known binding if one exists, this case and the other well-known cases occur only if there is a single group whose λ-expressions are all well known.

In this case, no closure is needed, and ρ' maps *rep* and the other variables bound by the group to $\perp$.

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

In this case, the closure is just the value of *x*, and ρ' maps *rep* and the other variables bound by the group to *x*.

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

In this case, the closure is a run-time allocated pair containing the values of *x* and *y*. ρ' maps each of the variables bound by the

group to *rep*, and code to create the pair and bind *rep* to the pair is generated as described in Section 3.6.

Case 1d: Well-known with three or more free variables $x \ \ldots$

In this case, the closure is a run-time allocated vector containing the values of $x \ \ldots$. ρ' maps each of the variables bound by the group to *rep*, and code to create the vector and bind *rep* to the vector is generated as described in Section 3.6.

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

In this case, the closure is a constant closure c containing just a code pointer, and ρ' maps *rep* and the other variables bound by the group to c.

Case 2b: Not well-known with one or more free variables $x \ \ldots$

In this case, the closure is a run-time allocated closure whose code pointer is the label of the single not-well-known binding in the group, and its free variable slots hold the values of $x \ \ldots$. ρ' maps each of the variables bound by the group to *rep*, and code to produce the closure and bind *rep* to the closure is generated as described in Section 3.6.

In cases 1c and 1d, we have a further opportunity, which is to locate a binding created by an outer `scletrec` that would have exactly the same set of free variables if the pair or vector were shared. For example, an outer closure might be represented as a pair with free variables x and y. If the new closure would also have just x and y as free variables, there is no harm in borrowing the pair used for the outer closure. Furthermore, if the new closure has f as a free variable as well as x and y, it can still use the outer pair, since f becomes a self reference and hence is not needed.

To implement this form of sharing, the algorithm maintains an additional compile-time environment, *bank*, that maps sets of free variables to the representative variables of closures available to be borrowed. We deposit into *bank* only closures represented as pairs or vectors, since we cannot borrow a closure with a code pointer without possibly retaining the code pointer too long, as described in Section 2.3. For any well-known procedure with more than one free variable, we check to see if a closure with a compatible set of free variables already exists in *bank* and, if so, map *rep* and the other variables bound by the group to the representative variable of the borrowed closure. While processing the body of a λ-expression, we withdraw from *bank* those closures whose names do not appear free in the λ-expression, since they are not visible in the body.

3.6 Rebuilding the code

The final step of the algorithm is to produce the output code. This step is performed for each `scletrec` during the traversal of the code described in Section 3.4, based on the decisions made in Section 3.5. The intermediate language for the final output is shown in Figure 9 and may require some or all of the primitive operations shown in Figure 10. Producing the output code involves:

- generating a `labels` form binding labels to lambda expressions,
- adding a closure-pointer (*cp*) variable to each λ-expression that requires its closure,
- rewriting variable references, and
- generating code to create the required pairs, vectors, and closures.

The generated `labels` form binds the label associated with each λ-expression in each group of the `scletrec` form to the corresponding λ-expression. Although the `labels` form appears where the `scletrec` form originally appeared, the scope of each label is the entire input program, so that any call can jump directly to the code produced by the corresponding λ-expression. In fact, since the resulting λ-expressions access their free variables, if any, through an explicit closure argument, all the `labels` forms and the λ-expressions within them can be moved to the top level of the program as part of this or some later transformation, if desired.

If a λ-expression has any free variables, it is given a closure-pointer (*cp*) variable; when code is ultimately generated for the λ-expression, *cp* must be assigned to the incoming closure (or pair or vector), just as the formal parameter x is assigned to the incoming actual parameter. In essence, *cp* is merely an additional formal parameter, and the closure is merely an additional actual parameter.

Each reference to a free variable within a λ-expression must be replaced by an expression to retrieve the value of the free variable from the closure. Similarly, a reference to a variable bound to a constant must be replaced by the constant, a reference to a variable bound to another variable must be replaced by a reference to the other variable, and references to unbound variables must be eliminated. Sufficient information already exists in the incoming environment ρ to handle the latter set of cases, where each reference to a variable x is simply replaced during the outermost to innermost traversal by ρx. Unbound variables arise only from well-known procedures and can thus appear only in the second (closure) position of a `call`, and only when the `call`'s label is not $\perp$. When this happens, the variable maps to $\perp$, and the reference to the variable is replaced in the `call` by $\perp$, which frees the caller from passing any sort of closure to the callee.

To handle free variables, ρ is augmented to create an environment ρ' as follows:

- for a self reference x, $\rho' x = cp$;
- for a procedure represented by one of its free variables x, $\rho' x = cp$;
- for a procedure represented by a pair of x and y, $\rho' x =$ `(car` *cp*`)`, and $\rho' y =$ `(cdr` *cp*`)`;
- for a procedure represented by a vector of $x_1 \ldots \ x_n$, $\rho' x_i =$ `(vector-ref` *cp* i`)` for $0 \leq i < n$;
- for a procedure represented by a closure of $x_1 \ldots \ x_n$, $\rho' x_i =$ `(closure-ref` *cp* i`)` for $0 \leq i < n$.

The augmented environment ρ' is used while processing the body of each `lambda` form as well as the body of the `scletrec` form, since references to the `letrec`-bound variables can occur in both contexts.

Generating code to create the required pairs, vectors, and closures is straightforward, with one minor twist. Since the links among the non-constant closures created for the set of groups in an `scletrec` necessarily form one or more cycles, the code to create them allocates all of the closures before storing the values of the free variables. For example, if an `scletrec` form with body e has two groups: one with representative r_1, label l_1, and free variables r_2 and y; and one with representative r_2, label l_2, and free variable r_1; the `scletrec` form is replaced with the following:

```
(let ([r1 (make-closure l1 2)]
      [r2 (make-closure l2 1)])
  (closure-set! r1 0 r2)
  (closure-set! r1 1 y)
  (closure-set! r2 0 r1)
  e)
```

The order of the bindings and the order of the `closure-set!` forms does not matter.

4. Results

Our implementation extends the algorithm described in Section 3 to support the full R6RS Scheme language [18]. To determine the effectiveness of closure optimization, we ran the optimization over a standard set of 67 R6RS benchmarks [6] and instrumented the compiler and resulting machine code to determine:

- statically,
  - the number of closures eliminated and
  - the reduction in the total number of free variables; and
- dynamically,
  - the reduction in allocation cost and
  - the reduction in the number of memory references.

Overall the optimization 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[3].

These numbers are gathered after a pass that performs aggressive inlining, constant propagation, constant folding, and copy propagation [21]. It also follows a pass that recognizes loops and converts them into the equivalent of labels and gotos. As a result, the numbers do not include the benefits of eliminating variables bound to constants or other variables, except where these situations arise during closure optimization. Inlining and loop recognition also eliminate the need for some closures and can impact the number of free variables (potentially increasing this number in some cases and decreasing it in others). Global variables are not counted as free variables, as they are stored at a fixed location and accessed via primitives that set or retrieve their values.

Constant closures and those replaced by a single free variable are considered eliminated in our numbers, since neither incurs run-time overhead. In counting the allocation of vectors and closures, we include the space required for the length (in the case of vectors) and the code pointer (in the case of closures), as well as the space required to hold the free-variable values[4].

In addition to the overall numbers, we ran in isolation optimizations that eliminate self-references, eliminate mutual references, share closures in strongly connected sets of bindings, borrow closures from outer sets of bindings, and select more efficient representations. Table 1 shows the breakdown in the percentage of eliminated closures, free-variables, memory references, and allocation by optimization. The table shows that running the optimizations together results in greater benefits than running the optimizations separately. This is because some of the optimizations can lead to opportunities for the others.

The results for the complete set of benchmarks are impressive but vary from benchmark to benchmark. Some of the benchmarks are simpler benchmarks for running functions like factorial, Tak, and Fibonacci. It is interesting to see how much these benefit, but ultimately, the larger programs within the benchmark suite help to provide a better indicator of how well these optimizations will work on real-world programs. The R6RS "compiler" benchmark is the largest, in terms of source code, and has the most closures initially. Statically, 40.85% of closures and 31.9% of free variables are eliminated, and dynamically, 31.52% of the allocation and 27.59% of the memory references attributable to closure creation and access are eliminated. This example might be more typical of the average real-world program.

Optimization	Closure	FV	Mem. Ref.	Alloc.
Self-ref.	0.00%	25.41%	45.64%	19.33%
Mutual-ref.	0.00%	7.91%	32.55%	6.14%
Representation	29.65%	3.48%	1.23%	20.78%
Sharing	1.91%	3.17%	0.00%	0.58%
Borrowing	0.20%	0.28%	0.00%	0.02%
All	56.94%	44.89%	58.58%	58.25%

Table 1. Eliminated closures (Closure), free-variables (FV), memory references (Mem. Ref.), and allocation (Alloc.) by optimization

The R6RS "simplex" benchmark is an example of a benchmark that does not benefit much from our closure optimization eliminating statically only 8.00% of the closures and 14.97% of free variables while eliminating dynamically 13.96% of the allocation and 9.84% of the memory references attributable to closure creation and access. The benchmark is written as a set of functions that mutate data structures pointed to from free variables. Since most of the free variables are not other closures (and some that are have already been inlined by the source optimization pass) there are not many closures that can be eliminated.

The R6RS "nucleic" benchmark is an example of a longer benchmark that performs better than average, eliminating 67.74% of the closures and 66.23% of the free variables statically, and 37.43% of the allocation and 72.69% of the memory references attributable to closure creation and access at run time. This benchmark has many top-level definitions that are either λ-expressions or constants, so the free variables tend to be other closures. While data structures are also mutated here, they tend to be passed as arguments rather than stored as free variables in the closure. In a program like this, we expect to see many of the closures eliminated.

In addition to the benchmarks, we also measured the effectiveness of the optimization algorithm when run on the sources for our own compiler. Statically, 45.67% of closures and 32.36% of free variables are eliminated, and dynamically, 47.52% of the allocation and 47.00% of the memory references attributable to closure creation and access are eliminated.

We also measured how our optimizations affected the run times of our benchmarks. The decrease in run times ranged from negligible up to 20%, with an average decrease of 3.6%. Run times actually increased for a few of the benchmarks. Since our optimizations supposedly guaranteed not to add overhead, we examined one of those whose run time increased and determined that its poor performance was due to bad caching. Rearranging the code led to equivalent performance between the optimized and unoptimized versions of the code. Several of the benchmarks spend most of their time in procedures recognized as loops earlier and do not benefit at all from the closure optimization. Larger programs tended to experience greater improvement in runtime, which correlates well with the other measurements. Memory allocation in our implementation is fast, averaging around three instructions plus one store to initialize each field. For implementations with slower or even out-of-line allocation, the decrease in run time due to reduction in closure allocation would be greater. Similarly, implementations that do not perform

[3] The percentage of memory stores attributable to closure initialization also decreases in direct proportion with the reduction in allocation.

[4] The numbers do not include pad words required to maintain object alignment, e.g., double-word alignment on 32-bit machines.

inlining or loop recognition would likely benefit more from the optimizations. On the other hand, systems with higher overhead in other places would likely benefit less.

Case 1 of Section 2.3 discusses the possibility of extending our implementation to support multiple code pointers so that all of the bindings of a strongly connected set of bindings can share the same closure, even if more than one is not well-known. Because making this change to our system would be a major undertaking, we decided to determine the potential benefit of the optimization before proceeding. Our test showed that this optimization would affect less then one tenth of one percent of `letrec` bindings, which led us to abandon the idea.

Our implementation of the optimization algorithm employs standard techniques to avoid the implied overhead of creating and updating the sets required by the algorithm. For example, it associates a *seen* flag with each variable to indicate when a free-variable has already been added to a set. Although we have yet to create a proof, we believe that the implementation is linear in the number of variables free in all λ-expressions in the program, which is the best any implementation of the flat-closure model can achieve [5]. In other words, our implementation adds at most constant overhead to the naive flat-closure model, and can sometimes improve the speed of downstream passes via the elimination of closure operations. Indeed, the optimization adds essentially no measurable compile-time overhead in our compiler, with compile times varying by an average of less than 1% with the optimization disabled or enabled.

5. Related Work

Our replacement of a well-known closure with a single free variable is a degenerate form of lambda lifting [12], 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 also 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, since we are replacing a single package of values with just one value.

Serrano [15] describes a closure optimization based on control-flow analysis [17]. 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, since 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 and Wand [19] 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 [13] 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 continuation passing style [11] 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 and Appel [16] 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. Since a substantial portion of the savings reported resulted from global variables [2], 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 and Métayer [9] 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.

Dragoş [7] 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. It also looks for heap-allocated free variables that are reachable from local variables or the stack to avoid adding them to closure. The optimization helps eliminate the closures for well-known calls by lambda lifting, if possible.

Appel [4] describes the elimination of 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, though in our optimization we only allow 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 to the existence of similar optimizations in various newsgroups and blogs. While other systems may implement some of the optimizations 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 presents 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.

[5] In the worst case, the number of such free variables is quadratic in the size of the program [16], although the worst case appears to be approached rarely in practice.

References

[1] M. D. Adams, A. W. Keep, J. Midtgaard, M. Might, A. Chauhan, and R. K. Dybvig. Flow-sensitive type recovery in linear-log time. In *OOPSLA*, pages 483–498, 2011.

[2] A. Appel. Private communication. 1994.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, 1992.

[4] A. W. Appel. *Closure conversion*, chapter 10, pages 103–124. Cambridge University Press, 40 West 20th Street, New York, NY 10011-4211, 1992.

[5] L. Cardelli. The functional abstract machine. Technical report, Bell Laboratories, 1983.

[6] W. D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`. Accessed Oct 20, 2011.

[7] I. Dragos. Optimizing Higher-Order Functions in Scala. In *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2008.

[8] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, Chapel Hill, Apr. 1987.

[9] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Trans. Program. Lang. Syst.*, 13:21–51, January 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/114005.102805. URL `http://doi.acm.org/10.1145/114005.102805`.

[10] A. Ghuloum and R. K. Dybvig. Fixing letrec (reloaded). In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*, pages 57–65, 2009.

[11] J. Guy L. Steele. Rabbit: A compiler for scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[12] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings 1985 Conference on Functional Programming Languages and Compiler Architecture*, pages 190–203. Springer-Verlag, 1985.

[13] D. A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, 1988.

[14] D. A. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. Orbit: An optimizing compiler for scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, 1986.

[15] M. Serrano. Control flow analysis: A functional languages compilation paradigm. In *Proceedings of the 1995 ACM symposium on Applied computing*, pages 118–122. ACM, 1995. doi: 10.1145/315891.315934.

[16] Z. Shao and A. W. Appel. Efficient and safe-for-space closure conversion. *ACM TOPLAS*, 22:129–161, 2000.

[17] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174. ACM, 1988. doi: 10.1145/53990.54007.

[18] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009. doi: 10.1017/S0956796809990074. URL `http://dx.doi.org/10.1017/S0956796809990074`.

[19] P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, 19:48–86, January 1997. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/239912.239915. URL `http://doi.acm.org/10.1145/239912.239915`.

[20] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[21] O. Waddell. *Extending the Scope of Syntactic Abstraction*. PhD thesis, Indiana University, 1999.

[22] O. Waddell, D. Sarkar, and R. K. Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-order and and symbolic computation*, 18(3/4):299–326, 2005.