# Flow-directed Inlining

Suresh Jagannathan and Andrew Wright

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
{suresh,wright}@research.nj.nec.com

**Abstract**

A *flow-directed inlining* strategy uses information derived from control-flow analysis to specialize and inline procedures for functional and object-oriented languages. Since it uses control-flow analysis to identify candidate call sites, flow-directed inlining can inline procedures whose relationships to their call sites are not apparent. For instance, procedures defined in other modules, passed as arguments, returned as values, or extracted from data structures can all be inlined. Flow-directed inlining specializes procedures for particular call sites, and can selectively inline a particular procedure at some call sites but not at others. Finally, flow-directed inlining encourages modular implementations: control-flow analysis, inlining, and post-inlining optimizations are all orthogonal components. Results from a prototype implementation indicate that this strategy effectively reduces procedure call overhead and leads to significant reduction in execution time.

## 1 Introduction

Functional languages like Scheme [7] or ML [17], and object-oriented languages like Self [5] or Java [24], provide abstraction through the use of first-class procedures and objects. These mechanisms encourage modular programming and enable elegant programming paradigms. However, expressivity of this kind has often come at the price of poor performance. A significant factor contributing to this performance loss is the overhead of pervasive procedure calls and method dispatches. In the absence of any compile-time optimizations, these overheads can be significant, especially for programs that make liberal use of data and control abstractions.

Inlining is an optimization that trades code space for time by replacing a procedure call with the called procedure's body. Inlining has two important benefits. First, it eliminates procedure call overhead. This overhead includes the cost of passing arguments, saving and restoring registers, building return linkage information, and branching to the procedure body. Second, by merging the procedure body with its calling context, inlining enables other optimizations to specialize the caller and callee together.

The same mechanisms that make functional and object-oriented languages more expressive conspire to make inlining more difficult. In functional languages, the called function may be computed by a complicated expression that returns a procedure value. In object-oriented languages, the value of a virtual method is based on the types of its argument objects. Thus the procedure or method applied at a particular call site may not be obvious. An inlining algorithm for these kinds of languages is unlikely to be very effective if it does not take a program's control- and data-flow properties into consideration.

*Flow-directed inlining* uses an approximation of a program's control- and data-flow to identify call sites where procedures may be inlined, and to estimate a procedure's size when specialized for a particular call site. Flow-directed inlining has several important attributes:

- *Generality.* All user-defined procedures are candidates for inlining. Inlining is not limited to system-defined primitives, global procedures, specially marked procedures, or procedures that have no free variables.

  Since inlining decisions are driven by a global control-flow analysis, inlining can be performed at call sites where the called procedure is not syntactically obvious. For higher-order languages, our system is capable of inlining procedures passed as arguments, imported from modules, extracted from data structures, or encapsulated as methods within objects. For object-oriented languages, flow-directed inlining can optimize calls to virtual methods. Thus, flow-directed inlining minimizes the penalty for liberally using higher-order procedures and objects to define layers of abstraction and encapsulation.

- *Selectivity.* Inlining decisions are highly selective. A particular procedure may be inlined at some call sites, but not at others. The decision to inline at a call site is based on an estimate of the procedure's size when specialized for that call site. Flow analysis facilitates constructing different size estimates for a procedure at different call sites based on the portions of the procedure that may be reached from those call sites. For example, large procedures can be aggressively inlined

To appear in the Proceedings of the 1996 ACM Symposium on Programming Language Design and Implementation. 

when determination of conditional tests in their bodies permits entire blocks of code to be pruned.

- *Modularity.* The structure of the flow analysis is independent of the design of the inlining algorithm. Inlining decisions are in turn made independently of any subsequent optimizations. Thus, different flow analyses can be substituted without necessitating changes to the inlining algorithm, and different inlining algorithms can be incorporated without modifying the flow analysis. This orthogonality makes it simple to upgrade a compiler that uses our strategy to include more flow powerful analysis techniques as they become available, or to incorporate new inlining strategies without modifying existing optimizations or analyses.

We have implemented a source-to-source inlining optimizer for R$^4$RS Scheme programs [7]. Initial experimental results from our prototype are very encouraging. For the programs in our benchmark suite, flow-directed inlining improves execution times by an average of 25%. For several of these benchmarks, performance doubles. On average, code sizes remain roughly the same as the original program. Based on these results, we believe flow-directed inlining is an effective optimization technique for functional and object-oriented languages.

The remainder of the paper is organized as follows. Section 2 gives an informal overview of flow-directed inlining. Section 3 gives a formal description of the algorithm. We present performance results in Section 4, and place flow-directed inlining in the context of related work in Section 5.

## 2 Overview

Flow-directed inlining consists of three steps: control-flow analysis, inlining, and local optimization.

### 2.1 Flow Analysis

We first perform a *polyvariant* [4, 16] control-flow analysis over the input program. Polyvariance is important for two reasons. First, because it disambiguates different uses of a procedure at different call sites, polyvariance yields very precise flow information. This enables many call sites to be identified as candidates for inlining. Second, because it analyses a procedure with respect to a specific call site, polyvariance enables specialization of the procedure for the call sites where it is inlined.

Inlining can be performed at a call site only if there is a unique procedure applied at that site. For example, in

```
(define g (λ (f x) (f x)))
```

the call site `(f x)` is considered a candidate for inlining only if flow analysis determines that there is a single *abstract* value associated with `f` in the call. This value must be an abstract closure. An abstract closure corresponds to a set of exact closures. These exact closures may be closed over different environments, but they must all share the same code. In other words, for `f` to be inlined, all calls to `g` must supply closures that are created from evaluation of the same λ-expression in the program.

To illustrate, consider the following object-oriented program fragment:

```
(define make-network
   (λ (args)
    ...
    (λ (msg)
      (case msg
        ((open)  (λ (addr) open a new port))
        ((close) (λ (port) close a port))
        ((send)  (λ (msg port)
                     send msg to port ))
        ((receive) (λ (port)
                     receive from port ))
        ...))))
```

A network is a procedure that dispatches on a request. A network is created by calling `make-network` with a set of arguments. Given a network `N`, the expression

```
((N 'open) "http://www.foo.com")
```

invokes a procedure to open a connection to the specified network. The flow analysis will associate `N` with the procedure

```
(λ (msg) (case msg ...))
```

and `(N 'open)` with the procedure

```
(λ (addr) open a new port).                    (*)
```

This approach to identifying potential inline sites is quite different from traditional approaches to inlining that rely on syntactic heuristics. For example, in the expression

```
(let ((f (λ (N)
           ...
           ((N 'open) "http://www.foo.com")
           ...)))
  ...
  (f (make-network args))
```

a conventional inlining optimizer will inline procedure (∗) at `(... "http://www.foo.com")` only if it also inlines `f`, performs significant simplification, and repeats the inline algorithm on the simplified program. In contrast, flow-directed inlining may treat the call `(... "http://www.foo.com")` as a candidate for inlining even if `f` is not inlined. All inline decisions are made prior to any simplification.

### 2.2 Inlining

The inlining algorithm proceeds depth-first over the program, inlining within a procedure $P$ before inlining $P$ itself. Closures at candidate call sites whose costs lie below a fixed threshold are inlined. Large procedures are not penalized by prohibiting inlining within their bodies. Even if a procedure $P$ is too big to be inlined at any of its call sites, procedures applied within $P$ can still be inlined within $P$'s body provided the constraints described above are satisfied. The algorithm builds a loop when inlining a call site would otherwise lead to infinite unfolding.

```
(define map
  (λ (f a1 . args)
    (letrec ([map1 (λ (f l)
                     (if (null? l)
                         '()
                         (cons (f (car l))
                               (map1 f (cdr l)))))]
             [map* (λ (lists)
                     (if (null? (car lists))
                         '()
                         (cons (apply f (map car lists))
                               (map* (map cdr lists)))))])
      (if (null? args)
          (map1 f a1)
          (map* (cons a1 args))))))
```

Figure 1: Scheme's `map` procedure

```
((λ (f a1 . args)
    (letrec ([map1 (λ (f l) ...)]
             [map* (λ (lists) ...)])
      (letrec ([map1 (λ (f l)
                       (if (null? l)
                           '()
                           (cons (car (car l))
                                 (map1 f (cdr l)))))])
        (map1 f a1))))
  car m)
```

Figure 2: Result of inlining `(map car m)` prior to simplification.

```
(letrec ([map1
          (λ (l)
            (if (null? l)
                '()
                (cons (car (car l))
                      (map1 (cdr l)))))])
    (map1 m))
```

Figure 3: Result of inlining `(map car m)` after simplification.

To illustrate the inlining algorithm, consider Fig. 1 which provides an implementation of Scheme's `map` procedure. Two local procedures are defined in `map`'s body: `map1` is called when `map` is invoked with a unary procedure, and `map*` is invoked for all other cases. Because `map1` operates over procedures whose arity is known, it is more efficient than `map*` whose implementation uses an expensive `apply` operator to handle the variable-arity case.

Flow analysis helps compute reasonably precise cost estimates. Consider the call `(map car m)`. The cost of inlining `map` at this call site is dependent on how much specialization can be performed at the inlined site. Because flow-directed inlining uses results from a polyvariant flow analysis, it can tailor its cost estimate based on the specific context in which a call occurs. In this call, flow analysis reveals that the conditional test `(null? args)` will be true, and hence the else-branch will not be taken. Thus, the cost estimate for this call need not include the else branch or the definition of `map*`. If `map` is inlined, the outer conditional test, the definition of `map*`, and its call will all be pruned in the inlined copy. Fig. 2 shows the result of inlining `(map car m)` prior to simplification.

### 2.3 Simplification

After inlining, $\beta_v$-substitution, dead-code elimination, and local code transformations are performed. These transformations include restructuring procedure definitions and calls to eliminate unused formal parameters. These optimizations are all syntactic and use no flow information. The simplifier performs no transformations that violate the results of the flow analysis. Hence, other optimizations could use flow information generated for the original program when operating over the inlined version. Fig. 3 shows the result of applying the simplifier to the inlined code for `(map car m)`.

## 3 Flow-directed inlining

In this section we give a precise account of our flow-directed inlining strategy and the flow analysis that drives it. We have used this same analysis to eliminate run-time checks from dynamically typed programs [15].

### 3.1 Source language

The abstract syntax for our Scheme-like source language has *labeled expressions* $e^l$ of the form

$$
\begin{array}{lcl}
e^l & ::= & c^l \mid f^l \mid x^l \\
 & \mid & \texttt{(}p\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l \\
 & \mid & \texttt{(call}\ e_0^{l_0}\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l \\
 & \mid & \texttt{(begin}\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l \\
 & \mid & \texttt{(if}\ e_1^{l_1}\ e_2^{l_2}\ e_3^{l_3}\texttt{)}^l \\
 & \mid & \texttt{(let ((}x_1\ e_1^{l_1}\texttt{)}\ldots\texttt{(}x_n\ e_n^{l_n}\texttt{))}\ e_b^{l_b}\texttt{)}^l \\
 & \mid & \texttt{(letrec ((}y_1\ f_1^{l_1}\texttt{)}\ldots\texttt{(}y_n\ f_n^{l_n}\texttt{))}\ e_b^{l_b}\texttt{)}^l \\
 & \mid & \texttt{(cons}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^l \\
 & \mid & \texttt{(car}\ e_1^{l_1}\texttt{)}^l \\
 & \mid & \texttt{(cdr}\ e_1^{l_1}\texttt{)}^l \\
 & \mid & \texttt{(set-car!}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^l \\
 & \mid & \texttt{(set-cdr!}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^l \\
 & & \\
f^l & ::= & \texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_b^{l_b}\texttt{)}^l
\end{array}
$$

where $c \in Const$ are *constants*, $p \in Prim$ are *primitives*, $x, y, z \in Var$ are *variables*, and $l \in Label$ are *labels*. Labels, which are not present in the concrete syntax, are discussed below. A $\lambda$-term constructs an anonymous procedure; terms beginning with $p$ are uses of primitive operations; `call` introduces a procedure call; `begin` and `if` are sequencing and conditional operations; `let` provides non-recursive local bindings; `letrec` constructs named recursive procedures; `cons` constructs mutable pairs; `car` and `cdr` are the first and second projections on pairs, and `set-car!` and `set-cdr!` mutate the components of pairs.

*Free* variables ($FV$) and *bound* variables ($BV$) are defined as usual for a lexically scoped language [3], with `let` binding its variables $x_1 \ldots x_n$ in $e_b$ and `letrec` binding its variables in all of $f_1 \ldots f_n$ and $e_b$. The bound expressions $f_1 \ldots f_n$ of a `letrec`-expression must be procedures. *Closed* programs are expressions with no free variables.

We sometimes distinguish different uses of variables as follows. First, a meta-variable named $y$ must be bound by a `letrec`-expression. Second, a subscript on a variable occurrence indicates how the variable is bound. A variable occurrence like $x^l_{[\lambda]}$ indicates that $x$ is bound by a $\lambda$-expression. A variable occurrence like $x^l_{[l']}$ or $y^l_{[l']}$ indicates that the variable is bound by the `let`- or `letrec`-expression with label $l'$, except that recursive variable occurrences within the bindings of a `letrec`-expression appear as $y^l_{[\mathrm{rec}]}$. The following expression illustrates the use of these conventions:

$$
\begin{array}{l}
\texttt{(let ((}x\ \texttt{0))} \\
\quad \texttt{(letrec ((}y\ \texttt{(}\lambda\ \texttt{(}z\texttt{) (call}\ y_{[\mathrm{rec}]}\ z_{[\lambda]}\texttt{))))} \\
\qquad \texttt{(call}\ y_{[l_2]}\ x_{[l_1]}\texttt{))}^{l_2}\texttt{)}^{l_1}
\end{array}
$$

In order to unambiguously name different components of a program, we assume programs have two properties. First, since we use labels to identify subterms of a program, each subterm of a program must have a unique label. Second, since we refer to a specific variable by its name, we assume that all free and bound variables in a program are distinct. This condition can be met by renaming bound variables appropriately.

Like Scheme, this source language is latently typed—no static typing discipline is imposed on programs. Like both Scheme and ML, this source language has an ordinary call-by-value semantics.

### 3.2 Flow analysis

Flow analysis determines the sets of values that may be bound to variables and returned from subexpressions. We name these sets *abstract values*; in other words, a single abstract value represents a set of exact values. Simple *monovariant* flow analyses like 0CFA [23] and SBA [13] associate a single abstract value with each variable and subexpression. *Polyvariant* analyses [4] associate multiple abstract values with each variable and expression, distinguishing them by *contours* that abstract the program's run-time state. We call these variable-contour pairs and label-contour pairs *program points*. Let *Avalue* be some set of abstract values and *Contour* be some set of contours. A *flow analysis* of a closed program $P$ is a function

$$F : (Var \times Contour) + (Label \times Contour) \xrightarrow{\text{fin}} Avalue$$

that maps the program points of $P$ to abstract values. $F\langle x, \kappa\rangle$ identifies the values that may be bound to $x$ in contour $\kappa$. $F\langle l, \kappa\rangle$ identifies the values that may be returned by the subexpression labeled $l$ in context $\kappa$.

Abstract values are defined as follows:

$$
\begin{array}{rcll}
a & \in & Avalue & = Aconst + Aclosure + Apair \\
\tau & \in & Aconst & = \{\mathsf{true}, \mathsf{false}, \mathsf{nil}, \mathsf{number}, \ldots\} \\
\langle l, \rho, \kappa\rangle_\lambda & \in & Aclosure & = Label \times Aenv \times Contour \\
\langle l, \kappa\rangle_\pi & \in & Apair & = Label \times Contour \\
\rho & \in & Aenv & = Var \xrightarrow{\text{fin}} Contour \\
\kappa & \in & Contour &
\end{array}
$$

An abstract value $a$ is a set of abstract constants, abstract closures, and abstract pairs. Abstract constants like $\mathsf{true}$, $\mathsf{false}$, and $\mathsf{nil}$ each denote a single exact value, while abstract constants like $\mathsf{number}$ denote a set of exact values. An abstract closure $\langle l, \rho, \kappa\rangle_\lambda$ identifies procedures created from the $\lambda$-expression $\texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)}^l$. The contour $\kappa$ of an abstract closure, paired with an argument $x_i$ or a label of a subexpression of $e_b$, determines the program points for the body of the abstract closure. Thus two abstract closures that share the same label but use different contours will have different program points. The abstract environment $\rho$ of an abstract closure records the contours in which its free variables are bound. An abstract pair $\langle l, \kappa\rangle_\pi$ identifies pairs constructed from $\texttt{(cons}\ e_1\ e_2\texttt{)}^l$ in contour $\kappa$.

Contours abstract different execution contexts and are used to distinguish argument bindings and subexpression results arising at different points in a program's execution. The set of contours must be finite to ensure that the analysis computes finite information, i.e., that the analysis terminates. Since the subsets of variables and labels that a program uses are also finite, a flow analysis can yield only finitely many abstract values for a particular program. Within this constraint, the choice of the contour set governs the precision of the analysis. Before discussing a particular set of contours suitable for flow-directed inlining, we introduce the analysis proper.

Given a program $P$ and a finite function $F$ of appropriate type, $F$ is a flow analysis or *flow graph* for $P$ if it satisfies the relation $\mathcal{A}$ in Fig. 4. The rules for constants, primitive applications, and $\lambda$-expressions force the corresponding program points or flow graph nodes to include appropriate values. The function *AbstractValOf* maps a constant to its abstract value; for example, $\mathit{AbstractValOf}(\texttt{\#f}) = \mathsf{false}$ and $\mathit{AbstractValOf}(\mathsf{5}) = \mathsf{number}$. The function *AbstractResultOf* maps a primitive to its abstract result; for example, $\mathit{AbstractResultOf}(+) = \mathsf{number}$.

The remaining rules introduce constraints or edges between nodes. The rule for variables constrains the abstract evaluation of a variable $x$ to include the abstract value bound to it in contour $\rho(x)$. For each abstract closure that can arise in the function position of a `call`, the application rule introduces two kinds of constraints. The first kind, $F\langle l_j, \kappa\rangle \subseteq F\langle x_j, \kappa'\rangle$, requires the formal parameters of the abstract closure to include the abstract values of the actual parameters. The second kind, $F\langle l_b, \kappa'\rangle \subseteq F\langle l, \kappa\rangle$, requires the application node to include the abstract value arising from the evaluation of the body of the abstract closure. In the figure, the notation $\rho[x_1 \ldots x_n \mapsto \kappa]$ means the functional update or extension of the environment $\rho$ at each of $x_1 \ldots x_n$ to $\kappa$. The constraint in the rule for `begin` returns the abstract value of the last subexpression as the result of the `begin` expression. In the rule for conditional expressions, evaluation of the then-clause (resp. else-clause) is conditional upon true (resp. false) arising as a possible value of the test. If the abstract value of the test includes both true and false, both the then-clause and the else-clause contribute to the result of the conditional expression. If the test diverges and its abstract evaluation yields the empty set, neither the then-clause nor the else-clause is evaluated.

None of the rules in the top part of Fig. 4 construct new contours. To provide polyvariance, the rules in the second section of the figure construct new contours for `let`-bound closures in a manner that mimics the actions of a polymorphic type system. Hence we call this form of polyvariance *polymorphic splitting* [15]. Under polymorphic splitting, contours are finite strings of labels. The $e_1$ subexpression of a `let`-expression evaluates in a contour $\kappa : l$ obtained by appending $l$ to the label string $\kappa$. This contour is captured in any abstract closures created by $e_1$. Since only `let`-expressions extend contours, the nesting depth of `let`-expressions within their $e_1$ subexpressions bounds the length of contours. At a subsequent use of the `let`-expression's variable, say $x^{l'}_{[l]}$, the contours in abstract closures bound to $x$ are modified by replacing $l$ with $l'$ (the notation $\kappa[l''/l]$ in the figure means the contour derived by replacing $l''$ with $l$ in $\kappa$). In this manner, different uses of the same abstract closure evaluate in different contours. The figure presents a rule for `let`-expressions with a single variable; the extension of this rule to multiple variables is straightforward, and is omitted here.

To illustrate polymorphic splitting, consider the following expression:

```
(let ((f (λ (x) x)¹))
  (begin (f² #t)
         (+ (f³ 0) 1)))⁰
```

The abstract closure constructed for `f` is $\langle 1, \emptyset, [0]\rangle_\lambda$ where 1 is the $\lambda$-expression's label, $\emptyset$ is the empty environment, and the $[0]$ is contour consisting of the label of the `let`-expression. At $\texttt{f}^2$, this closure is split to $\langle 1, \emptyset, [2]\rangle_\lambda$, hence the application $(\texttt{f}^2\ \texttt{\#t})$ binds `x` to $\{\mathsf{true}\}$ in contour $[2]$. At $\texttt{f}^3$, `f`'s closure is split to $\langle 1, \emptyset, [3]\rangle_\lambda$, hence the application $(\texttt{f}^3\ \texttt{0})$ binds `x` to $\{\mathsf{number}\}$ in contour $[3]$. Since the two applications of `f` evaluate in different contours, the application $(\texttt{f}^3\ \texttt{0})$ yields the abstract value $\{\mathsf{number}\}$. Were the two applications evaluated in the same contour, $(\texttt{f}^3\ \texttt{0})$ would yield the less precise result $\{\mathsf{number}, \mathsf{true}\}$. For additional explanation and examples of polymorphic splitting, we refer the reader to an earlier paper [15].

The third part of Fig. 4 presents a polymorphic splitting rule for `letrec`-expressions. As in a typical polymorphic type system, only occurrences of variable $y$ in the body $e_b$ of $(\texttt{letrec}\ ((y\ \ f))\ e_b)$ are polyvariant. We call these *non-recursive* occurrences, while the occurrences of $y$ within $f$ are *recursive* occurrences. Recursive uses of $y$ evaluate in the most recent contour of a non-recursive use of $y$. To illustrate, consider the following expression:

```
(letrec ((last (λ (l)
                 (if (null? (cdr l))
                     (car l)
                     (last (cdr l))))¹))
  (begin (last² (cons 1 (cons 2 '())))
         (last³ (cons "a" (cons "b" '())))))
```

The abstract closure created for `last` is $\langle 1, \emptyset, [\,]\rangle_\lambda$ where 1 is the $\lambda$-expression's label. At $\texttt{last}^2$, the abstract closure for `last` is split to yield $\langle 1, [\texttt{last} \mapsto [2], [2]\rangle_\lambda$. During abstract evaluation of $(\texttt{last}^2\ \ldots)$, recursive references to `last` yield the same abstract closure $\langle 1, [\texttt{last} \mapsto [2], [2]\rangle_\lambda$. Similarly, during abstract evaluation of $(\texttt{last}^3\ \ldots)$, the abstract closure for `last` is split to yield $\langle 1, [\texttt{last} \mapsto [3], [3]\rangle_\lambda$, and recursive references to `last` yield the abstract closure $\langle 1, [\texttt{last} \mapsto [3], [3]\rangle_\lambda$. In each case, the recursive calls evaluate in the same contour as the outermost call. Hence `last`'s argument `l` is bound in different contours for the entire evaluation of each recursive application of `last`.

The last part of Fig. 4 presents rules for construction, projection, and mutation of pairs. The rules merge values in the store only when the program's control flow makes this necessary.

For a given program, there are many flow graphs that satisfy relation $\mathcal{A}$. We want the least such graph. It is not difficult to prove that for any program $P$ the least flow graph exists and is unique. Nor is it difficult to implement an algorithm that constructs this flow graph [15].

$$
\begin{array}{rcl}
\mathcal{A}[\![c^{l}]\!]\rho\kappa & \Rightarrow & \mathit{AbstractValOf}(c) \in F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(}p\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathit{AbstractResultOf}(p) \in F\langle l, \kappa\rangle \text{ and for all } i = 1 \ldots n \ \ \mathcal{A}[\![e_i^{l_i}]\!]\rho\kappa \\
\mathcal{A}[\![\texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \langle l, \rho, \kappa\rangle_\lambda \in F\langle l, \kappa\rangle \\
\mathcal{A}[\![x^{l}_{[\lambda]}]\!]\rho\kappa & \Rightarrow & F\langle x, \rho(x)\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(call}\ e_0^{l_0}\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \text{for all } i = 0 \ldots n \ \ \mathcal{A}[\![e_i^{l_i}]\!]\rho\kappa \text{ and} \\
& & \text{for all } \langle l', \rho', \kappa'\rangle_\lambda \in F\langle l_0, \kappa\rangle \text{ where } \texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_b^{l_b}\texttt{)}^{l'} \in P \\
& & \quad F\langle l_j, \kappa\rangle \subseteq F\langle x_j, \kappa'\rangle \text{ for all } j = 1 \ldots n \text{ and} \\
& & \quad \mathcal{A}[\![e_b^{l_b}]\!]\rho'[x_1 \ldots x_n \mapsto \kappa']\kappa' \text{ and} \\
& & \quad F\langle l_b, \kappa'\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(begin}\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \text{for all } i = 1 \ldots n \ \ \mathcal{A}[\![e_i^{l_i}]\!]\rho\kappa \text{ and } F\langle l_n, \kappa\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(if}\ e_1^{l_1}\ e_2^{l_2}\ e_3^{l_3}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa \text{ and} \\
& & (\mathsf{true} \in F\langle l_1, \kappa\rangle \ \Rightarrow\ \mathcal{A}[\![e_2^{l_2}]\!]\rho\kappa \text{ and } F\langle l_2, \kappa\rangle \subseteq F\langle l, \kappa\rangle) \text{ and} \\
& & (\mathsf{false} \in F\langle l_1, \kappa\rangle \ \Rightarrow\ \mathcal{A}[\![e_3^{l_3}]\!]\rho\kappa \text{ and } F\langle l_3, \kappa\rangle \subseteq F\langle l, \kappa\rangle)
\end{array}
$$

---

$$
\begin{array}{rcl}
\mathcal{A}[\![\texttt{(let ((}x\ e_1^{l_1}\texttt{))}\ e_b^{l_b}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho(\kappa : l) \text{ and } F\langle l_1, \kappa : l\rangle \subseteq F\langle x, \kappa\rangle \text{ and} \\
& & \mathcal{A}[\![e_b^{l_b}]\!]\rho[x \mapsto \kappa]\kappa \text{ and } F\langle l_b, \kappa\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![x^{l}_{[l'']}]\!]\rho\kappa & \Rightarrow & \tau \in F\langle l, \kappa\rangle \text{ for all } \tau \in F\langle x, \rho(x)\rangle \text{ and} \\
& & \langle l', \kappa'\rangle_\pi \in F\langle l, \kappa\rangle \text{ for all } \langle l', \kappa'\rangle_\pi \in F\langle x, \rho(x)\rangle \text{ and} \\
& & \langle l', \rho', \kappa'[l''/l]\rangle_\lambda \in F\langle l, \kappa\rangle \text{ for all } \langle l', \rho', \kappa'\rangle_\lambda \in F\langle x, \rho(x)\rangle
\end{array}
$$

---

$$
\begin{array}{rcl}
\mathcal{A}[\![\texttt{(letrec ((}y\ f^{l_1}\texttt{))}\ e_b^{l_b}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \langle l_1, \rho, \kappa\rangle_\lambda \in F\langle y, \kappa\rangle \text{ and } \mathcal{A}[\![e_b^{l_b}]\!]\rho[y \mapsto \kappa]\kappa \text{ and } F\langle l_b, \kappa\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![y^{l}_{[l'']}]\!]\rho\kappa & \Rightarrow & \text{for all } \langle l', \rho', \kappa'\rangle_\lambda \in F\langle y, \rho(y)\rangle \\
& & \quad \langle l', \rho'[y \mapsto \kappa' : l''], \kappa' : l''\rangle_\lambda \in F\langle y, \kappa' : l''\rangle \text{ and} \\
& & \quad \langle l', \rho'[y \mapsto \kappa' : l''], \kappa' : l''\rangle_\lambda \in F\langle l, \kappa\rangle \\
\mathcal{A}[\![y^{l}_{[\mathrm{rec}]}]\!]\rho\kappa & \Rightarrow & F\langle y, \rho(y)\rangle \subseteq F\langle l, \kappa\rangle
\end{array}
$$

---

$$
\begin{array}{rcl}
\mathcal{A}[\![\texttt{(cons}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1]\!]\rho\kappa \text{ and } \mathcal{A}[\![e_2]\!]\rho\kappa \text{ and } \langle l, \kappa\rangle_\pi \in F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(car}\ e_1^{l_1}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1]\!]\rho\kappa \text{ and for all } \langle l', \kappa'\rangle_\pi \in F\langle l_1, \kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \in P \\
& & \quad F\langle l_a, \kappa'\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(cdr}\ e_1^{l_1}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1]\!]\rho\kappa \text{ and for all } \langle l', \kappa'\rangle_\pi \in F\langle l_1, \kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \in P \\
& & \quad F\langle l_b, \kappa'\rangle \subseteq F\langle l, \kappa\rangle \\
\mathcal{A}[\![\texttt{(set-car!}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1]\!]\rho\kappa \text{ and } \mathcal{A}[\![e_2]\!]\rho\kappa \text{ and } \mathsf{void} \in F\langle l, \kappa\rangle \text{ and} \\
& & \text{for all } \langle l', \kappa'\rangle_\pi \in F\langle l_1, \kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \in P \\
& & \quad F\langle l_2, \kappa\rangle \subseteq F\langle l_a, \kappa'\rangle \\
\mathcal{A}[\![\texttt{(set-cdr!}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^{l}]\!]\rho\kappa & \Rightarrow & \mathcal{A}[\![e_1]\!]\rho\kappa \text{ and } \mathcal{A}[\![e_2]\!]\rho\kappa \text{ and } \mathsf{void} \in F\langle l, \kappa\rangle \text{ and} \\
& & \text{for all } \langle l', \kappa'\rangle_\pi \in F\langle l_1, \kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \in P \\
& & \quad F\langle l_2, \kappa\rangle \subseteq F\langle l_b, \kappa'\rangle
\end{array}
$$

Figure 4: Relation $\mathcal{A}$ defining flow analysis $F$ of program $P$.

While our implementation of flow-directed inlining uses polymorphic splitting, adapting our inlining strategy to other polyvariant analyses is straightforward.

### 3.3 Identifying inlining sites

A polyvariant flow analysis determines abstract values for the function position of each call site. If the function position for a particular call site contains the same single abstract closure in every contour, it is possible to specialize and inline the procedure at that call site. This leads to the following inlining condition.

**Inlining Condition 1.** *If* $(\texttt{call}\ e_0^{l_0}\ e_1 \ldots e_p)^l$ *is a call site such that*

$$\bigcup_{\kappa' \in Contour} F\langle l_0, \kappa' \rangle \;=\; \{\langle m, \rho, \kappa \rangle_\lambda\}$$

*then* $(\lambda\ (x_1 \ldots x_n)\ e_b)^m$ *can be specialized to contour* $\kappa$ *and inlined at call site* $l$*, provided* $n = p$.

The proviso $n = p$ ensures that we do not eliminate applications which raise an error due to an argument count mismatch.[1] We describe specialization below. For now, let $(\lambda\ (x_1 \ldots x_n)\ e_b')$ be the specialized procedure. If condition 1 holds, we can inline the specialized procedure by replacing the call at $l$ with

$$(\texttt{call}\ (\lambda\ (w\ x_1 \ldots x_n)\ e_b')\ e_0\ e_1 \ldots e_p)$$

where $w$ is a fresh variable. Adding the extra argument $w$ ensures that our transformation preserves termination and side effect behavior that may result from evaluating $e_0$. Moreover, as we describe in Section 3.5, the extra argument will also be used to ensure proper access to the inlined procedure's free variables. Simple local transformations now suffice to eliminate the procedure call overhead.

### 3.4 Specializing inlined procedures

Inlining Condition 1 determines an initial set of call sites where inlining can be performed. For such a call site $l$, we can specialize the unique procedure $m$ called at $l$. Specialization involves $(i)$ pruning code within $m$ that is never reached from call site $l$, and $(ii)$ recursively inlining at call sites within $m$.

Pruning unreachable code is most important for conditional expressions. Let $\langle m, \rho, \kappa \rangle_\lambda$ be the unique closure called at $l$. To prune unreachable code at a conditional expression $(\texttt{if}\ e_1^{l_1}\ e_2\ e_3)$ within $m$, we use contour $\kappa$ to find an upper bound $F\langle l_1, \kappa \rangle$ on the set of values that the test expression $e_1$ can yield when $m$ is called from $l$. If $F\langle l_1, \kappa \rangle$ does not include true, the consequent $e_2$ can be pruned. If $F\langle l_1, \kappa \rangle$ does not include false, the alternative $e_3$ can be pruned. If $F\langle l_1, \kappa \rangle$ includes neither true nor false, both $e_2$ and $e_3$ can be pruned.

[1] If the arities do not match, we leave the application alone. We could also issue a warning, raise a compile-time error, or insert a call to an error handler.

Pruning is also possible for most other expression forms. For example, assuming subexpressions are evaluated from left to right in an application, all subexpressions to the right of a divergent subexpression whose abstract value is empty can be pruned. To simplify the presentation, we have included pruning only for conditionals. Our implemented algorithms for flow analysis and inlining include pruning wherever possible.

The second kind of specialization is recursive inlining at call sites within a procedure that is being inlined. Inlining Condition 1 identifies call sites where inlining is possible within the unspecialized procedures of the original program. To identify candidates for inlining within a specialized procedure, we employ a similar condition that takes into account the specialization contour.

**Inlining Condition 2.** *Let* $\langle m, \rho, \kappa \rangle_\lambda$ *be an abstract closure that is being inlined at call site* $l$ *and specialized to contour* $\kappa$*. A call site* $(\texttt{call}\ \hat{e}_0^{\hat{l}_0}\ \hat{e}_1 \ldots \hat{e}_{\hat{p}})^{\hat{l}}$ *within* $m$ *can be inlined in the specialized version of* $m$ *if*

$$F\langle \hat{l}_0, \kappa \rangle \;=\; \{\langle \hat{m}, \hat{\rho}, \hat{\kappa} \rangle_\lambda\}$$

*and closure* $\hat{m}$ *has arity* $\hat{p}$.

That is, we can inline $\hat{m}$ at call site $\hat{l}$ within the specialized version of $m$ if the flow analysis identifies a unique abstract closure at program point $\langle \hat{l}_0, \kappa \rangle$. Support for recursive inlining follows naturally from a polyvariant flow analysis.

### 3.5 Inlining procedures with free variables

Our discussion of inlining has so far neglected an important issue in a language with higher-order procedures and lexical scoping. How do inlined procedures gain access to the values of their free variables?

To answer this question, we define a target language that adds an ordered list of free variables $[z_1 \ldots z_m]$ to each $\lambda$-expression:

$$(\lambda\ [z_1 \ldots z_m]\ (x_1 \ldots x_n)\ e_b)$$

The target language also includes an expression form

$$(\texttt{cl-ref}\ e_1\ n)$$

to access the $n^{th}$ element of a closure's free variable list. The `cl-ref` operator (short for closure-reference) requires $e_1$ to evaluate to a closure.

When a procedure with free variables is inlined, all references to these variables are replaced with `cl-ref` operations. To illustrate, suppose that at $(\texttt{call}\ e_0\ e_1 \ldots e_n)$ we inline $(\lambda\ (x_1 \ldots x_n)\ e_b)$ which has $\{z_1 \ldots z_m\}$ free. The original $\lambda$-expression is replaced with

$$(\lambda\ [z_1 \ldots z_m]\ (x_1 \ldots x_n)\ e_b).$$

We replace the call with

$$(\texttt{call}\ (\lambda\ (w\ x_1 \ldots x_n)\ e_b')\ e_0\ e_1 \ldots e_n)$$

where $w$ is a new variable and $e'_b$ is a specialized copy of $e_b$ with references to $z_i$ replaced by `(cl-ref` $w$ $i$`)`.

The `cl-ref` operation is easy to implement in a compiler that uses flat closures [2]. A closure record consists of a code pointer and the values of the variables in the free variable list for that procedure. A compiler that uses linked closures may require additional information in the `cl-ref` operation to indicate the forms of the linkage structures.

### 3.6 An algorithm

Fig. 5 collects the above observations into a complete inlining algorithm.

Given a flow analysis $F$ for a program $P$, the transformation $\mathcal{I}[\![e]\!]\kappa\mu$ takes a subexpression $e$ of $P$, a contour $\kappa$, and a loop map $\mu$, and produces an equivalent inlined and specialized expression. The initial contour for $P$ is the special contour ? and the initial loop map is the empty map. The contour ? denotes the union of all possible contours, as in Inlining Condition 1. In other words,

$$F\langle l, ?\rangle \;=\; \bigcup_{\kappa \in Contour} F\langle l, \kappa\rangle.$$

The contour parameter $\kappa$ of the algorithm selects a particular context in which to specialize subexpressions (see the condition $F\langle l_0, \kappa\rangle = \{\langle l', \rho', \kappa'\rangle_\lambda\}$ in the rule for applications, and the various conditions in the rule for `if`-expressions). Since the original copy of a $\lambda$-expression must not be specialized to any specific contour, the transformation of its body is performed in the special contour ?.

To prevent uncontrolled unwinding of recursive calls, the loop map $\mu$ maps label-contour pairs to variables. If, while inlining procedure $l'$ in contour $\kappa'$, the algorithm encounters another call site for $l'$ in contour $\kappa'$ (see the second case of the rule for applications), it constructs a loop by binding a fresh variable to the inlined procedure at the initial call site with `letrec`. In fact, the algorithm introduces a `letrec` for every inlined procedure, whether recursive or not, but subsequent local simplification removes any unnecessary bindings.

This method of controlling unwinding avoids unrolling loops. Loop unrolling can be a valuable optimization and would be easy to include in this framework. We have intentionally avoided unrolling loops in order to isolate the benefits of inlining.

### 3.7 Making inlining decisions

Inlining every call site that is a candidate for inlining is impractical as it can lead to exponential code growth. To limit inlining to those sites where it is likely to be most profitable, Fig. 5 requires the predicate *Inline?* to hold for the procedure body when specialized for the contour and loop map under consideration. This predicate estimate the sizes of the generated code for the inlined procedure at a particular call site, and limits inlining to cases where the generated code is smaller than some threshold size[2]. In the next section, we present performance results for various inlining thresholds.

### 3.8 Local Simplification

After inlining, we perform some simple optimizations that are based purely on local syntactic criteria. These include $\beta_v$ reductions that do not significantly increase code size, simple constant propagation and constant folding, eliminating unused bindings, and discarding purely functional expressions whose result is never used.

## 4 Performance

We have implemented a source-to-source flow-directed inlining optimization for the full R$^4$RS Scheme language [7]. Given a Scheme program and an inline threshold $T$, our optimizer inlines procedures whose specialized size is estimated to be less than $T$. We use Chez Scheme [11] to compile the optimized programs to native code.

Without access to Chez's internal data representations, it is impossible to implement `cl-ref` with the same efficiency as a variable reference. Indeed, using a faithful implementation of the algorithm from the previous section, we found the premium for accessing variables via `cl-ref` masked the benefit of inlining. Therefore, the results presented in this section reflect an inlining strategy in which the *Inline?* predicate requires inlined procedures be closed up to top-level variables. Under this constraint, the inlining algorithm never generates `cl-ref` operations. The algorithm will still allow a procedure $P$ that has a free variable $x$ to be inlined at call site $C$ if ($i$) $x$ occurs in a conditional branch which can be eliminated in the specialized copy of $P$ at $C$, or ($ii$) $x$ refers to a procedure that that will be inlined. In either case, a free variable in the source does not appear in the specialized version. Surprisingly, this inlining strategy leads to consistent performance improvements for all programs we have tested. For many of the benchmarks in our test suite, the performance improvements are significant. We would expect even greater improvements with an efficient implementation of `cl-ref` since this would enable inlining open procedures.

We applied our optimizer to the benchmarks listed in Table 1. These benchmarks are characteristic of a wide range of Scheme programs, and none were written with knowledge of the inlining strategy used.

The program Lattice enumerates the lattice of maps between two lattices, and is mostly first-order. Boyer is a term-rewriting theorem prover. Graphs counts the number of directed graphs with a distinguished root and $k$ vertices, each having out-degree at most 2. It makes extensive use of higher-order procedures, and is written in a continuation-passing style. Given an $n \times n$ random matrix $M$ with {+1, -1} entries, Matrix tests whether $M$ is maximal among all

[2]The algorithm in Fig. 5 requires constructing the specialized version of the procedure body before deciding if it should be inlined. Our implementation uses an equivalent but more efficient algorithm that estimates the size of the specialized procedure without actually constructing it.

$$\mathcal{I}[\![c]\!]\kappa\mu \;=\; c$$

$$\mathcal{I}[\![\texttt{(}p\; e_1\ldots e_n\texttt{)}]\!]\kappa\mu \;=\; \texttt{(}p\; \mathcal{I}[\![e_1]\!]\kappa\mu \ldots \mathcal{I}[\![e_n]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(}\lambda\;\texttt{(}x_1\ldots x_n\texttt{)}\; e_b\texttt{)}]\!]\kappa\mu \;=\; \texttt{(}\lambda\;[z_1\ldots z_m]\;\texttt{(}x_1\ldots x_n\texttt{)}\;\mathcal{I}[\![e_b]\!]?\mu\texttt{)} \quad \text{where} \quad [z_1\ldots z_m] = FV\,\texttt{((}\lambda\;\texttt{(}x_1\ldots x_n\texttt{)}\; e_b\texttt{))}$$

$$\mathcal{I}[\![x]\!]\kappa\mu \;=\; x$$

$$\mathcal{I}[\![\texttt{(call}\; e_0^{l_0}\; e_1\ldots e_n\texttt{)}]\!]\kappa\mu \;=\; \begin{cases}
\begin{array}{l}
\texttt{(letrec} \\
\quad \texttt{((}y\;\texttt{(}\lambda\;\texttt{(}w\;x_1\ldots x_n\texttt{)} \\
\qquad\qquad \texttt{(let ((}z_1\;\texttt{(cl-ref}\;w\;1\texttt{)))} \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \texttt{((}z_m\;\texttt{(cl-ref}\;w\;m\texttt{)))} \\
\qquad\qquad\quad \mathcal{I}[\![e_b]\!]\kappa'\mu'\texttt{))))} \\
\quad \texttt{(call}\;y\;\mathcal{I}[\![e_0]\!]\kappa\mu\;\mathcal{I}[\![e_1]\!]\kappa\mu\ldots\mathcal{I}[\![e_n]\!]\kappa\mu\texttt{))}
\end{array} &
\begin{array}{ll}
\text{if} & e_0 \neq \texttt{(}\lambda\ldots\texttt{)} \\
\text{and} & F\langle l_0,\kappa\rangle = \{\langle l',\rho',\kappa'\rangle_\lambda\} \\
\text{and} & \langle l',\kappa'\rangle \notin \mathrm{Dom}(\mu) \\
\text{and} & \mathit{Inline?}\,(\mathcal{I}[\![e_b]\!]\kappa'\mu') \\
\text{where} & \texttt{(}\lambda\;\texttt{(}x_1\ldots x_n\texttt{)}\;e_b\texttt{)}^{l'} \in P \\
 & y, w \text{ fresh} \\
 & \mu' = \mu[\langle l',\kappa'\rangle \mapsto y] \\
 & [z_1\ldots z_m] = FV\,\texttt{((}\lambda\;\texttt{(}x_1\ldots x_n\texttt{)}\;e_b\texttt{))}
\end{array} \\[2ex]
\texttt{(call}\;y\;\mathcal{I}[\![e_0]\!]\kappa\mu\;\mathcal{I}[\![e_1]\!]\kappa\mu\ldots\mathcal{I}[\![e_n]\!]\kappa\mu\texttt{)} &
\begin{array}{ll}
\text{if} & F\langle l_0,\kappa\rangle = \{\langle l',\rho',\kappa'\rangle_\lambda\} \\
\text{and} & \mu\langle l',\kappa'\rangle = y
\end{array} \\[2ex]
\texttt{(call}\;\mathcal{I}[\![e_0]\!]\kappa\mu\;\mathcal{I}[\![e_1]\!]\kappa\mu\ldots\mathcal{I}[\![e_n]\!]\kappa\mu\texttt{)} & \text{otherwise}
\end{cases}$$

$$\mathcal{I}[\![\texttt{(begin}\; e_1\ldots e_n\texttt{)}]\!]\kappa\mu \;=\; \texttt{(begin}\;\mathcal{I}[\![e_1]\!]\kappa\mu\ldots\mathcal{I}[\![e_n]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(if}\; e_1^{l_1}\; e_2\; e_3\texttt{)}]\!]\kappa\mu \;=\; \begin{cases}
\texttt{(if}\;\mathcal{I}[\![e_1]\!]\kappa\mu\;\mathcal{I}[\![e_2]\!]\kappa\mu\;\mathcal{I}[\![e_3]\!]\kappa\mu\texttt{)} & \text{if } \{\mathsf{true},\mathsf{false}\} \subseteq F\langle l_1,\kappa\rangle \\
\texttt{(begin}\;\mathcal{I}[\![e_1]\!]\kappa\mu\;\mathcal{I}[\![e_2]\!]\kappa\mu\texttt{)} & \text{if } \{\mathsf{true}\} \subseteq F\langle l_1,\kappa\rangle \\
\texttt{(begin}\;\mathcal{I}[\![e_1]\!]\kappa\mu\;\mathcal{I}[\![e_3]\!]\kappa\mu\texttt{)} & \text{if } \{\mathsf{false}\} \subseteq F\langle l_1,\kappa\rangle \\
\mathcal{I}[\![e_1]\!]\kappa\mu & \text{otherwise}
\end{cases}$$

$$\mathcal{I}[\![\texttt{(let ((}x\; e_1\texttt{))}\; e_2\texttt{)}^l]\!]\kappa\mu \;=\; \texttt{(let ((}x\;\mathcal{I}[\![e_1]\!](\kappa:l)\mu\texttt{))}\;\mathcal{I}[\![e_2]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(letrec ((}y\; e_1^{l_1}\texttt{))}\; e_2\texttt{)}^l]\!]\kappa\mu \;=\; \texttt{(letrec ((}y\;\mathcal{I}[\![e_1]\!]\kappa\mu'\texttt{))}\;e_2\texttt{)} \quad \text{where} \quad \mu' = \mu[\langle l_1,\kappa\rangle \mapsto y]$$

$$\mathcal{I}[\![\texttt{(cons}\; e_1\; e_2\texttt{)}]\!]\kappa\mu \;=\; \texttt{(cons}\;\mathcal{I}[\![e_1]\!]\kappa\mu\;\mathcal{I}[\![e_2]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(car}\; e_1\texttt{)}]\!]\kappa\mu \;=\; \texttt{(car}\;\mathcal{I}[\![e_1]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(cdr}\; e_1\texttt{)}]\!]\kappa\mu \;=\; \texttt{(cdr}\;\mathcal{I}[\![e_1]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(set-car!}\; e_1\; e_2\texttt{)}]\!]\kappa\mu \;=\; \texttt{(set-car!}\;\mathcal{I}[\![e_1]\!]\kappa\mu\;\mathcal{I}[\![e_2]\!]\kappa\mu\texttt{)}$$

$$\mathcal{I}[\![\texttt{(set-cdr!}\; e_1\; e_2\texttt{)}]\!]\kappa\mu \;=\; \texttt{(set-cdr!}\;\mathcal{I}[\![e_1]\!]\kappa\mu\;\mathcal{I}[\![e_2]\!]\kappa\mu\texttt{)}$$

Figure 5: A flow-directed inlining algorithm

| Program | Source Lines | Analysis Times (in secs.) | Ratio of object code size to original object code size for various inlining thresholds | | | | |
|---|---|---|---|---|---|---|---|
| | | | 50 | 100 | 200 | 500 | 1000 |
| Lattice | 207 | .2 | 1.02 | 1.02 | 1.06 | 1.69 | 3.09 |
| Boyer | 381 | 1.4 | .95 | 1.05 | 1.10 | .97 | .95 |
| Graphs | 489 | .3 | .79 | .87 | .83 | .81 | .80 |
| Matrix | 527 | .4 | .93 | .97 | .96 | 1.13 | 1.21 |
| Maze | 568 | .5 | .94 | .98 | .96 | .93 | .89 |
| Splay | 934 | 4.0 | .94 | .94 | .95 | .95 | .95 |
| Nbody | 964 | 2.6 | 1.25 | 1.53 | 1.66 | 2.29 | 2.62 |
| Dynamic | 2046 | 110.3 | 1.52 | 1.65 | 2.36 | 2.48 | 2.48 |

Table 1: Benchmark programs.

matrices of the same dimension obtainable by simple reordering of rows and columns, and negation of any subset of rows and columns. Like Graphs, this program is written in continuation-passing style. Maze generates a random maze and computes a path through it using a union-find algorithm. It makes extensive use of records [25] and uses Scheme's `call-with-current-continuation` operator, but is primarily a first-order program. Splay is an implementation of splay trees. It makes extensive use of higher-order procedures and pattern matching macros [25]. N-Body is a Scheme implementation [26] of the Greengard multipole algorithm [12] for computing gravitational forces on point-masses distributed uniformly in a cube. Dynamic is an implementation of a tagging optimization algorithm [14] for Scheme. It is primarily a first-order program, but has complex control-flow, and many deeply-nested conditional expressions.

The first column of Table 1 indicates the size of each program in lines of source code after prepending necessary library procedures, removing comments, expanding macros, and performing local simplifications as described in Section 3.8. The next column indicates the time taken by the flow analysis to analyze the program. The remaining columns indicate code sizes after inlining for various size thresholds. For example, the call `(map car m)` from Figs. 1, 2, and 3 is inlined at thresholds above 60.

Code size ratios less than one indicate that the inlined program is *smaller* than the original. For example, at threshold 1000, the inlined version of Graphs is still 20% smaller than the original program. For most benchmarks, object code size grows quite slowly as the inlining threshold increases. There are two reasons for this. First, Scheme programs typically consist of many small procedures, so higher inlining thresholds are not likely to expose many more opportunities for inlining. Second, even when relatively large procedures become candidates for inlining, specialization and local simplification significantly reduce the size of the inlined copy.

Fig. 6 presents execution times for these benchmarks under different inlining thresholds. These times were gathered on an SGI 150 MHz MIPS R4400 workstation. The execution time for each inlined program is normalized to the execution time of the original program after local simplification (i.e., threshold 0) when run under Chez Scheme 5.0a at its highest optimization level. At this level, the compiler will inline calls to primitives and small built-in procedures. Furthermore, the generated code is unsafe: inlined primitives do not perform any type or bounds checking. Thus, performance improvements shown in these graphs measure gains *above* what an aggressive optimizing compiler would ordinarily achieve.

The graphs in Fig. 6 separate execution time into mutator time and collector time. The axis on the left of each graph measures normalized total execution time, which is indicated by the total heights of the bars. The axis on the right of each graph measures mutator execution time, which is indicated by the tops of the dark bars. Mutator time is the time taken by the application when garbage collection is discounted. The light part of each bar indicates time spent performing garbage collection. For example, at size threshold 200, `Maze` shows a 40% performance improvement in total execution time. Discounting garbage collection time, which inlining cannot (directly) affect, mutator time has been improved by nearly 60%.

For all of the programs we have tested, flow-directed inlining improves execution times. Furthermore, as size thresholds are increased, performance either increases or remains roughly constant. This indicates that our size metric is benign, and inlining decisions rarely impact performance negatively. On average, the best performance occurs at size thresholds between 200 and 500. At smaller thresholds, our implementation inlines small library procedures like `cadr` or a specialized version of `map`, but is unlikely to inline non-trivial user-defined procedures. At higher thresholds, more user-defined procedures are inlined, and more opportunities for specialization of these procedures become apparent. Few additional procedures are inlined at thresholds above 500; thus, we see little further performance improvement. Both Nbody and Matrix show performance improvement of only roughly 10%. A relatively large percentage of these program's execution time is spent in tight loops, and not in calls to out-of-line procedures. Hence, inlining exposes few optimization opportunities. On the other hand, Maze and Boyer both make many out-of-line calls relative to the overall computation. Inlining these calls removes significant overhead in these benchmarks. The same conclusion holds to a slightly lesser extent for Graphs and Lattice.

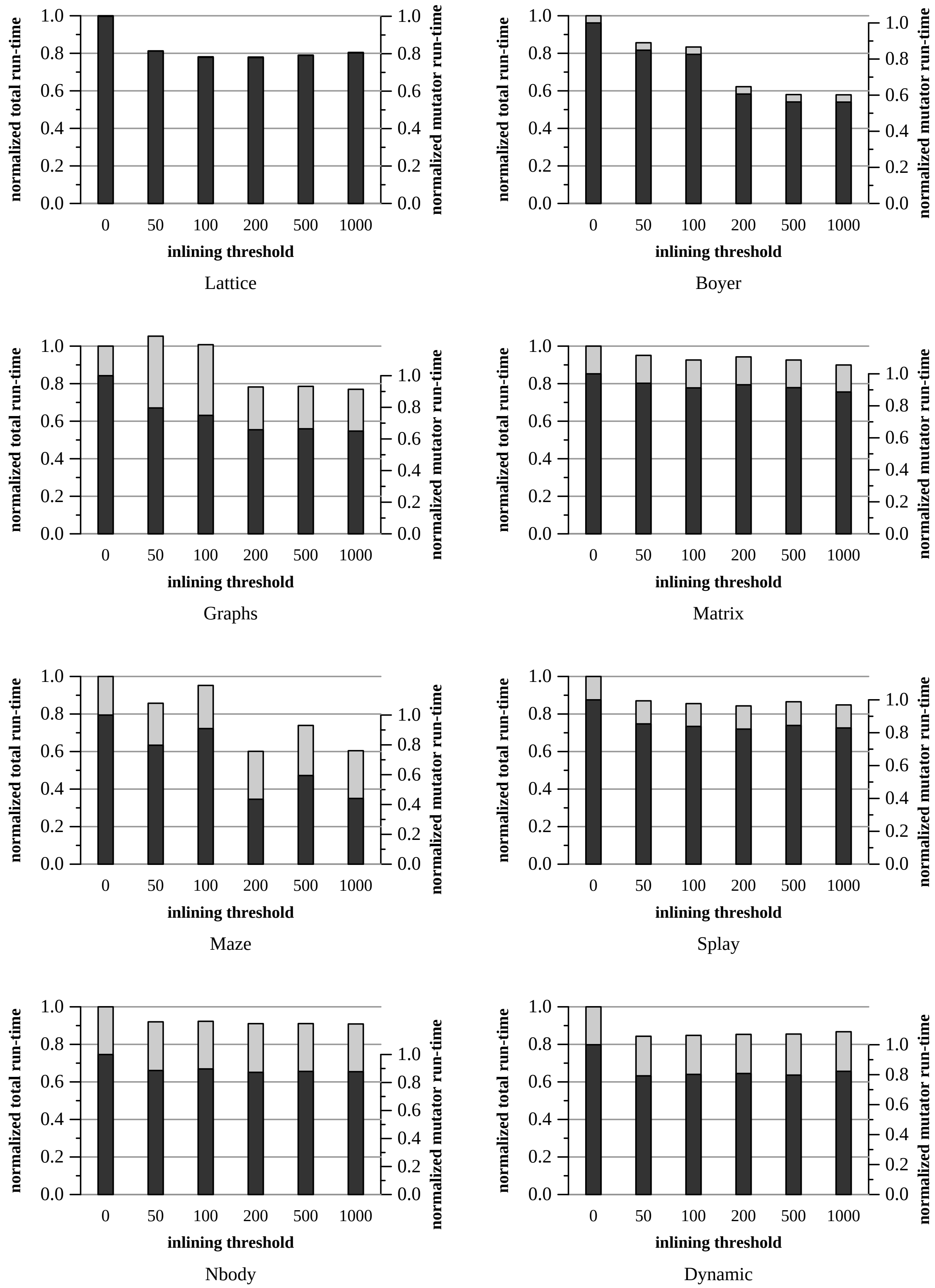


Figure 6: Execution times under different inlining thresholds. Threshold 0 indicates no inlining. The dark section of each bar indiates mutator time; the light section indicates collector time.

There is little change in collection time for most of the benchmarks in Fig. 6. This implies that inlining does not generally alter the amount of data allocated or the lifetime of objects. Indeed, for most benchmarks, overall allocation remains roughly constant. However, for Graphs, there is a dramatic increase in collection time (and allocation) at thresholds 50 and 100. We have three conjectures for this behavior. First, inlining may cause more closures to be allocated. When a procedure is inlined, any local procedures within it are copied, and hence will construct new closures. Also, recall that a local `letrec` is introduced for every inlined procedure. If some of these procedures are not compiled as tail-recursive loops, closures will be allocated for them as well. Second, inlining may increase closure allocation costs. Chez Scheme uses a flat closure representation. As a result, inlining will increase the cost of allocating closures by introducing additional free variables into procedures. Third, with Chez Scheme's closure representation, introducing additional free variables into procedures may cause data to be retained longer than it would be otherwise. We suspect that using Shao and Appel's "safe for space complexity" rule [21] would ameliorate some of these problems. At higher thresholds, the collection times for Graphs drop. More opportunities for simplification are exposed which presumably eliminate the troublesome closures.

## 5 Related work

Two independent topics related to our work have been investigated by other researchers—flow analysis and inlining.

### 5.1 Flow analysis

The literature on flow analysis for functional and object-oriented languages is vast [13, 18, 22]. We limit our discussion of flow analysis to systems with polyvariance, which we believe is key to building optimizations that scale with program size.

A well-known approach to polyvariance proposed by Shivers [22] is to distinguish different procedure calls by call site. In a framework similar to ours, contours are finite strings of call site labels. An $N$-CFA analysis uses the most recent $N$ call sites to distinguish different invocations of a procedure. While call-string based analyses are highly precise, they also appear to be quite expensive to compute [15]. In general, polymorphic splitting provides as much, if not more, accuracy than polyvariant call-string analyses, but at significantly lower cost.

### 5.2 Inlining

Cooper, Hall, and Torczon [8, 9] studied the effects of inlining for Fortran programs. They determined inlining sites by hand, and were able to eliminate the majority of dynamically executed procedure calls from their benchmark programs. Furthermore, they found that the opportunities revealed by inlining mitigated object code growth—object code grew by only 10% when inlining doubled the size of the source code. However, they were unable to demonstrate consistent speedups for the inlined programs. They speculate that inlining did enable useful optimizations, but the speedup from these optimizations was masked by effects like increased register pressure, and pessimistic assumptions made by the compiler about aliasing which lead to poor instruction scheduling.

Several differences between Scheme and Fortran may explain our better results. First, our inlined programs have smaller procedures than the inlined Fortran programs, so register pressure is less significant. Second, Scheme compilers cannot optimistically assume that parameters to procedure calls do not alias. Hence inlining introduces no additional requirements on instruction scheduling. Finally, and most importantly, as Scheme programs tend to have more procedure calls, there is more procedure call overhead to remove.

Chang *et al.* [6] constructed an automatic inlining optimization for C programs. As with Cooper *et al.*'s experiments with Fortran, Chang *et al.* were able to eliminate the majority of procedure calls while increasing object code size by only 16%. They obtained consistent performance improvements that averaged about 10%.

Much attention has been devoted to reducing the overhead of dynamically bound method dispatches in object-oriented languages. Recently, static analyses have been applied to either inline most dispatches or replace them with direct procedure calls [1, 10, 19, 20]. But with the exception of Cecil where dispatching consumes a large fraction of execution time [10], speedups are minimal at best. We suspect that these systems fail to improve execution times because their compilers do not use the larger contexts constructed by inlining to enable further optimization.

Functional language compilers like Standard ML of New Jersey [2] use syntactic heuristics to guide inlining decisions. A call site is a candidate for inlining if the called procedure is syntactically evident; typically, the function expression is a variable that is bound to a $\lambda$-expression by `let`. Inlining is intertwined with other syntactic optimizations, so additional inlining candidates can arise as optimization proceeds. In contrast, flow-directed inlining separates inlining decisions from post-inlining simplifications. Heuristics that estimate object code size, similar to ours, are used to control code growth. But these heuristics are necessarily more conservative, as they cannot take into account future simplifications that may take place if inlining is performed. Furthermore, because candidates for inlining are identified syntactically, procedures used in a higher-order manner will not be inlined unless other simplifications reduce the higher-order uses to syntactic ones. We believe that for programs which make heavy use of data and procedural abstraction, flow-directed inlining is likely to identify more profitable sites for inlining.

## 6 Future Work

We have used the same flow analysis described in this paper to eliminate run-time checks from Scheme programs [15]. We plan to combine our inlining and run-time check optimizations along with other optimizations that use the same flow information. This combination should yield significant performance improvements without compromising safety.

**Acknowledgments**

We are grateful to Henry Cejtin, Richard Kelsey and the PLDI referees for many helpful suggestions.

**References**

[1] AGESEN, O., AND HOLZLE, U. Type Feedback vs. Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1995), pp. 91–107.

[2] APPEL, A. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] BARENDREGT, H. *The Lambda Calculus.* North-Holland, Amsterdam, 1981.

[4] BULYONKOV, M. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica 21* (1984), 473–484.

[5] CHAMBERS, C., AND UNGAR, D. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation* (New York, June 1989), ACM, pp. 146–160.

[6] CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., AND HWU, W.-M. W. Profile-Guided Automatic Inline Expansion for C Programs. *Software Practice and Experience* (May 1992), 349–369.

[7] CLINGER, W., AND REES, EDITORS, J. Revised$^4$ Report on the Algorithmic Language Scheme. *ACM Lisp Pointers 4*, 3 (July 1991).

[8] COOPER, K., HALL, M., AND TORCZON, L. An Experiment with Inline Substitution. *Software: Practice and Experience 21*, 6 (1991), 581–601.

[9] COOPER, K., HALL, M., AND TORCZON, L. Unexpected Side Effects of Inline Substitution: A Case Study. *ACM Letters on Programming Languages and Systems 1*, 1 (1992), 22–32.

[10] DEAN, J., CHAMBERS, C., AND GROVE, D. Selective Specialization for Object-Oriented Languages. In *ACM Conference on Programming Language Design and Implementation* (1995), pp. 93–102.

[11] DYBVIG, K. *The Scheme Programming Language.* Prentice-Hall, Inc., 1987.

[12] GREENGARD, L. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, 1987.

[13] HEINTZE, N. Set-Based Analysis of ML Programs. In *ACM Symposium on Lisp and Functional Programming* (1994), pp. 306–317.

[14] HENGLEIN, F. Global Tagging Optimization by Type Inference. In *ACM Symposium on Lisp and Functional Programming* (1992), pp. 205–215.

[15] JAGANNATHAN, S., AND WRIGHT, A. Effective Flow-Analysis for Avoiding Runtime Checks. In *Second Interntational Symposium on Static Analysis* (1995), pp. 207–225. Springer-Verlag LNCS 983.

[16] JONES, N., AND MUCHNICK, S. Flow Analysis and Optimization of Lisp-like Structures. In *$6^{th}$ ACM Symposium on Principles of Programming Languages* (January 1979), pp. 244–256.

[17] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* MIT Press, 1990.

[18] PALSBERG, J., AND SWARTZBACH, M. I. Object-Oriented Type Inference. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1991), pp. 146–161.

[19] PANDE, H. D., AND RYDER, B. G. Static Type Determination for C++. In *Usenix C++ Conference Proceedings* (1994), pp. 85–97.

[20] PLEVYAK, J., AND CHIEN, A. Precise Concrete Type Inference for Object-Oriented Languages. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1994), pp. 324–340.

[21] SHAO, Z., AND APPEL, A. Space-Efficient Closure Representations. In *ACM Symposium on Lisp and Functional Programming* (1994), pp. 150–161.

[22] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[23] SHIVERS, O. The Semantics of Scheme Control-Flow Analysis. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1991), pp. 190–198.

[24] SUN MICROSYSTEMS, INC. *The Java Language Specification.* Mountain View, Calif., 1995.

[25] WRIGHT, A. K., AND DUBA, B. F. Pattern Matching for Scheme. Unpublished document available from http://www.neci.nj.nec.com/homepages/wright.html, 1993.

[26] ZHAO, F. An $O(N)$ Algorithm for Three-Dimensional N-Body Simulations. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.