The Effectiveness of Flow Analysis for Inlining[*]

J. Michael Ashley
University of Kansas
Snow Hall 415
Lawrence, Kansas 66045
`jashley@eecs.ukans.edu`

Abstract

An interprocedural flow analysis can justify inlining in higher-order languages. In principle, more inlining can be performed as analysis accuracy improves. This paper compares four flow analyses to determine how effectively they justify inlining in practice. The paper makes two contributions. First, the relative merits of the flow analyses are measured with all other variables held constant. The four analyses include two monovariant and two polyvariant analyses that cover a wide range of the accuracy/cost spectrum. Our measurements show that the effectiveness of the inliner improves slightly as analysis accuracy improves, but the improvement is offset by the compile-time cost of the accurate analyses. The second contribution is an improvement to the previously reported inlining algorithm used in our experiments. The improvement causes flow information provided by a polyvariant analysis to be selectively merged. By merging flow information depending on the inlining context, the algorithm is able to expose additional opportunities for inlining. This merging technique can be used in any program transformer justified by a polyvariant flow analysis. The revised algorithm is fully implemented in a production Scheme compiler.

1 Introduction

A flow analysis is a tool that can assist program transformers in assessing the legality of transformations. An interprocedural flow analysis computes flow information across procedure boundaries. While straightforward for first-order languages, interprocedural flow analysis is more complex for higher-order languages. Since procedures are data, both control-flow and data-flow information must be computed simultaneously. Work in the last several years, however, has made flow analyses a more viable tool for program transformations [5, 12, 26], including compiler optimizations [2, 4, 17, 18, 23].

A flow analysis is either monovariant or polyvariant. A monovariant analysis uses a single abstract evaluation context to model the evaluation of an expression. Thus, a single abstract value is associated with each expression in the program. A polyvariant analysis is more accurate, modeling the evaluation of an expression using multiple abstract evaluation contexts. Multiple values may be associated with each expression, one value for each abstract evaluation context. Program transformers can use the additional information by cloning and specializing an expression to a particular evaluation context using the more accurate flow information.

Compilers can use flow information to justify optimizations. One such optimization is inlining. Inlining replaces a procedure call's operator with the code of the called procedure. Besides eliminating calls, a compiler may be able to take advantage of inlining to improve intraprocedural optimizations such as register allocation, constant folding, and copy propagation. In order to legally inline a procedure at a call site, the compiler must be able to establish that only one procedure will arrive at the call site and that it is legal to substitute the procedure's text at that call site. Furthermore, in order for inlining to be effective the compiler must also be able to determine if eliminating the call would speed up the program. Issues involved with this latter decision include register usage and the code size of the inlined program.

This paper measures the effectiveness of flow analysis for inlining using an improved version of Jagannathan and Wright's inlining algorithm [18]. The algorithm uses dataflow information to establish the legality of inlining and to guide decisions on whether or not calls should be inlined. It is improved so that when flow information must be merged, the merging is done selectively to retain as many inlining candidates as possible. This modification has no effect when a monovariant analysis is used since a monovariant analysis merges all information as the analysis is run. It can, however, increase in-

[*]This work supported by NSF grant CCR-9623753.

lining opportunities when a polyvariant analysis is used.

Since the algorithm is dependent on the quality of flow information, the implementation is evaluated using four flow analyses of differing levels of accuracy. One is a fast, monovariant analysis that makes at most two passes over a program. The second is equivalent to 0CFA [24]. The third is a polyvariant analysis more accurate than 0CFA but less accurate than 1CFA [24]. The fourth analysis is similar to 1CFA. All four analyses are instantiations of a single, parameterized flow analysis framework [2].

The four analyses are evaluated using a variety of benchmarks. The polyvariant analyses are able to justify somewhat more inlining than the monovariant analyses but at a significantly greater cost. Code growth is also larger when a polyvariant analysis is used, although this may not hold in a compiler that is better able to optimize larger intraprocedural contexts created by inlining.

The rest of the paper is outlined as follows. Section 2 describes the assumptions made about the flow analysis and in particular specifies assumptions made about flow values and contours. Section 3 describes the inlining algorithm, and Section 4 describes its implementation. Section 5 evaluates the algorithm using the four four flow analyses described above. Section 6 provides related work, and Section 7 gives our conclusions.

2 Flow analysis assumptions

Figure 1 describes the source and target language of the inlining algorithm. It is a language of labeled expressions that corresponds to the relevant subset of Scheme, which we take as a representative higher-order language. Applications are further decorated with keys, which may be used by a polyvariant flow analysis to guide splitting decisions. The three sets *ExpLabels*, *ProcLabels*, and *Keys* are all assumed to be finite and specific to the particular program being transformed. For any program e we assume a function $P : \mathit{Labels} \rightarrow E$ such that if e_0^l is a subexpression of e, $P(l) = e_0^l$.

Given a program e for inlining, the inlining transformation assumes an off-line flow analysis has analyzed the program and produced a function F that describes the fruits of the analysis.

$$\begin{array}{rcl} F & : & \mathit{Labels} \times \widehat{\mathit{Contours}} \rightarrow \mathcal{P}(\widehat{\mathit{Values}}) \\ \kappa \in \widehat{\mathit{Contours}} & = & \mathit{Keys}^* \\ \widehat{\mathit{Values}} & = & \{\mathit{true}, \mathit{false}\} \cup \\ & & (\mathit{ProcLabels} \times \widehat{\mathit{Contours}}) \end{array}$$

The function F takes a label and a *contour* describing an analysis context. It returns a set of abstract values. A contour is a sequence of keys, and an abstract value is a finite approximation of a possibly infinite set of actual values. Although Scheme supports a rich set of data types, and flow analyses typically reflect that in a rich domain of abstract values, the only values that are important to the inlining transformation are booleans and abstract closures. Furthermore, while many analyses will produce abstract closures containing potentially useful information such as abstract environments, the inlining algorithm only needs the label and the contour. The label identifies the lambda abstraction from which the closure was derived, and the contour describes the analysis context in which the abstract closure was created. The function F can therefore be viewed as a projection function that maps relatively rich abstract values to the simpler abstract values needed by the inliner. Many flow analyses, both monovariant [2, 4, 6, 15, 16, 23, 25] and polyvariant [3, 11, 17] produce results that can be cast into the form of function F.

Polyvariant analyses use contours to segregate abstract values and improve the accuracy of the analysis. Most analyses create a new contour at each variable binding point and use it while analyzing the code in the lexical contour established by the binding point. How contours are created, however, differs from analysis to analysis, and the choice dictates an analysis' splitting strategy. There are two popular strategies. The first uses the variables' bindings to guide contour creation [11]. The second uses the calling context in which the bindings are established. The second strategy is often referred to as the *call-string* approach [15, 17, 25].

Consider the following program as an example.

```
(let ((f (λ (x) x)))
  (begin
    (f 2)
    (f 5)))
```

In this program the procedure bound to `f` is called at two points. Depending on the splitting strategy, a polyvariant analysis might create either one or two contours in which to analyze the procedure's body. If the strategy is based on the types of the bindings, only one contour would be created since the procedure is called with only integers. If the strategy is based on call sites then two contours would be created. One contour would represent the evaluation context when the procedure is applied to `2`. The other would represent the context when the procedure is applied to `5`. Determining which splitting strategy is better depends on the consumer of the flow information.

Although it is not a requirement, the inlining transformation in Section 3 assumes a splitting strategy based on call sites. The grammar of the source and target language supports this strategy. At a call site $(e_0 \ldots e_n)_t$, the code of an applied abstract closure $\langle l, \kappa \rangle$ is assumed to be analyzed in the contour $t : \kappa$, *i.e.*, the contour obtained by prepending the sequence κ with the tag t.

$$
\begin{array}{rcl}
e \in E & = & M^p \,|\, (\lambda\ (\mathtt{x}_1\ \ldots\ \mathtt{x}_n)\ E)^q \\
M & = & \mathtt{c} \,|\, \mathtt{x} \,|\, (\mathtt{begin}\ E_1\ \ E_2) \,|\, (\mathtt{if}\ \ E_1\ \ E_2\ \ E_3) \,|\, (E_0\ E_1\ \ldots\ E_n)_t \,|\, (\mathtt{letrec}\ \ ((\mathtt{x}\ \ E_1))\ E_2) \\
p & \in & \mathit{ExpLabels} \\
q & \in & \mathit{ProcLabels} \\
l \in \mathit{Labels} & = & \mathit{ExpLabels} \cup \mathit{ProcLabels} \\
t & \in & \mathit{Keys}
\end{array}
$$

Figure 1: Source and target language for the inlining algorithm.

Many implemented analyses are based on this splitting strategy. An analysis is monovariant if *Keys* is a singleton set and all call sites are tagged with the same key. The analysis is similar to Shivers's 1CFA analysis if each call site is tagged with a unique key. A less accurate analysis that is suitable for inlining is obtained by tagging with a unique key each call site that

- has an operator that is a variable reference referring to a syntactically recognized `let`- or `letrec`-bound procedure, and
- does not represent a direct recursive call to the procedure.

All other calls are tagged with the "generic" key t_g. This strategy is the one used for polyvariant analysis of programs in this paper.

Examples

Some examples are useful for understanding the differences between monovariant and polyvariant analyses. The examples are written in the syntax of Figure 1 but `let` expressions are used to abbreviate direct applications of `lambda` expressions, and the examples are only partially labeled to avoid clutter. It is also assumed that the abstract value of a nonprocedure value is its type except for booleans, which maintain their truth value. For example, the abstract value of `5` is $\{\mathit{integer}\}$, and the abstract value of `#t` is $\{\mathit{true}\}$.

As a first example consider the following.

```
(let ((f (λ (x) x^p)))
  (begin
    (f 2)_{t_g}
    (f #f)_{t_g}))
```

A monovariant analysis of this expression begins by analyzing the expression $(\lambda\ (\mathtt{x})\ \mathtt{x}^p)$ in the empty contour $\langle\rangle$. The resulting abstract closure is then applied twice. The first application causes the body $\mathtt{x}^p$ to be analyzed in the contour $\langle t_g\rangle$. The second causes the body to be analyzed again in the contour $\langle t_g\rangle$. The analysis is forced to merge the information from the two call sites, and therefore, $F(\mathtt{x}^p, \langle t_g\rangle) = \{\mathit{integer}, \mathit{false}\}$.

Below is the same program but annotated for a polyvariant analysis.

```
(let ((f (λ (x) x^p)))
  (begin
    (f 2)_{t_0}
    (f #f)_{t_1}))
```

In this case a polyvariant analysis begins by analyzing the expression $(\lambda\ (\mathtt{x})\ \mathtt{x}^p)$ in the empty contour $\langle\rangle$. The first application causes the body $\mathtt{x}^p$ to be analyzed in the contour $\langle t_0\rangle$. The second application, however, causes the body to be analyzed in the different contour $\langle t_1\rangle$. Consequently, $F(\mathtt{x}^p, \langle t_0\rangle) = \{\mathit{integer}\}$ and $F(\mathtt{x}^p, \langle t_1\rangle) = \{\mathit{false}\}$. This is more precise than the information collected by the previous monovariant analysis.

As another example consider this program partially labeled for a monovariant analysis.

```
(let ((f (λ (a) (λ (b) (if a b #f))^{q_1})^{q_0}))
  (let ((g (if (h) (f #t)_{t_g} (f #f)_{t_g})))
    (g 5)_{t_g}))
```

In this example the abstract value of `f` is the singleton set containing the closure $\langle q_0, \langle\rangle\rangle$. Since the result of the call `(h)` is unknown, the closure is applied to both `#t` and `#f`. The result of the first call is the abstract value $\{\langle q_1, \langle t_g\rangle\rangle\}$, and the result of the second call is the same value. Thus the abstract value of `g` is $\{\langle q_1, \langle t_g\rangle\rangle\}$. At the call `(g 5)` the body of this second closure is evaluated in the contour $\langle t_g, t_g\rangle$. Since the value of `a` is $\{\mathit{true}, \mathit{false}\}$, the result of the call is $\{\mathit{integer}, \mathit{false}\}$.

For a polyvariant analysis the above program is labelled

```
(let ((f (λ (a) (λ (b) (if a b #f))^{q_1})^{q_0}))
  (let ((g (if (h) (f #t)_{t_0} (f #f)_{t_1})))
    (g 5)_{t_2}))
```

In this case the abstract value of `g` is the set $\{\langle q_0, \langle t_0\rangle\rangle, \langle q_0, \langle t_1\rangle\rangle\}$. The first closure represents the lambda expression evaluated in a context where `a` is bound to $\{\mathit{true}\}$, and the second represents the expression evaluated in a context where `a` is bound to $\{\mathit{false}\}$. At the call site `(g 5)` both closures are applied. Applying the first closure returns the abstract value $\{\mathit{integer}\}$

while the second returns the abstract value $\{\mathit{false}\}$. The result of the whole program is the union of these two values: $\{\mathit{integer}, \mathit{false}\}$.

3 The inlining algorithm

The inlining algorithm is specified in Figure 2. It is an improved version of the algorithm presented by Jagannathan and Wright [18]. In the figure the notation $\mathcal{P}(A)$ denotes the powerset of A and FV is a function that takes an expression and returns a sequence of the variables that occur free in that expression.

The inliner is a function $\mathcal{I}$ that takes an expression, a contour set, and a loop map. It returns an expression with possibly some or all call sites inlined. The contour set and loop map represent the inlining context, and the role of each will be explained together with the algorithm. Given a program e, the value of $\mathcal{I}[\![e]\!]\emptyset\emptyset$ is the inlined program. The target language of the inlining algorithm is a closure-converted [26] variant of the source. Closure-conversion is required, since procedures may be inlined outside of the lexical scope of their free variables.

The behavior of the function $\mathcal{I}$ depends first on the form of input expression. Constants and variables are mapped to themselves. A `begin` expression is rebuilt by inlining its constituent subexpressions. A λ expression must be residualized in all possible calling contexts. This explains why the inliner must maintain a set of contours as opposed to a single contour. During flow analysis a procedure activation occurs in a single analysis context, but during transformation a procedure may be residualized without knowing how it will be applied. The inliner must therefore assume that the procedure will be evaluated in all contexts in which the procedure may be applied. Hence the need for a contour set. The possible calling contexts for the lambda expression are obtained by using the function *add* to prepend each contour in the contour set γ with each of the elements of *Keys*. For example, if γ is the set $\{\langle t_1\rangle, \langle t_0\rangle\}$ and *Keys* is the set $\{t_g, t_0, t_1\}$, then the new contour set is $\{\langle t_g, t_1\rangle, \langle t_g, t_0\rangle, \langle t_0, t_1\rangle, \langle t_0, t_0\rangle, \langle t_1, t_1\rangle, \langle t_1, t_0\rangle\}$.

A conditional expression is inlined depending on the abstract value of its test. If the abstract value represents a value that is both true and false, both arms of the conditional are inlined and the entire conditional is reconstructed. If the value is definitely true or definitely false, then one arm or the other can be pruned in the output expression. If the abstract value is neither true nor false, then the arms of the conditional are considered dead, and the test is evaluated for effect.

A `letrec` expression is inlined by rebuilding the expression using inlined versions of the subexpressions. The right-hand side is evaluated in an updated loop map. The updated loop map is used to record the inlining context of a `letrec`-bound expression if the value of the expression is a merged abstract closure derived from a single lambda abstraction.

An application can be transformed in one of three ways. If no inlining can be justified the call is simply reconstructed. The other two cases accommodate an inlined call.

In the first case the following conditions must hold:

- the operator evaluates to a single merged abstract closure,
- the transformed procedure body is small enough to inline according to the predicate *inline*?, and
- there is not already a binding for the procedure, confirmed by examining the domain of the loop map μ.

If these conditions hold, the procedure is inlined at the call site and recursively bound to a fresh variable `f`. Although not specified in the figure, the procedure may be inlined directly at the operator position of the call if there are no references to `f` in the transformed procedure's body.

The second way in which an application may be inlined is if the operator evaluates to a single merged abstract closure, and there already exists an entry in the loop map for the procedure. In this case, the application is reconstructed with the the new operator being a reference to the binding variable for the preexisting specialization.

As an example consider the program in Figure 3(a). It is the factorial program annotated for and analyzed with a 1CFA analysis. The application `(f 5)` is inlined as well as the recursive call. The recursive call creates a new copy of the procedure, because the labels on the two call sites are different. At the next recursive call, however, the inliner finds an entry in the loop map and residualizes the recursive call.

Merging abstract values

Since the inlining context is composed of a set of contours, the abstract value of an expression cannot be obtained directly using the function F. Instead, the abstract values of the expression in all the contexts of γ must be merged. In the definition of M, the set V represents the union of the abstract values of the expression in each of the contours specified by γ.

The function M further maximizes the utility of the set V by merging the abstract closures in the set. The abstract closures derived from a label p are collapsed into a single merged abstract closure. The merged closure consists of the label p and a set of contours built from the abstract closures in V that contain the label p. While this merging is not helpful for residualizing

$$\begin{array}{rcl} \mathcal{I} & : & E \times \widetilde{Contours} \times \mathit{Loops} \to E \\ \gamma \in \widetilde{Contours} & = & \mathcal{P}(\widehat{Contours}) \\ \mu \in \mathit{Loops} & = & \mathit{ProcLabels} \times \widetilde{Contours} \to \mathit{Variables} \end{array}$$

$\mathcal{I}\llbracket \mathtt{c} \rrbracket \gamma\mu = \mathtt{c}$

$\mathcal{I}\llbracket \mathtt{x} \rrbracket \gamma\mu = \mathtt{x}$

$\mathcal{I}\llbracket (\mathtt{begin}\ \mathtt{e}_1\ \mathtt{e}_2) \rrbracket \gamma\mu = (\mathtt{begin}\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_2 \rrbracket \gamma\mu)$

$\mathcal{I}\llbracket (\lambda\ (\mathtt{x}_1\ \ldots\ \mathtt{x}_n)\ \mathtt{e}) \rrbracket \gamma\mu = (\lambda\ [\mathtt{z}_1\ \ldots\ \mathtt{z}_m]\ (\mathtt{x}_1\ \ldots\ \mathtt{x}_n)\ \mathcal{I}\llbracket \mathtt{e} \rrbracket \mathit{add}(\mathit{Keys}, \gamma)\mu)$

where $\langle \mathtt{z}_1, \ldots, \mathtt{z}_m \rangle = FV((\lambda\ (\mathtt{x}_1\ \ldots\ \mathtt{x}_n)\ \mathtt{e}))$

$$\mathcal{I}\llbracket (\mathtt{if}\ \mathtt{e}_1^l\ \mathtt{e}_2\ \mathtt{e}_3) \rrbracket \gamma\mu = \begin{cases} (\mathtt{if}\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_2 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_3 \rrbracket \gamma\mu) & \text{if } \mathit{true?}(M(l,\gamma)) \wedge \mathit{false?}(M(l,\gamma)) \\ (\mathtt{begin}\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_2 \rrbracket \gamma\mu) & \text{if } \mathit{true?}(M(l,\gamma)) \wedge \neg\mathit{false?}(M(l,\gamma)) \\ (\mathtt{begin}\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_3 \rrbracket \gamma\mu) & \text{if } \neg\mathit{true?}(M(l,\gamma)) \wedge \mathit{false?}(M(l,\gamma)) \\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu & \text{otherwise} \end{cases}$$

$\mathcal{I}\llbracket (\mathtt{letrec}\ ((\mathtt{x}\ \mathtt{e}_1^l))\ \mathtt{e}_2) \rrbracket \gamma\mu = (\mathtt{letrec}\ ((\mathtt{x}\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \mathit{add}(\mathit{Keys}, \gamma)\mu'))\ \mathcal{I}\llbracket \mathtt{e}_2 \rrbracket \gamma\mu)$

$$\text{where} \quad \mu' = \begin{cases} \mu[\langle q, t_g \rangle \to x] & \text{if } M(l,\gamma) = \{\langle q, \gamma' \rangle\} \\ \mu & \text{otherwise} \end{cases}$$

$$\mathcal{I}\llbracket (\mathtt{e}_0^{l_0}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n)_t \rrbracket \gamma\mu = \begin{cases} \begin{array}{l} ((\mathtt{letrec}\ ((\mathtt{f}\ (\lambda\ (\mathtt{w}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n) \\ \qquad\qquad ((\lambda\ [\mathtt{z}_1\ \ldots\ \mathtt{z}_m]\ \mathcal{I}\llbracket \mathtt{e} \rrbracket \gamma'\mu') \\ \qquad\qquad\ (\mathtt{closure\text{-}ref}\ \mathtt{w}\ 1) \\ \qquad\qquad\ \vdots \\ \qquad\qquad\ (\mathtt{closure\text{-}ref}\ \mathtt{w}\ m))))) \\ \quad \mathtt{f}) \\ \ \mathcal{I}\llbracket \mathtt{e}_0 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu \ldots \mathcal{I}\llbracket \mathtt{e}_n \rrbracket \gamma\mu)_t \end{array} & \begin{array}{l} \text{if } \mathit{inline?}(\mathcal{I}\llbracket \mathtt{e}_0 \rrbracket \gamma'\mu') \text{ and } \langle q, \gamma' \rangle \notin \mathit{dom}(\mu) \\ \text{where } M(l_0, \gamma) = \{\langle q, \gamma'' \rangle\} \\ \qquad P(q) = (\lambda\ (\mathtt{x}_1\ \ldots\ \mathtt{x}_n)\ \mathtt{e}) \\ \qquad \langle \mathtt{z}_1, \ldots, \mathtt{z}_m \rangle = FV((\lambda\ (\mathtt{x}_1\ \ldots\ \mathtt{x}_n)\ \mathtt{e})) \\ \qquad \gamma' = \mathit{add}(\{t\}, \gamma'') \\ \qquad f \text{ is fresh} \\ \qquad \mu' = \mu[\langle q, \gamma' \rangle \to f] \end{array} \\ (\mathtt{f}\ \mathcal{I}\llbracket \mathtt{e}_0 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu\ \ldots\ \mathcal{I}\llbracket \mathtt{e}_n \rrbracket \gamma\mu)_t & \text{if } \mu(\langle q, \mathit{add}(\{t\}, \gamma') \rangle) = f \text{ where } M(l_0, \gamma) = \{\langle q, \gamma' \rangle\} \\ (\mathcal{I}\llbracket \mathtt{e}_0 \rrbracket \gamma\mu\ \mathcal{I}\llbracket \mathtt{e}_1 \rrbracket \gamma\mu \ldots \mathcal{I}\llbracket \mathtt{e}_n \rrbracket \gamma\mu)_t & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} M & : & \mathit{Labels} \times \widetilde{Contours} \to \mathcal{P}(\{\mathit{true}, \mathit{false}\} \cup (\mathit{ProcLabels} \times \widetilde{Contours})) \\ M(l,\gamma) & = & (V \cap \{\mathit{true}, \mathit{false}\}) \cup \{\langle q, \gamma \rangle \mid q \in \mathit{ProcLabels} \wedge \gamma = \{\kappa \mid \langle q, \kappa \rangle \in V\} \wedge \gamma \neq \emptyset\} \\ & & \text{where } V = \bigcup \{F(l, \kappa) \mid \kappa \in \gamma\} \end{array}$$

$$\begin{array}{rcl} \mathit{add} & : & \mathit{Keys} \times \widetilde{Contours} \to \widetilde{Contours} \\ \mathit{add}(T, \gamma) & = & \{t : \kappa \mid t \in T \wedge \kappa \in \gamma\} \end{array}$$

Figure 2: The inlining algorithm.

```
(letrec ((f (λ (x) (if (= x 0) 1 (∗ x (f (− x 1))$_{t_g}$)))))
  (f 5)$_{t_0}$)
```

```
(letrec ((f (λ (x) (if (= x 0) 1 (∗ x (f (− x 1))$_{t_g}$)))))
  ((λ (g$_0$ x)
    (if (= x 0)
        1
        (∗ x ((letrec ((g$_1$ (λ (g$_2$ x) (if (= x 0) 1 (∗ x (g$_1$ f (− x 1))$_{t_g}$)))))
                g$_1$)
              f
              (− x 1))$_{t_g}$)))
   f
   5)$_{t_0}$)
```

(a) The factorial program before and after inlining.

```
(let ((f (λ (a) (λ (b) (if a b #f))$^{q_1}$)$^{q_0}$))
  (let ((g (if (h) (f #t)$_{t_0}$ (f #f)$_{t_1}$)))
    (g 5)$_{t_2}$))
```

```
(let ((f (λ (a) (λ (b) (if a b #f))$^{q_1}$)$^{q_0}$))
  (let ((g (if (h)
               ((λ (g$_0$ a) (λ (b) b)$^{q_1}$) f #t)$_{t_0}$
               ((λ (g$_1$ a) (λ (b) #f)$^{q_1}$) f #f)$_{t_1}$)))
    ((λ (g$_2$ b)
       (let ((a (closure-ref g$_2$ 1)))
         (if a b #f)))
     g
     5)$_{t_2}$))
```

(b) A higher-order program before and after inlining.

Figure 3: Two examples of inlining justified by a 1CFA analysis. The programs are only partially labeled.

conditionals, the merged closures can be useful when residualizing calls.

Figure 3(b) is an example of inlining with merged closures. It is the last example from Section 2 analyzed using a polyvariant analysis. Since the abstract value of `f` is a singleton set containing one closure, both calls to `f` would be inlined assuming that the *inline*? predicate was satisfied. At the call site (`g 5`), however, the abstract value of the operator is the set consisting of two closures: $\{\langle q_1, \langle t_0\rangle\rangle, \langle q_1, \langle t_1\rangle\rangle\}$. From the definition of M, this value would be transformed into the value $\{\langle q_1, \{\langle t_0\rangle, \langle t_1\rangle\}\rangle\}$. Thus without merging the closures, the call (`g 5`) could not be inlined, but after merging it can be inlined.

The example also illustrates how maintaining the inlining context can improve residualization of lambda expressions. When the calls (`f #t`) and (`f #f`) are inlined, the expression (λ (`b`) (`if a b #f`)) is residualized in both contexts. By maintaining the inlining context it is possible for the inliner to prune the conditionals in both cases. When the call (`g 5`) is inlined the conditional cannot be pruned because it is inlined in a context where the abstract value of `a` is the set $\{\mathit{true}, \mathit{false}\}$.

4 Implementation

The inlining algorithm is fully implemented in the *Chez Scheme* [9] compiler. It is implemented as specified with two differences. First, the inliner's output is not closure-converted at the source level. Subsequent simplification may eliminate free variables, so it is desirable to avoid selecting a closure's layout until later in compilation. Second, a free variable referenced through an operator's closure is avoided if the free variable is visible at the reference point.

The implementation approximates *inline?*. Instead of querying whether a specialized procedure body is small enough for inlining, the implementation examines the original procedure body with conditionals pruned where flow information deems it safe to do so. A simple node count of the parse tree is computed and the procedure is inlined if the count is under a threshold set before inlining is begun.

The inliner is run after the flow analysis and before procedure call optimizations that are also justified by flow information [2]. Immediately after inlining a simplification pass is performed to profit from the larger intraprocedural contexts introduced by inlining. Also, the inlining algorithm cannot preserve flow information to justify procedure call optimizations, so the flow analysis must be run again after simplification. After the second flow analysis compilation is continued.

4.1 Simplification

Inlining introduces opportunities for constant folding, copy propagation, and useless-binding elimination. A one-pass simplifier performs these operations on the inliner's output. With one exception it is a simple version of the optimizer described by Waddell and Dybvig [27] but does not perform procedure integration as part of copy propagation.

The exception is the treatment of inlined procedure calls that result in a recursive call to the inlined procedure. Consider this possible output from the inliner as an example.

```
((letrec ((g₀ (λ (g₁ x) (g₀ g₁ x)))) g₀)
 f
 5)
```

The variable g_1 is unneeded, and the program should be transformed into

```
((letrec ((g₀ (λ (x) (g₀ x)))) g₀)
 5)
```

A naïve simplifier would be unable to perform this transformation, however, since the variable g_1 is referenced for value. The implemented simplifier is aware, however, of recursive bindings introduced by the inliner and recursive calls to those bindings. The simplifier is therefore able to eliminate such useless bindings whenever possible.

As two complete examples, the simplified versions of the inlined programs from Figure 3 are given in Figure 4. In Figure 4(a), the outer `letrec` expression is eliminated because there is no longer a reference to `f`. The bindings for `b` and `x` are eliminated since all references to them are propagated. Assuming that the calls to $-$ and `=` are inlined primitives, the outer calls `(= x 0)` and `(- x 1)` can be folded. This then permits the outer conditional to be folded as well.

The simplified output in Figure 4(b) is obtained in a similar fashion. Constants are propagated to their reference points, and the bindings rendered useless by propagation are eliminated.

4.2 Preserving flow information

The inlining algorithm preserves flow information about nonprocedure values but it violates the soundness of information about procedures. Consider this program annotated for and analyzed with a monovariant analysis.

```
(let ((f (λ (a) (λ (b) 5)^{q₁})^{q₀}))
  (let ((g (if (h) (f 1)_{t_g} (f 2)_{t_g})))
    (g 3)_{t_g}))
```

After the analysis the abstract value of `g` is the set $\{\langle q_1, \langle t_g \rangle\rangle\}$.

Assume that during inlining the calls to `f` are inlined but the call to `g` is not. After inlining and simplification the output is

```
(let ((g (if (h)
             (λ (b) 5)^{q₁}
             (λ (b) 5)^{q₁})))
  (g 3)_{t_g})
```

At this point the abstract value of `g` is unsafe. The value indicates that only one procedure may arrive at the call site at run time when in fact either of two procedures may arrive. Thus the information is unsound for flow-based procedure call optimizations.

Consequently, the flow analysis must be run again after inlining in order to recompute flow information. If the inliner performs closure-conversion at the source level it does not matter how accurate the analysis is. In our implementation, however, an analysis at least as accurate as 0CFA must be used in order to correctly resolve `closure-ref` operations later in the compiler.

5 Evaluation

The goal of our evaluation is to determine how the quality of flow information affects the inlining algorithm's ability to optimize programs. To do this, four flow analyses were used to evaluate the inliner. In all four analyses, conditional tests were used when possible to restrict the abstract values of variables in the arms of conditionals [16]. The analyses are sub0CFA$_1$, 0CFA, sub1CFA$_1$, and 1CFA.

0CFA and 1CFA are relatively well-understood analyses. The sub0CFA$_1$ and sub1CFA$_1$ analyses are approximations. From the point of view of abstract interpretation, 0CFA is the monovariant analysis obtained by allowing the interpreter to evaluate an expression an

```
(* 5 ((letrec ((g$_0$ (λ (x) (if (= x 0)$_{t_g}$ 1 (* x (g$_0$ (− x 1))$_{t_g}$)))))
        g$_0$)
      4)$_{t_g}$)
```

(a) The simplified factorial program.

```
(let ((g (if (h) (λ (b) b)$^{q_1}$ (λ (b) #f)$^{q_1}$)))
  (let ((a (closure-ref g 1)))
    (if a 5 #f)))
```

(b) The simplified higher-order program.

Figure 4: The two examples from Figure 3 after simplication has been performed.

unlimited number of times. For a constraint-based analysis, this corresponds to allowing the solver to update a constraint variable an unlimited number of times. Limiting the number of evaluations, however, yields a class of analyses we call *sub0CFA*. In particular, sub0CFA$_n$ is the analysis obtained by limiting the analysis to n evaluations before forcing it to select a safe solution that may be less precise than the least solution. The class forms a hierarchy with sub0CFA$_0$ the least accurate analysis in the class and 0CFA the limit as n goes to infinity. The development of the sub1CFA class of analyses is analogous.

The relative accuracy of the two classes is clear in some cases but not in others. 0CFA is less accurate than 1CFA, and for any n, sub0CFA$_n$ is less accurate than sub1CFA$_n$. On the other hand, for any n, 0CFA is neither more nor less accurate than sub1CFA$_n$. For some programs 0CFA will be more accurate and for others sub1CFA$_n$ will be more accurate.

The implementation was evaluated using the benchmarks described in Table 1. The benchmarks were compiled as whole programs. Furthermore, the dynamic, graphs, lattice, matrix, maze, nbody, and splay benchmarks had many primitive procedures explicitly included in their source. Examples of such procedures include `map`, `assq`, and `cadr`. These modified benchmarks were the same benchmarks used by Jagannathan and Wright [18].

Table 2 gives statistics on the cost of analyzing the benchmarks for each of the four flow analyses. These statistics were collected on an Intel 80686 processor with a 512k level two cache, 128Mb of memory, and 100MB of swap space.

The statistics support the claim that polyvariant analyses are in practice more expensive than monovariant analyses. In all cases, more cpu time is required by the polyvariant analyses than by the monovariant analyses. Furthermore, the amount of cpu time required by a polyvariant analysis can be significant. For example, the 1CFA analysis needs three seconds to analyze matrix yet the benchmark is only 575 lines of code, and the 1CFA analysis exhausted virtual memory analyzing the interpret benchmark.

Table 3 shows the run-time reduction in calls in order to measure the analyses' effectiveness at identifying sites where inlining may occur. The data includes only calls that could be inlined, *i.e.*, calls to top-level bound variables and inlined primitives are omitted. It also does not include direct calls that are equivalent to `let` expressions. The inliner was run at thresholds 10 and 30, and the ratios are given as the number of calls performed by the inlined program over the number of calls performed by a baseline with no inlining or simplification performed. All numbers are less than one, since inlining cannot introduce new calls other than calls that are treated as `let` expressions. The data supports the claim that more inlining is justified as the accuracy of the analysis improves, but the relatively inaccurate analyses are also able to justify a significant amount of inlining.

Table 4 shows change in object code size. Again, the numbers are given as ratios over a baseline with no inlining or simplification performed. The threshold 10 is relatively safe as it does not cause any significant increase in code growth.

At threshold 30, however, there is sometimes significant code growth. Furthermore, inlining justified by a polyvariant analysis can cause code growth larger than when justified by a monovariant analysis. As illustrated in Figure 3(a), inlining when justified by a polyvariant analysis causes unrolling that does not occur when inlining is justified by a monovariant analysis. The aim in these cases is to take advantage of the polyvariant analysis' extra precision to perform pruning or other optimizations that would not be possible with a monovariant analysis. Figure 4(a) shows this is possible, but as the numbers for the benchmark dynamic show, this optimism may dramatically increase code growth if code

benchmark	lines	Description
boyer	500	logic programming benchmark originally by Bob Boyer
conform	450	type checker by Jim Miller
dynamic	2,275	a dynamic type inferencer applied to itself
earley	650	Earley's parser by Marc Feeley
graphs	500	counts directed graphs with distinguished root and k vertices, each having out-degree at most 2
interpret	1,000	Marc Feely's Scheme interpreter evaluating takl
lattice	200	enumerates the lattice of maps between two lattices
matrix	575	tests whether an $n \times n$ matrix satisfies a particular property
maze	800	computes path through a random maze using a union-find algorithm
nbody	1,500	Greengard multipole algorithm for computing gravitational forces
peval	600	Feeley's simple Scheme partial evaluator
simplex	175	simplex algorithm
splay	950	an implementation of splay trees

Table 1: Benchmarks used to evaluate the inlining algorithm.

benchmark	sub0CFA$_1$	0CFA	sub1CFA$_1$	1CFA
boyer	0.01	0.01	0.05	0.07
conform	0.04	0.06	0.18	0.31
dynamic	0.32	0.32	1.99	2.09
earley	0.07	0.07	0.35	0.45
graphs	0.05	0.05	0.17	0.2
interpret	0.39	6.11	1.18	—
lattice	0.01	0.02	0.08	0.19
matrix	0.05	0.07	0.5	3.02
maze	0.05	0.05	0.17	0.19
nbody	0.13	0.22	1.13	1.98
peval	0.07	0.07	0.42	0.41
simplex	0.03	0.03	0.16	0.19
splay	0.06	0.06	0.83	0.95
average	0.1	0.55	0.55	0.84

Table 2: Statistics on cpu time used to analyzes each benchmark. Times are in seconds.

simplification does not occur.

Figure 5 gives run-time performance improvements with simplification enabled. The numbers are given as ratios over a baseline with no inlining or simplification performed. Speedups improve as the inlining threshold is increased, but this may not hold on machines with large caches, since code growth may interfere with cache behavior. The lattice benchmark shows the largest improvement, running 20% faster when optimized using the 1CFA analysis rather than the 0CFA analysis. Most benchmarks, however, improve by only a few percent if at all.

The data on speedups and reduction in calls indicates that inlining can improve performance primarily by enabling intraprocedural optimizations elsewhere in the compiler, *e.g.*, the simplification phase run after inlining. Procedure call overhead is not very large because the compiler already optimizes calls, and therefore, eliminating procedure calls does not improve performance significantly. For example, the lattice benchmark had 20% fewer procedure calls when optimized using the 1CFA analysis over the 0CFA analysis, but it was only 3% faster.

6 Related work

Our work is most closely related to Jagannathan and Wright's [18]. Our work elaborates on their results by comparing a number of different flow analyses and improving upon their inlining algorithm. They judged their inlining algorithm using a single polyvariant flow analysis while we evaluated the algorithm using four analyses, both monovariant and polyvariant. Our algorithm improves upon their algorithm in that it exposes

benchmark	10				30			
	sub0CFA_1	0CFA	sub1CFA_1	1CFA	sub0CFA_1	0CFA	sub1CFA_1	1CFA
boyer	0.98	0.98	0.98	0.98	0.89	0.89	0.84	0.84
conform	0.31	0.31	0.31	0.31	0.07	0.07	0.06	0.06
dynamic	0.34	0.34	0.34	0.34	0.32	0.32	0.19	0.19
earley	0.59	0.59	0.59	0.59	0.52	0.52	0.45	0.45
graphs	0.91	0.91	0.91	0.91	0.67	0.67	0.50	0.50
interpret	0.83	0.83	0.83	—	0.83	0.83	0.83	—
lattice	0.99	0.99	0.99	0.99	0.53	0.53	0.34	0.34
matrix	0.69	0.69	0.77	0.69	0.53	0.53	0.53	0.46
maze	0.31	0.31	0.31	0.31	0.29	0.29	0.29	0.29
nbody	0.96	0.96	0.96	0.96	0.74	0.74	0.53	0.53
peval	0.93	0.93	0.93	0.93	0.49	0.49	0.45	0.44
simplex	0.32	0.32	0.32	0.32	0.32	0.32	0.30	0.30
splay	0.57	0.55	0.61	0.54	0.48	0.45	0.52	0.44
average	0.67	0.67	0.68	0.66	0.51	0.51	0.45	0.40

Table 3: Ratio of run-time calls over the number or calls performed by a baseline with no inlining performed. Ratios are measured for inlining thresholds 10 and 30.

benchmark	10				30			
	sub0CFA_1	0CFA	sub1CFA_1	1CFA	sub0CFA_1	0CFA	sub1CFA_1	1CFA
boyer	1.03	1.03	1.03	1.03	1.47	1.47	2.33	2.33
conform	0.94	0.94	0.94	0.94	1.06	1.06	1.19	1.19
dynamic	2.00	2.00	2.08	2.00	3.56	3.56	4.80	4.72
earley	1.07	1.07	1.07	1.07	1.06	1.06	1.13	1.13
graphs	0.71	0.71	0.71	0.71	0.73	0.73	0.82	0.82
interpret	1.23	1.23	1.23	—	2.06	2.06	2.21	—
lattice	1.05	1.05	1.05	1.04	1.20	1.20	1.51	1.51
matrix	1.03	1.03	1.03	1.02	1.11	1.11	1.24	1.23
maze	0.86	0.86	0.86	0.86	0.88	0.88	0.89	0.89
nbody	1.06	1.06	1.06	1.06	1.29	1.29	1.38	1.37
peval	1.11	1.11	1.11	1.11	1.20	1.20	1.31	1.31
simplex	0.89	0.89	0.89	0.89	0.89	0.89	0.92	0.92
splay	1.30	1.33	1.23	1.27	1.42	1.44	1.42	1.46
average	1.10	1.10	1.10	1.08	1.38	1.38	1.63	1.57

Table 4: Ratio of object code size over the size of the baseline at inlining thresholds 10 and 30. Numbers smaller than 1 indicate a reduction in code size.

	10				30			
benchmark	sub0CFA$_1$	0CFA	sub1CFA$_1$	1CFA	sub0CFA$_1$	0CFA	sub1CFA$_1$	1CFA
boyer	1.01	1.01	1.01	1.01	0.99	0.99	0.97	0.98
conform	0.70	0.70	0.70	0.70	0.58	0.58	0.57	0.57
dynamic	0.92	0.92	0.91	0.93	0.90	0.90	0.98	0.95
earley	0.93	0.93	0.93	0.93	0.91	0.91	0.90	0.90
graphs	0.89	0.89	0.89	0.89	0.83	0.83	0.79	0.79
interpret	0.89	0.89	0.89	—	0.89	0.90	0.89	—
lattice	1.02	1.02	1.02	1.02	0.97	0.97	0.75	0.75
matrix	0.98	0.98	0.98	0.98	0.95	0.95	0.95	0.92
maze	0.84	0.84	0.84	0.84	0.79	0.79	0.79	0.79
nbody	0.98	0.98	0.98	0.98	0.94	0.94	0.91	0.91
peval	0.96	0.96	0.96	0.96	0.87	0.87	0.88	0.87
simplex	0.72	0.72	0.72	0.72	0.72	0.72	0.72	0.72
splay	0.95	0.94	0.93	0.92	0.92	0.92	0.90	0.90
average	0.91	0.91	0.90	0.91	0.87	0.87	0.85	0.84

Table 5: Ratio of run time over the baseline at inlining thresholds 10 and 30. Numbers smaller than 1 indicate that the inlined program is faster than its baseline.

more opportunities for inlining. It does this by building more precise flow information for contexts in which lambda expressions are residualized. In particular, if their algorithm is rewritten in terms of ours, the function M would not merge closures, and lambda expressions would be residualized in the contour set $\widehat{Contours}$.

Jagannathan and Wright report better speedups for some benchmarks given in this paper. The differences may be due to the fact that their implementation inlines procedures bound in the top-level environment. Calls to top-level procedures are significantly more expensive than calls to procedures lexically bound by `let` or `letrec` expressions. This unintended side-effect of their implementation masks the actual benefits of inlining. In our experiments, on the other hand, each benchmark was given in complete form to the analysis, the inliner, and the rest of the compiler. Hence the compiler was able to treat procedures that would have been bound in the top-level environment as lexically-bound procedures. As a result, our speedups and code growth are comparable to those reported by others [10, 13].

Static analyses have been used to reduce method dispatch overhead in object-oriented languages by either inlining calls or converting them to static dispatches [1, 14, 20, 21, 22]. Agesen and Hölze obtain good speedups for Cecil as well, averaging about 10% improvement. Plevyak and Chien obtain excellent speedups for C++ when combining their optimizations with others [21]. Their combination of inlining with other optimizations makes the precise benefits of inlining difficult to determine, but it appears to make at least a 50% improvement in speed on their benchmarks while no statistics on code growth were given. These results are to be expected, since C++ and Cecil are ideally suited for inlining, where method dispatch is expensive and methods to be inlined are small.

Our improved inlining algorithm is related to the specialization phase of an off-line partial evaluator [19]. Inlining a call is analogous to unfolding a call. To the best of our knowledge, merging abstract closures to expose more opportunities for unfolding is a technique not used in specialization algorithms for partial evaluators.

7 Conclusions

We have evaluated four flow analyses to determine their effectiveness for justifying inlining. The analyses represent a range of cost/accuracy tradeoffs, from a fast analysis that is less accurate than 0CFA to a polyvariant analysis similarly to 1CFA. The inlining algorithm incorporates a novel notion of merging that exposes more opportunities for inlining when justified by information collected by a polyvariant analysis.

The improvement to the inlining algorithm can expose additional opportunities for inlining when a polyvariant analysis is used. The improvement selectively merges flow information depending on the specialization context of the inliner. The merging folds multiple abstract closures obtained from a single lambda expression into a single abstract closure. This merging technique could be used by any program transformer that uses the results of a polyvariant flow analysis.

The analyses and inlining algorithm were evaluated using several benchmarks. More procedure calls were eliminated as the accuracy of the analyses improved. Speedups also improved somewhat as analysis accuracy increased, but the speedups obtained from the accurate analyses were not significantly better than the improvements from the less precise analyses. Given the cost of accurate flow analyses, it may not be worthwhile to use a polyvariant analysis during development and testing and instead use it only during the final stages of development. Using a parameterized flow analysis framework [2], however, allows the compiler to be constructed such that either analysis can be run without changing the optimizations passes.

While speedups improved using a polyvariant analysis, code growth also increased. Better heuristics are needed to determine when it is profitable to inline procedures. One possibility is to combine simplification with inlining in order to better judge what effect inlining will have on code size. Taking cues from work on inlining for Cecil and C, it may also be worthwhile to combine flow analysis with dynamic profiling information [7] to make inlining decisions.

The choice of source and target language in Figure 1 assumes a splitting strategy based on call strings. The inlining is not dependent on this choice, however, and other strategies could be substituted with little difficulty. One possibility is to base the decision on types. This would perhaps lead to more pruning in a called procedure, causing the inliner to integrate the procedure in cases where it otherwise would not.

The inlining algorithm does not preserve the soundness of flow information. Thus the flow analysis must be run again if subsequent compiler passes depend on flow information. This complicates a compiler whose optimizations are based on flow analyses, since optimizations must be ordered based on what flow analysis is run and at what time during compilation.

The compiler does not currently take full advantage of inlining, and speedups would perhaps improve if it did. While the simplification pass is able to use the inlining results to perform more constant folding, copy propagation, and useless-binding elimination, the register allocator [8] does not take advantage of the larger procedure bodies introduced by inlining. As Davidson and Holler [13] indicate, it can even be detrimental to inline without cooperation from the register allocator.

Acknowledgments: We thank Suresh Jagannathan, Andrew Wright, and Marc Feeley for making their benchmarks available.

References

[1] Ole Agesen and Urs Hölze. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA '95, 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–107. ACM, 1995.

[2] J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 184–194. ACM, 1996.

[3] J. Michael Ashley and Charles Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1431–1448, 1994.

[4] Andrew E. Ayers. *Abstract Analysis and Optimization of Scheme.* PhD thesis, MIT, 1993.

[5] Anders Bondorf. *Similix Manual, System Version 5.0.* DIKU, University of Copenhagen, Denmark, 1993.

[6] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on Partial Evaluation*, 1993.

[7] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. Submitted for publication.

[8] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 130–138. ACM, 1995.

[9] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme System Manual, Rev. 2.4*, July 1994.

[10] Pohua P. Chang, Scott A. Mahlke, and William Y. Chen. Profile-guided automatic inline expansion for C programs. *Software – Practice and Experience*, 25(5):349–369, May 1192.

[11] Charles Consel. Polyvariant binding-time analysis for higher-order, applicative languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93*, pages 66–77, 1993.

[12] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93*, pages 145–154, 1993.

[13] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software – Practice and Experience*, 18(8):775–790, August 1988.

[14] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 93–102. ACM, 1995.

[15] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[16] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–317. ACM, 1994.

[17] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the 1995 International Static Analysis Symposium*, volume 854 of *Lecture Notes in Computer Science*, pages 207–224. Springer-Verlag, 1995.

[18] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 193–205. ACM, 1996.

[19] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[20] Hemant D. Pande and Barbara G. Ryder. Static type determination and aliasing for C++. Technical Report LCSR-TR-250-A, Rutgers University Department of Computer Science, October 1995.

[21] John Plevyak and Andrew A. Chien. Automatic interprocedural optimization for object-oriented languages. Submitted for publication.

[22] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94, 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340. ACM, 1995.

[23] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 50–61. ACM, 1996.

[24] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991. CMU-CS-91-145.

[25] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '91*, New Haven, Connecticut, 1991.

[26] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

[27] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure integration. Submitted for publication.