Effective Flow Analysis for Avoiding Run-Time Checks*

Suresh Jagannathan and Andrew Wright

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

Abstract. This paper describes a general purpose program analysis that computes global control-flow and data-flow information for higher-order, call-by-value programs. This information can be used to drive global program optimizations such as inlining and run-time check elimination, as well as optimizations like constant folding and loop invariant code motion that are typically based on special-purpose local analyses.

The analysis employs a novel approximation technique called *polymorphic splitting* that uses let-expressions as syntactic clues to gain precision. Polymorphic splitting borrows ideas from Hindley-Milner polymorphic type inference systems to create an analog to polymorphism for flow analysis.

Experimental results derived from an implementation of the analysis for Scheme indicate that the analysis is extremely precise and has reasonable cost. In particular, it eliminates significantly more run-time checks than simple flow analyses (*i.e.* 0CFA) or analyses based on type inference.

1 Introduction

Advanced programming languages such as Scheme [3] and ML [13] encourage a programming style that makes extensive use of data and procedural abstraction. Higher degrees of abstraction generally entail higher run-time overheads; hence, sophisticated compiler optimizations are essential if programs written in these languages are to compete with those written in lower-level languages such as C. Furthermore, because Scheme and ML procedures tend to be small, these compiler optimizations must be interprocedural if they are to be effective.

Interprocedural optimizations require interprocedural control- and data-flow information that expresses where procedures are called and where data values are passed. In its purest form, the flow analysis problem for these languages involves determining the set of procedures that can reach a given application point and the set of values that can reach a given argument position in an application. This information can be used to drive global program optimizations such as inlining and run-time check elimination, as well as optimizations like constant folding and loop invariant code motion that are typically based on special-purpose local analyses.

---

* A preliminary version of this paper appears as part of Aarhus University Technical Report, DAIMI-PB 493, May, 1995.

We present a flow analysis framework for a call-by-value, higher-order language. The framework is parameterized over different approximations of exact values to abstract values, and hence can be used to construct a spectrum of analyses with different cost and accuracy characteristics [10, 20]. In particular, we study a novel approximation technique called *polymorphic splitting* that uses let-expressions as syntactic clues to gain precision. Polymorphic splitting borrows ideas from Hindley-Milner polymorphic type inference systems to create an analog to polymorphism for flow analysis. Polymorphic splitting allows the analysis to avoid merging information between unrelated calls to a polymorphic function, yielding more precise flow information than would otherwise be possible.

To investigate its effectiveness, we have implemented the analysis for Scheme. The implementation handles all of the constructs and operators specified in the R^4RS report [3], including variable-arity procedures, data structures, first-class continuations, and assignment. We use the analysis to avoid unnecessary run-time type checks at primitive operations. Run-time type checks can be avoided at an application of a primitive operator if the types of the arguments can be proven to conform to the expected signature of the operator being applied. As our analysis yields sets of abstract values that approximate the types of the arguments, it is easy to determine when a run-time check can be avoided: each argument's abstract value or type must be a subset of the type expected.

Experimental results for realistic Scheme programs indicate that polymorphic splitting is extremely precise and has reasonable cost. The analysis eliminates significantly more run-time checks than comparable simple analyses (*e.g.* 0CFA [19]) or type-inference based techniques (*e.g.* soft typing [22]). While the computational cost of our analysis appears to be higher than that of type-inference based methods, analysis times are still within reason for including the analysis in an optimizing compiler. Furthermore, and perhaps surprisingly, our polymorphic splitting analysis is often faster than coarse analyses such as 0CFA. We discuss the reasons for this in Section 5.

The remainder of the paper is organized as follows. The next section informally motivates polymorphic splitting, and Section 3 defines our framework in a formal manner. Section 3.1 discusses polymorphic splitting, Section 4 discusses the implementation, and Section 5 provides performance measurements. The final section places our results in the context of related work.

2 Motivation

In the absence of any compile-time analysis, Scheme implementations must ensure *at run-time* that primitive operations are applied to arguments of a sensible type. Even in ML, where a static type discipline prevents some primitives from being applied to incorrect values, many primitive operations (*e.g.* `hd`) require run-time checks. These run-time checks can have a significant negative impact on program performance.

Although one can construct *ad hoc* heuristics to determine when certain run-time checks can be safely removed, a more satisfactory solution is to use a general interprocedural analysis. Many analyses with different cost and accuracy characteristics are possible, and the right combination is not readily apparent for a given optimization. For example, consider the following Scheme expression:

```
(let ((f (lambda (x) x)))
  (f 1))
```

Simple control-flow analyses [20] or set-based analyses [7] determine that the application of `f` to `1` produces `1` as the result. These simple low-order polynomial-time analyses effectively determine all potential call sites for all procedures, but *merge* the values of arguments from all call sites. The *type* of a procedure's argument is therefore the union of the types of all values to which the argument is possibly bound. This set of *abstract values* is propagated among expressions in the procedure's body, yielding a similarly merged set for the procedure's result.

Because of the merging performed at calls and returns, these analyses fail to recognize that a run-time check is unnecessary on the addition operation in the following expression:

```
(let ((f (lambda (x) x)))
  (+ (f 1))
  (f #t)))
```

The arguments (`1` and `#t`) to the two calls to `f` are merged by such analyses to yield {`1`,`#t`} as the abstract value for `x` in `f`'s body. Consequently, any application of `f` in this framework yields {`1`,`#t`} as its result. Run-time check optimizations based on these analyses are unable to eliminate the unnecessary run-time check at the addition operation.

A more sophisticated analysis [20] avoids merging information in syntactically distinct calls to the same procedure by treating the call site at which the procedure is applied as a disambiguation context; this analysis is referred to as 1CFA. In the above example, a 1CFA analysis deduces that `f` in the first call is applied to `1` and returns `1` as its result and, that in the second call, `f` is applied to `#t` and returns `#t`. With 1CFA, the run-time check at the addition operation can be eliminated.

1CFA is an instance of a general flow-analysis framework based on *call-string* approximations. Call-string based analyses use the N most dynamically recent call sites as a disambiguation context, for some small, fixed N. Because the point at which a procedure is defined may be far removed from its point of use, call-string based analyses are highly sensitive to program structure. It is easy to construct a simple extension to the above example for which 1CFA computes overly conservative information:

```
(let ((f (lambda (x) x))
      (g (lambda (a b) (a b))))
  (+ (g f 1))
  (g f #t))
```

In this example, there is an intervening call between the point where `f` is first referenced and the point where it is applied. 1CFA computes an abstract value set for `f` 's result that contains both `1` and `#t` . Correctly disambiguating the two calls to `(lambda (x) x)` made via the two calls to `g` requires a 2CFA analysis that preserves two levels of call history.

In contrast, polymorphic type inference algorithms [12, 22] correctly infer the proper type for `f` without requiring such tuning. Polymorphic type systems disambiguate different calls to a polymorphic procedure by effectively duplicating the procedure wherever it is referenced. This yields an analysis that is insensitive to the dynamic sequence of calls separating the construction of a value from its use. Consequently, analyses based on call-string abstractions are only marginally useful in capturing polymorphism.

We therefore consider an alternative abstraction called *polymorphic splitting* that uses let-expressions as syntactic clues to determine when abstract evaluation contexts should be disambiguated. Let f be a procedure bound by a let-expression, and let $f_1, \ldots, f_n$ be syntactically distinct occurrences of f within the let-expression's body. An analysis incorporating polymorphic splitting associates a distinct abstract closure with each of the f_i. Thus, different applications of f will construct different abstract bindings for the arguments. The abstract values of these bindings are not merged with abstract values associated with bindings of other applications of f in the let-body. It is well-known that quantification rules for type variables can be discarded in favor of a substitution rule on polymorphic variables [14]. Polymorphic splitting captures the essence of polymorphism by incorporating this observation into a flow-analysis framework.

Like polymorphic type inference, polymorphic splitting relies only on lexical structure and is independent of dynamic contexts. In the above example, there are four distinct occurrences of let-bound variables in the let-body. The analysis effectively substitutes the closure values of these variables at the application points, and thereby avoids merging bindings from the different applications. A run-time check optimization based on this analysis will eliminate all run-time checks from this example.

Some technical machinery is required to implement the intuition of polymorphism as substitution without actually performing an inefficient code transformation. We discuss this issue in Section 3.1.

3 The Framework

We present a formal definition of our analysis framework for a functional core language. The definition is sufficiently parameterized that we can easily modify it to obtain a spectrum of analyses with different cost and accuracy characteristics.

We consider a simple call-by-value language with *labeled expressions* e^l of the form

$$\begin{aligned} e^l \ ::= \ & c^l \ \big| \ x^l \ \big| \ (\lambda x.e_1^{l_1})^l \ \big| \ (e_1^{l_1}\ e_2^{l_2})^l \\ \big| \ & x^l_{[l']} \ \big| \ (\mathsf{let}\ x_{[l]} = e_1^{l_1}\ \mathsf{in}\ e_2^{l_2})^l \\ \big| \ & (\mathsf{if}\ e_1^{l_1}\ \mathsf{then}\ e_2^{l_2}\ \mathsf{else}\ e_3^{l_3})^l \end{aligned}$$

where $c \in \mathit{Const}$ are *constants*, $x \in \mathit{Var}$ are *variables*, and $l \in \mathit{Label}$ are *labels*. Constants include simple values like $\mathsf{0}, \mathsf{1}, \mathsf{true}$, and false. Free variables (FV) and bound variables (BV) are defined as usual [2], except that a subscripted variable $x^l_{[l']}$ must be bound by a let-expression with label l', and an unsubscripted variable x^l must be bound by a λ-expression. A *program* $e_0^{l_0}$ is an expression with no free variables. We assume that bound variables are appropriately renamed so that all bound variables in a program are distinct. We require every subexpression of a program to have a unique label, and occasionally omit labels from expressions to avoid clutter. The exact semantics for this language is an ordinary call-by-value semantics [18]. Recursive procedures can be constructed with the call-by-value Y combinator.

Our analysis yields sets of *abstract values.* Abstract values are defined as follows:

$$\begin{aligned} v &\in \mathit{Value} &&= \mathit{Label} + \mathit{Closure} \\ \langle l, \rho, \kappa \rangle &\in \mathit{Closure} &&= \mathit{Label} \times \mathit{Env} \times \mathit{Contour} \\ \rho &\in \mathit{Env} &&= \mathit{Var} \rightarrow \mathit{Contour} \\ \kappa &\in \mathit{Contour} \end{aligned}$$

A single abstract value v is either a label l or a closure $\langle l, \rho, \kappa \rangle$. An abstract value $v = l$ identifies a particular occurrence of a constant c^l, and hence represents a single exact value. An abstract value $v = \langle l, \rho, \kappa \rangle$ identifies a *set* of functions created by evaluations of a λ-expression $(\lambda x.e)^l$. In a typical operational semantics, a closure is a pair consisting of a λ-expression and an environment mapping its free variables to values. A single λ-expression may thus produce many different closures. An abstract closure $v = \langle l, \rho, \kappa \rangle$ approximates a set of exact closures created from the same λ-expression.

Several exact closures are mapped to a single abstract closure by collapsing their environments. An abstract closure's environment maps variables to *contours* rather than values. Contours are finite strings of labels. We can make different choices for the set of contours, and the set of contours selected governs the precision of the analysis. In call-string based analyses, contour labels identify application sites. In our analysis, contour labels identify either let-expressions or uses of let-bound variables.

A *flow analysis* of a program $e_0^{l_0}$ is a function

$$F : \mathit{ProgramPoint} \rightarrow \; 2^{\mathit{Values}}$$

that maps *program points* to sets of abstract values. A program point is either a $\mathit{Var} \times \mathit{Contour}$ or $\mathit{Label} \times \mathit{Contour}$ pair. $\mathit{Var} \times \mathit{Contour}$ pairs represent bindings constructed by function applications. $\mathit{Label} \times \mathit{Contour}$ pairs represent the results of expressions. Informally, a program point associates an abstract program state with an identifier or expression.

F is a flow analysis of a program $e_0^{l_0}$ if $\mathcal{A}[\![e_0^{l_0}]\!]\rho_0\kappa_0$ holds, where $\mathcal{A}$ is the relation defined in Fig. 1 and implicitly parameterized by F, and ρ_0 and κ_0 are a specific initial environment and initial contour. There are many functions which are flow analyses of $e_0^{l_0}$. The *least flow analysis* is the least such function. Intuitively, $\mathcal{A}$ specifies constraints on a graph defined by F. Edges in the graph correspond to subset constraints, and nodes in the graph correspond to program points. The addition of new information is modeled in the framework in terms of satisfiability of these constraints. The following theorem establishes the existence of a least function F.

$$\mathcal{A}[\![c^l]\!]\rho\kappa \Rightarrow l \in F\langle l, \kappa\rangle$$

$$\mathcal{A}[\![(\lambda x.e)^l]\!]\rho\kappa \Rightarrow \langle l, \rho|_{FV(\lambda x.e)}, \kappa\rangle \in F\langle l, \kappa\rangle$$

$$\mathcal{A}[\![x^l]\!]\rho\kappa \Rightarrow F\langle x, \rho(x)\rangle \subseteq F\langle l, \kappa\rangle$$

$$\begin{array}{rl}
\mathcal{A}[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]\rho\kappa \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa \text{ and whenever } F\langle l_1, \kappa\rangle \neq \emptyset \\
& \quad \mathcal{A}[\![e_2^{l_2}]\!]\rho\kappa \text{ and whenever } F\langle l_2, \kappa\rangle \neq \emptyset \\
& \qquad \mathcal{A}[\![e_b^{l_b}]\!]\rho'[x \mapsto \kappa']\kappa' \text{ and} \\
& \qquad F\langle l_2, \kappa\rangle \subseteq F\langle x, \kappa'\rangle \text{ and} \\
& \qquad F\langle l_b, \kappa'\rangle \subseteq F\langle l, \kappa\rangle \\
& \qquad \text{where } (\lambda x.e_b^{l_b})^{l'} \in e_0^{l_0} \\
& \quad \text{for all } \langle l', \rho', \kappa'\rangle \in F\langle l_1, \kappa\rangle
\end{array}$$

$$\begin{array}{rl}
\mathcal{A}[\![(\mathsf{let}\ x = e_1^{l_1}\ \mathsf{in}\ e_2^{l_2})^l]\!]\rho\kappa \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa' \text{ and whenever } F\langle l_1, \kappa'\rangle \neq \emptyset \\
& \quad F\langle l_1, \kappa'\rangle \subseteq F\langle x, \kappa\rangle \text{ and} \\
& \quad F\langle l_2, \kappa\rangle \subseteq F\langle l, \kappa\rangle \text{ and} \\
& \quad \mathcal{A}[\![e_2^{l_2}]\!]\rho[x \mapsto \kappa]\kappa \\
& \text{where } \kappa' = \kappa l
\end{array}$$

$$\begin{array}{rl}
\mathcal{A}[\![x_{[l_0]}^l]\!]\rho\kappa \Rightarrow & l' \in F\langle l, \kappa\rangle \\
& \quad \text{for all } l' \in F\langle x, \rho(x)\rangle \\
& \langle l', \rho', \kappa'[l_0/l]\rangle \in F\langle x, \rho(x)\rangle \\
& \quad \text{for all } \langle l', \rho', \kappa'\rangle \in F\langle x, \rho(x)\rangle
\end{array}$$

$$\begin{array}{rl}
\mathcal{A}[\![\mathsf{if}\ e_1^{l_1}\ \mathsf{then}\ e_2^{l_2}\ \mathsf{else}\ e_3^{l_3}]\!]\rho\kappa \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa \text{ and whenever } F\langle l_1, \kappa\rangle \neq \emptyset \\
& \quad \mathcal{A}[\![e_2^{l_2}]\!]\rho\kappa \text{ and} \\
& \quad \mathcal{A}[\![e_3^{l_3}]\!]\rho\kappa \text{ and} \\
& \quad F\langle l_2, \kappa\rangle \subseteq F\langle l, \kappa\rangle \text{ if } \mathsf{true}^{l'} \in e_0^{l_0} \text{ and } l' \in F\langle l_1, \kappa\rangle \\
& \quad F\langle l_3, \kappa\rangle \subseteq F\langle l, \kappa\rangle \text{ if } \mathsf{false}^{l'} \in e_0^{l_0} \text{ and } l' \in F\langle l_1, \kappa\rangle
\end{array}$$

The notation $\rho[x \mapsto \kappa]$ means the functional update or extension of ρ at x to κ. The notation $\kappa[l/l']$ means the contour derived by replacing l with l' in κ.

Fig. 1. Relation $\mathcal{A}$

Theorem (Least Flow Analysis). *Given environment ρ_0, contour κ_0 and closed expression $e_0^{l_0}$, there exists a unique minimal function F such that $\mathcal{A}[\![e_0^{l_0}]\!]\rho_0\kappa_0$ holds.*

Proof Sketch. Since the analysis only manipulates a finite set of values, we can enumerate the set S of functions F for which $\mathcal{A}$ holds, and impose a natural partial ordering on S. Suppose F_1 and F_2 are in S. By the structure of the constraint rules, we show that $\mathcal{A}$ must also hold for $F_1 \cap F_2$. ∎

The application rule introduces two constraints. The first requires that the abstract value of the argument be a subset of the value of the formal; in other words, information flows from the actual parameter of the call to the function's formal. The second requires that the abstract value of the function body evaluated at this call site be a subset of the abstract value of the application; in other words, information flows from the function body to the call site. Besides these two constraints, the rule introduces an explicit strictness ordering. If the abstract value of the function position at a call is unspecified, *i.e.* no information flows into the node represented by the corresponding program point, the argument position need not be evaluated. A similar strictness constraint is introduced to cutoff evaluation of the function body if the abstract value of the argument position is unspecified. This strictness constraint corresponds to a reachability assertion [10], and significantly reduces the number of abstract values generated by the analysis.

3.1 Polymorphic Splitting

The last two rules of Fig. 1 for let-expressions and let-bound variables are the only rules that introduce new contours. For programs that do not use let-expressions, contours are essentially useless—all expressions are evaluated in the initial contour. Thus, in the absence of let, the analysis computes the same information as a 0CFA analysis. Polymorphic splitting uses different contours created at let-expressions and let-bound variables to disambiguate different uses of a let-bound procedure.

A let-binding evaluates in a new contour κ' which is constructed by appending the let-expression's label l to the let-expression's contour κ. The maximum length of a contour string is therefore the maximum nesting depth of let-bindings. Any closures created while evaluating the let-binding will capture the extended contour κ' as their third component (see the rule for λ-expressions). To illustrate, we adapt the second example from Section 2 to our formal core language:

$$(\mathsf{let}\ f = (\lambda x.x^{l_1})^{l_2}\ \mathsf{in}\ ((\lambda d.(f_{[l_0]}^{l_3}\ 1^{l_4})^{l_5})^{l_6}\ (f_{[l_0]}^{l_7}\ \#\mathsf{t}^{l_8})^{l_9})^{l_{10}})^{l_0}$$

The let-binding evaluates in contour $\kappa_0 l_0$, and returns the abstract value set $\{\langle l_2, \emptyset, \kappa_0 l_0\rangle\}$. This abstract value set is bound to f in program point $\langle f, \kappa_0\rangle$.

When a let-bound variable $x_{l'}^{l}$ is used, abstract closures bound to the variable are split. For each abstract closure c that captured the let-expression label l', we define a new closure c' whose contour is derived by substituting l for l' in c's contour component. We return the new closure c' rather than c. In the example above, the abstract value set for f at $f_{[l_0]}^{l_3}$ is $\{\langle l_2, \emptyset, \kappa_0 l_0\rangle\}$. We split the closure in this set by substituting l_3 for l_0 in the closure's contour, yielding $\{\langle l_2, \emptyset, \kappa_0 l_3\rangle\}$ as the abstract value set for $f_{[l_0]}^{l_3}$. Similarly, $\{\langle l_2, \emptyset, \kappa_0 l_7\rangle\}$ is the abstract value set for $f_{[l_0]}^{l_7}$. When these closures are applied, their argument bindings are created in different contours ($\kappa_0 l_3$ and $\kappa_0 l_7$) and the closures evaluate in different contours. The analysis thereby avoids merging bindings from the two calls.

As another example, consider the expression

$$((\lambda f.((f\ \mathsf{0})\ (f\ \mathsf{true})))\ (\lambda x.(\lambda y.x)))$$

which evaluates to 0, and whose least flow analysis is shown in Fig. 2. The set of abstract values for the result of this expression (program point $\langle l_0, \kappa_0\rangle$) is $\{l_2,\ l_5\}$. The analysis produces an imprecise result because the two applications $(f\ \mathsf{0})^{l_3}$ and $(f\ \mathsf{true})^{l_6}$ each create the same abstract closure. In an exact semantics, these two applications create different closures binding different values for their free variable x.

$$\begin{array}{llll}
\langle l_1, \kappa_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0\rangle\} & & \langle l_9, \kappa_0\rangle \mapsto \{l_2,\ l_5\} \\
\langle l_2, \kappa_0\rangle \mapsto \{l_2\} & & \langle l_{10}, \kappa_0\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0\rangle\} \\
\langle l_3, \kappa_0\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0\rangle\} & & \langle l_{11}, \kappa_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0\rangle\} \\
\langle l_4, \kappa_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0\rangle\} & & \langle l_0, \kappa_0\rangle \mapsto \{l_2,\ l_5\} \\
\langle l_5, \kappa_0\rangle \mapsto \{l_5\} & & \langle f, \kappa_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0\rangle\} \\
\langle l_6, \kappa_0\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0\rangle\} & & \langle x, \kappa_0\rangle \mapsto \{l_2,\ l_5\} \\
\langle l_7, \kappa_0\rangle \mapsto \{l_2,\ l_5\} & & \langle y, \kappa_0\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0\rangle\} \\
\langle l_8, \kappa_0\rangle \mapsto \{\langle l_8, \emptyset, \kappa_0\rangle\} & &
\end{array}$$

Fig. 2. The least flow analysis of
$$e_0^{l_0} \ = \ ((\lambda f.((f^{l_1}\ \mathsf{0}^{l_2})^{l_3}\ (f^{l_4}\ \mathsf{true}^{l_5})^{l_6})^{l_7})^{l_8}\ (\lambda x.(\lambda y.x^{l_9})^{l_{10}})^{l_{11}})^{l_0}$$

Rewriting the above example using a let-expression enables polymorphic splitting to take place:

$$\mathsf{let}\ \ f = \lambda x.\lambda y.x\ \ \mathsf{in}\ \ ((f\,\mathsf{0}\,)\ (f\,\mathsf{true}\,))$$

Fig. 3 presents the least flow graph of this rewritten expression. Several closures are now created for f with contours $\kappa_0 l_0$, $\kappa_0 l_1$, and $\kappa_0 l_4$. Since there are now two separate bindings for x corresponding to the two arguments passed to f, this analysis yields a more precise result.

3.2 Comparison to Type Inference

For several important idioms, polymorphic-splitting results in finer precision than Hindley-Milner typing or safety analysis [16] as embodied by a 0CFA analysis. As a simple illustration, consider the expression:[2]

[2] The examples in this section use natural extensions for begin and primitive operators.

$$\begin{array}{ll}
\langle l_1, \kappa_0 l_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_1\rangle\} & \langle l_9, \kappa_0 l_1\rangle \mapsto \{l_2\} \\
\langle l_2, \kappa_0 l_0\rangle \mapsto \{l_2\} & \langle l_{10}, \kappa_0 l_1\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_1], \kappa_0 l_1\rangle\} \\
\langle l_3, \kappa_0 l_0\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_1], \kappa_0 : l_1\rangle\} & \langle l_{10}, \kappa_0 l_4\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_4], \kappa_0 l_4\rangle\} \\
\langle l_4, \kappa_0 l_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_4\rangle\} & \langle l_{11}, \kappa_0 l_0\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_0\rangle\} \\
\langle l_5, \kappa_0 l_0\rangle \mapsto \{l_5\} & \langle l_0, \kappa_0 l_0\rangle \mapsto \{l_2\} \\
\langle l_6, \kappa_0 l_0\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_4], \kappa_0 l_4\rangle\} & \langle f, [l_0]\rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_0\rangle\} \\
\langle l_7, \kappa_0 l_0\rangle \mapsto \{l_2\} & \langle x, [l_1]\rangle \mapsto \{l_2\} \\
 & \langle x, [l_4]\rangle \mapsto \{l_5\} \\
 & \langle y, [l_1]\rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_4], \kappa_0 l_4\rangle\}
\end{array}$$

Fig. 3. The least flow analysis of
$e_0^{l_0} \;=\; (\mathsf{let}\ f_{l_0} = (\lambda x.(\lambda y.x^{l_9})^{l_{10}})^{l_{11}}\ \ \mathsf{in}\ \ ((f_{[l_0]}^{l_1}\ \mathsf{0}^{l_2})^{l_3}\ (f_{[l_0]}^{l_4}\ \mathsf{true}^{l_5})^{l_6})^{l_7})^{l_0}$

$$\begin{array}{l}
(\mathsf{let}\ \ h = (\lambda x.\lambda y. \\
\qquad\qquad \mathsf{if}\ \ x^{l_1} \\
\qquad\qquad \mathsf{then}\ \ \mathsf{false} \\
\qquad\qquad \mathsf{else}\ \ y^{l_2} + 1)^{l_3} \\
\ \mathsf{in}\ \ \mathsf{begin} \\
\qquad (1 + (h_{[l_0]}^{l_4}\ \mathsf{false}\ \ 1)^{l_5})^{l_6} \\
\qquad (h_{[l_0]}^{l_7}\ \mathsf{true}\ \ \text{"foo"})^{l_8} \\
\quad\ \mathsf{end}\ )^{l_0}
\end{array}$$

Under a standard Hindley-Milner type discipline, this program would fail to type-check for two reasons. First, the branches of the conditional have different types. Second, y is assumed to be of type Int based on the context in which it is used in the procedure body; the second call to h would violate this assumption. Under a 0CFA analysis, the two calls to h would be merged producing an abstract value for y that contains both 1 (from the first call) and "foo" (from the second). Depending on how the analysis is used, this merging would either lead to the program being rejected, or would result in run-time type-checks being required at both calls to h, and in the addition operation within h's body.

Both static type inference and 0CFA ignore inter-variable dependencies. Consequently, they are unable to capture the fact that y is an integer precisely when x is false . Flow analyses employing polymorphic splitting capture this information. In particular, the abstract values at all program points with label l_2 do not include the symbol "foo" since control never reaches the false branch of the conditional in the second call. Moreover, the abstract value at all program points with label l_8 will contain only false, thus allowing such calls to be used in a Boolean context even though the conditional yields an integer in the false branch. This kind of precision captures a form of polymorphism well-suited to Scheme programs, and has a direct impact on performance.

Although polymorphic splitting records inter-variable dependencies, the approximation defined by relation $\mathcal{A}$ does lose precision relative to Hindley-Milner typing in some cases. In particular, free occurrences of a polymorphic procedure defined within another may get merged because of the way contours are extended and substituted. Consider the following example:

$$
\begin{array}{l}
(\mathsf{let}\ \ f = (\lambda\, x.\, x^{l_f}) \\
\ \mathsf{in}\ \ (\mathsf{let}\ \ g = \lambda x.\lambda y.\ \ldots (f^{l_5}\ x)\ \ldots (f^{l_6}\ y)\ \ldots \\
\qquad \mathsf{in}\ \ (g^{l_3}\ \ldots) \\
\qquad\quad (g^{l_4}\ \ldots))^{l_1})^{l_0}
\end{array}
$$

To faithfully model polymorphism, we require f to be evaluated in four distinct contours. However, because g's closure binds f to contour $\kappa_0 l_0$, the first occurrence of f in g will be merged over the two calls to g, as will the second. For example, the abstract value yielded by evaluating the first reference to f in g will be a set containing the abstract closure $\langle l_f, \rho_0, [\kappa_0 l_5]\rangle$; the evaluation of this occurrence of f in the second call to g will be the same.

It is possible to extend $\mathcal{A}$'s definition to disambiguate polymorphic references such that it fully captures the behavior of Hindley-Milner type inference. For example, by simply evaluating the body of a let-expression in the same (extended) contour that the corresponding let-binding is evaluated within, we would avoid the inappropriate merging in the above example. Although we can clearly make $\mathcal{A}$'s definition more precise, it is not clear whether the greater disambiguation provided by this added precision can be realized without making the implementation of the analysis too expensive for realistic programs. Our experimental results indicate that the potential loss of precision for programs that have nested polymorphic procedures does not compromise the practical utility of polymorphic splitting in eliminating run-time checks.

4 Implementation

The implementation of the analysis and the run-time type-check optimization consists of approximately 4000 lines of Scheme code. The output of the analysis is used to control the placement of three kinds of run-time checks:

1. An *arity check* is required at a λ-expression if the procedure may be applied to an inappropriate number of arguments.
2. An *application check* is required at an application if the expression in the call position may yield a value other than a closure.
3. A *primitive check* is required at a primitive operation if the operation may be applied to a value of inappropriate type.

The analysis operates over the entire Scheme language, and thus supports variable-arity procedures, continuations, data structures, and assignment. While most extensions to the functional core are straightforward, there are several worthy of mention:

1. *Data Structures.* Elements of Scheme data structures can be mutated. The analysis tracks such assignments by recording them in the program point that holds the value of the corresponding sub-expression. Unlike other static type systems proposed for Scheme [22], assigning to data structures causes no loss in precision. Assigning to a field in a pair, for example, simply augments the abstract value set for that field.
2. *Implicit Allocation.* Certain Scheme constructs implicitly allocate storage. The most obvious example is variable-arity procedures, *i.e.*, procedures that take an optional number of arguments. Optional arguments are bundled into lists which are implicitly allocated and bound to a *rest* argument when the procedure is applied. Since expressions in the body of the procedure can walk down the rest argument using regular list operations, the implementation uses extra labels at each application to hold rest arguments.
3. *Type Predicates and Conditionals.* A common Scheme idiom is to use type predicates in conditional tests to guarantee that variables have a desired type in appropriate branches of the conditional. Failure to take such type predicates into account in the analysis can lead to unnecessary merging of abstract value sets and loss of precision. Our implementation recognizes type predicates that satisfy simple syntactic conditions, and evaluates the branches of a conditional in separate environments. These environments bind the variable upon which the type predicate test is performed to different abstract values consistent with the predicate check. (This environment splitting is *not* reflected in Fig. 1.)

The implementation constructs the least flow analysis for a program $e_0^{l_0}$ by building a graph in which *nodes* represent program points, nodes *contain* sets of abstract values, and *edges* represent containment constraints between sets of values at nodes. The algorithm constructs the graph incrementally by propagating values across edges to successor nodes and building new pieces of the graph as new closures arrive at applications. The algorithm maintains a work list of nodes with values that have not yet been propagated to all successors of a node, and proceeds by repeatedly picking a node from the work list and propagating its values to its successors. The algorithm terminates when the work list is empty.

Procedure Flow, presented in Fig. 4, constructs the graph for an expression reached in a particular environment and contour. For constants and λ-expressions, Flow just adds the abstract value to program point $\langle l, \kappa \rangle$. For variable references, Flow adds a simple edge $(\langle \cdot, \cdot \rangle \rightsquigarrow \langle \cdot, \cdot \rangle)$ between the appropriate program points. For applications, Flow constructs the graph for e_1, and then adds a special *application edge* $(\langle \cdot, \cdot \rangle \rightsquigarrow \lambda \cdots)$ from the program point representing the result of e_1. The target of this edge is a procedure that is invoked for every value propagated across the edge; *i.e.* for every abstract value computed by e_1. For every such value that is a closure, Flow constructs the graph for e_2, memoizing to ensure that this subgraph is built only once. Flow also constructs the graph for the body of the λ-expression being invoked, adds an edge to propagate argument values to the formal parameter, and adds an edge to propagate result values back to the call site.

$$
\begin{array}{ll}
\mathrm{Flow}(c^l, \rho, \kappa) & = \text{add } l \text{ to } \langle l, \kappa\rangle \\
\mathrm{Flow}((\lambda x.e)^l, \rho, \kappa) & = \text{add } \langle l, \rho|_{FV(\lambda x.e)}, \kappa\rangle \text{ to } \langle l, \kappa\rangle \\
\mathrm{Flow}(x^l, \rho, \kappa) & = \text{add } \langle x, \rho(x)\rangle \rightsquigarrow \langle l, \kappa\rangle \\
\mathrm{Flow}((e_1^{l_1}\ e_2^{l_2})^l, \rho, \kappa) & = \mathrm{Flow}(e_1^{l_1}, \rho, \kappa) \\
& \quad \text{add } \langle l_1, \kappa\rangle \rightsquigarrow \lambda v. \\
& \qquad\qquad \text{if } v = \langle l', \rho', \kappa'\rangle \text{ where } (\lambda x.e_b^{l_b})^{l'} \in e_0^{l_0} \\
& \qquad\qquad\quad \mathrm{MemoFlow}(e_2^{l_2}, \rho, \kappa) \\
& \qquad\qquad\quad \mathrm{MemoFlow}(e_b^{l_b}, \rho'[x \mapsto \kappa'], \kappa') \\
& \qquad\qquad\quad \text{add } \langle l_2, \kappa\rangle \rightsquigarrow \langle x, \kappa'\rangle \\
& \qquad\qquad\quad \text{add } \langle l_b, \kappa'\rangle \rightsquigarrow \langle l, \kappa\rangle
\end{array}
$$

Fig. 4. The procedure Flow builds a flow graph corresponding to the least flow analysis of a program.

5 Performance

Our implementation runs on top of Chez [4], a commercially available implementation of Scheme. At optimize-level 3, Chez eliminates almost all run-time checks, making no safety guarantees. By feeding the output of our analysis to a procedure that inserts explicit run-time checks based on the categories described in the previous section, the resulting program can be safely executed at optimize-level 3.

We have used the analysis to eliminate run-time checks from moderately sized Scheme programs. Fig. 5 lists the benchmarks used to test the analysis, their size in number of lines of code, the number of sites where run-time checks would ordinarily be required in the absence of any optimizations, and the time to analyze them under polymorphic splitting, soft typing [22], a 0CFA implementation, and a 1CFA implementation. The times were gathered on a 150 MHz MIPS R4400 with 1 GByte of memory.

The program Lattice enumerates the lattice of maps between two lattices, and is purely functional. Browse is a database searching program that allocates extensively. The program Check is a simple static type checker for a subset of Scheme. Graphs counts the number of directed graphs with a distinguished root and k vertices, each having out-degree at most 2. This program makes extensive use of mutation and vectors. Boyer is a term-rewriting theorem prover that allocates heavily. N-Body is a Scheme implementation [23] of the Greengard multipole algorithm [6] for computing gravitational forces on point-masses distributed uniformly in a cube. Dynamic is an implementation of a tagging optimization algorithm [8] for Scheme. Nucleic is a constraint satisfaction algorithm used to determine the three-dimensional structure of nucleic acids [5]. It is floating-point intensive and uses an object package implemented using macros and vectors. Nucleic2 is a modified version of Nucleic described below.

For several of the benchmarks, the analysis time required by soft typing is less than the time required for any of the flow analyses. Because soft typing

Benchmark	Lines	Sites	Analysis Time (in seconds)			
			Polymorphic Splitting	Soft Typing	0CFA	1CFA
Lattice	215	252	.22	.46	.15	.67
Browse	233	281	.21	.96	.26	2.57
Check	278	377	1.64	1.76	9.14	132.95
Graphs	621	411	.30	.73	.20	1.13
Boyer	632	211	1.48	3.69	31.03	71.01
N-Body	874	1227	0.89	2.75	.36	8.14
Dynamic	2331	2369	81.77	4.68	401.42	*
Nucleic	3334	910	32.28	6.44	34.70	34.34
Nucleic2	3264	1126	181.58	21.67	174.53	195.04

* For Dynamic, the 1CFA analysis exhausted heap space.

Fig. 5. Benchmark programs, their size (in lines of code), static incidences of run-time checks for these programs in the absence of any run-time check optimization, and analysis times under polymorphic splitting, soft typing, 0CFA, and 1CFA.

is based on a Hindley-Milner type inference framework [9, 12], the body of a λ-expression is evaluated only once. Applications unify the type signature for a procedure's arguments with the type inferred for the formal, and thus do not require re-analysis of the procedure body. In contrast, the implementation of the constraint rules specified for our flow analysis propagates the value of a procedure's argument to the sites where it is referenced within the procedure body. Thus, expressions within a procedure may be evaluated each time an application of the procedure is evaluated. Despite the potential for re-evaluation of expressions in a λ-body, the analysis times for polymorphic splitting are reasonably close to that of soft typing; Dynamic, Nucleic, and Nucleic2 are exceptions.

In many cases, the analysis time for polymorphic splitting is less than the analysis time for 0CFA. On the surface, this is counter-intuitive: one might imagine a more precise analysis would have greater cost in analysis time. The reason for the significant difference in analysis times is because 0CFA yields coarser approximations, and thus induces more merging. More merging leads to more propagation, which in turn leads to more re-evaluation. Consider an abstract procedure value P. This value is represented as a set of abstract closures. As the size of this set becomes bigger, abstract applications of P require analyzing the body of more λ-expressions since the application rule applies *each* element in the set defined by P. Because polymorphic splitting results in less merging than 0CFA, abstract value sets are smaller, which leads to shorter analysis times.

Fig. 6 shows the percentage of *static* checks remaining in these benchmarks under the different analyses after run-time check optimization has been applied. The static counts measure the textual number of run-time checks remaining in the programs. Fig. 7 shows the percentage of *dynamic* checks remaining in the benchmarks. The dynamic counts measure the number of times these checks are evaluated in the actual execution of the program. Dynamic counts provide a reasonable metric to determine the effectiveness of an analysis—a good analysis will eliminate run-time checks along control-paths most likely to be exercised during program execution.

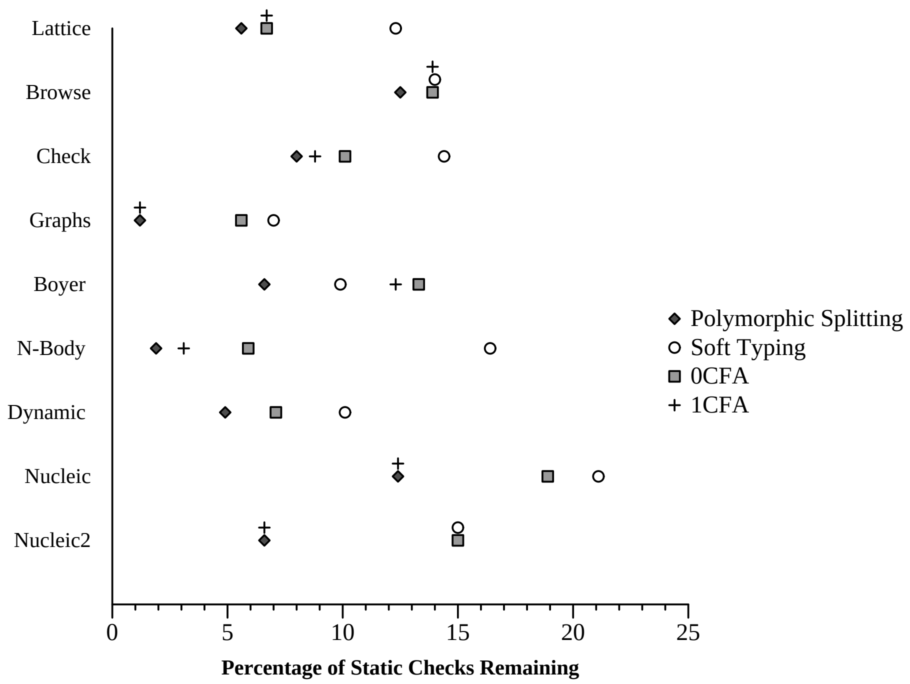

Fig. 6. Percentage of static checks remaining after run-time check optimization.

For each of the benchmarks, polymorphic splitting requires fewer run-time checks than either 0CFA or soft typing. In many cases, the difference is significant. Interestingly, 0CFA also outperforms soft typing on several benchmarks. For example, the `Lattice` program has roughly half as many static checks when analyzed using 0CFA than using soft typing. The primary reason for this is the problem of *reverse flow* [22] in the soft-typing framework that leads to imprecise typing. Reverse flow refers to type information that flows both with and counter to the direction of value flow. Conditionals often cause reverse flow because the type rules for a conditional require both its branches to have the same type. Consequently, type information specific to one branch may cross over to the other, yielding a weaker type signature than necessary for the entire expression. Because of reverse flow, a cascading effect can easily occur in which many unnecessary run-time checks are required because of overly conservative type signatures.

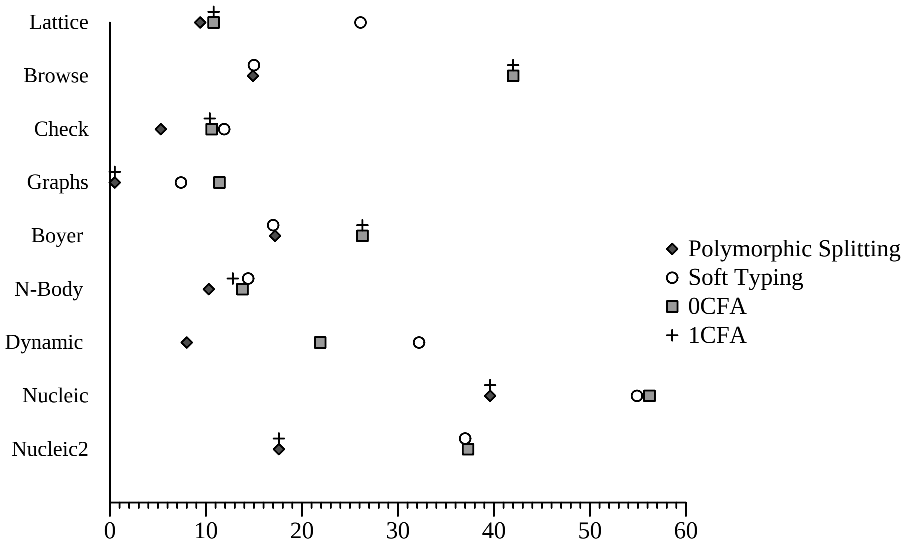

Fig. 7. Percentage of dynamic checks remaining after run-time check optimization.

The dynamic check count figures indicate that polymorphic splitting eliminates run-time checks at important program sites. The improvement over soft typing is more striking here. For example, about one third as many run-time checks are encountered during the execution of Lattice optimized under polymorphic splitting than under soft typing. About one quarter as many checks are executed for Dynamic. Here also, a simple 0CFA flow analysis generally does better than soft typing, although for some benchmarks like Browse, Boyer, and Graphs, 0CFA is worse. This indicates that the approximation used by polymorphic splitting is superior to 0CFA. Insofar as the benchmark programs use common Scheme idioms and programming styles, we conclude that Scheme programs use polymorphism in interesting and non-trivial ways. Analyses which are sensitive to polymorphism will outperform those that are not.

The counts of static and dynamic checks for Nucleic are significantly higher than for the other benchmarks. Nucleic implements a structure package that supports a simple form of inheritance; the main data objects manipulated by the program are defined in terms of structure instances. Structures are implemented as vectors, and tags (represented as symbols) are interspersed in this vector to demarcate distinct substructures in the inheritance hierarchy. Since the analyses do not retain distinct type information for individual vector slots, references to these structures invariably incur a run-time check. Thus, although the vast majority of references in the program are to floating-point data, all the analyses require run-time checks at arithmetic operations that are applied to elements of these structures. Nucleic2 is a modified version of Nucleic that uses an alternative representation for structures. This representation separates the tags from the actual data stored in structure slots, enabling the analyses to avoid run-time checks when extracting elements from structures. Nucleic2 incurs significantly fewer run-time checks than Nucleic for all analyses, although these improvements come at the cost of increased analysis times.

6 Conclusions and Related Work

Flow analysis can enable and facilitate a number of important optimizations necessary to implement realistic high-level languages efficiently. Run-time check elimination is one such example that is relevant in the context of languages such as Scheme or ML.

Although the flow analysis problem has been well-studied [11], and although parameterizable systems have been investigated elsewhere [10, 20], the applicability of such frameworks for optimizing realistic programs has enjoyed relatively little examination. When viewed in the context of run-time check elimination, flow analysis bears an interesting relationship to type inference [17]. However, there has been little work on extending the relation to polymorphism or to understand its implications on implementations.

One important kind of run-time check optimization is elimination of type tags [21]. Shivers [19] used the term *type recovery* to describe a type-tag elimination optimization based on flow analysis. Because this framework relies on call-string abstractions, and did not study the possibility of exploiting polymorphism, its success was limited. Moreover, since no attempt was made to implement the interpretation for the entire Scheme language, making a quantified assessment of its utility is difficult. Henglein [8] describes a tagging optimization based on type inference. While the analysis can reduce tagging overheads in many Scheme programs, it does not consider polymorphism or union types, and uses a coarse type approximation that is significantly less precise than control-flow analysis obtained via polymorphic splitting.

Soft Scheme [22] is a Scheme implementation that employs traditional type inference techniques to derive type information which can be then used to eliminate or obviate run-time checks. The type system used in Soft Scheme (called *soft typing*) is a generalization of static type checking that accommodates both dynamic typing and static typing in one framework. A soft type checker infers types for identifiers and inserts run-time checks to transform untypable programs to typable form. Aiken *et al.* [1] describe a more sophisticated soft type system for a functional language. This system uses conditional types in a more powerful type language that should yield more a precise analysis than Soft Scheme's type system, but no implementation is available for a realistic programming language.

Besides type inference and abstract interpretation, there has been recent work on using constraint systems [7, 15] to analyze high-level programs. These systems are based on an operational semantics that ignores all inter-variable dependencies. Consequently, while efficient implementations of these analyses can be built, it is unclear whether they provide the necessary precision to perform useful run-time check optimizations. Refinements on these approaches that take into account polymorphism are possible [7], but are *ad hoc* and do not fit neatly within the constraint framework.

We conclude that flow analysis offers the possibility of distilling more precise information useful for run-time check elimination than unification-based type inference procedures. Because flow analysis tracks dynamic control-flow, it can avoid incorporating information propagated from dead or unreachable code in abstract values. Furthermore, because these analyses record the creation points of values, they can be used to build a more refined notion of the set of values that can be associated with a given program point. This refinement can facilitate other kinds of optimizations beside run-time check elimination. Moreover, in sharp contrast to traditional abstract interpretation systems, a direct implementation of the least flow graph generated by the constraint rules can be easily applied to analyze incrementally-defined programs.

Our results indicate that flow analysis offers a promising platform upon which to implement run-time check elimination. If the framework is equally adept in supporting other optimizations, we expect to use the ideas upon which it is based in an optimizing compiler for Scheme.

References

1. Alexander Aiken and Wimmers, Edward and T.K Lakshman. Soft Typing with Conditional Types. In *21th ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.
2. Henk Barendregt. *The Lambda Calculus*. North-Holland, Amsterdam, 1981.
3. William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.
4. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., 1987.
5. Marc Feeley, Marc Turcotte, and Guy Lapalme. Using MultiLisp for Solving Constraint Satistfaction Problems: An Application to Nucleic Acid 3D Structure Determination. *Lisp and Symbolic Computation*, 7(2/3):231–248, 1994.
6. Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. ACM Press, 1987.
7. Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 306–317, 1994.
8. Fritz Henglein. Global Tagging Optimization by Type Inference. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 205–215, 1992.
9. R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
10. Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *22th ACM Symposium on Principles of Programming Languages*, pages 392–401, January 1995.
11. Neil Jones and Stephen Muchnick. Flow Analysis and Optimization of Lisp-like Structures. In *6th ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.
12. Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
13. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

14. John Mitchell. *Handbook of Theoretical Computer Science: Volume B*, chapter Type Systems for Programming Languages, pages 367–453. MIT Press, 1990.
15. Jens Palsberg. Global Program Analysis in Constraint Form. In *Proceedings of the 1994 Colloquium on Trees in Algebra and Programming*, pages 276–290. Springer-Verlag, 1994. Appears as LNCS 787.
16. Jens Palsberg and Patrick O'Keefe. A Type System Equivalent to Flow Analysis. In *22^{th} ACM Symposium on Principles of Programming Languages*, pages 367–378, January 1995.
17. Jens Palsberg and Michael Schwartzbach. Safety Analysis versus Type Inference. *Information and Computation, to appear.*
18. Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
19. Olin Shivers. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation.* MIT Press, 1990.
20. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.
21. Peter Steenkiste and John Hennessy. Tags and type checking in lisp. In *Proceedings of the Second Architectural Support for Programming Languages and Systems Symposium*, pages 50–59, 1987.
22. Andrew Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 250–262, 1994.
23. Feng Zhao. 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.