Polymorphic Splitting: An Effective Polyvariant Flow Analysis

ANDREW K. WRIGHT
and
SURESH JAGANNATHAN
NEC Research Institute

This paper describes a general-purpose program analysis that computes global control-flow and data-flow information for higher-order, call-by-value languages. The analysis employs a novel form of polyvariance called *polymorphic splitting* that uses let-expressions as syntactic clues to gain precision. The information derived from the analysis is used both to eliminate run-time checks and to inline procedures.

The analysis and optimizations have been applied to a suite of Scheme programs. Experimental results obtained from the prototype implementation indicate that the analysis is extremely precise and has reasonable cost. Compared to monovariant flow analyses such as 0CFA, or analyses based on type inference such as soft typing, the analysis eliminates significantly more run-time checks. Run-time check elimination and inlining together yield an average performance improvement of 40% for the programs in the benchmark suite, with some programs running four times as fast.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Applicative Languages*; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General terms: Performance, Experimentation

Additional Key Words and Phrases: Flow analysis, polyvariance, run-time checks, inlining

1. INTRODUCTION

Advanced programming languages such as Scheme [Clinger, Rees, et al. 1991] and ML [Milner, Tofte, and Harper 1990] 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.

This paper combines and extends two conference papers that use flow analysis based on based on polymorphic splitting to eliminate run-time checks [Jagannathan and Wright 1995] and to inline procedures [Jagannathan and Wright 1996].
Authors' Addresses: NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. email: wright@research.nj.nec.com, suresh@research.nj.nec.com.

Interprocedural optimizations require interprocedural control-flow 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 program optimizations such as run-time check elimination, inlining, and unboxing that are global in nature, as well as optimizations such as constant folding and loop-invariant code motion that are typically based on special-purpose local analyses.

This paper presents a flow analysis framework for call-by-value, higher-order languages. The implementation of the framework is parameterized over different approximations of exact values to abstract values, and can thus be used to construct a spectrum of analyses with different cost and accuracy characteristics. A simple *monovariant analysis* such as 0CFA [Shivers 1991] or set-based analysis (SBA) [Heintze 1994] associates with each subexpression of the program a single *abstract value* describing its potential results. More precise *polyvariant* analyses [Jones and Muchnick 1979; Bulyonkov 1984] associate several abstract values with each subexpression, corresponding to the different contexts in which the subexpressions are reached. We introduce a novel polyvariance technique called *polymorphic splitting* that uses let-expressions as syntactic clues to provide increased precision. Polymorphic splitting borrows ideas from Hindley-Milner [Hindley 1969; Milner 1978] polymorphic type inference systems to create a form of polyvariance that mimics polymorphism. Polymorphic splitting allows a flow analysis to avoid merging information between unrelated calls to a polymorphic function, yielding more precise flow information.

To investigate its utility, we have used our analysis to drive two optimizations. The first is run-time check elimination. Dynamically typed languages such as Scheme require run-time checks at every use of a primitive operator. Furthermore, every function application requires a run-time check to ensure that the function subexpression evaluates to a procedure. Run-time checks are also needed in statically typed languages such as ML that allow operations to be applied to arguments whose domain is is smaller than their static type. For instance, the operation `hd` has type $\alpha\ \mathit{list} \rightarrow \alpha$, but may be applied only to non-empty lists. Run-time checks can add significant overhead to program execution, but checks at a given operator can be safely avoided if all possible values for the operator's arguments can be proven to lie within the operator's domain. As our analysis yields sets of abstract values that approximate the values of the arguments, it is easy to determine when a run-time check can be eliminated: the arguments' abstract values must denote a subset of the operator's domain.

The second optimization we have studied is inlining. Inlining trades code space for time by replacing a procedure call with the called procedure's body. Our flow analysis enables effective inlining because it identifies the set of procedures that may be invoked at each call site in a program. At a call-site, our analysis associates a set of abstract closures with the function position of the call. These abstract closures approximate the exact closures that may be applied at that call-site. (Informally, a single abstract closure might correspond to a set of exact closures having the same code pointer, but closed over different environments.) It is now easy to determine when a call site is a candidate for inlining: there must be a single abstract closure applied at the call-site. Our particular flow analysis based on polymorphic splitting also facilitates constructing different size estimates for a procedure at different call sites based on the portions of the procedure that may be reached from those sites. For example, large procedures can be aggressively inlined when static determination of conditional tests in their bodies permits entire blocks of code to be pruned.

The implementation of our analysis handles all of the constructs and operators specified in the R^4RS Scheme report, including variable-arity procedures, data structures, first-class continuations, and assignment. Experimental results for realistic Scheme programs indicate that polymorphic splitting is extremely precise and has reasonable cost. Furthermore, our polyvariant analysis is often cheaper to compute than less precise monovariant analyses.

We have used our analysis to optimize a collection of Scheme programs. We perform optimization by source-to-source transformation, and use Chez Scheme [Dybvig 1987] to compile the resulting programs to native machine code. Together, our run-time check elimination and inlining optimizations yield significant performance improvements over the unoptimized programs. Our optimizer yields an average improvement of 40% for the programs in our benchmark suite, with some programs running four times as fast.[1] Comparisons of polymorphic splitting to simple flow analyses (closure analysis [Palsberg 1995] and 0CFA [Shivers 1990]) and to soft-typing [Wright and Cartwright 1997] indicate that polymorphic splitting is significantly more effective than these analyses in eliminating run-time checks.

The remainder of the paper is organized as follows. The next section discusses our motivation for exploring flow analysis as an optimization tool. Section 3 presents a precise definition of our flow analysis framework, including a discussion of polymorphic splitting. The fourth section describes an algorithm for computing flow information. Section 5 describes our run-time check elimination and flow-directed inlining optimizations, and presents experimental results. Section 6 discusses related work, and Section 7 provides a summary.

2. MOTIVATION

Higher-order functional languages provide rich and expressive programming platforms. However, realizing efficient implementations of these languages has proven to be challenging. This is primarily because the abstract programming style encouraged by functional languages is significantly different from the programming style found in more traditional (imperative) languages. Because higher-order procedures, abstract datatypes, and sophisticated control abstractions (e.g., continuations) are the fundamental building blocks of functional languages, conventional compiler optimization techniques developed for imperative languages have little effect when applied to programs using these features. A basic premise underlying the motivation of this paper is that optimization strategies developed for imperative languages will be only marginally useful in building high performance implementations of functional languages *unless* they are used in conjunction with more powerful *interprocedural* analyses.

In conventional imperative languages, procedures are large and their call sites are

[1]The unoptimized programs were compiled by Chez Scheme under maximum safe optimization (optimize-level 2) after performing the same suite of local optimizations as for the optimized programs.

syntactically apparent. Hence intraprocedural local analyses can gather sufficient information to enable useful optimizations. Such local analyses can be adapted to obtain some useful information for functional languages. For example, syntactic heuristics can sometimes be used to identify the procedures that may be applied at a given call site. Thus, in the following Scheme expression,

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

it is easy to recognize that `f` is only applied at the call `(f 1)`. However, simple syntactic analyses are usually foiled by a slightly more complicated expression:

```
(let ([g (lambda (h) h)]
      [f (lambda (x) x)])
  ((g f) 1))
```

In order for a syntactic analysis to correctly determine that the call `(g f)` yields `f` and that `f` is only applied at `((g f) 1)`, the analysis must track how `f` is passed through `g`. Since procedure `g` might be arbitrarily complex, effective syntactic analysis of its body is impossible.

For higher-order languages, a flow analysis must construct some approximation to the set of procedures applied at a given a call-site, and the set of values returned from a call. Control-flow frameworks such as 0CFA compute such information. Such monovariant analyses are low-order polynomial-time analyses that effectively determine all potential call sites for all procedures, but *merge* the values of arguments from all call sites. In the above example, 0CFA will correctly indicate that `(g f)` can return only `f`. The analysis associates an *abstract value* with `(g f)` that describes procedure `f`. Such an abstract value might be a label that uniquely identifies the lambda-expression bound to `f`.

Because of the merging performed at calls and returns, monovariant analyses fail to separate the argument values supplied by the two calls to `f` in the following expression:

```
(let ([f (lambda (x) x)])
  (begin
    (+ (f 1) 0)
    (f #t)))
```

A monovariant analysis computes an abstract value for `x` that describes the set {`1`,`#t`}. Consequently, the abstract result of any application of `f` includes {`1`,`#t`}. Type-specific optimizations must therefore assume that `x` can be either an integer or a boolean. Hence monovariant analyses are too weak to eliminate the run-time check at `+`.

A more sophisticated polyvariant flow analysis avoids merging information from different calls by associating a distinct context with each call. For instance, the call-site where a procedure is applied can be used to disambiguate different bindings to the procedure's arguments. In the above example, such an analysis (commonly referred to as 1CFA) deduces that `f` is applied to `1` and returns `1` in the first call, and that `f` is applied to `#t` and returns `#t` in the second call. With 1CFA, two separate abstract values are constructed for `x`, one for each call site of `f`. Hence 1CFA is powerful enough to eliminate the run-time check at `+`.

1CFA is an instance of a polyvariant flow-analysis framework based on *call-strings*. 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, however, call-string based analyses are highly sensitive to program structure. It is easy to extend the above example so that 1CFA computes overly conservative information:

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

Here, 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 for `f`'s result that describes both `1` and `#t`. Correctly disambiguating the two calls to `f` made via the two calls to `g` requires a 2CFA analysis that preserves two levels of call history.

Another polyvariance technique that has been used in implementations of some object-oriented languages relies on the types of a procedure's or method's arguments to provide a disambiguation context [Chambers and Ungar 1990; Plevyak and Chien 1994]. Under such a *type-directed* analysis [Jagannathan, Weeks, and Wright 1997], the two calls to `g` in the above example are discriminated by their arguments' types. The first call to `g` is analyzed in the context $\langle \mathit{procedure}, \mathit{integer} \rangle$, while the second call is analyzed in $\langle \mathit{procedure}, \mathit{boolean} \rangle$. Because type-directed analyses rely on approximations to the *values* of a call's arguments (the arguments' types), type-directed analyses are less dependent on the syntactic structure of a program than call-string based analyses. Nonetheless, their effectiveness is also closely tied to how much procedure call history is preserved. While a type-directed analysis need preserve only one level of call history to avoid unwanted merging in the above example, it is easy to construct similar examples where more history is necessary.

In contrast to these flow analysis techniques, polymorphic type inference algorithms [Milner 1978] handle all of the above examples without undesirable merging. Polymorphic type systems disambiguate different calls to a polymorphic procedure by effectively duplicating the procedure wherever it is referenced. Polymorphic type systems are insensitive to the dynamic sequence of calls separating the construction of a value from its use. Consequently, they are less dependent on the syntactic structure of programs.

We therefore consider an alternative form of polyvariance called *polymorphic splitting* that uses let-expressions as clues to determine when to introduce new analysis contexts. 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 occurrences 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 the usual quantification rules for type variables are equivalent to a substitution rule [Mitchell 1990]:

$$\frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2[x \mapsto e_1] : \tau_2}{A \vdash \texttt{(let ([}x\ e_1\texttt{])}\ e_2\texttt{)} : \tau_2}$$

Polymorphic splitting captures the essence of polymorphism by incorporating this observation into a flow-analysis framework. Like polymorphic type inference, a flow analysis based on polymorphic splitting correctly disambiguates the two calls to `g` and `f` in the above example. The body of `f` is analyzed in four different contexts, just as using the above substitution rule would result in four copies of the body of `f`. Like polymorphic type inference, an analysis using polymorphic splitting has exponential worst-case complexity [Kanellakis, Mairson, and Mitchell 1991], but exhibits reasonable analysis times in practice.

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 the next section.

3. FLOW ANALYSIS FRAMEWORK

This section presents a formal definition of our flow analysis framework that obtains polyvariance through polymorphic splitting. We first describe a simple higher-order source language that includes let-expressions as a distinct syntactic form. We present a small-step operational semantics for an enriched version of this language that has explicit notation for closures and local environments. We then use a constraint relation to define a flow analysis of a program in this enriched language, and show a soundness result that relates the meaning of expressions as defined by the semantics to the abstract values computed by the flow analysis. Finally, we show how to accommodate recursive definitions, data constructors, and assignment in the analysis.

3.1 Source language

The abstract syntax for our Scheme-like source language has *labeled expressions* $e^l \in \mathit{Exp}$ of the form

$$\begin{array}{rcl}
e^l & ::= & c^l \mid f^l \mid x^l \\
 & \mid & \texttt{(}p\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l \\
 & \mid & \texttt{(}e_0^{l_0}\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l \\
 & \mid & \texttt{(if}\ e_1^{l_1}\ e_2^{l_2}\ e_3^{l_3}\texttt{)}^l \\
 & \mid & \texttt{(let ([}x\ e_1^{l_1}\texttt{])}\ e_b^{l_b}\texttt{)}^l \\
 & & \\
f^l & ::= & \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e_b^{l_b}\texttt{)}^l
\end{array}$$

where $c \in \mathit{Const}$ are *constants*, $p \in \mathit{Prim}$ are *primitives* (eg. $+$, $\div$), $x, y, z \in \mathit{Var}$ are *variables*, and $l \in \mathit{Label}$ are *labels*. Labels, which are not present in the surface syntax available to programmers, are discussed below. A λ-term constructs an anonymous procedure; terms beginning with p are uses of primitive operations; `if` provides conditional execution; and, `let` provides non-recursive local bindings. *Free* variables (FV) and *bound* variables (BV) are defined as usual for a lexically-scoped language [Barendregt 1984], with `let` binding its variable x in e_b.

Programs are expressions with no free variables. In order to unambiguously name different components of a program, we assume programs have two properties. First, since we use labels to identify subterms of a program, each subterm of a program must have a unique label. Second, since we refer to a specific variable by its name, we assume that all free and bound variables in a program are distinct. This condition can be met by renaming bound variables appropriately.

We sometimes distinguish different uses of variables as follows. A subscript on a variable occurrence indicates how the variable is bound. A variable occurrence such as $x^l_{[\lambda]}$ indicates that x is bound by a λ-expression. A variable occurrence such as $x^l_{[l']}$ indicates that the variable is bound by the `let`-expression with label l'. The following expression illustrates the use of these conventions:

$$
\begin{array}{l}
\texttt{(let ([}x\texttt{ (lambda (}z\texttt{) }z_{[\lambda]}\texttt{)])} \\
\quad \texttt{(}x_{[l_1]}\texttt{ 11)} \\
\quad \texttt{(}x_{[l_1]}\texttt{ 12))}^{l_1}
\end{array}
$$

The `let`-expression has label l_1, and the labels of the other subexpressions have been omitted.

3.2 Exact semantics

Like Scheme and ML, our source language has an ordinary call-by-value semantics. We specify the semantics as a small-step contextual rewriting system. To represent intermediate states of evaluation, we extend the source language with constructs to represent closure values and local environments as follows:

$$
\begin{array}{rcl}
e^l & ::= & \ldots \;\big|\; \texttt{(close } \rho\ f\texttt{)}^l \;\big|\; \texttt{(bind } \rho\ e\texttt{)}^l \\
v^l & ::= & c^l \;\big|\; \texttt{(close } \rho\ f\texttt{)}^l \\
\rho & ::= & [x_1 \mapsto v_1^{l_1}] \ldots [x_n \mapsto v_n^{l_n}]
\end{array}
$$

We identify a subset of expressions as *values*, denoted v. A `close`-expression (closure) represents the value of a procedure, and is a value. A `bind`-expression represents an activation record, holding values for the free variables of a procedure being evaluated. The first component of a `close`- or `bind`-expression is a finite list containing pairs of variables and values. We treat this list as an environment, i.e., a finite function. We write $\rho(x_i)$ for the unique v_i associated with x_i in ρ.

We use two forms of evaluation contexts [Barendregt 1984; Felleisen and Friedman 1986] to encode control information, i.e., to find the next redex:

$$
\begin{array}{rcl}
E \in \mathit{EvalContext} & ::= & [\,] \;\big|\; \texttt{(}v^*\ E\ e^*\texttt{)} \;\big|\; \texttt{(}p\ v^*\ E\ e^*\texttt{)} \;\big|\; \texttt{(if } E\ e\ e\texttt{)} \\
 & \big| & \texttt{(let ([}x\ E\texttt{]) } e\texttt{)} \;\big|\; \texttt{(bind } \rho\ E\texttt{)} \\
S \in \mathit{StackContext} & ::= & [\,] \;\big|\; \texttt{(}v^*\ S\ e^*\texttt{)} \;\big|\; \texttt{(}p\ v^*\ S\ e^*\texttt{)} \;\big|\; \texttt{(if } S\ e\ e\texttt{)} \\
 & \big| & \texttt{(let ([}x\ S\texttt{]) } e\texttt{)}
\end{array}
$$

An evaluation context (E or S) is a term with a single "hole" $[\,]$ in the place of one sub-term. We write $E[e]$ to denote the expression resulting from "filling" the hole of context E with e, with free variables of e captured by bindings in E. The grammar for evaluation contexts specifies where holes may appear, and thus controls the order in which the sub-expressions of a term are evaluated. For example, the arguments of primitive operations and procedure calls are evaluated from left to right, as the

$$
\begin{array}{ll}
E[\texttt{(bind } \rho\ S[x^l]\texttt{)}] & \textbf{(lookup)} \\
\quad \longmapsto E[\texttt{(bind } \rho\ S[v^l]\texttt{)}] \quad \text{where } \rho(x) = v^{l'} & \\
E[\texttt{(bind } \rho\ S[\texttt{(lambda (}x_1 \ldots x_n\texttt{) } e\texttt{)}^l]\texttt{)}] & \textbf{(make-closure)} \\
\quad \longmapsto E[\texttt{(bind } \rho\ S[\texttt{(close } \rho|_{FV(\texttt{(lambda (}x_1 \ldots x_n\texttt{) } e\texttt{)})}\ \texttt{(lambda (}x_1 \ldots x_n\texttt{) } e\texttt{)}^l\texttt{)}^l]\texttt{)}] & \\
E[\texttt{((close } \rho\ \texttt{(lambda (}x_1 \ldots x_n\texttt{) } e\texttt{)) } v_1 \ldots v_n\texttt{)}^l] & \textbf{(apply-closure)} \\
\quad \longmapsto E[\texttt{(bind } \rho[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]\ e\texttt{)}^l] & \\
E[\texttt{(}p\ v_1 \ldots v_n\texttt{)}^l] & \textbf{(primitive)} \\
\quad \longmapsto E[\delta(p, v_1 \ldots v_n)^l] \quad \text{if } \delta(p, v_1 \ldots v_n) \text{ is defined} & \\
E[\texttt{(bind } \rho\ v^{l'}\texttt{)}^l] & \textbf{(return)} \\
\quad \longmapsto E[v^l] & \\
E[\texttt{(bind } \rho\ S[\texttt{(let ([}x\ v\texttt{]) } e\texttt{)}^l]\texttt{)}] & \textbf{(let-bind)} \\
\quad \longmapsto E[\texttt{(bind } \rho\ S[\texttt{(bind } \rho[x \mapsto v]\ e\texttt{)}^l]\texttt{)}] & \\
E[\texttt{(if } v\ e_1\ e_2\texttt{)}^l] & \textbf{(if-true)} \\
\quad \longmapsto E[e_1^l] \quad \text{if } v \neq \texttt{\#f} & \\
E[\texttt{(if \#f } e_1\ e_2\texttt{)}^l] & \textbf{(if-false)} \\
\quad \longmapsto E[e_2^l] &
\end{array}
$$

Fig. 1. Exact semantics.

grammar requires all arguments left of the hole to be values. *Singular* contexts S are a subset of evaluation contexts E, and are used by various reductions to gain access to the closest lexical binding of a variable.

Figure 1 presents the rewriting rules for the semantics. The first rule indicates that a variable's value is obtained from the environment of the closest enclosing `bind`-expression. Evaluating a `lambda`-expression yields a closure value whose environment contains bindings for all of the `lambda`-expression's free variables, i.e., a flat closure. When a closure is applied to a value, **apply-closure** constructs a new activation record that holds bindings for the closure's free variables and its formal parameters.[2] Rule **primitive** uses a partial function δ to handle primitive applications. The **return** rule discards activation frames that are no longer needed. The remaining rules handle `let`-expressions and conditionals.

Note that reductions are label-preserving since the contractum of a redex inherits the same label as the redex. For example, **lookup** replaces x^l with v^l even though $\rho(x) = v^{l'}$. This property will be important later when we relate the abstract values computed by the flow analysis for an expression to the exact values specified by the exact semantics.

3.3 Flow analysis

A flow analysis determines the sets of values that may be bound to variables and returned from subexpressions. We name these sets *abstract values*; in other words, a single abstract value represents a set of exact values. Polyvariant analyses associate multiple abstract values with each variable and expression, distinguishing them by *contours* that abstract a program's run-time state. We call these variable-contour pairs and label-contour pairs *program points*. Let *Avalue* be some set of abstract

[2]This semantics is not tail-recursive. It is straightforward to add a tail-call rule that pops useless `bind`-expressions before pushing the new activation record.

values and *Contour* be some set of contours. A *flow analysis* of a program P is a function

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

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

Abstract values for our framework are defined as follows:

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

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

Our polymorphic splitting analysis represents contours as finite strings of labels. Contours abstract different execution contexts and are used to distinguish argument bindings and subexpression results arising at different points in a program's execution. The set of contours must be finite to ensure that the analysis computes finite information, i.e., that the analysis terminates. Since the subsets of variables and labels that a program uses are also finite, and since contours are represented as strings of distinct labels, a flow analysis can yield only finitely many abstract values for a particular program.

3.4 Constraints for safety

Figure 2 defines a relation

$$\vdash^P : \mathit{Flow} \times \mathit{Exp} \times \mathit{Aenv} \times \mathit{Contour}$$

that imposes *safety* constraints on a flow function F. These constraints ensure that F conservatively approximates the behavior of P, as we show in Section 3.7.

The rules for constants and λ-expressions require the corresponding program points to include appropriate values. The function *AbsValOf* maps a constant to its abstract value. Thus $F, \rho, \kappa \vdash^P 4^l$ holds if and only if $\mathsf{number} \in F\langle l, \kappa\rangle$, given that $\mathit{AbsValOf}(4) = \mathsf{number}$. Similarly, $F, \rho, \kappa \vdash^P (\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l$ holds if and only if the abstract closure $\langle l, \theta|_{FV((\texttt{lambda}\ (x_1 \ldots x_n)\ e))}, \kappa\rangle_\lambda$ is in $F\langle l, \kappa\rangle$.

The rule for primitive applications "evaluates" the subexpressions $e_1 \ldots e_n$ from left to right by requiring $F, \theta, \kappa \vdash^P e_i^{l_i}$ to hold whenever $F\langle l_j, \kappa\rangle \neq \emptyset$ for all $j < i$. In other words, $F, \theta, \kappa \vdash^P e_i^{l_i}$ must hold whenever the preceeding subexpressions might yield values. The notation $X \Rightarrow Y$ means that Y must hold whenever X holds. When $F\langle l_i, \kappa\rangle \neq \emptyset$ for all i, the rule for primitives also requires $\mathit{AbsResultOf}(p) \in$

$$F,\theta,\kappa \vdash^P c^l \text{ iff } \mathit{AbsValOf}(c) \in F\langle l,\kappa\rangle \qquad \textbf{(const)}$$

$$F,\theta,\kappa \vdash^P \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e\texttt{)}^l \text{ iff } \langle l, \theta|_{FV(\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e\texttt{)})}, \kappa\rangle_\lambda \in F\langle l,\kappa\rangle \qquad \textbf{(lam)}$$

$$\begin{array}{lll} F,\theta,\kappa \vdash^P \texttt{(}p\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l & \text{iff} & F\langle l_j,\kappa\rangle \neq \emptyset \text{ for all } j < i \\ & & \quad \Rightarrow F,\theta,\kappa \vdash^P e_i^{l_i} \quad \text{for } i = 1 \ldots n \\ & & F\langle l_n,\kappa\rangle \neq \emptyset \Rightarrow \mathit{AbsResultOf}(p) \in F\langle l,\kappa\rangle \end{array} \qquad \textbf{(prim)}$$

$$F,\theta,\kappa \vdash^P x^l_{[\lambda]} \text{ iff } F\langle x,\theta(x)\rangle \subseteq F\langle l,\kappa\rangle \qquad \textbf{(lam-var)}$$

$$\begin{array}{lll} F,\theta,\kappa \vdash^P \texttt{(}e_0^{l_0}\ e_1^{l_1} \ldots e_n^{l_n}\texttt{)}^l & \text{iff} & F\langle l_j,\kappa\rangle \neq \emptyset \text{ for all } j < i \\ & & \quad \Rightarrow F,\theta,\kappa \vdash^P e_i^{l_i} \quad \text{for } i = 0 \ldots n \\ & & F\langle l_n,\kappa\rangle \neq \emptyset \Rightarrow \\ & & \quad \text{for all } \langle l',\theta',\kappa'\rangle_\lambda \in F\langle l_0,\kappa\rangle \\ & & \quad \text{where } \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e_b^{l_b}\texttt{)}^{l'} \text{ in } P \\ & & \qquad F,\theta'[x_1 \ldots x_n \mapsto \kappa'],\kappa' \vdash^P e_b^{l_b} \\ & & \qquad F\langle l_i,\kappa\rangle \subseteq F\langle x_i,\kappa'\rangle \text{ for all } i = 1 \ldots n \\ & & \qquad F\langle l_b,\kappa'\rangle \subseteq F\langle l,\kappa\rangle \end{array} \qquad \textbf{(ap)}$$

$$\begin{array}{lll} F,\theta,\kappa \vdash^P \texttt{(if }e_1^{l_1}\ e_2^{l_2}\ e_3^{l_3}\texttt{)}^l & \text{iff} & F,\theta,\kappa \vdash^P e_1^{l_1} \\ & & \mathsf{true} \in F\langle l_1,\kappa\rangle \Rightarrow F,\theta,\kappa \vdash^P e_2^{l_2} \text{ and } F\langle l_2,\kappa\rangle \subseteq F\langle l,\kappa\rangle \\ & & \mathsf{false} \in F\langle l_1,\kappa\rangle \Rightarrow F,\theta,\kappa \vdash^P e_3^{l_3} \text{ and } F\langle l_3,\kappa\rangle \subseteq F\langle l,\kappa\rangle \end{array} \qquad \textbf{(if)}$$

$$\begin{array}{lll} F,\theta,\kappa \vdash^P \texttt{(let ([}x\ e_1^{l_1}\texttt{])}\ e_b^{l_b}\texttt{)}^l & \text{iff} & F,\theta,[\kappa:l] \vdash^P e_1^{l_1} p \\ & & F\langle l_1,[\kappa:l]\rangle \subseteq F\langle x,\kappa\rangle \\ & & F\langle x,\kappa\rangle \neq \emptyset \Rightarrow \\ & & \quad F,\theta[x \mapsto \kappa],\kappa \vdash^P e_b^{l_b} \text{ and } F\langle l_b,\kappa\rangle \subseteq F\langle l,\kappa\rangle \end{array} \qquad \textbf{(let)}$$

$$\begin{array}{lll} F,\theta,\kappa \vdash^P x^l_{[l_0]} & \text{iff} & b \in F\langle l,\kappa\rangle \text{ for all } b \in F\langle x,\theta(x)\rangle \\ & & \langle l',\theta',cn'[l_0/l]\rangle_\lambda \in F\langle l,\kappa\rangle \text{ for all } \langle l',\theta',\kappa'\rangle_\lambda \in F\langle x,\theta(x)\rangle \end{array} \qquad \textbf{(let-var)}$$

$$F,\theta,\kappa \vdash^P \texttt{(close }\rho\ e^{l'}\texttt{)}^l \text{ iff } F,\theta' \models^P \rho \text{ for some } \langle l',\theta',\kappa'\rangle_\lambda \in F\langle l,\kappa\rangle \qquad \textbf{(close)}$$

$$\begin{array}{lll} F,\theta,\kappa \vdash^P \texttt{(bind }\rho\ \mathtt{e}^{l'}\texttt{)}^l & \text{iff} & \text{for some } \theta',\kappa' \\ & & \quad F,\theta',\kappa' \vdash^P e^{l'} \\ & & \quad F\langle l',\kappa'\rangle \subseteq F\langle l,\kappa\rangle \\ & & \quad F,\theta' \models^P \rho \end{array} \qquad \textbf{(bind)}$$

$$\begin{array}{lll} F,\theta \models^P \rho & \text{iff} & \text{for all } x \in \mathrm{dom}(\rho) \text{ where } \rho(x) = v^l \\ & & \quad F,\theta',\kappa' \vdash^P v^l \text{ and } F\langle l,\kappa'\rangle \subseteq F\langle x,\theta(x)\rangle \text{ for some } \theta',\kappa' \end{array}$$

Fig. 2. Relation $\vdash^P$ defining flow analysis F of program P.

$F\langle l, \kappa\rangle$. The function $\mathit{AbsResultOf}$ maps a primitive to its abstract result; for example, $\mathit{AbsResultOf}(+) = \mathsf{number}$.

The remaining rules introduce constraints between program points. These constraints ensure that the abstract values of program points safely approximate the exact values those program points can take on during execution. The rule for lambda-bound variables constrains the abstract evaluation of a variable x to include the abstract value bound to it in contour $\rho(x)$. For each abstract closure that can arise in the function position of an application (i.e., for each abstract closure in $F\langle l_0, \kappa\rangle$), the application rule introduces two kinds of constraints. The first kind, $F\langle l_i, \kappa\rangle \subseteq F\langle x_i, \kappa'\rangle$, requires the formal parameters of the abstract closure to include the abstract values of the actual parameters. The second kind, $F\langle l_b, \kappa'\rangle \subseteq F\langle l, \kappa\rangle$, requires the application point to include the abstract value arising from the evaluation of the body of the abstract closure. In the figure, the notation $\rho[x_1 \ldots x_n \mapsto \kappa]$ means the functional update or extension of the environment ρ at each of $x_1 \ldots x_n$ to κ.

In the rule for `if`-expressions, evaluation of the then-clause (resp. the else-clause) is conditional upon `#t` (resp. `#f`) arising as a possible value of the test. If the abstract value of the test includes both true and false, both the then-clause and the else-clause contribute to the result of the conditional expression. If the test always diverges and its abstract evaluation yields the empty set, neither the then-clause nor the else-clause is evaluated. The analysis is thus sensitive to whether subexpressions are reachable. Suppose the program contains `(if` $e_1^{l_1}$ $e_2^{l_2}$ $e_3^{l_3}$`)` and the analysis determines that e_1 can never yield `#t` in any contour, i.e., $\mathsf{true} \notin F\langle l_1, \kappa\rangle$ for any κ. Then abstract evaluation of e_2 never commences since

$$\mathsf{true} \in F\langle l_1, \kappa\rangle \Rightarrow F, \theta, \kappa \vdash^P e_2^{l_2} \text{ and } F\langle l_2, \kappa\rangle \subseteq F\langle l, \kappa\rangle$$

holds without requiring $F, \theta, \kappa \vdash^P e_2^{l_2}$ to hold. Similarly, the conditional relations in the rules for primitives and calls cause the analysis to respect left-to-right order of evaluation of subexpressions. Sensitivity to reachability is significant for polyvariant analyses because many expressions are unreachable in particular contours.

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

The third part of Figure 2 includes rules for `close`- and `bind`-expressions. Although these expressions do not appear in source programs, their flow analysis rules are necessary for the inductive proof of soundness. The rules use an auxiliary *agree-*

$$
\begin{array}{rcl}
\langle 0,[\,]\rangle & \mapsto & \{\mathsf{number}\} \\
\langle 1,[\,]\rangle & \mapsto & \{\langle 10,\emptyset,[1]\rangle_\lambda\} \\
\langle 2,[\,]\rangle & \mapsto & \{\mathsf{number}\} \\
\langle 3,[\,]\rangle & \mapsto & \{\langle 9,[\mathtt{x}\mapsto[1]],[1]\rangle_\lambda\} \\
\langle 4,[\,]\rangle & \mapsto & \{\langle 10,\emptyset,[4]\rangle_\lambda\} \\
\langle 5,[\,]\rangle & \mapsto & \{\mathsf{true}\} \\
\langle 6,[\,]\rangle & \mapsto & \{\langle 9,[\mathtt{x}\mapsto[4]],[4]\rangle_\lambda\} \\
\langle 7,[\,]\rangle & \mapsto & \{\mathsf{number}\}
\end{array}
\qquad
\begin{array}{rcl}
\langle 8,[1]\rangle & \mapsto & \{\mathsf{number}\} \\
\langle 9,[1]\rangle & \mapsto & \{\langle 9,[\mathtt{x}\mapsto[1]],[1]\rangle_\lambda\} \\
\langle 9,[4]\rangle & \mapsto & \{\langle 9,[\mathtt{x}\mapsto[4]],[4]\rangle_\lambda\} \\
\langle 10,[0]\rangle & \mapsto & \{\langle 10,\emptyset,[0]\rangle_\lambda\} \\
\langle \mathtt{f},[0]\rangle & \mapsto & \{\langle 10,\emptyset,[0]\rangle_\lambda\} \\
\langle \mathtt{x},[1]\rangle & \mapsto & \{\mathsf{number}\} \\
\langle \mathtt{x},[4]\rangle & \mapsto & \{\mathsf{true}\} \\
\langle \mathtt{y},[1]\rangle & \mapsto & \{\langle 9,[\mathtt{x}\mapsto[4]],[4]\rangle_\lambda\}
\end{array}
$$

Fig. 3. Least flow analysis of $\begin{array}{l}\mathtt{(let\ ([f\ (lambda\ (x)\ (lambda\ (y)\ x^{8})^{9})^{10}])} \\ \quad \mathtt{((f^{1}\ 0^{2})^{3}\ (f^{4}\ \#t^{5})^{6})^{7})^{0}}\end{array}$

$$
\begin{array}{rcl}
\langle 0,[\,]\rangle & \mapsto & \{\mathsf{number},\ \mathsf{true}\} \\
\langle 1,[\,]\rangle & \mapsto & \{\langle 11,\emptyset,[\,]\rangle_\lambda\} \\
\langle 2,[\,]\rangle & \mapsto & \{\mathsf{number}\} \\
\langle 3,[\,]\rangle & \mapsto & \{\langle 10,[\mathtt{x}\mapsto[\,]],[\,]\rangle_\lambda\} \\
\langle 4,[\,]\rangle & \mapsto & \{\langle l_{11},\emptyset,\kappa_0\rangle_\lambda\} \\
\langle 5,[\,]\rangle & \mapsto & \{\mathsf{true}\} \\
\langle 6,[\,]\rangle & \mapsto & \{\langle 10,[\mathtt{x}\mapsto[\,]],[\,]\rangle_\lambda\} \\
\langle 7,[\,]\rangle & \mapsto & \{\mathsf{number},\ \mathsf{true}\}
\end{array}
\qquad
\begin{array}{rcl}
\langle 8,[\,]\rangle & \mapsto & \{\langle 8,\emptyset,[\,]\rangle_\lambda\} \\
\langle 9,[\,]\rangle & \mapsto & \{\mathsf{number},\ \mathsf{true}\} \\
\langle 10,[\,]\rangle & \mapsto & \{\langle 10,[\mathtt{x}\mapsto[\,]],[\,]\rangle_\lambda\} \\
\langle 11,[\,]\rangle & \mapsto & \{\langle 11,\emptyset,[\,]\rangle_\lambda\} \\
\langle \mathtt{f},[\,]\rangle & \mapsto & \{\langle 11,\emptyset,[\,]\rangle_\lambda\} \\
\langle \mathtt{x},[\,]\rangle & \mapsto & \{\mathsf{number},\ \mathsf{true}\} \\
\langle \mathtt{y},[\,]\rangle & \mapsto & \{\langle 10,[\mathtt{x}\mapsto[\,]],[\,]\rangle_\lambda\}
\end{array}
$$

Fig. 4. Least flow analysis of $\begin{array}{l}\mathtt{((lambda\ (f)\ ((f^{1}\ 0^{2})^{3}\ (f^{4}\ \#t^{5})^{6})^{7})^{8}} \\ \quad \mathtt{(lambda\ (x)\ (lambda\ (y)\ x^{9})^{10})^{11})^{0}}\end{array}$

ment relation $\models$ to relate an exact environment ρ to an abstract environment θ. This agreement relation requires the variables defined by θ to have abstract values which approximate the exact values defined by ρ.

3.5 Discussion

To illustrate polymorphic splitting, consider the expression

```
(let ([f (lambda (x) (lambda (y) x))])
  ((f 0) (f #t)))
```

which yields `0` and whose least flow analysis is shown in Figure 3. Because `f` is a `let`-bound procedure, each of its two calls are analyzed in separate contours. In the first call (f 0), abstract values for the variables and subexpressions of `f` are recorded in contour [1]. In the second call (f #t), abstract values are recorded in contour [4]. Hence abstract values from the two calls are not merged, and the result of the analysis for program point $\langle 0,[\,]\rangle$ is $\{\mathsf{number}\}$.

The following semantically equivalent expression leads to a less precise result:

```
((lambda (f) ((f 0) (f #t)))
  (lambda (x) (lambda (y) x)))
```

Figure 4 presents its least flow analysis. The abstract value for the result of this expression is $\{\mathsf{number},\ \mathsf{true}\}$. Because a polymorphic splitting analysis introduce new contours only at references to `let`-bound procedures, the two applications `(f 0)` and `(f #t)` are analyzed in the same contour. Hence arguments to `f` from the two calls are merged.

Although polymorphic splitting mimics polymorphism, the approximation defined by the flow analysis is less precise than Hindley-Milner typing in some cases.

Consider the following example:

```
(let ([f (lambda (x) x)²])
  (let ([g (lambda (x)
             (lambda (y)
               ... (f⁵ x)
               ... (f⁶ y) ...))])
    (begin
      (g³ ...)
      (g⁴ ...)))¹)⁰
```

Here `(begin` e_1 e_2`)` abbreviates `((lambda (`v_1`)` e_2`)` e_1`)` where $v_1 \notin FV(e_2)$. To faithfully model polymorphism, we require `f` to be evaluated in four distinct contours. However, because `g`'s closure binds `f` to contour [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 $\langle 2, \emptyset, [5]\rangle_\lambda$; the evaluation of this occurrence of `f` in the second call to `g` will be the same.

It is possible to refine the specification to fully capture the behavior of Hindley-Milner type inference. By 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. However, this increased precision appears to entail significantly higher analysis costs. 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.

3.6 $\vdash$ is well-defined

Because $\vdash$ is not defined inductively on the structure of expressions, as witnessed by the rule for applications, it is not immediately clear how to interpret Figure 2. Following Neilson and Neilson [1997], we define $\vdash$ *coinductively* as the greatest fixed-point of the underlying functional.

Let $R_P : \mathit{Flow} \times \mathit{Exp} \times \mathit{Aenv} \times \mathit{Contour}$ be a family of relations for program P. Define a functional $A_P : R_P \to R_P$, similar to Figure 2, as follows:

$$\begin{aligned}
A_P(R_P) \;=\;& \{\langle F, \theta, \kappa, c^l\rangle \mid \mathit{AbsValOf}(c) \in F\langle l, \kappa\rangle\} \\
\cup\;& \{\langle F, \theta, \kappa, \texttt{(lambda ...)}^l\rangle \mid \langle l, \theta|_{FV(\texttt{(lambda ...)})}, \kappa\rangle_\lambda \in F\langle l, \kappa\rangle\} \\
\cup\;& \left\{\langle F, \theta, \kappa, (p\ e_1^{l_1} \ldots e_n^{l_n})^l\rangle \;\middle|\; \begin{array}{l} F\langle l_j, \kappa\rangle \neq \emptyset \text{ for } j < i \Rightarrow \langle F, \theta, \kappa, e_i^{l_i}\rangle \in R_P \\ F\langle l_n, \kappa\rangle \neq \emptyset \Rightarrow \mathit{AbsResultOf}(p) \in F\langle l, \kappa\rangle \end{array}\right\} \\
\cup\;& \ldots \text{ remaining cases adapted } \textit{mutatis mutandis}
\end{aligned}$$

We define $F, \theta, \kappa \vdash^P e$ as the greatest fixed-point of this functional. This fixed-point is guaranteed to exist because A_P is monotonic. We want the greatest fixed-point, rather than the least fixed-point, because many useful programs would not have flow functions under the least fixed-point. (Consider a program containing a loop whose integer bounds are abstracted to number. For $\vdash$ to hold for the loop body, $\vdash$ must hold for the iterative application within the loop body. But this application requires $\vdash$ to hold for the loop body.)

We now show that for any program P, there exists a unique least flow analysis that adheres to the safety constraints imposed by $\vdash^P$.

THEOREM 3.1 (LEAST FLOW ANALYSIS). *Given program P, initial contour κ_0 and closed expression ${e_0}^{l_0}$, there exists a unique minimal function F such that $F, \phi, \kappa_0 \vdash^P e_0^{l_0}$ holds.*

PROOF. For any given program P, we can easily construct a flow function $F_{\max}$ such that $F_{\max}, \phi, \kappa_0 \vdash^P e_0^{l_0}$ by including all possible abstract closures and abstract constants relevant to P in every program point. $F_{\max}$ is finite since it uses only finitely many contours. We then impose a natural partial ordering on flow functions for P, and write $F_1 \subseteq F_2$ when F_1 is smaller than F_2. By coinduction, we prove that if $F_1, \theta_0, \kappa_0 \vdash^P e_0^{l_0}$ and $F_2, \theta_0, \kappa_0 \vdash^P e_0^{l_0}$ then $F_1 \cap F_2, \theta_0, \kappa_0 \vdash^P e_0^{l_0}$. □

3.7 Soundness

A flow analysis is *sound* if the abstract values it computes for various program points are conservative approximations of the exact values that those program points take on during execution. We prove soundness via a subject reduction theorem [Wright and Felleisen 1994] that relates reductions in the exact semantics to the constraints found in the specification of the flow analysis. In the following, we say that an expression e in program P is *safe* if $F, \theta, \kappa \vdash^P e$ holds for some θ and κ. Subject reduction states that if an expression e_1 reduces to e_2 and e_1 is safe, then e_2 must also be safe.

THEOREM 3.2 (SUBJECT REDUCTION). *If $F, \theta, \kappa \vdash^P e_1$ and $e_1 \longmapsto e_2$ then $F, \theta, \kappa \vdash^P e_2$.*

PROOF. We require two supporting lemmas for the proof. The first lemma states that the redex of a safe expression is also safe.

LEMMA 3.1. *If $F, \theta, \kappa \vdash^P E[e]$ then there exist θ' and κ' such that $F, \theta', \kappa' \vdash^P e$. Furthermore, if $E = S$ for some S, then $\theta' = \theta$.*

This lemma is proven by induction on the structure of E.

The second lemma states that the replacement of a safe redex by a safe contractum yields a safe expression.

LEMMA 3.2. *If $F, \theta, \kappa \vdash^P E[e_1^l]$ and $F, \theta', \kappa' \vdash^P e_1^l$ and $F, \theta', \kappa' \vdash^P e_2^l$ then $F, \theta, \kappa \vdash^P E[e_2^l]$.*

This lemma is proven by induction on the structure of E.

The proof of the subject reduction theorem proceeds by case analysis on the form of the reduction.

Case $E[(\texttt{bind}\ \rho\ S[x^l])] \longmapsto E[(\texttt{bind}\ \rho\ S[v^l])]$ where $\rho(x) = v^{l_0}$. We have $F, \theta, \kappa \vdash^P (\texttt{bind}\ \rho\ S[x^l])$ by Lemma 3.1. Hence there exist θ', κ' such that

$$F, \theta' \models^P \rho \tag{1}$$

and

$$F, \theta', \kappa' \vdash^P S[x^l] \tag{2}$$

by **bind**. From (1) by the definition of $\models$ we have

$$F, \theta'', \kappa'' \vdash^P v^{l_0} \tag{3}$$

and

$$F\langle l_0, \kappa''\rangle \subseteq F\langle x, \theta'(x)\rangle. \tag{4}$$

From (2) we have

$$F, \theta', \kappa''' \vdash^P x^l \tag{5}$$

by lemma 3.1.

Suppose $v = c$. Then $AbsValOf(c) \in F\langle l_0, \kappa''\rangle$ by **const** with (3). Hence $AbsValOf(c) \in F\langle x, \theta'(x)\rangle$ by (4). By either **let-var** or **lam-var** according to how x is bound, $AbsValOf(c) \in F\langle l, \kappa'''\rangle$. Hence

$$F, \theta', \kappa''' \vdash^P v^l \tag{6}$$

by **const**. The result follows from (5) and (6) by lemma 3.2.

Otherwise $v = (\texttt{close}\ \rho'\ e'^{l'})$. Then $F, \theta_v \models^P \rho'$ and $\langle l', \theta_v, \kappa_v\rangle_\lambda \in F\langle l_0, \kappa''\rangle$ by **close** with (3). Hence $\langle l', \theta_v, \kappa_v\rangle_\lambda \in F\langle x, \theta'(x)\rangle$ by (4). By either **let-var** or **lam-var** according to how x is bound, $\langle l', \theta_v, \kappa'_v\rangle_\lambda \in F\langle l, \kappa'''\rangle$ (where κ'_v is different from κ_v if x is bound by `let`). Hence

$$F, \theta', \kappa''' \vdash^P v^l \tag{7}$$

by **close**. The result follows from (5) and (7) by lemma 3.2.

Case $E[(\texttt{bind}\ \rho\ S[(\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l])] \longmapsto$
$E[(\texttt{bind}\ \rho\ S[(\texttt{close}\ \rho|_{FV((\texttt{lambda}\ (x_1\ \ldots\ x_n)\ e))}\ (\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l)^l])]$.

By lemma 3.1, we have $F, \theta, \kappa \vdash^P (\texttt{bind}\ \rho\ S[(\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l])$. Hence

$$F, \theta' \models^P \rho \tag{8}$$

and

$$F, \theta', \kappa' \vdash^P S[(\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l]$$

by **bind**. From the latter we have

$$F, \theta', \kappa' \vdash^P (\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l \tag{9}$$

by lemma 3.1. Hence

$$\langle l, \theta'|_{FV((\texttt{lambda}\ (x_1\ \ldots\ x_n)\ e))}, \kappa'\rangle_\lambda \in F\langle l, \kappa'\rangle \tag{10}$$

by **lam**. Then from (8) and (10) we have

$$F, \theta', \kappa' \vdash^P (\texttt{close}\ \rho|_{FV((\texttt{lambda}\ (x_1\ \ldots\ x_n)\ e))}\ (\texttt{lambda}\ (x_1 \ldots x_n)\ e)^l)^l \tag{11}$$

by **close**. The result follows from (9) and (11) by lemma 3.2.

Case $E[((\texttt{close}\ \rho\ (\texttt{lambda}\ (x_1 \ldots x_n)\ e_b^{l_b})^{l_c})\ v_1 \ldots v_n)^l] \longmapsto$
$E[(\texttt{bind}\ \rho[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]\ e_b)^l]$.

By lemma 3.1 we have

$$F, \theta, \kappa \vdash^P ((\texttt{close}\ \rho\ (\texttt{lambda}\ (x_1 \ldots x_n)\ e_b))\ v_1 \ldots v_n)^l. \tag{12}$$

By **ap** we have

$$F, \theta, \kappa \vdash^P v_i^{l_i} \tag{13}$$

for $i = 1 \ldots n$ and

$$F, \theta, \kappa \vdash^P \texttt{(close } \rho \texttt{ (lambda (} x_1 \ldots x_n \texttt{) } e_b \texttt{)).} \tag{14}$$

From the latter by **close** we have

$$F, \theta' \models^P \rho \tag{15}$$

and $\langle l_c, \theta', \kappa' \rangle_\lambda \in F\langle l_1, \kappa \rangle$. Together with (13) we have

$$F, \theta'[x \mapsto \kappa'], \kappa' \vdash^P e_b^{l_b} \tag{16}$$

and

$$F\langle l_i, \kappa' \rangle \subseteq F\langle x_i, \kappa' \rangle \tag{17}$$

for $i = 1 \ldots n$, as well as

$$F\langle l_b, \kappa' \rangle \subseteq F\langle l, \kappa \rangle \tag{18}$$

by **ap**. By the variable conventions, $x \notin \text{dom}(\rho)$, hence with (15) we have

$$F, \theta'[x \mapsto \kappa'] \models^P \rho.$$

With (13) and (17) we have

$$F, \theta'[x \mapsto \kappa'] \models^P \rho[x \mapsto v^{l_2}]$$

and together with (16) and (18) we have

$$F, \theta, \kappa \vdash^P \texttt{(bind } \rho[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \ e_b \texttt{)}^l \tag{19}$$

by **bind**. The result follows from (12) and (19) by lemma 3.2.

Case $E[\texttt{(bind } \rho \ v^{l'} \texttt{)}^l] \longmapsto E[v^l]$. We have

$$F, \theta, \kappa \vdash^P \texttt{(bind } \rho \ v^{l'} \texttt{)}^l \tag{20}$$

by lemma 3.1. Then

$$F, \theta', \kappa' \vdash^P v^{l'} \tag{21}$$

and

$$F\langle l', \kappa' \rangle \subseteq F\langle l, \kappa \rangle \tag{22}$$

by **bind**.

Suppose $v = c$. Then $\mathit{AbsValOf}(c) \in F\langle l', \kappa' \rangle$ by **const** with (21). Hence $\mathit{AbsValOf}(c) \in F\langle l, \kappa \rangle$ by (22). Hence

$$F, \theta, \kappa \vdash^P v^l \tag{23}$$

by **const**. The result follows from (20) and (23) by lemma 3.2.

Otherwise $v = \texttt{(close } \rho' \ e''^{l'} \texttt{)}$. Then $F, \theta_v \models^P \rho'$ and $\langle l', \theta_v, \kappa_v \rangle_\lambda \in F\langle l', \kappa' \rangle$ by **close** with (21). Hence $\langle l', \theta_v, \kappa_v \rangle_\lambda \in F\langle l, \kappa \rangle$ by (22). Hence

$$F, \theta', \kappa \vdash^P v^l \tag{24}$$

by **close**. The result follows from (20) and (24) by lemma 3.2.

Case $E[(\texttt{bind}\ \rho\ S[(\texttt{let}\ ([x\ v^{l_1}])\ e_b^{l_b})^l])] \longmapsto$
$E[(\texttt{bind}\ \rho\ S[(\texttt{bind}\ \rho[x \mapsto v^{l_1}]\ e_b^{l_b})^l])]$.
Then

$$F, \theta, \kappa \vdash^P (\texttt{bind}\ \rho\ S[(\texttt{let}\ ([x\ v^{l_1}])\ e_b^{l_b})^l])$$

by lemma 3.1. Hence

$$F, \theta' \models^P \rho \tag{25}$$

and

$$F, \theta', \kappa' \vdash^P S[(\texttt{let}\ ([x\ v^{l_1}])\ e_b^{l_b})^l] \tag{26}$$

by **bind**. From the latter, $F, \theta', \kappa'' \vdash^P (\texttt{let}\ ([x\ v^{l_1}])\ e_b^{l_b})^l$ by lemma 3.1. Hence

$$F, \theta', [\kappa'' : l] \vdash^P v^{l_1} \tag{27}$$

and

$$F\langle l_1, [\kappa''\ :\ l]\rangle \subseteq F\langle x, \kappa''\rangle \tag{28}$$

by **let**. From (27) and (28) by **lam**, $F\langle x, \kappa''\rangle \neq \emptyset$, hence

$$F, \theta'[x \mapsto \kappa''], \kappa'' \vdash^P e_b^{l_b} \tag{29}$$

and

$$F\langle l_b, \kappa''\rangle \subseteq F\langle l, \kappa''\rangle \tag{30}$$

by **let**. Now, by the definition of $\models$ with (27), (28), and (25), we have

$$F, \theta'[x \mapsto \kappa''] \models^P \rho[x \mapsto v^{l_1}].$$

Together with (29) and (30) we obtain

$$F, \theta', \kappa'' \vdash^P (\texttt{bind}\ \rho[x \mapsto v^{l_1}]\ e_b^{l_b})$$

by **bind**. The result follows by lemma 3.2 together with (26).

The remaining cases for primitive application and `if` are straightforward. □

3.8 Recursion, data constructors, and assignment

To add recursion, data constructors, and assignment, we extend our source language with several new syntactic forms:

$$\begin{array}{rcl} e^l & ::= & \ldots \\ & | & (\texttt{letrec}\ ([y\ f^{l_1}])\ e_b^{l_b})^l \\ & | & (\texttt{cons}\ e_1^{l_1}\ e_2^{l_2})^l \\ & | & (\texttt{car}\ e_1^{l_1})^l \\ & | & (\texttt{cdr}\ e_1^{l_1})^l \\ & | & (\texttt{set-car!}\ e_1^{l_1}\ e_2^{l_2})^l \\ & | & (\texttt{set-cdr!}\ e_1^{l_1}\ e_2^{l_2})^l \end{array}$$

The `letrec` form constructs named recursive procedures; `cons` constructs mutable pairs; `car` and `cdr` are the first and second projections on pairs, and `set-car!` and `set-cdr!` mutate the components of pairs. A `letrec`-expression binds its

$$
\begin{array}{rll}
F,\theta,\kappa \vdash^P \texttt{(letrec ([}y\ f^{l_1}\texttt{])}\ e_b^{l_b}\texttt{)}^l & \text{iff } F,\theta,[\kappa : l] \vdash^P f^{l_1} & \textbf{(letrec)} \\
& F\langle l_1,[\kappa : l]\rangle \subseteq F\langle y,\kappa\rangle & \\
& F,\rho[y \mapsto \kappa],\kappa \vdash^P e_b^{l_b} & \\
& F\langle l_b,\kappa\rangle \subseteq F\langle l,\kappa\rangle & \\
F,\theta,\kappa \vdash^P y^l_{[l'']} & \text{iff for all } \langle l',\rho',\kappa'\rangle_\lambda \in F\langle y,\rho(y)\rangle & \textbf{(non-rec-occur)} \\
& \quad \langle l',\rho'[y \mapsto \kappa'[l''/l],\kappa'[l''/l]\rangle_\lambda \in F\langle y,\kappa'[l''/l]\rangle & \\
& \quad \langle l',\rho'[y \mapsto \kappa'[l''/l],\kappa'[l''/l]\rangle_\lambda \in F\langle l,\kappa\rangle & \\
F,\theta,\kappa \vdash^P y^l_{[\mathsf{rec}]} & \text{iff } F\langle y,\theta(y)\rangle \subseteq F\langle l,\kappa\rangle & \textbf{(rec-occur)} \\
F,\theta,\kappa \vdash^P \texttt{(cons}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^l & \text{iff } F,\theta,\kappa \vdash^P e_1^{l_1} & \textbf{(cons)} \\
& F\langle l_1,\kappa\rangle \neq \emptyset \Rightarrow F,\theta,\kappa \vdash^P e_2^{l_2} & \\
& F\langle l_2,\kappa\rangle \neq \emptyset \Rightarrow \langle l,\kappa\rangle_\pi \in F\langle l,\kappa\rangle & \\
F,\theta,\kappa \vdash^P \texttt{(car}\ e_1^{l_1}\texttt{)}^l & \text{iff } F,\theta,\kappa \vdash^P e_1^{l_1} & \textbf{(car)} \\
& \text{for all } \langle l',\kappa'\rangle_\pi \in F\langle l_1,\kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \text{ in } P & \\
& \quad F\langle l_a,\kappa'\rangle \subseteq F\langle l,\kappa\rangle & \\
F,\theta,\kappa \vdash^P \texttt{(cdr}\ e_1^{l_1}\texttt{)}^l & \text{iff } F,\theta,\kappa \vdash^P e_1^{l_1}\rho\kappa & \textbf{(cdr)} \\
& \text{for all } \langle l',\kappa'\rangle_\pi \in F\langle l_1,\kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \text{ in } P & \\
& \quad F\langle l_b,\kappa'\rangle \subseteq F\langle l,\kappa\rangle & \\
F,\theta,\kappa \vdash^P \texttt{(set-car!}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^l & \text{iff } F,\theta,\kappa \vdash^P e_1^{l_1} & \textbf{(set-car!)} \\
& F\langle l_1,\kappa\rangle \neq \emptyset \Rightarrow F,\theta,\kappa \vdash^P e_2^{l_2} & \\
& F\langle l_2,\kappa\rangle \neq \emptyset \Rightarrow \mathsf{void} \in F\langle l,\kappa\rangle & \\
& \text{for all } \langle l',\kappa'\rangle_\pi \in F\langle l_1,\kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \text{ in } P & \\
& \quad F\langle l_2,\kappa\rangle \subseteq F\langle l_a,\kappa'\rangle & \\
F,\theta,\kappa \vdash^P \texttt{(set-cdr!}\ e_1^{l_1}\ e_2^{l_2}\texttt{)}^l & \text{iff } F,\theta,\kappa \vdash^P e_1^{l_1} & \textbf{(set-cdr!)} \\
& F\langle l_1,\kappa\rangle \neq \emptyset \Rightarrow F,\theta,\kappa \vdash^P e_2^{l_2} & \\
& F\langle l_2,\kappa\rangle \neq \emptyset \Rightarrow \mathsf{void} \in F\langle l,\kappa\rangle & \\
& \text{for all } \langle l',\kappa'\rangle_\pi \in F\langle l_1,\kappa\rangle \text{ where } \texttt{(cons}\ e_a^{l_a}\ e_b^{l_b}\texttt{)}^{l'} \text{ in } P & \\
& \quad F\langle l_2,\kappa\rangle \subseteq F\langle l_b,\kappa'\rangle &
\end{array}
$$

Fig. 5. Extensions to relation $\vdash$ for recursion, data constructors, and assignment.

variable y in f and e_b. The bound expression f of a `letrec`-expression must be a procedure. We extend our conventions for naming `let`-bound variables to `letrec`-bound variables, except that recursive references to `letrec` variables (i.e., references to y appearing within f) appear as $y^l_{[\mathsf{rec}]}$. We also reserve the meta-variable y for variables bound by `letrec`-expressions.

We extend the definition of abstract values to include abstract pairs:

$$
\begin{array}{rcl}
a \in \mathit{Avalue} & = & \ldots + \mathit{Apair} \\
\langle l,\kappa\rangle_\pi \in \mathit{Apair} & = & \mathit{Label} \times \mathit{Contour}
\end{array}
$$

An abstract pair $\langle l,\kappa\rangle_\pi$ identifies pairs constructed from expression $\texttt{(cons}\ e_1\ e_2\texttt{)}^l$ in contour κ. Figure 5 extends relation $\vdash^P$ to the new syntactic forms. Both theorems 3.1 and 3.2 hold for these extensions.

In a Hindley-Milner polymorphic type system, polymorphic recursion is not supported. Thus, occurrences of a variable y bound in $\texttt{(letrec ([}y\ f\texttt{])}\ e_b\texttt{)}$ are polymorphic only in e_b, and not in f. Similarly, our polyvariant flow analysis introduces new abstract closures only at occurrences of y within e_b. We call these *non-recursive* occurrences, while the occurrences of y within f are *recursive* occurrences. Recursive uses of y evaluate in the most recent contour of a non-recursive use of y. To illustrate, consider the following expression:

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

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

The remaining rules of Figure 5 handle construction, projection, and mutation of pairs. We have claimed that polymorphic splitting attempts to model Hindley-Milner polymorphism in a flow-analysis context. In light of the above discussion, it is reasonable to question its effectiveness in the presence of imperative features such as mutation on pairs. It is well-known that the intuitive explanation of let-polymorphism as a semantics-preserving substitution of the let-bound expression for all free occurrences of the let-variable in the let-body fails in the presence of mutation [Tofte 1990; Wright 1995]. The reason for this is that the contents (and therefore the type) of a polymorphic reference may change over time, and may be used in inappropriate contexts.

A flow analysis using polymorphic splitting neither preserves any temporal ordering on the elements of a mutable structure, nor differentiates procedures found in mutable structures from those that are not. For example, consider the following program fragment:

```
(let ((f (cons (lambda (x) x) '())))
  (begin
    (set-car! f (lambda (x) (not x)))
    ((car f) true)))
```

The abstract values associated with `f` in the call, `((car f) 1)` includes the abstract closure for both `(lambda (x) x)` and `(lambda (x) (not x))`. The analysis does not evaluate the two occurrences of `f` in the `let`-body in separate contours since splitting only occurs for `let`-bound variables whose abstract values are closures. Moreover, because the analysis does not impose any temporal order on the elements in the set of abstract values found at a program point, both `(lambda (x) x)` and `(lambda (x) (not x))` are merged in the `car` of `f`. Consequently, a run-time type-check optimization would be unable to eliminate a run-time check at the application of `(car f)` even though the program's exact dataflow ensures no run-time exception would be raised.

Note, however, that our polyvariant approximation will not merge abstract values of pairs *returned* by `let`-bound procedures. Each of these pairs will be associated with a different contour for each syntactically-distinct occurrence of the procedure in the `let`-body. Consider the following expression:

```
(let* ((f (lambda (x) (cons x '())))
       (g (f (lambda (y) (+ y 1))))
       (h (f (lambda (z) (not z)))))
  ((car g) 1)
  ((car h) true))
```

The abstract value of `g` is a pair whose first component is the procedure associated with `(lambda (y) (+ y 1))`, and `h`'s abstract value is a pair whose `car` is the procedure associated with `(lambda (z) (not z))`. No merging of these two procedures takes place through `x` because the analysis associates two different abstract closures with `f` for each of its two occurrences; `x` is bound to distinct contours in the two calls.

4. IMPLEMENTATION

The implementation of our analysis consists of approximately 4000 lines of Scheme code. The implementation constructs the least flow analysis for a program P by building a graph in which *nodes* represent program points containing sets of abstract values, and *edges* represent containment constraints between the sets of values at nodes [Henglein 1992; Jagannathan and Weeks 1995]. 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 holding 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, defined in Figure 6, constructs the graph for an expression reached in a particular environment and contour. The figure includes cases for only some of our language constructs; the remaining cases are similar. For constants and λ-expressions, Flow just adds the appropriate abstract value to program point $\langle l, \kappa\rangle$. For variable references, Flow adds a simple edge $\langle x, \kappa\rangle \rightsquigarrow \langle l', \kappa'\rangle$ between the appropriate program points; l' denotes the label of the reference. Such an edge indicates that every element arriving at $\langle x, \kappa\rangle$ must be propagated to $\langle l', \kappa'\rangle$.

For applications, we use *active edges* to construct new parts of the program flow graph as new values arrive. An active edge $\langle l, \kappa\rangle \overset{\lambda}{\rightsquigarrow} f$ calls function f with each new element that is added to $\langle l, \kappa\rangle$. Indeed, we can use active edges to define simple edges:

$$\langle l, \kappa\rangle \rightsquigarrow \langle l', \kappa'\rangle \;=\; \langle l, \kappa\rangle \overset{\lambda}{\rightsquigarrow} (\lambda v.\ \text{add } v \text{ to } \langle l', \kappa'\rangle).$$

A *one-shot edge* $\langle l, \kappa\rangle \overset{1}{\rightsquigarrow} f$ calls f with an uninteresting value only the first time a value arrives at $\langle l, \kappa\rangle$, i.e., when $\langle l, \kappa\rangle$ becomes non-empty. One-shot edges, which can also be defined in terms of active edges $\overset{\lambda}{\rightsquigarrow}$, allow the implementation to build pieces of the the flow graph incrementally, adhering to various reachability constraints. In this sense, one-shot edges are related to the representation of conditional types in [Aiken, Wimmers, and Lakshman 1994]; implementations of other

$$
\begin{array}{lcl}
\mathrm{Flow}(c^l, \theta, \kappa) & = & \text{add } \mathit{AbsValOf}(c) \text{ to } \langle l, \kappa \rangle \\
\mathrm{Flow}(\texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)}^l, \theta, \kappa) & = & \text{add } \langle l, \theta|_{FV(\texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)})}, \kappa \rangle_\lambda \text{ to } \langle l, \kappa \rangle \\
\mathrm{Flow}(x^l, \theta, \kappa) & = & \text{add } \langle x, \theta(x) \rangle \rightsquigarrow \langle l, \kappa \rangle \\
\mathrm{Flow}(\texttt{(}e_0^{l_0} \ldots e_n^{l_n}\texttt{)}^l, \theta, \kappa) & = & \mathrm{Flow}(e_0, \theta, \kappa); \\
 & & \text{add } \langle l_i, \kappa \rangle \overset{1}{\rightsquigarrow} (\lambda v.\ \mathrm{Flow}(e_i, \theta, \kappa)) \quad \text{for } i = 0 \ldots n-1; \\
 & & \text{add } \langle l_n, \kappa \rangle \overset{1}{\rightsquigarrow} (\lambda v.\ \text{add } \langle l_0, \kappa \rangle \overset{\lambda}{\rightsquigarrow} f) \\
 & & \text{where } f = \lambda v. \\
 & & \qquad\qquad \text{if } v = \langle l', \theta', \kappa' \rangle_\lambda \\
 & & \qquad\qquad\quad \text{let } \texttt{(}\lambda\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_b^{l_b}\texttt{)}^{l'} \in P \\
 & & \qquad\qquad\quad \mathrm{MemoFlow}(e_b, \theta'[x \mapsto \kappa'], \kappa') \\
 & & \qquad\qquad\quad \text{add } \langle l_i, \kappa \rangle \rightsquigarrow \langle x_i, \kappa' \rangle \quad \text{for } i = 1 \ldots n \\
 & & \qquad\qquad\quad \text{add } \langle l_b, \kappa' \rangle \rightsquigarrow \langle l, \kappa \rangle \\
\mathrm{Flow}(\texttt{(if}\ e_1^{l_1}\ e_2^{l_2}\ e_3^{l_3}\texttt{)}^l, \theta, \kappa) & = & \mathrm{Flow}(e_1, \theta, \kappa); \\
 & & \text{add } \langle l_1, \kappa \rangle \overset{\lambda}{\rightsquigarrow} \lambda v. \\
 & & \qquad\qquad \text{if } v = \texttt{true} \\
 & & \qquad\qquad\quad \mathrm{Flow}(e_2, \theta, \kappa) \\
 & & \qquad\qquad\quad \text{add } \langle l_2, \kappa \rangle \rightsquigarrow \langle l, \kappa \rangle \\
 & & \qquad\qquad \text{if } v = \texttt{false} \\
 & & \qquad\qquad\quad \mathrm{Flow}(e_3, \theta, \kappa) \\
 & & \qquad\qquad\quad \text{add } \langle l_3, \kappa \rangle \rightsquigarrow \langle l, \kappa \rangle \\
\mathrm{Flow}(\texttt{(let ([}x\ e_1^{l_1}\texttt{])}\ e_b^{l_b}\texttt{)}^l, \theta, \kappa) & = & \mathrm{Flow}(e_1, \theta, [\kappa : l]); \\
 & & \text{add} \langle l_1, [k : l] \rangle \rightsquigarrow \langle x, k \rangle \\
 & & \text{add} \langle x, \kappa \rangle \overset{1}{\rightsquigarrow} \lambda v. \mathrm{MemoFlow}(e_b^{l_b}, \theta[x \mapsto \kappa], \kappa); \\
 & & \qquad\qquad \text{add} \langle l_b, \kappa \rangle \rightsquigarrow \langle l, \kappa \rangle \\
\mathrm{Flow}(x^l_{[l'']}, \theta, \kappa) & = & \text{add} \langle x, \theta(x) \rangle \overset{\lambda}{\rightsquigarrow} \lambda v. \text{if } v = \langle l', \theta', \kappa' \rangle \\
 & & \qquad\qquad\quad \text{then add } \langle l', \kappa'[l''/l'] \rangle \rightsquigarrow \langle l, \kappa \rangle \\
 & & \qquad\qquad\quad \text{else add } v \text{ to } \langle l, \kappa \rangle \\
\mathrm{MemoFlow} & = & \mathrm{Memo}(\mathrm{Flow})
\end{array}
$$

Fig. 6. Procedure Flow for constructing a least flow analysis.

constraint-based control-flow systems (e.g., set-based analysis [Heintze 1994]) have a similar structure.

When an application is reached, procedure Flow first constructs the graph for the first subexpression e_0. It then adds one-shot edges that will evaluate subsequent subexpressions as the nodes for their predecessors become non-empty. Finally, when the node for the last subexpression e_n becomes non-empty, Flow establishes an active edge from the node for e_0 that, when a new closure arrives, will construct the graph for the body of the closure and establish appropriate argument and return edges. The argument edges propagate values from the nodes for the subexpressions e_i to the nodes for the formal parameters x_i. The return edge propagates values from the node for the closure body e_b to the call expression's node. In constructing the graph for the body of the closure, some care is needed to ensure that the algorithm does not loop. MemoFlow is a memoized version of Flow that simply cuts off evaluation when called with the same arguments as some previous call.

For conditionals, Flow constructs the subgraph for e_1. It then establishes an active edge from the node for e_1 that will construct the graph for e_2 or e_3 as true or false values arrive from e_1. For languages such as Scheme, where e_2 is executed if e_1 yields any value other than `#f`, a little more work beyond changing "$v = \texttt{true}$"

to "$v \neq$ `false`" must be done to ensure that $\text{Flow}(e_2, \theta, \kappa)$ is only called once.

For `let` expression with label l, Flow constructs the subgraph for the `let`-binding in a new contour constructed by the extending the current contour with l. Edges are added between the node for the binding-variable, and the node holding the result of the binding. Finally, the flow graph for the body is constructed, and a new edge is established between the node holding the result of the `let`-body and the node holding the result of the `let`-expression.

The implementation of `let`-bound variables creates an active edge between the variable node $\langle x, \kappa \rangle$ holding the value of the variable, and a procedure responsible for filtering new values propagated to the variable. This filter splits the contour of abstract closures that arrive at $\langle x, \kappa \rangle$, and leaves all other abstract values untouched.

5. OPTIMIZATIONS

This section describes two optimizations driven by our flow analysis: run-time check elimination and inlining. For both optimizations, we provide experimental results indicating their effectiveness. For run-time check elimination, we also provide a comparison between polymorphic splitting and several other kinds of analyses. We show that polymorphic splitting is often more precise and less expensive to compute. We elaborate on our notion of precision below.

The experimental results reported in the following sections were obtained by using Chez Scheme 5.0a [Dybvig 1987] to compile programs to native MIPS code. The compiled programs were run on a 150 MHz SGI Challenge with 1GByte of 8-way interleaved physical memory. Times were collected by taking the least of five runs.

5.1 Run-time check elimination

Dynamically-typed languages like Scheme require run-time checks at every use of a primitive operator to ensure that the operator's arguments lie its domain. There are three kinds of run-time checks that may be required in Scheme programs:

(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.

Ordinarily, Scheme compilers conservatively insert run-time checks at *all* primitive operations and calls except in the simplest of cases. Using the information derived by flow-analysis, the need for many run-time checks can be avoided.

Run-time checks at primitive operations can be eliminated if the values of all arguments can be proven to lie within the expected domains of the primitive operation. In our experimental framework, we assume that there are two versions of every primitive operation: one that includes a run-time check, and one that does not. In a real compiler, run-time checks for each argument could be handled independently. Let $AbsArgOf(p, i)$ be a set of abstract values describing the expected domain of the i^{th} argument of primitive operator p. For example, $AbsArgOf(+, i) = \{\mathsf{num}\}$ for $i = 1, 2$. Then a run-time check can be eliminated if the following condition holds.

Benchmark	Lines	Sites	Analysis Time (in seconds)				
			Polymorphic Splitting	0CFA	1CFA	Type Based	Soft Typing
Lattice	231	252	0.17	0.10	0.55	0.10	0.43
Boyer	559	204	1.40	29.42	58.39	29.3	3.26
Graphs	573	411	0.26	0.19	0.88	0.19	0.69
Matrix	649	502	0.35	0.28	3.96	0.28	0.85
Maze	767	615	0.49	0.27	0.72	0.27	1.40
Splay	949	685	3.74	0.71	3.94	1.06	2.23
Nbody	1659	1454	1.91	1.04	12.75	1.30	2.84
Dynamic	2147	2370	109.12	237.23	*	56.5	4.69
Nucleic	9708	910	26.24	22.95	28.47	6.63	5.92
Nucleic2	12547	1126	157.22	163.86	165.89	50.37	20.51

Fig. 7: Benchmark programs, their size (in lines of code) after macro expansion, static incidences of run-time checks for these programs in the absence of any run-time check optimization, and analysis times under polymorphic splitting, 0CFA, 1CFA, type-directed analysis, and soft typing. Under 1CFA, Dynamic exhausted 1GByte of heap space without completing.

Run-time check elimination condition. *No run-time check is necessary at* $(p\ e_1^{l_1} \ldots e_n^{l_n})$ *if*

$$\bigcup_{\kappa \in Contour} F\langle l_i, \kappa \rangle \ \subseteq\ AbsArgOf(p, i)$$

for all $i = 1 \ldots n$.

Similar conditions can be stated for eliminating arity checks and procedure checks at applications.

5.2 Experimental results for run-time check elimination

Figure 7 lists the benchmarks used to test the analysis, their size in pretty-printed lines of source code after prepending library routines, stripping comments, and expanding macros, the number of sites where run-time checks would ordinarily be required in the absence of any optimizations, and the times to analyze them under polymorphic splitting, a 0CFA implementation, a 1CFA implementation, a type-directed analysis,[3] and an implementation of soft typing [Wright and Cartwright 1997]. These benchmarks are characteristic of a wide range of Scheme programs, and none were written with knowledge of the analysis or inlining strategies used.

The program Lattice enumerates the lattice of maps between two lattices, and is mostly first-order. Boyer is a term-rewriting theorem prover. Graphs counts the number of directed graphs with a distinguished root and k vertices, each having out-degree at most 2. It makes extensive use of higher-order procedures, and is written in a continuation-passing style. Given an $n \times n$ random matrix M with {+1, -1} entries, Matrix tests whether M is maximal among all matrices of the same dimension obtainable by simple re-ordering of rows and columns, and negation of any subset of rows and columns. Like Graphs, this program is written in continuation-passing style. Maze generates a random maze and computes a path through it using a union-find algorithm. It makes extensive use of records [Wright and Duba 1993]

[3]The type-directed analysis uses the outermost type constructors of the arguments to the two most recent procedure calls to determine the contour in which to analyse a procedure body.

and uses Scheme's `call-with-current-continuation` operator, but is primarily a first-order program. Splay is an implementation of splay trees. It makes extensive use of higher-order procedures and pattern matching macros [Wright and Duba 1993]. Nbody is a Scheme implementation [Zhao 1987] of the Greengard [1987] multipole algorithm for computing gravitational forces on point-masses distributed uniformly in a cube. Dynamic is an implementation of a tagging optimization algorithm [Henglein 1992] for Scheme. It is primarily a first-order program, but has complex control-flow, and many deeply-nested conditional expressions. Nucleic is a constraint satisfaction algorithm used to determine the three-dimensional structure of nucleic acids [Feeley, Turcotte, and Lapalme 1994]. It is floating-point intensive and uses an object package implemented with macros and vectors. Nucleic2 uses a different object package as described below.

For the larger benchmarks, soft typing requires less analysis time than any of the flow analyses. Because the soft typing implementation uses a Hindley-Milner type inference framework, the body of a procedure is analyzed only once. Type constraints within the procedure body are calculated using type variables to express the dependence between unspecified inputs and outputs. These constraints are duplicated for different calls to the procedure via polymorphic type variable instantiation without being recalculated from scratch. In contrast, our flow analysis algorithm establishes multiple copies of the constraint rules between program points of a procedure's body, one copy for each contour in which the body is analyzed. If a procedure is never reached, our algorithm is cheaper than type inference because it never calculates any constraints. But in the more common case where a procedure is reached under many contours, type inference is cheaper because it calculates the constraints only once. Nevertheless, we believe the times obtained from our *prototype* flow analysis implementation indicate that this technique should be suitable for inclusion in an optimizing compiler.

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 bodies 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.

Figure 8 shows the percentage of *static* checks remaining in the benchmarks under the different analyses after run-time check elimination. The static counts measure the textual number of run-time checks remaining in the programs. For each of the benchmarks, polymorphic splitting requires fewer run-time checks than either 0CFA or soft typing.[4] 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

[4]Note that the soft typing implementation analyzes several of these programs more precisely if allowed to use its own special definitions of library routines such as `map`, `append`, and `apply`.

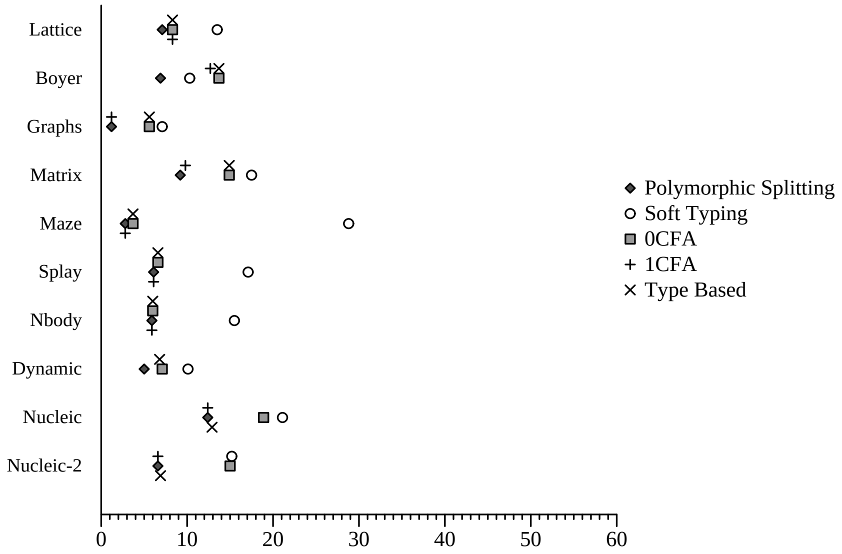

Fig. 8. Percentage of static checks remaining after run-time check optimization.

using soft typing. The primary reason for this is the problem of *reverse flow* [Wright and Cartwright 1997] 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. As the specification shown in Fig. 2 reveals, flow analysis does not suffer from the reverse flow problem. Soft typing implementations that use a richer type system may not exhibit this problem [Aiken, Wimmers, and Lakshman 1994].

Figure 9 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. 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 performs 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

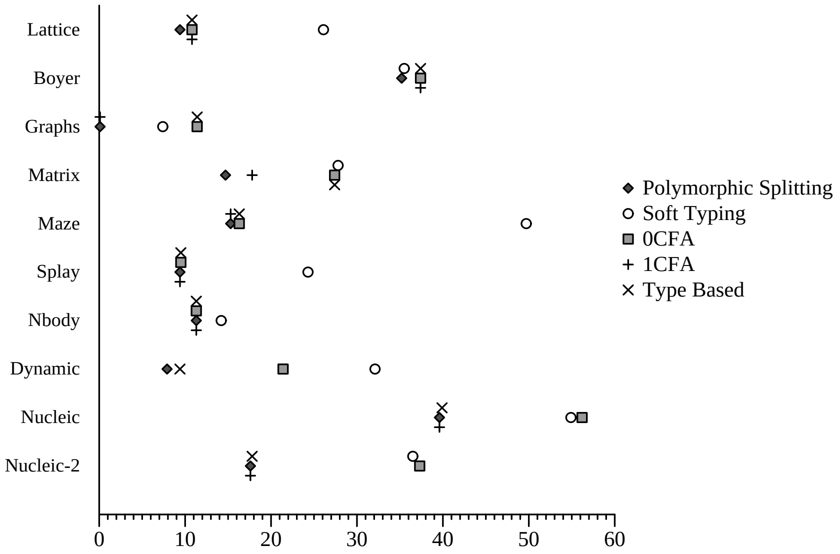

Fig. 9. Percentage of dynamic checks remaining after run-time check optimization.

will outperform those that are not. The other two polyvariant analyses studied, 1CFA and the type-directed analysis, reduce run-time checks as well as polymorphic splitting on six of the eight benchmarks. While 1CFA seems to be prohibitively expensive, type-directed flow analyses appear to share many of the positive features of polymorphic splitting [Jagannathan, Weeks, and Wright 1997].

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 symbolic tags 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.

Figure 10 presents execution times for the benchmarks after polymorphic splitting has been used to eliminate run-time checks. The top of the graph (1.0 on the y-axis) represents the normalized execution time of the unoptimized program. Execution time is further broken down by color into mutator time excluding run-time checks (black), garbage collection time (grey), and run-time checking remaining (white). The white sections of the bars indicate run-time checking overhead that the analysis was unable to safely eliminate. For Graphs, Matrix, Dynamic, and Nucleic2, the analysis was sufficiently precise to eliminate essentially all run-time check

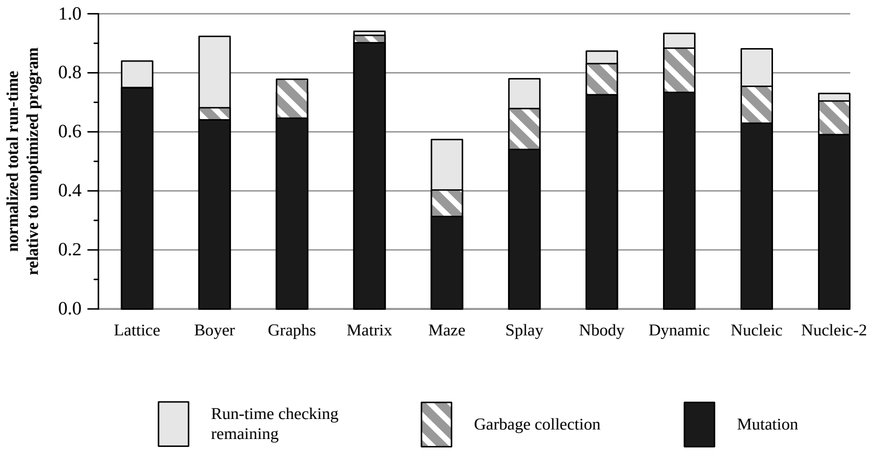

Fig. 10. Execution time after run-time check elimination.

overhead. Even for those benchmarks where run-time check costs were measurable, the optimization was still able to yield significant performance improvement (as indicated by the gap between 1.0 and the top of each bar).

5.3 Inlining

The second interprocedural optimization we consider is inlining. Polyvariant flow analysis is especially useful here for two reasons. First, because it disambiguates different uses of a procedure at different call sites, polyvariance yields very precise flow information. This enables many call sites to be identified as candidates for inlining. Second, because it analyses a procedure with respect to a specific call site, polyvariant flow analysis enables specialization of the procedure for the call sites where it is inlined.

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

```
(define g (lambda (f x) (f x)))
```

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

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

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

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

The proviso $n = p$ ensures that we do not eliminate applications which raise an error due to an argument count mismatch.

To illustrate how flow analysis enables potential inline sites to be identified, consider the following object-oriented program fragment:

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

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

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

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

```
(lambda (msg) (case msg ...))
```

and `(N 'open)` with the procedure

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

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

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

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

Besides identifying potential inline sites, flow analysis also helps to compute reasonably efficient cost estimates. To illustrate, consider Figure 11 which provides an implementation of Scheme's `map` procedure. Two local procedures are defined in `map`'s body: `map1` is called when `map` is invoked with a unary procedure, and `map*` is invoked for all other cases. Because `map1` operates over procedures whose arity is known, it is more efficient than `map*` whose implementation uses an expensive `apply` operator to handle the variable-arity case.

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

Fig. 11. Scheme's `map` procedure

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

Fig. 12. Result of inlining `(map car m)` prior to simplification.

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

Fig. 13. Result of inlining `(map car m)` after simplification.

Now, consider the call `(map car m)`. The cost of inlining `map` at this call site is dependent on how much specialization can be performed at the inlined site. Because the inlining algorithm uses results from a polyvariant flow analysis, it can tailor its cost estimate based on the specific context in which a call occurs. In this call, flow analysis reveals that the conditional test `(null? args)` will be true, and hence the else-branch will not be taken. Thus, the cost estimate for this call need not include the else branch or the definition of `map*`. If `map` is inlined, the outer conditional test, the definition of `map*`, and its call will all be pruned in the inlined copy. Figure 12 shows the result of inlining `(map car m)` prior to simplification.

After inlining, β_v-substitution, dead-code elimination, and local code transformations are performed. These transformations include restructuring procedure definitions and calls to eliminate unused formal parameters. These optimizations are all syntactic and use no flow information. The simplifier performs no transformations that violate the results of the flow analysis. Hence, other optimizations could use flow information generated for the original program when operating over the inlined version. Figure 13 shows the result of applying the simplifier to the inlined code for `(map car m)`.

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

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

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

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

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

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

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

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

Inlining procedures with free variables

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

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

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

The target language also includes an expression form

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

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

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

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

We replace the call with

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

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

The `cl-ref` operation is easy to implement in a compiler that uses flat closures [Appel 1992]. A closure record consists of a code pointer and the values of the variables in the free variable list for that procedure. A compiler that uses linked closures may require additional information in the `cl-ref` operation to indicate the forms of the linkage structures.

An inlining algorithm

Figure 14 collects the above observations into a complete inlining algorithm. Given a flow analysis F for a program P, the transformation $\mathcal{I}[\![e]\!]\kappa\mu$ takes a subexpression e of P, a contour κ, and a loop map μ, and produces an equivalent inlined and specialized expression. The initial contour for P is the special contour ? and the initial loop map is the empty map. The contour ? denotes the union of all possible contours, as in Inlining Condition 1. In other words,

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

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

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

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

where $[z_1 \ldots z_m] = FV(\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)})$

$$\mathcal{I}[\![\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)}]\!]\kappa\mu = \texttt{(lambda}\ [z_1 \ldots z_m]\ \texttt{(}x_1 \ldots x_n\texttt{)}\ \mathcal{I}[\![e_b]\!]?\mu\texttt{)}$$

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

$$\begin{array}{rcl}\mathcal{I}[\![\texttt{(}e_0^{l_0}\ e_1 \ldots e_n\texttt{)}]\!]\kappa\mu & = & \texttt{(letrec} \\ & & \quad \texttt{([}y\ \texttt{(lambda (}w\ x_1 \ldots x_n\texttt{)} \\ & & \qquad\quad \texttt{(let ((}z_1\ \texttt{(cl-ref}\ w\ 1\texttt{)))} \\ & & \qquad\qquad\qquad \vdots \\ & & \qquad\qquad \texttt{((}z_m\ \texttt{(cl-ref}\ w\ m\texttt{)))} \\ & & \qquad\quad \mathcal{I}[\![e_b]\!]\kappa'\mu'\texttt{))])} \\ & & \quad \texttt{(}y\ \mathcal{I}[\![e_0]\!]\kappa\mu\ \mathcal{I}[\![e_1]\!]\kappa\mu \ldots \mathcal{I}[\![e_n]\!]\kappa\mu\texttt{))}\end{array}$$

if $e_0 \neq \texttt{(lambda} \ldots\texttt{)}$ and $F\langle l_0, \kappa\rangle = \{\langle l', \rho', \kappa'\rangle_\lambda\}$ and $\langle l', \kappa'\rangle \notin \mathrm{Dom}(\mu)$ and *Inline?* $(\mathcal{I}[\![e_b]\!]\kappa'\mu')$
where $\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)}^{l'}$ in P
$\mu' = \mu[\langle l', \kappa'\rangle \mapsto y]$
$[z_1 \ldots z_m] = FV(\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ e_b\texttt{)})$
y, w fresh

$$\mathcal{I}[\![\texttt{(}e_0^{l_0}\ e_1 \ldots e_n\texttt{)}]\!]\kappa\mu = \texttt{(}y\ \mathcal{I}[\![e_0]\!]\kappa\mu\ \mathcal{I}[\![e_1]\!]\kappa\mu \ldots \mathcal{I}[\![e_n]\!]\kappa\mu\texttt{)}$$

if $e_0 \neq \texttt{(lambda} \ldots\texttt{)}$ and $F\langle l_0, \kappa\rangle = \{\langle l', \rho', \kappa'\rangle_\lambda\}$ and $\langle l', \kappa'\rangle \in \mathrm{Dom}(\mu)$ where $\mu\langle l', \kappa'\rangle = y$

$$\mathcal{I}[\![\texttt{(}e_0^{l_0}\ e_1 \ldots e_n\texttt{)}]\!]\kappa\mu = \texttt{(}\mathcal{I}[\![e_0]\!]\kappa\mu\ \mathcal{I}[\![e_1]\!]\kappa\mu \ldots \mathcal{I}[\![e_n]\!]\kappa\mu\texttt{)}$$

if $e_0 = \texttt{(lambda} \ldots\texttt{)}$ or $F\langle l_0, \kappa\rangle \neq \{\langle l', \rho', \kappa'\rangle_\lambda\}$ or not *Inline?* $(\mathcal{I}[\![e_b]\!]\kappa'\mu')$ where $\mu\langle l', \kappa'\rangle = y$

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

if $\mathsf{true} \in F\langle l_1, \kappa\rangle$ and $\mathsf{false} \in F\langle l_1, \kappa\rangle$

$$\mathcal{I}[\![\texttt{(if}\ e_1^{l_1}\ e_2\ e_3\texttt{)}]\!]\kappa\mu = \texttt{(if}\ \mathcal{I}[\![e_1]\!]\kappa\mu\ \mathcal{I}[\![e_2]\!]\kappa\mu\ \mathcal{I}[\![e_3]\!]\kappa\mu\texttt{)}$$

if $\mathsf{true} \in F\langle l_1, \kappa\rangle$ and $\mathsf{false} \notin F\langle l_1, \kappa\rangle$

$$\mathcal{I}[\![\texttt{(if}\ e_1^{l_1}\ e_2\ e_3\texttt{)}]\!]\kappa\mu = \texttt{(begin}\ \mathcal{I}[\![e_1]\!]\kappa\mu\ \mathcal{I}[\![e_2]\!]\kappa\mu\texttt{)}$$

if $\mathsf{true} \notin F\langle l_1, \kappa\rangle$ and $\mathsf{false} \in F\langle l_1, \kappa\rangle$

$$\mathcal{I}[\![\texttt{(if}\ e_1^{l_1}\ e_2\ e_3\texttt{)}]\!]\kappa\mu = \texttt{(begin}\ \mathcal{I}[\![e_1]\!]\kappa\mu\ \mathcal{I}[\![e_3]\!]\kappa\mu\texttt{)}$$

if $\mathsf{true} \notin F\langle l_1, \kappa\rangle$ and $\mathsf{false} \notin F\langle l_1, \kappa\rangle$

$$\mathcal{I}[\![\texttt{(if}\ e_1^{l_1}\ e_2\ e_3\texttt{)}]\!]\kappa\mu = \mathcal{I}[\![e_1]\!]\kappa\mu$$

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

$$\mathcal{I}[\![\texttt{(letrec ((}y\ e_1^{l_1}\texttt{))}\ e_2\texttt{)}^l]\!]\kappa\mu = \texttt{(letrec ((}y\ \mathcal{I}[\![e_1]\!]\kappa\mu'\texttt{))}\ e_2\texttt{)}$$

where $\mu' = \mu[\langle l_1, \kappa\rangle \mapsto \bullet]$

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

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

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

Fig. 14. A flow-directed inlining algorithm

To prevent uncontrolled unwinding of recursive calls, the loop map μ maps label-contour pairs to either variables or the special symbol $\bullet$. If, while inlining procedure l' in contour κ', the algorithm encounters another call site for l' in contour κ' (see the second case of the rule for applications), it constructs a loop by binding a fresh variable to the inlined procedure at the initial call site with `letrec`. In fact, the algorithm introduces a `letrec` for every inlined procedure, whether recursive or not, but subsequent local simplification removes any unnecessary bindings. For `letrec`-expressions occurring in the program, and not introduced by the inlining transformation, the `letrec`-bound variable y is bound to $\bullet$ in the loop-map. This mapping ensures that calls to y are not transformed to calls which pass an extra closure argument since only inlined procedures bound to `letrec`-expressions introduced by the transformation expect an additional closure record as their first argument.

This method of controlling unwinding avoids unrolling loops. Loop unrolling can be a valuable optimization and would be easy to include in this framework. We have intentionally avoided unrolling loops in order to isolate the benefits of inlining. Similarly, the algorithm would be simpler if we closure-converted all procedures regardless of whether they were inlined or not, but would have been expensive because our transformations are all performed at the source-language level.

Making inlining decisions

Inlining every call site that is a candidate for inlining is impractical as it can lead to exponential code growth. To limit inlining to those sites where it is likely to be most profitable, Figure 14 requires the predicate *Inline?* to hold for the procedure body when specialized for the contour and loop map under consideration. This predicate estimate the sizes of the generated code for the inlined procedure at a particular call site, and limits inlining to cases where the generated code is smaller than some threshold size.[5] In the next section, we present performance results for various inlining thresholds.

5.4 Experimental results for inlining

We have implemented a source-to-source flow-directed inlining optimization for the full R^4RS Scheme language [Clinger, Rees, et al. 1991]. Given a Scheme program and an inline threshold T, our optimizer inlines procedures whose specialized size is estimated to be less than T.

Without access to Chez's internal data representations, it is impossible to implement `cl-ref` with the same efficiency as a variable reference. Indeed, using a faithful implementation of the algorithm from the previous section, we found the premium for accessing variables via `cl-ref` masked the benefit of inlining. Therefore, the results presented in this section reflect an inlining strategy in which the *Inline?* predicate requires inlined procedures be closed up to top-level variables. Under this constraint, the inlining algorithm never generates `cl-ref` operations. The algorithm will still allow a procedure P that has a free variable x to be inlined at call site C if (i) x occurs in a conditional branch which can be eliminated in

[5]The algorithm in Figure 14 requires constructing the specialized version of the procedure body before deciding if it should be inlined. Our implementation uses an equivalent but more efficient algorithm that estimates the size of the specialized procedure without actually constructing it.

Program	Ratio of object code size to original object code size for various inlining thresholds				
	50	100	200	500	1000
Lattice	0.86	0.86	0.88	1.38	2.49
Boyer	0.92	1.00	1.03	0.89	0.87
Graphs	0.74	0.80	0.75	0.73	0.72
Matrix	0.92	0.96	0.95	1.00	1.16
Maze	0.89	0.91	0.88	0.86	0.83
Splay	0.91	0.91	0.91	0.91	0.91
Nbody	1.12	1.38	1.47	2.01	2.28
Dynamic	1.41	1.53	2.15	2.25	2.25
Nucleic	0.77	0.79	1.09	1.22	1.61
Nucleic2	0.86	0.87	1.15	1.15	1.69

Fig. 15. Code size ratios for various inlining thresholds.

the specialized copy of P at C, or (ii) x refers to a procedure that that will be inlined. In either case, a free variable in the source does not appear in the specialized version. Surprisingly, this inlining strategy leads to consistent performance improvements for most programs we have tested. For many of the benchmarks in our test suite, the performance improvements are significant. We would expect even greater improvements with an efficient implementation of `cl-ref` since this would enable inlining open procedures.

We applied our optimizer to the benchmarks listed in Table 7. Figure 15 indicate code sizes after inlining for various size thresholds.[6] For example, the call `(map car m)` from Figs. 11, 12, and 13 is inlined at thresholds above 60.

Code size ratios less than one indicate that the inlined program is *smaller* than the original. For example, at threshold 1000, the inlined version of Graphs is still 28% smaller than the original program. For most benchmarks, object code size grows quite slowly as the inlining threshold increases. There are two reasons for this. First, Scheme programs typically consist of many small procedures, so higher inlining thresholds expose relatively fewer opportunities for inlining. Second, even when relatively large procedures become candidates for inlining, specialization and local simplification significantly reduce the size of the inlined copy.

Figure 16 presents execution times for these benchmarks under different inlining thresholds. The leftmost bar indicates the execution times of the original programs *after* local simplification (i.e., threshold 0). These times reflect inlining of primitives and small built-in procedures by the compiler. The dark section of each bar indicates mutator time, and the grey section indicates collector time. The y-axis of each graph measures normalized total execution time, which is indicated by the total heights of the bars. Mutator execution time, the time taken by the application when garbage collection overhead is discounted, is indicated by the tops of the dark bars. For example, at size threshold 200, `Graphs` shows a nearly 40% performance improvement in total execution time.

As size thresholds increase, performance either increases or remains roughly constant. This indicates that our size metric is benign, and inlining decisions rarely

[6]Code sizes are smaller than reported in our conference paper [Jagannathan and Wright 1996] because we improved the simplifier.

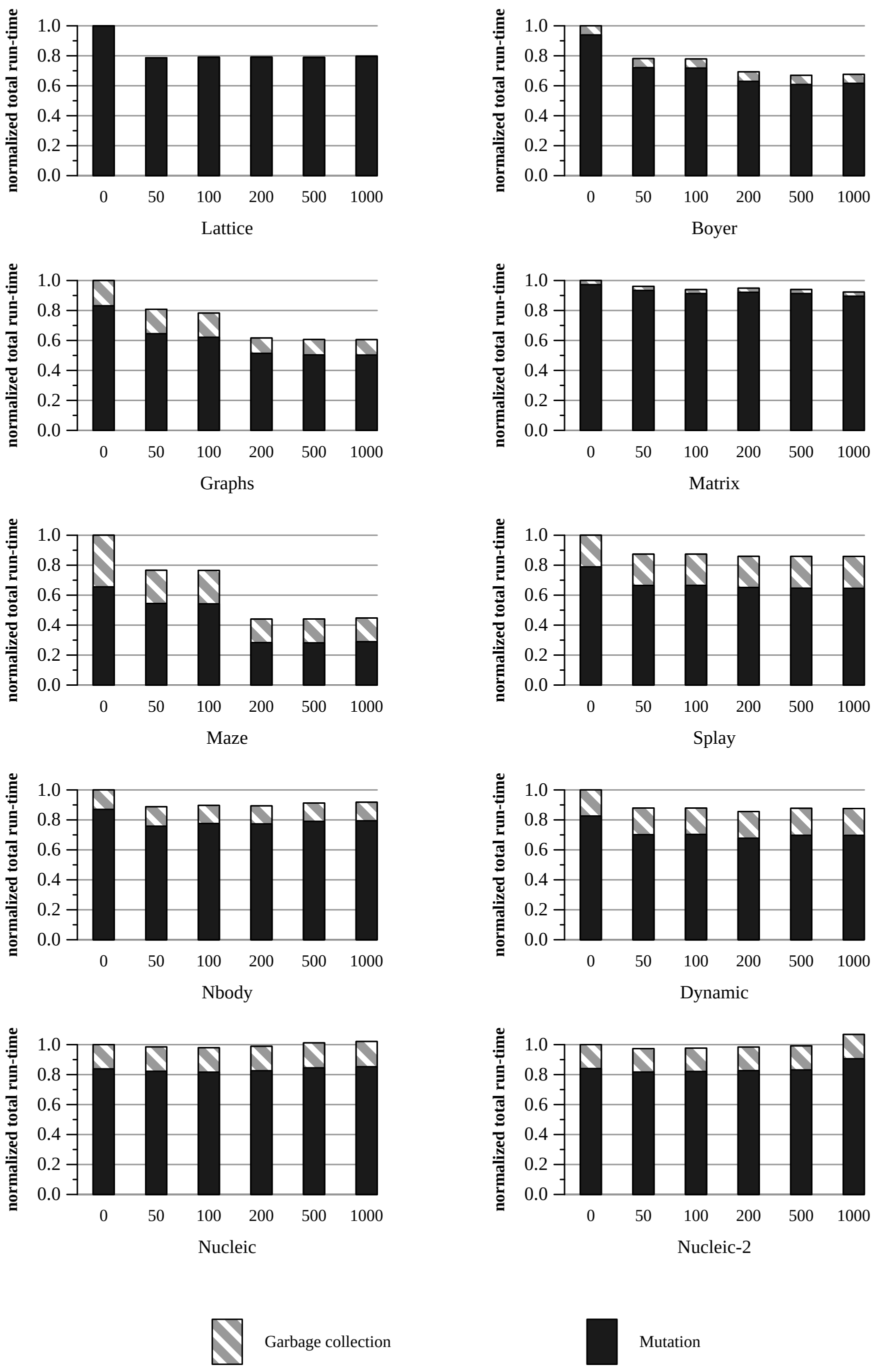

Fig. 16. Execution times under different inlining thresholds.

impact performance negatively. On average, the best performance occurs at size thresholds of about 200. At smaller thresholds, our implementation inlines small library procedures like `cadr` or a specialized version of `map`, but is unlikely to inline non-trivial user-defined procedures. At higher thresholds, more user-defined procedures are inlined, and more opportunities for specialization of these procedures become apparent. Few additional procedures are inlined at thresholds above 500; thus, we see little further performance improvement. Both Nbody and Matrix show performance improvement of roughly 10%, and Nucleic exhibits no improvement. A relatively large percentage of the execution time for these programs is spent in tight loops, and not in calls to out-of-line procedures. Hence, inlining exposes few optimization opportunities. On the other hand, Graphs, Boyer, and Maze make many out-of-line calls relative to the overall computation. Inlining these calls removes significant overhead in these benchmarks.

Garbage collection time and space allocated are not affected by inlining for most of the benchmarks in Figure 16. The exception is Graphs, which shows a reduction of 50% in both collection time. This reduction is a consequence of reducing allocation by 43%. Thus inlining can even improve space usage by eliminating intermediate data structures.

5.5 Combining run-time check elimination and inlining

As the information computed by flow analysis is not destroyed by run-time check optimization, we can drive both run-time check elimination and inlining with the information from one analysis pass. But a more intelligent combination of the two optimizations is possible than simply applying one after the other. The specialization contours of our inlining optimization enable a stronger condition for detecting when a run-time check can be eliminated than our original condition described on page 22. When a procedure containing a primitive operation is being inlined at contour κ, this stronger condition need only inspect the flow function F at contour κ to determine if the primitive operation needs a run-time check.

Run-time check elimination under inlining condition. *Let $\langle m, \rho, \kappa \rangle_\lambda$ be an abstract closure that is being inlined and specialized to contour κ. No run-time check is necessary at primitive* ($p\ e_1^{l_1} \ldots e_n^{l_n}$) *within the specialized version of m if*

$$F\langle l_i, \kappa \rangle \ \subseteq \ AbsArgOf(p, i)$$

for $i = 1 \ldots n$.

Consequently, inlining and run-time check elimination can function synergistically to reduce run-time check overhead below what run-time check elimination alone can achieve.

Figure 17 presents execution times for our benchmarks under this combination of run-time check elimination and inlining. The leftmost bar indicates the original, unoptimized program. The dark section of each bar indicates mutator time; the grey section indicates collector time; and the white section indicates run-time check overhead. Lattice and Maze show the synergistic effect of combining run-time check elimination and inlining. The run-time check overhead remaining in Lattice with no inlining (threshold 0) is cut by one half at inline thresholds of 50 or greater. Similarly, the run-time check overhead remaining in Maze is significantly reduced at thresholds of 200 or greater. Taken together, run-time check elimination and

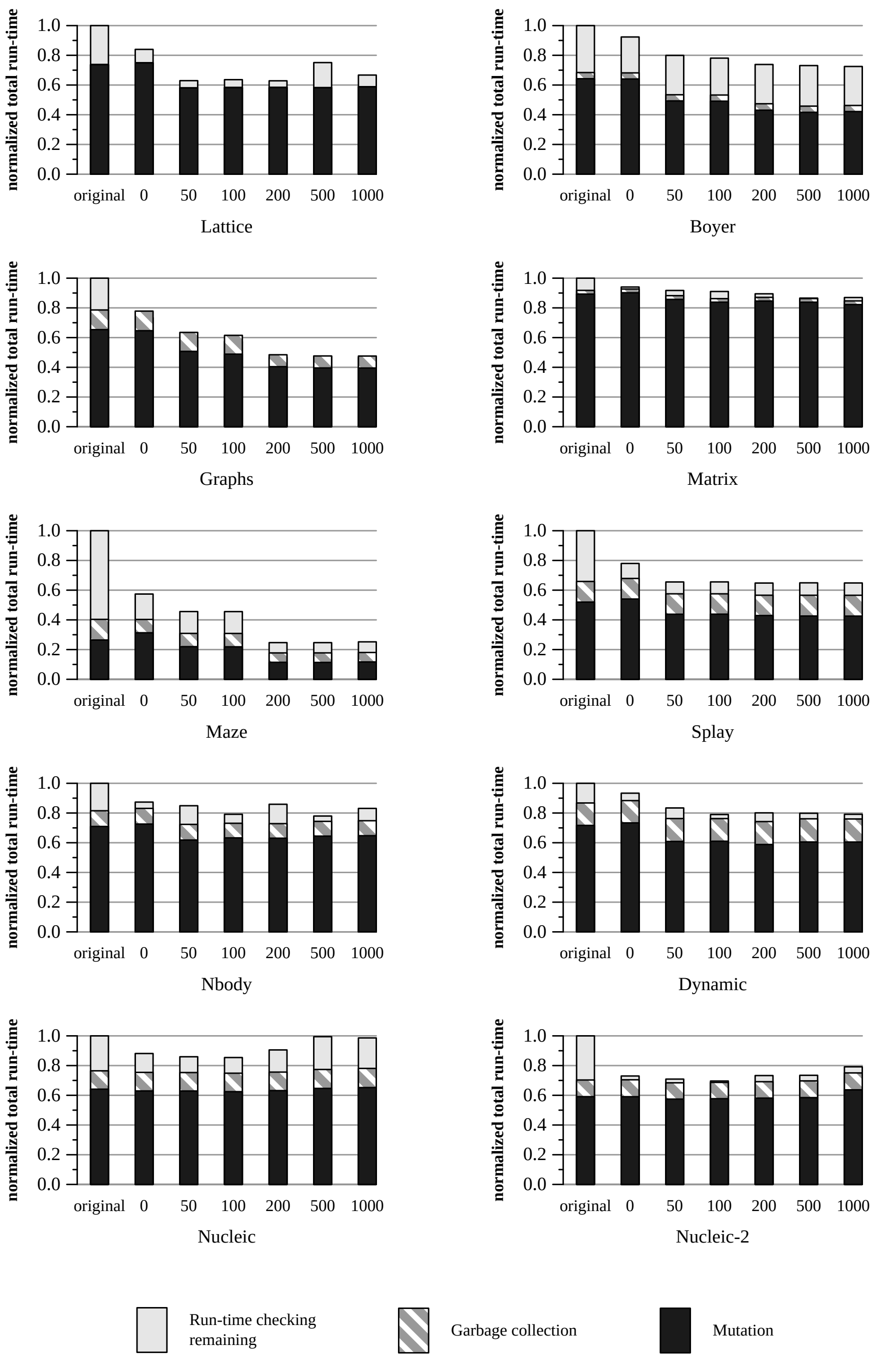

Fig. 17. Execution times for run-time check elimination and inlining combined.

inlining driven by a polyvariant flow analysis can have a significant impact on program performance.[7]

6. RELATED WORK

Three independent topics related to our work have been investigated by other researchers: flow analysis, run-time check elimination, and inlining.

6.1 Flow analysis

Flow analysis can facilitate a number of optimizations essential to implementing realistic high-level languages efficiently. Although flow analysis for imperative languages has been studied for many years [Jones and Muchnick 1979], its application to functional languages is relatively recent. Efforts at building serious implementations of functional languages that employ flow analysis are even rarer. Besides our work, we are aware of only a few implementations of interprocedural flow analysis for functional languages [Ashley 1996; Flanagan and Felleisen 1995; Heintze 1994; Serrano and Feeley 1996]. All of these implementations are based on a simple monovariant (OCFA) analysis of the source program.

A well-known approach to polyvariance proposed by Shivers [1991] is to distinguish different procedure calls by call site. In a framework similar to ours, contours are finite strings of call site labels. An N-CFA analysis uses the most recent N call sites to distinguish different invocations of a procedure. While call-string based analyses are highly precise, they also appear to be quite expensive to compute, as we illustrated in Section 5.2. Polymorphic splitting provides as good or better precision than polyvariant call-string analyses at significantly lower cost.

Type-directed analyses have been investigated for optimizing object-oriented languages [Chambers and Ungar 1990; Diwan, Moss, and McKinley 1996]. For example, the Self compiler performs significant program analysis to optimize method calls based on the expected type of the method's arguments. Unlike call-string approaches, type-directed analyses for languages like Scheme appear to offer reasonable precision with less cost.

6.2 Run-time check elimination

For many years, optimizing compilers for Lisp-like languages have used type recovery procedures based on simple syntactic properties and local flow analysis to eliminate unnecessary run-time checks [Beer 1987; Kaplan and Ullman 1980; Ma and Kessler 1990; MacLachlan 1992]. These techniques can be reasonably effective [Steenkiste and Hennessy 1987], but because they are intraprocedural they do not handle higher-order functions well.

Shivers [1990] describes a type recovery method based on his polyvariant flow analysis framework using call-strings. Our experiments indicate that a flow analysis based on polymorphic splitting is less costly and more effective.

Type inference is an alternative to flow analysis for recovering type information for untyped programs. Soft typing [Wright and Cartwright 1997; Aiken, Wimmers,

[7] We were unable to determine the reasons behind the unusual increases in run-time check overhead in Lattice at threshold 500 and in Nucleic at higher thresholds. We conjecture that register allocation or instruction selection led to additional spill code or cache conflicts. Indeed, when these programs are run on a Pentium, the magnitudes of these spikes are greatly reduced.

and Lakshman 1994; Cartwright and Fagan 1991] uses traditional polymorphic type inference techniques, extended with union types and recursive types, to derive type information that is then used to eliminate run-time checks. Our experimental results indicate that, for at least the soft-typing system described by Wright and Cartwright [1997], flow analysis is more precise. Richer soft-typing systems such as that described by Aiken, Wimmers, and Lakshman [1994] which also support conditional types and subtyping may yield more precise type information, but we are unaware of implementations of these systems for Scheme. Henglein [1992] describes a tagging and checking optimization based on an extension to simple 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. Henglein and Rehof [1995] extend Henglein's system for optimizing run-time tagging and checking to include polymorphism and union types. They use their system to translate Scheme programs to typable ML programs, injecting values tagged into an ML datatype with variants for the different Scheme types. By inferring both tag checking and tagging coercions, their system also permits using untagged data representations (uninjected ML values). Aiken and Fahndrich [1995] show how to use set constraints to infer tagging and checking coercions. Their system subsumes Henglein's original system and is quite similar to Henglein and Rehof's system. While the information inferred by these systems is ideal for eliminating run-time checks, it is of less use for other optimizations like inlining.

6.3 Inlining

Cooper, Hall, and Torczon [1991, 1992] studied the effects of inlining for Fortran programs. They determined inlining sites by hand, and were able to eliminate the majority of dynamically executed procedure calls from their benchmark programs. Furthermore, they found that the opportunities revealed by inlining mitigated object code growth—object code grew by only 10% when inlining doubled the size of the source code. However, they were unable to demonstrate consistent speedups for the inlined programs. They speculate that inlining did enable useful optimizations, but the speedup from these optimizations was masked by effects like increased register pressure, and pessimistic assumptions made by the compiler about aliasing which lead to poor instruction scheduling.

In a later study [Cooper, Hall, and Torczon 1992], they examined one program that ran 8.5% slower after inlining despite a decrease in code size of 9%. They found that inlining had forced the compiler to make pessimistic assumptions about aliasing and hence scheduled instructions poorly in an inner loop. Better instruction scheduling was possible in the non-inlined program because the Fortran standard allows compilers to assume that the arguments to a procedure call do not alias.

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

Chang *et al* [1992] constructed an automatic inlining optimization for C programs. As with Cooper *et al*'s experiments with Fortran, Chang *et al* were able to eliminate the majority of procedure calls while increasing object code size by only 16%. They obtained consistent performance improvements that averaged about 10%.

Much attention has been devoted to reducing the overhead of dynamically-bound method dispatches in object-oriented languages. Method calls that can lead to only one receiver can be replaced with direct procedure calls or inlined. Even if multiple receivers are possible, the most likely receiver can be inlined by preceding the inlined code with an appropriate conditional test. Another solution is to streamline method dispatches for the most common case using either profiling information from test runs [Grove, Dean, Garrett, and Chambers 1995] or inlining trials [Dean and Chambers 1994]. An inlining trial captures the effect of an inlining optimization at a call-site in a persistent database that is consulted when making future inlining decisions at similar calls encountered elsewhere in the program or during later compiles. More recently, static analyses have been applied to either inline most dispatches or replace them with direct procedure calls [Agesen and Hölzle 1995; Dean, Chambers, and Grove 1995; Diwan, Moss, and McKinley 1996; Pande and Ryder 1994].

Functional language compilers like Standard ML of New Jersey [Appel 1992] use syntactic heuristics to guide inlining decisions. A call site is a candidate for inlining if the called procedure is syntactically evident; typically, the function expression is a variable that is bound to a λ-expression by `let`. Inlining is intertwined with other syntactic optimizations, so additional inlining candidates can arise as optimization proceeds. In contrast, flow-directed inlining separates inlining decisions from post-inlining simplifications. Heuristics that estimate object code size, similar to ours, are used to control code growth. But these heuristics are necessarily more conservative, as they cannot take into account future simplifications that may take place if inlining is performed. Furthermore, because candidates for inlining are identified syntactically, procedures used in a higher-order manner will not be inlined unless other simplifications reduce the higher-order uses to syntactic ones. We believe that for programs which make heavy use of data and procedural abstraction, flow-directed inlining is likely to identify more profitable sites for inlining.

7. CONCLUSION

Until recently, research in global program analysis has been primarily theoretical. We have proposed a new polyvariant global program analysis that has a firm theoretical grounding. Additionally, we have experimentally investigated the effectiveness of this analysis for driving global program optimizations. Our results indicate that precise polyvariant analyses like ours can be effective for optimizing functional languages. Indeed, our experiments reveal that precise interprocedural flow information can be computed without prohibitive cost, and can often be cheaper to compute than coarser information. We expect that implementations of similar analyses and optimizations for other high-level functional and object-oriented languages will yield similar results. We anticipate that further positive results will lead to the inclusion of sophisticated program analyses into next generation optimizing compilers for advanced programming languages.

ACKNOWLEDGMENTS

We thank Henry Cejtin, Richard Kelsey, Stephen Weeks, and the anonymous referees for many helpful suggestions.

REFERENCES

AGESEN, O. AND HÖLZLE, U. 1995. Type Feedback vs. Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *10^{th} Annual Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, New York, pp. 91–107. ACM Press.

AIKEN, A. AND FÄHNDRICH, M. 1995. Dynamic Typing and Subtype Inference. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, New York, pp. 182–191. ACM Press.

AIKEN, A., WIMMERS, E. L., AND LAKSHMAN, T. K. 1994. Soft Typing with Conditional Types. In *Proc. 21st Annual Symposium on Principles of Programming Languages*, New York, pp. 163–173. ACM Press.

APPEL, A. 1992. *Compiling with Continuations.* Cambridge University Press, New York.

ASHLEY, J. M. 1996. A Practical and Flexible Flow Analysis for Higher-Order Languages. In *Proc. 23^{rd} Annual Symposium on Principles of Programming Languages*, New York, pp. 184–194. ACM Press.

BARENDREGT, H. P. 1984. *The Lambda Calculus: Its Syntax and Semantics* (Revised ed.), Volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam.

BEER, R. D. 1987. Preliminary report on a practical type inference system for Common Lisp. *Lisp Pointers 1,* 2, 5–11.

BULYONKOV, M. 1984. Polyvariant Mixed Computation for Analyzer Programs. *Acta Inf. 21*, 473–484.

CARTWRIGHT, R. AND FAGAN, M. 1991. Soft Typing. In *Proc. Conf. on Programming Language Design and Implementation*, New York, pp. 278–292. ACM Press.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proc. Conf. on Programming Language Design and Implementation*, New York, pp. 150–164. ACM Press.

CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., AND HWU, W.-M. W. 1992. Profile-Guided Automatic Inline Expansion for C Programs. *Softw. Pract. Exper. 22,* 5 (May), 349–369.

CLINGER, W., REES, J., ET AL. 1991. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers IV,* 3 (July), 1–55.

COOPER, K., HALL, M., AND TORCZON, L. 1991. An Experiment with Inline Substitution. *Softw. Pract. Exper. 21,* 6, 581–601.

COOPER, K., HALL, M., AND TORCZON, L. 1992. Unexpected Side Effects of Inline Substitution: A Case Study. *ACM Lett. Program. Lang. Syst. 1,* 1, 22–32.

DEAN, J. AND CHAMBERS, C. 1994. Towards Better Inlining Decisions Using Inlining Trials. In *Proc. Conf. on Lisp and Functional Programming*, New York, pp. 294–306. ACM Press.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1995. Selective Specialization for Object-Oriented Languages. In *Proc. Conf. on Programming Language Design and Implementation*, New York, pp. 93–102. ACM Press.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *11^{th} Annual Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, New York. ACM Press.

DYBVIG, K. 1987. *The Scheme Programming Language.* Prentice-Hall, Inc., London.

FEELEY, M., TURCOTTE, M., AND LAPALME, G. 1994. Using MultiLisp for Solving Constraint Satisfaction Problems: An Application to Nucleic Acid 3D Structure Determination. *J. Lisp Symb. Comput. 7,* 2/3, 231–248.

FELLEISEN, M. AND FRIEDMAN, D. P. 1986. Control Operators, the SECD-Machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pp. 193–217. North-Holland.

FLANAGAN, C. AND FELLEISEN, M. 1995. Set-Based Analysis for Full Scheme and Its Use in Soft-Typing. Tech. Rep. TR95-253 (Oct.), Rice University.

GREENGARD, L. 1987. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, New York.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C. 1995. Profile-Guided Receiver Class Prediction. In *10th Annual Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, New York, pp. 108–123. ACM Press.

HEINTZE, N. 1994. Set-Based Analysis of ML Programs. In *Proc. Conf. on Lisp and Functional Programming*, New York, pp. 306–317. ACM Press.

HENGLEIN, F. 1992. Global Tagging Optimization by Type Inference. In *Proc. Conf. on Lisp and Functional Programming*, New York, pp. 205–215. ACM Press.

HENGLEIN, F. AND REHOF, J. 1995. Safe polymorphic type inference for a Dynamically Typed Language: Translating Scheme to ML. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, New York, pp. 192–203. ACM Press.

HINDLEY, R. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. American Mathematical Society 146*, 29–60.

JAGANNATHAN, S. AND WEEKS, S. 1995. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *Proc. 22nd Annual Symposium on Principles of Programming Languages*, New York, pp. 392–401. ACM Press.

JAGANNATHAN, S., WEEKS, S., AND WRIGHT, A. 1997. Type-Directed Flow Analysis for Typed Intermediate Languages. Tech. Rep. 97-3 (May), NEC Research Institute.

JAGANNATHAN, S. AND WRIGHT, A. 1995. Effective Flow-Analysis for Avoiding Runtime Checks. In *Proc. 2nd Intl. Static Analysis Symposium, LNCS 983*, Berlin, pp. 207–225. Springer-Verlag.

JAGANNATHAN, S. AND WRIGHT, A. 1996. Flow-directed Inlining. In *Proc. Conf. on Programming Language Design and Implementation*, New York, pp. 193–205. ACM Press.

JONES, N. AND MUCHNICK, S. 1979. Flow Analysis and Optimization of Lisp-like Structures. In *Proc. 6th Annual Symposium on Principles of Programming Languages*, New York, pp. 244–256. ACM Press.

KANELLAKIS, P., MAIRSON, H., AND MITCHELL, J. 1991. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pp. 444–478. Cambridge, Massachusetts: MIT Press.

KAPLAN, M. A. AND ULLMAN, J. D. 1980. A Scheme for the Automatic Inference of Variable Types. *Journal of the Association for Computing Machinery 27,* 1 (Jan.), 128–145.

MA, K. L. AND KESSLER, R. R. 1990. TICL—A type inference system for Common Lisp. *Softw. Pract. Exper. 20,* 6 (June), 593–623.

MACLACHLAN, R. A. 1992. The Python Compiler for CMU Common Lisp. In *Proc. Conf. on Lisp and Functional Programming*, New York, pp. 235–246. ACM Press.

MILNER, R. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences 17*, 348–375.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts.

MITCHELL, J. 1990. *Handbook of Theoretical Computer Science: Volume B*, Chapter Type Systems for Programming Languages, pp. 367–453. MIT Press, Cambridge, Massachusetts.

NIELSON, F. AND NIELSON, H. R. 1997. Infinitary Control-Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. 24rd Annual Symposium on Principles of Programming Languages*, pp. 332–345.

PALSBERG, J. 1995. Closure Analysis in Constraint Form. *ACM Transactions on Programming Languages and Systems 17,* 1, 47–62.

PALSBERG, J. AND O'KEEFE, P. 1995. A Type System Equivalent to Flow Analysis. In *22th Annual Symposium on Principles of Programming Languages*, New York, pp. 367–378. ACM Press.

PANDE, H. D. AND RYDER, B. G. 1994. Static Type Determination for C++. In *Usenix C++ Conference Proceedings*, Cambridge, Massachusetts, pp. 85–97. MIT Press.

PLEVYAK, J. AND CHIEN, A. 1994. Precise Concrete Type Inference for Object-Oriented Languages. In 9^{th} *Annual Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, New York, pp. 324–340. ACM Press.

SERRANO, M. AND FEELEY, M. 1996. Storage Use Analysis and its Applications. In *Proc. Intl. Conf. on Functional Programming*, New York. ACM Press.

SHIVERS, O. 1990. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation*. Cambridge, Massachusetts: MIT Press.

SHIVERS, O. 1991. Control-Flow Analysis of Higher-Order Languages or Taming Lambda. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University.

STEENKISTE, P. AND HENNESSY, J. 1987. Tags and Type Checking in Lisp. In *Proc.* 2^{nd} *Architectural Support for Programming Languages and Systems Symposium*, New York, pp. 50–59. ACM Press.

TOFTE, M. 1990. Type Inference for Polymorphic References. *Information and Computation 89,* 1, 1–34.

WRIGHT, A. 1995. Simple Imperative Polymorphism. *LISP and Symbolic Computation 8*, 343–355.

WRIGHT, A. AND CARTWRIGHT, R. 1997. A Practical Soft Type System for Scheme. *ACM Transactions on Programming Languages and Systems 19,* 1, 87–152.

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

WRIGHT, A. K. AND FELLEISEN, M. 1994. A Syntactic Approach to Type Soundness. *Information and Computation 115,* 1, 38–94.

ZHAO, F. 1987. An $O(N)$ Algorithm for Three-Dimensional N-Body Simulations. M.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Received September 1996