SRC Technical Note
1999 - 001

January 28, 1999

Flexible Polyvariance in Constraint-Based Analyses

Cormac Flanagan

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Abstract

Existing constraint-based analyses only support a limited form of polyvariance, based on syntactic clues such as let-polymorphism. These analyses become imprecise when the call sites for a particular function are not syntactically obvious, such as in object-oriented method invocations, or when a function is referenced across a module boundary.

This paper describes how to extend the constraint-based approach to encompass a much wider range of polyvariance strategies. We achieve this by deferring decisions about when to instantiate constraint schemas to the solution phase, where these decisions can exploit information inferred about the program's run-time behavior.

Our approach yields a polyvariant constraint-based analysis framework parameterized over a strategy for controlling schema instantiation. Instantiations of the framework cover a hierarchy of analyses with different cost/precision tradeoffs. The framework combines the efficiency benefits of staged, constraint-based analyses with the flexible polyvariance strategies of monolithic analysis frameworks.

1 Introduction

MrSpidey is a static debugger for Scheme that checks the soundness of all primitive operations based on a static program analysis [FFK+96]. The analysis is a constraint-based system similar to Heintze's set-based analysis of ML programs [Hei94]. It consists of two phases: a *specification* phase, which derives constraints describing the data flow relationships of the analyzed program, and a *solution* phase, which solves these constraints.

To accurately large programs, which tend to re-use functions in a polymorphic manner, the analysis is *polyvariant*. That is, it duplicates the constraint systems for function definitions at appropriate points in the program, in order to avoid merging information between unrelated calls to these functions. These polyvariance decisions about to duplicate these constraint systems crucially affect both the cost and the precision of the analysis.

Existing polyvariant constraint-based analyses make these decisions during the specification phase [AWL94, FF97, Fla97]. Hence these crucial polyvariance decisions cannot exploit information about the run-time behavior of the program computed in the solution phase, and instead, must be made based on simple syntactic criteria, such as **let**-polymorphism. These syntactic criteria are often sub-optimal. In particular, if the call-sites of a function are not syntactically obvious, then the specification phase cannot duplicate the function's constraint system at those call sites, and hence information is merged between these call sites. This situation often

occurs in object-oriented method invocations, or in large programs when a function is referenced across a module boundary.

In an attempt to overcome this problem, MrSpidey's specification phase includes an additional, first-order analysis designed to better identify the call sites of particular functions. This approach suffered from two disadvantages: it was still unable to identify all call sites where constraint duplication would be appropriate, and it significantly complicated the analysis implementation, since the first-order analysis duplicated much of the functionality of the overall analysis.

In direct contrast to this problem with staged, constraint-based analyses, *monolithic* analyses based on abstract interpretation [CC77] have much more flexibility in their choice of polyvariance strategies, because they can exploit information inferred about the program's run-time behavior to drive polyvariance decisions. This flexibility leads to a rich hierarchy of analyses with varying cost/precision tradeoffs [Oli91, JW95b, JW95a, SZ94]. Many of these analyses cannot be expressed in the constraint-based framework.

This paper explores how to incorporate the flexible polyvariance strategies of monolithic analyses within the constraint-based framework. The crucial insight is that the polyvariance decisions about when to duplicate constraint systems must be deferred to the solution phase, when sufficient information is available to make these decisions intelligently. The result of this approach is a polyvariant constraint-based analysis framework. The framework is parameterized over a polyvariance strategy, which controls when constraint systems are duplicated in the solution phase. Possible instantiations of the framework cover a range of analyses, from a simple monomorphic analysis at one extreme, to an extremely precise but uncomputable analysis at the other extreme, with a variety of intermediate analyses covering different cost/precision tradeoffs. This framework combines the efficiency benefits of the constraint-based analyses with the flexible polyvariance strategies of monolithic analysis frameworks.

The presentation of our results proceeds as follows. Section 2 introduces a simple source language. Section 3 describes a monomorphic set-based analysis for that language, and section 4 extends that analysis with **let**-polymorphism. Section 5 describes our polyvariant set-based analysis framework, and section 6 describes some instantiations of that framework. Section 7 concludes with directions for future research.

2 The Source Language

For simplicity, we present our analysis framework for a λ-calculus-like language with constants and labeled expressions. It is straightforward to extend the analysis

Syntax:

$$\begin{array}{rcll} M & \in & \Lambda & = & x \mid V \mid (M\ M) \mid M^l & \text{(Expressions)} \\ V & \in & \mathit{Value} & = & b \mid (\lambda^t x.M) & \text{(Values)} \\ x & \in & \mathit{Vars} & = & \{x, y, z, \ldots\} & \text{(Variables)} \\ b & \in & \mathit{BasicConst} & & & \text{(Basic constants)} \\ t & \in & \mathit{Tag} & & & \text{(Function tags)} \\ l & \in & \mathit{Label} & & & \text{(Expression labels)} \end{array}$$

Evaluator:

$$\begin{array}{rcll} \mathit{eval} : \Lambda^0 & \multimap & \mathit{Value} & \\ \mathit{eval}(M) & = & V & \text{if } M \longmapsto^* V \end{array}$$

Reduction Rules:

$$\begin{array}{rcll} \mathcal{E}[\ ((\lambda^t x.M)\ V)\ ] & \longmapsto & \mathcal{E}[\ M[x \mapsto V]\ ] & (\beta_v) \\ \mathcal{E}[\ V^l\ ] & \longmapsto & \mathcal{E}[\ V\ ] & (\mathit{unlabel}) \end{array}$$

Evaluation Contexts:

$$\mathcal{E} = [\ ] \mid (\mathcal{E}\ M) \mid (V\ \mathcal{E}) \mid \mathcal{E}^l$$

Figure 1: The source language Λ: syntax and semantics

to a realistic language including assignments, recursive data structures, objects and modules along the lines described in an earlier report [Fla97].

The syntax and semantics of the language are described in figure 1. Expressions in the language are either variables, values, function applications, **let**-expressions, or labeled expressions. Values include basic constants and functions. Functions have identifying tags so that we can reconstruct a call-graph from the results of the analysis. We use labels to identify those program expressions whose values we wish to predict. We work with the usual conventions and terminology of the λ_v-calculus when discussing syntactic issues. In particular, the substitution operation $M[x \leftarrow V]$ replaces all free occurrences of x within M by V, and Λ^0 denotes the set of closed terms, also called *programs*.

We specify the meaning of programs via a reduction semantics based on the reduction rules β_v and (*unlabel*). The β_v rule is conventional, and (*unlabel*) rule removes the label from an expression once its value is needed.

Notation: $\mathcal{P}$ to denotes the power-set constructor; $\mathcal{P}_{\text{fin}}$ to denote the finite power-set constructor; and $\multimap$ denotes the partial function space constructor.

3 Monomorphic Set-Based Analysis

Set-based analysis consists of two phases: a specification phase and a solution phase.[1] The specification phase derives constraints on the sets of values that program expressions may assume. These constraints describe the data flow relationships of the analyzed program. The solution phase solves these constraints to yield a conservative approximation of the set of possible values for each labeled expression in the program.

3.1 The Constraint Language

A *set expression* τ is either a set variable, a constant, or a function set expression:

$$\begin{array}{rcll} \tau & \in & \mathit{SetExp} & = \quad \alpha \mid c \mid (\beta_1 \rightarrow^t \beta_2) \\ \alpha, \beta, \gamma & \in & \mathit{SetVar} & \supset \quad \mathit{Label} \\ c & \in & \mathit{Const} & = \quad \mathit{BasicConst} \cup \mathit{Tag} \end{array}$$

The meta-variables $\alpha, \beta, \gamma, \ldots$ range over set variables, and we include program labels in the collection of set variables. A function set expression $(\beta_1 \rightarrow^t \beta_2)$ includes the identifying tag t of the corresponding function definition, as well as set variables describing the argument set (β_1) and result set (β_2) of the function. Constants include both basic constants and function tags. A set of constants $C \in \mathcal{P}(\mathit{Const})$ represents a set of run-time values (relative to a given program) according to the relation $V \ \mathit{in}\ C$:

$$\begin{array}{rcl} b \ \mathit{in}\ C & \text{iff} & b \in C \\ (\lambda^t x.M) \ \mathit{in}\ C & \text{iff} & t \in C \end{array}$$

The language of constraints is:

$$\begin{array}{rcll} \mathcal{C} & \in & \mathit{Constraint} & = \quad [\tau \leq \alpha] \\ & & & \mid \quad [\alpha \leq (\gamma_1 \rightarrow \gamma_2)] \end{array}$$

The constraint $[\tau \leq \alpha]$ states the the set of values described by α includes those described by τ. The constraint $[\alpha \leq (\gamma_1 \rightarrow \gamma_2)]$ is an *application constraint*; it extracts into γ_1 and γ_2 the argument and result sets of functions described by β. A *constraint system* $\mathcal{S}$ is a finite collection of constraints:

$$\mathcal{S} \ \in \ \mathit{ConstraintSystem} \ = \ \mathcal{P}_{\text{fin}}(\mathit{Constraint})$$

[1] Cousot and Cousot [CC95] showed that the results of set-based analysis can alternatively be computed via an abstract interpretation based on chaotic iteration.

$$\Gamma \cup \{x : \alpha\} \vdash x : \alpha, \emptyset \qquad (var)$$

$$\Gamma \vdash b : \alpha, \{b \leq \alpha\} \qquad (const)$$

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash M^l : \alpha, \mathcal{S} \cup \{\alpha \leq l\}} \qquad (label)$$

$$\frac{\Gamma \cup \{x : \beta_1\} \vdash M : \beta_2, \mathcal{S}}{\Gamma \vdash (\lambda^t x.M) : \alpha, \mathcal{S} \cup \{(\beta_1 \rightarrow^t \beta_2) \leq \alpha\}} \qquad (abs)$$

$$\frac{\Gamma \vdash M_i : \beta_i, \mathcal{S}_i}{\Gamma \vdash (M_1\ M_2) : \alpha, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\beta_1 \leq (\beta_2 \rightarrow \alpha)\}} \qquad (app)$$

Figure 2: Constraint derivation rules.

3.2 Deriving Constraint Systems

The specification phase of set-based analysis derives constraints on the sets of values that program expressions may assume. Following Aiken *et al*. [AWL94] and Palsberg and O'Keefe [PO95], we formulate this derivation as a proof system.

The derivation proceeds in a syntax-directed manner according to the constraint derivation rules presented in figure 2. Each rule infers a judgement of the form $\Gamma \vdash M : \alpha, \mathcal{S}$, where the *derivation context* Γ maps the free variables of M to set variables; α names the result set of M; and the constraint system $\mathcal{S}$ describes the data flow relationships of M.

The constraint derivation rule (*var*) for a variable reference extracts the corresponding set variable from the derivation context. The (*const*) rule ensures that the value set for a constant expression includes that constant. The (*label*) rule records the possible values of a labeled expression M^l in l. The (*abs*) rule for a function definition records both the function's tag and the set variables denoting its argument and result sets. The (*app*) rule for a function application propagates values from the argument expression into the argument set of the function and from the range of the function into the result of the application expression.

3.3 Solving Constraint Systems

We solve a constraint system by closing it under the inference rules Θ described in figure 3. Intuitively, these rules infer all the data-flow paths in the program, which are described by constraints of the form $\beta \leq \gamma$ (for $\beta, \gamma \in \mathit{SetVar}$), and propagate values along those data-flow paths. Specifically, the rule (*prop*) propagates information about constants and functions forward along data-flow paths, and the rule (*app*) infers data-flow paths from argument expressions to the corresponding formal parameters and from function results to the corresponding call sites. We write $\mathcal{S} \vdash_\Theta \mathcal{C}$ if $\mathcal{S}$ proves $\mathcal{C}$ via the rules Θ, and use $\mathit{close}_\Theta(\mathcal{S})$ to denote the closure of $\mathcal{S}$ under Θ, *i.e.*, the set $\{\mathcal{C} \mid \mathcal{S} \vdash_\Theta \mathcal{C}\}$.

$$\frac{\tau \leq \alpha \qquad \alpha \leq \beta}{\tau \leq \beta} \qquad (\mathit{prop})$$

$$\frac{(\beta_1 \rightarrow^t \beta_2) \leq \alpha \qquad \alpha \leq (\gamma_1 \rightarrow \gamma_2)}{\gamma_1 \leq \beta_1 \qquad \beta_2 \leq \gamma_2} \qquad (\mathit{app})$$

Figure 3: The rules $\Theta = \{(\mathit{prop}), (\mathit{app})\}$

MrSpidey uses a worklist algorithm to compute the closure of $\mathcal{S}$ under Θ efficiently. The worklist keeps track of all constraints in $\mathcal{S}$ whose consequences under Θ may not be in $\mathcal{S}$. The algorithm repeatedly removes a constraint from the worklist, and for each consequences under Θ that is not already in $\mathcal{S}$, it adds that consequence both to $\mathcal{S}$ and to the worklist. The process iterates until the worklist is empty, at which point $\mathcal{S}$ is closed under Θ. The complete algorithm can be found in a related report [Fla97].

This closure process propagates all information concerning the constant and function values for a labeled expression M^l into constraints of the form $[c \leq l]$ or $[(\beta_1 \rightarrow^t \beta_2) \leq l]$. We define the set based analysis of a program as a function that computes an *analysis result* from this information, where an analysis result is a mapping from program labels to sets of possible basic constants and function tags:

$$R \in \mathit{AnalysisResult} = \mathit{Label} \longrightarrow \mathcal{P}(\mathit{Const}) \; .$$

Definition 3.1 *$sba : \Lambda^0 \longrightarrow \mathit{AnalysisResult}$*
If $P \in \Lambda^0$, $\emptyset \vdash P : \alpha$, $\mathcal{S}$ and $\mathcal{S}' = \mathit{close}_\Theta(\mathcal{S})$ then $sba(P) = R$ where

$$\begin{aligned} R(l) = \quad & \{b \mid [b \leq l] \in \mathcal{S}'\} \\ \cup \quad & \{t \mid [(\beta_1 \rightarrow^t \beta_2) \leq l] \in \mathcal{S}'\} \; . \end{aligned}$$

The solution $sba(P)$ conservatively approximates the value sets for each labeled expression in the program, as described by the following theorem.

Theorem 3.1 (Correctness of *sba*)
If $sba(P) = R$ *and* $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ *then* V *in* $R(l)$.

This result follows from a subject reduction proof along the lines of Wright and Felleisen [WF94] and Palsberg [Pal95] and is contained in a related report [Fla97].

4 Polymorphic Set-Based Analysis

The analysis described in the previous section is a *monomorphic* analysis. That is, it merges information between distinct calls to the same function. This is a useful approximation for small to medium sized programs, but becomes increasingly imprecise on larger programs, because such programs tend to re-use functions in a polymorphic fashion.

To illustrate this idea, consider the program:[2]

$$P \equiv (\textbf{let}\ (f\ (\lambda^t x.x))\ (\textbf{begin}\ (f\ \texttt{true})^{l_1}\ (f\ 0)^{l_2}))$$

This program binds f to the identity function, and then applies f to the constants `true` and 0. These two argument values get merged in the body of f, and hence the analysis results for the two application expressions are:

$$\begin{aligned} sba(P)(l_1) &= \{\texttt{true}, 0\} \\ sba(P)(l_2) &= \{\texttt{true}, 0\} \end{aligned}$$

These results are overly approximate, since the evaluation of expression l_1 can never yield 0, and the evaluation of expression l_2 can never yield `true`.

To accurately analyze programs that use such polymorphic functions, we need to package the constraints for those functions into *constraint schemas*, and duplicating or *instantiating* these constraint schemas at different points in the program. These decisions about when to instantiate constraint schemas are typically made during the specification phase according to simple syntactic criteria, such as **let**-polymorphism [AWL94, FF97].

To modify the monomorphic analysis in this fashion, we extend the source language with **let**-expressions, which introduce polymorphic bindings, and we duplicate the constraints for **let**-bound functions at each polymorphic reference, in a manner analogous to the **let** polymorphism of ML [MTH90, Tof90].

[2]For clarity, this example is expressed in terms of **let** and **begin**, but these constructs can be expanded into the language Λ in the usual manner.

The syntax of the **let** construct is:

$$M \quad \in \quad \Lambda \quad = \quad \ldots \mid (\textbf{let}\ (x\ V)\ M)$$

and its semantics is given by the reduction rule:

$$\mathcal{E}[\ (\textbf{let}\ (x\ V)\ M)\ ] \quad \longmapsto \quad \mathcal{E}[\ M[x \mapsto V]\ ] \qquad\qquad (\beta_{let})$$

$$\frac{\begin{array}{c}\Gamma \vdash V : \beta, \mathcal{S} \\ \overline{\alpha} = FV(\mathcal{S}) \setminus (FV[range(\Gamma)] \cup Label) \\ \Gamma \cup \{x : \forall\overline{\alpha}.\,(\beta, \mathcal{S})\} \vdash M : \gamma, \mathcal{S}_M\end{array}}{\Gamma \vdash (\textbf{let}\ (x\ V)\ M) : \gamma, \mathcal{S}_M} \qquad (let)$$

$$\frac{\psi \text{ is a substitution of set variables for } \overline{\alpha}}{\Gamma \cup \{x : \forall\overline{\alpha}.\,(\beta, \mathcal{S})\} \vdash x : \psi(\beta), \psi(\mathcal{S})} \qquad (inst)$$

Figure 4: Constraint derivation rules for polymorphism.

The new constraint derivation rules are given in figure 4. The rule (*let*) produces a *constraint schema* $\sigma = \forall\overline{\alpha}.\,(\beta, \mathcal{S})$ for the polymorphic, **let**-bound values [AWL94, Tof90]. The set variable β names the result of the value; the system of constraints $\mathcal{S}$ describes the data-flow relationships of the value, using β; and the set $\overline{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$ contains those internal set variables of the constraint system that must be duplicated at each reference to the **let**-bound variable via the rule (*inst*).

A derivation context Γ now maps program variables to either set variables or constraint schemas:

$$\begin{array}{rcrcl} \Gamma & \in & DerivCtxt & = & Var \multimap SetVar \cup Schema \\ \sigma & \in & Schema & = & \forall\overline{\alpha}.\,(\beta, \mathcal{S}) \end{array}$$

We use $FV(\mathcal{S})$ to denote the collection of set variables in a constraint system $\mathcal{S}$, and we use $FV[range(\Gamma)]$ to denote the free set variables in the range of Γ. The free set variables of a constraint schema $\forall\overline{\alpha}.\,(\beta, \mathcal{S})$ are those in $\mathcal{S}$ but not in $\overline{\alpha}$, and the free variables of a set variable is simply the set variable itself.

The rest of the development of the previous section can be adapted *mutatis mutandis*. In particular, since this extension does not change the constraint language, we can re-use the constraint solution algorithm of the previous section.

Since excessive constraint duplication can be costly, a better approach is to first simplify these constraint schemas (as described in a related report [FF97]), thus reducing the number of constraints that need to duplicated at each polymorphic reference. On a standard set of benchmarks, this technique reduces analysis times by a factor of 2 to 4, and makes the polyvariant analysis time comparable to, and sometimes faster than, the monomorphic analysis [Fla97, p. 86].

5 Polyvariant Set-Based Analysis

Instantiating constraint schemas based on syntactic criteria such as **let**-polymorphism is a straightforward strategy for introducing polyvariance. This strategy works well for programs, such as the example program of the previous section, where the call sites of polymorphic functions are syntactically obvious. However, if the call sites of a function are not obvious, then this approach can lead to insufficient polyvariance and hence imprecise analysis results. This situation often occurs in object-oriented method invocations, or when a function is referenced across a module boundary.

To illustrate this problem, consider what happens if we split the program P into two modules, where we express the modular structure of the program using function abstraction and application. The first module defines the identity function:

$$M_1 \equiv (\lambda^t x.x)$$

The second module imports that function, and applies it twice:

$$M_2 \equiv (\lambda^{t_2} f.(\textbf{begin } (f\ \texttt{true})^{l_1}\ (f\ 0)^{l_2}))$$

The complete program links these two modules together via a function application:

$$P \equiv (M_2\ M_1)$$

In the polymorphic analysis of this modular program, the two call sites of the identity function are not syntactically obvious. Hence the constraint system for the identity function cannot be duplicated at each call site, and so the **let**-polymorphic analysis of this program yields the same imprecise results as the monomorphic analysis, *i.e.*:

$$\begin{aligned} sba(P)(l_1) &= \{\texttt{true}, 0\} \\ sba(P)(l_2) &= \{\texttt{true}, 0\} \end{aligned}$$

In an attempt to tackle this problem, MrSpidey's specification phase included an additional, first-order analysis designed to identify the call sites of certain polymorphic functions. This approach was partially successful, but suffers from two

disadvantages. First, the first-order analysis was unable to identify all call sites of those functions, and hence the resulting analysis was still quite imprecise. Second, the approach significantly complicated the analysis implementation, since the first-order analysis had to duplicate much of the functionality of the overall analysis.

In contrast to this limitation of constraint-based analyses, *monolithic* analyses based on abstract interpretation [CC77] have much more flexibility in their choice of polyvariance strategies, because they can exploit information inferred about the program's run-time behavior to drive polyvariance decisions. This flexibility leads to a rich hierarchy of analyses with varying cost/precision tradeoffs [Oli91, JW95b, JW95a]. Many of these analyses cannot be expressed in the constraint-based framework. Indeed, it is unclear if the traditional constraint-based approach can express any of these analyses other than 0CFA. Given this context, the following comment [JW95b, p. 16] appears justified:

> ... while efficient implementations of (constraint-based) analyses can be built, it is unclear whether they provide the necessary precision to perform useful run-time check optimization. Refinements of these approaches that take into account polymorphism are possible [Hei94], but are *ad hoc* and do not fit neatly within the constraint framework.

5.1 Flexible Polyvariance in Constraint-Based Analyses

The goal of this paper is to describe an approach whereby we can cleanly incorporate the flexible polyvariance strategies of monolithic analyses within the constraint-based framework. The crucial insight is that we need to defer polyvariance decisions about when to instantiate constraint schemas to the solution phase, where we can make these decisions based on additional information about the program's run-time behavior. The remainder of this section describes how to extend the set-based analysis of section 3 in this fashion.

5.2 The Constraint Language

In order to defer schema instantiation decisions to the solution phase, we need to express schemas within the constraint language. Hence we extend the constraint language as follows:

$$
\begin{array}{lcll}
\tau & \in & \mathit{SetExp} & = \quad \alpha \mid c \mid \sigma \\
\sigma & \in & \mathit{Schema} & = \quad \forall\overline{\alpha}.\,(\beta_1 \rightarrow^t \beta_2, \mathcal{S}) \\
\mathcal{C} & \in & \mathit{Constraint} & = \quad [\tau \leq \alpha] \\
 & & & \mid \quad [\alpha \leq (\gamma_1 \rightarrow \gamma_2)] \\
\mathcal{S} & \in & \mathit{ConstraintSystem} & = \quad \mathcal{P}_{\text{fin}}(\mathit{Constraint})
\end{array}
$$

We merge function set expressions into constraint schemas in order to avoid the complications of having two representations for function values in the constraint language. We use $FV(\mathcal{S})$ to denote the free set variables in a constraint system $\mathcal{S}$, which *excludes* those variables that are bound in a schema by the enclosing universal quantifier.

5.3 Deriving Constraints

Since constraint schemas can now appear in constraints, and not just in derivation contexts, we can introduce constraint schemas for all function definition, and not just for **let**-bound functions. The new constraint derivation rule for function definitions is:

$$\frac{\begin{array}{c}\Gamma \cup \{x : \beta_1\} \vdash M : \beta_2, \mathcal{S} \\ \overline{\alpha} = FV(\mathcal{S}) \setminus (FV[range(\Gamma)] \cup Label) \\ \sigma = \forall\overline{\alpha}.\, (\beta_1 \rightarrow^t \beta_2, \mathcal{S})\end{array}}{\Gamma \vdash (\lambda^t x.M) : \alpha, \{\sigma \leq \alpha\}} \qquad (abs)$$

This rule creates a constraint schema for each function definition. This schema combines the function set expression for the function with the constraint system representing the internal data-flow paths of the function. This constraint schema is instantiated as appropriate by the constraint solution algorithm, which is described below.

5.4 Computing a Solution

Intuitively, solving a constraint system with schemas involves inferring all of the data-flow paths in the program, and propagating values along those data-flow paths. Propagating a function's constraint schema to a corresponding call site for that function yields the following pair of constraints:

$$\begin{array}{c}\forall\overline{\alpha}.\, (\beta_1 \rightarrow^t \beta_2, \mathcal{S}) \leq \delta \\ \delta \leq (\gamma_1 \rightarrow \gamma_2)\end{array}$$

To propagate values between the call site and the function definition, we need to instantiate the constraint schema by choosing an appropriate substitution ψ for the bound variables of the constraint schema, and then record the appropriate data-flow paths for the argument and result values. This is achieved by the following inference rule (*inst-app*):

$$\frac{\begin{array}{c}\forall\overline{\alpha}.\,(\beta_1 \rightarrow^t \beta_2, \mathcal{S}) \leq \delta \\ \delta \leq (\gamma_1 \rightarrow \gamma_2)\end{array}}{\begin{array}{rcl}\gamma_1 & \leq & \psi(\beta_1) \\ \psi(\beta_2) & \leq & \gamma_2 \\ \multicolumn{3}{c}{\psi(\mathcal{S})}\end{array}} \quad \text{where } \psi \text{ is a subst. on } \overline{\alpha} \qquad (\textit{inst-app})$$

This inference rule is parameterized by a substitution ψ on the bound variables of the schema. Different strategies for choosing ψ yield different degrees of polyvariance in the analysis. One extreme is to always choose ψ to be the identity substitution, which yields the monomorphic analysis of section 3. The other extreme is to always choose substitutions that introduce fresh variables. This approach yields an extremely precise but uncomputable analysis that instantiates a constraint schema according each function call in the program's execution. In between these two extremes lie a range of analyses, with different cost/precision tradeoffs, generated by other strategies for choosing the substitution ψ. We outline some of these analyses in the following section. The polyvariance strategy used for choosing the substitution ψ does not affect its correctness of the analysis.

In addition to the (*inst-app*) rule, we also need the (*prop*) for propagating values along data-flow paths. Thus the complete set of inference rules for the constraint language is:

$$\Delta = \{(\textit{prop}), (\textit{inst-app})\}\ .$$

We write $\mathcal{S} \vdash_\Delta \mathcal{C}$ if $\mathcal{S}$ proves $\mathcal{C}$ via the rules Δ, and we write $\mathcal{S} \vdash_\Delta \mathcal{S}'$ if $\mathcal{S} \vdash_\Delta \mathcal{C}$ for each $\mathcal{C} \in \mathcal{S}'$.

Since the substitution ψ is unspecified in the rule (*inst-app*), do not have a unique closed form for each constraint system. Instead, the relation $close_\Delta$ relates each constraint system to a number of different closed forms. Each closed forms correspond to a different choice of substitutions in applications of the rule (*inst-app*).

Definition 5.1 (*$close_\Delta \subseteq ConstraintSystem \times ConstraintSystem$*)
The relation $close_\Delta(\mathcal{S}, \mathcal{S}')$ holds if and only if:

1. $\mathcal{S} \vdash_\Delta \mathcal{S}'$,
2. *$\mathcal{S}'$ is closed under the rule* (*prop*)*, and*
3. *for all* (*inst-app*) *antecedent in $\mathcal{S}'$ there exists at least one corresponding (inst-app) consequent in $\mathcal{S}'$.*

The closure process propagates all information concerning the constant and function values for a labeled expression M^l into constraints of the form $[c \leq l]$ or $[\forall\overline{\alpha}.(\beta_1 \rightarrow^t \beta_2, \mathcal{S}) \in l]$. Hence we can compute an analysis result from each closed form. We define set-based analysis as a relation that associates each program with this set of analysis results.

Definition 5.2 (*sba* $\subseteq \Lambda^0 \times$ *AnalysisResult*)
If $P \in \Lambda^0$ *and* $\emptyset \vdash P : \alpha, \mathcal{S}$ *then* $sba(P, R)$ *holds if and only if there exists* $\mathcal{S}'$ *such that* $close_\Delta(\mathcal{S}, \mathcal{S}')$ *and*

$$\begin{aligned} R(l) = \quad & \{b \mid [b \leq l] \in \mathcal{S}'\} \\ \cup \quad & \{t \mid [\forall\overline{\alpha}.(\beta_1 \rightarrow^t \beta_2, \mathcal{S}'') \leq l] \in \mathcal{S}'\}\ . \end{aligned}$$

5.5 Correctness of Polyvariant Set-Based Analysis

Any analysis result produced by *sba* conservatively approximates the value sets for each labeled expression in the program, as described by the following theorem.

Theorem 5.1 (Correctness of *sba*)
If $sba(P, R)$ *and* $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ *then* V *in* $R(l)$.

This result follows from a subject reduction proof along the lines of Wright and Felleisen [WF94] and Palsberg [Pal95]. The final paper will contain an outline of the proof.

6 Expressiveness of Polyvariant SBA

The polyvariant set-based analysis framework encompasses a hierarchy of analyses with varying cost/precision tradeoffs. This section presents a number of instantiations of the framework.

6.1 Expressing Monovariant SBA

Suppose that the substitution ψ used in each application of the (*polyapp*) rule is identity substitution. Then, since $\psi(\mathcal{S}) = \mathcal{S}$, the analysis never duplicates constraint systems, and hence the analysis is monomorphic.

This monomorphic analysis is actually slightly more accurate than the monomorphic analysis of section 3 on programs that contain dead code. That is, if the analyzed program defines a function that has no call sites, then the constraint schema for that function is never instantiated, and hence those constraints do not affect the analysis results.

To illustrate this idea, consider the program:

$$Q \equiv (\textbf{let } (f\ (\lambda^f x.x^l))\ (\lambda^g d.(f\ 0)))$$

Since the function tagged g is never called, the constraint for the application of f to 0 does not affect the analysis results, and hence the value set of the expression x^l is correctly the empty set. In contrast, the monomorphic analysis of section 3 would yield $\{0\}$ as the value set for x^l.

6.2 Expressing contour-based flow analyses

There are a number of contour-based flow analysis frameworks [Oli91, SZ94, JW95a], of which Jagannathan and Weeks' framework appears to be the most expressive. Although space restrictions do not allow us to provide a review of their framework, this subsection outlines instantiations of their framework can be expressed as polyvariant set-based analyses.

We associate with each application constraint $\mathcal{C} = [\gamma \leq (\alpha \rightarrow \beta)]$ a label $\mathit{Label}(\mathcal{C})$ and an abstract contour $\mathit{Contour}(\mathcal{C})$. We use the contour information to choose the substitution ψ used in the each application of the (*inst-app*) rule. For each contour-schema pair, we have a corresponding substitution that introduces fresh variables. An application of the (*polyapp*) rule based on the antecedents $[\sigma \leq \alpha]$ and $\mathcal{C} = [\alpha \leq (\gamma_1 \rightarrow \gamma_2)]$ uses the substitution corresponding to σ and $\mathit{Contour}(\mathcal{C})$.

We infer contour information for application constraints as follows. Each application constraint is generated by the (*app*) or (*inst-app*) rule. An application constraint $\mathcal{C}$ generated via the (*app*) rule corresponds to from an application expression in the analyzed program. In this case we set both $\mathit{Label}(\mathcal{C})$ and $\mathit{Contour}(\mathcal{C})$ to that application expression.

For the (*inst-app*) case, suppose we have the (*inst-app*) antecedents $[\overline{\forall\alpha}.\ (\beta_1 \rightarrow^t \beta_2, \mathcal{S}) \leq \delta]$ and $\mathcal{C}' = [\delta \leq (\gamma_1 \rightarrow \gamma_2)]$, and that $\mathcal{S}$ contains an application constraint $\mathcal{C}$ that is copied to $\psi(\mathcal{C})$. In this case we set:

$$\begin{array}{rcl} \mathit{Label}(\psi(\mathcal{C})) & = & \mathit{Label}(\mathcal{C}) \\ \mathit{Contour}(\psi(\mathcal{C})) & = & \mathit{Label}(\mathcal{C}').\mathit{Contour}(\mathcal{C}') \end{array}$$

Proposition 6.1 *The above parameterization of the polyvariant set-based analysis framework yields the same results as Jagannathan and Weeks' framework, for any choice of abstract contours.*

7 Conclusions and Directions

Our polyvariant set-based analysis framework is parameterized over an algorithm that controls the duplication of constraint systems. Different choices for this algorithm yield a hierarchy of analyses with varying cost/precision tradeoffs. The framework can express monomorphic and **let**-polymorphic set-based analysis, Shivers' nCFA [Oli91], and the analyses generated by Jagannathan and Week's framework [JW95a].

We intend to experimentally explore other instantiations of the framework in a second-generation version of MrSpidey, which we are implementing for Java. This system will combine the flexible polyvariance framework of this paper with the constraint simplification algorithms of [FF97, Fla97]. We expect this system to successfully handle program of up to 100K lines of code, and we intend to use the system to investigate how various polyvariance strategies affect the analysis cost and precision on programs of that size. The long term goal of this research is to develop an efficient and accurate static debugging tool for large, real-world Java applications.

References

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the Symposium on the Principles of Programming Languages*, pages 163–173, 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analyses of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[CC95] P. Cousot and R. Cousot. Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In *Functional Programming and Computer Architecture*, pages 170–181, 1995.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, June 1997.

[FFK+96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Finding bugs in the web of program

invariants. In *Programming Language Design and Implementation*, pages 23–32, 1996.

[Fla97] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[JW95a] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *22nd ACM Symposium on Principles of Programming Languages*, pages 393–407, 1995.

[JW95b] Suresh Jagannathan and Andrew K. Wright. Effective flow analysis for avoiding run-time checks. In **2nd** *International Static Analysis Symposium, Lecture Notes in Computer Science, vol. 983*, pages 207–224. Springer-Verlag, September 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Oli91] Olin Shivers. *Control-flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, Carnegie-Mellon University, 1991.

[Pal95] Jens Palsberg. Closure analysis in constraint form. *Transactions on Programming Languages and Systems*, 17(1):47–62, 1995.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the Symposium on the Principles of Programming Languages*, pages 367–378, 1995.

[SZ94] Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher order functional programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 318–327, 1994.

[Tof90] M. Tofte. Type inference for polymorphic references. *Info. Comput.*, 89(1):1–34, November 1990.

[WF94] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Info. Comput.*, 115(1):38–94, 1994.