# Selectors Make Analyzing `case-lambda` Too Hard*


Philippe Meunier[†‡] Robby Findler[‡] Paul A. Steckler[†] Mitchell Wand[†]

[†]College of Computer Science
Northeastern University
Boston, MA 02115
{meunier,steck,wand}@ccs.neu.edu

[‡]Department of Computer Science, MS 132
Rice University
Houston, TX 77005-1892
robby@cs.rice.edu



## Abstract

Flanagan's set-based analysis (SBA) uses *selectors* to choose data flowing through expressions. For example, the `rng` selector chooses the ranges of procedures flowing through an expression. The MrSpidey static debugger for PLT Scheme is based on Flanagan's formalism. In PLT Scheme, a `case-lambda` is a procedure with possibly several argument lists and clauses. When a `case-lambda` is applied at a particular call site, at most one clause is actually invoked, chosen by the number of actual arguments. Therefore, an analysis should propagate data only through appropriate `case-lambda` clauses. MrSpidey propagates data through all clauses of a `case-lambda`, lessening its usefulness as a static debugger. Wishing to retain Flanagan's framework, we extended it to better analyze `case-lambda` with rest parameters by annotating selectors with arity information. The resulting analysis gives strictly better results than MrSpidey. Unfortunately, the improved analysis is too expensive because of overheads imposed by the use of selectors. Nonetheless, a closure-analysis style SBA eliminates these overheads and can give comparable results within cubic time.


## 1 Introduction

In 1990, Dybvig and Hieb introduced a variable-arity construct for Scheme, called `lambda*` [5]. The `case-lambda` construct in PLT Scheme is essentially the same as `lambda*`. A `case-lambda` expression is like a `lambda` expression, except that there may be several clauses in a `case-lambda` procedure, each with its own parameter list and body. When such a procedure is applied, one of the clauses may be selected based on the number of actual arguments, and the corresponding body executed. In this way, `case-lambda` offers overloading by number of arguments. If none of the clause argument lists matches the number of actual arguments, an arity error occurs. Both PLT Scheme [16] and Chez Scheme [3] provide `case-lambda`. In both implementations, a `case-lambda` clause parameter list may have a rest parameter, just as for `lambda` in Scheme [13].[1]

In this paper, we describe our extension of the formalism developed by Flanagan [9, 10] for the MrSpidey static debugger [17] to handle `case-lambda` and rest parameters. In Flanagan's set-based analysis (SBA), the *selectors* `dom`, `rng`, `car`, and `cdr` are used to choose data flowing through expressions. For example, the `rng` selector chooses the ranges of procedures that flow through an expression; the `car` selector obtains the first elements of lists that flow through the expression. The matching of selectors from procedures and application sites controls the flow of data into and out of procedures.

MrSpidey analyzes nearly all of PLT Scheme, including `case-lambda`, though Flanagan does not provide a formal treatment of that part of its analysis. Unfortunately, the existing implementation incorrectly propagates values through *all* clauses of `case-lambda`, even unused clauses, and also propagates values in the presence of arity errors. As a consequence, MrSpidey often flags errors where none exists.

In our modification of Flanagan's analysis, we annotate selectors with arity and argument-position information, which assures that data flows into and out of appropriate `case-lambda` clauses. As we shall show, this approach improves upon the existing MrSpidey implementation. Selector annotations also allow us to analyze rest arguments.

While our approach yields sound results, its time cost is too great. Selectors impose two time burdens. First, all selectors associated with procedures are propagated in addition to the procedures themselves. Second, data flowing into and out of procedures is propagated through selector pairs matched according to argument position and arity annotations. Obtaining these pairs requires searching through a list of candidate selectors and checking the annotations for each candidate.

Fortunately, a closure-analysis style SBA (CA-SBA) [14] can provide similar results, and compute it with a lower asymptotic time upper bound. With those insights, we conclude that the annotated-selector approach to SBA is not suitable for analysis of a Scheme with `case-lambda`. Therefore, we plan to use a CA-SBA analyzer in a future version of DrScheme, in place of MrSpidey [7].

The paper is organized as follows. We begin in Section 2 by presenting limitations of the MrSpidey implementation. In Section 3, we review Flanagan's account of set-based analysis with selectors. Next, in Section 4, we describe our extension to Flanagan's system for `case-lambda` programs without rest arguments. In Section 5, we add rest arguments to the analysis. In Section 6, we obtain the complexity of the resulting analysis. In Section 7, we present a closure-analysis style SBA that handles `case-lambda` and rest parameters. In Section 8, we give empirical results, comparing MrSpidey, our modified selector analysis, and a closure-analysis style


*Meunier, Findler, and Steckler were partially supported by National Science Foundation grants CCR-9619756, CDA-9713032, and CCR-9708957, and a State of Texas ATP grant; Wand was supported by National Science Foundation grant CCR-9804115. For Meunier and Steckler, most of this work was done at Rice University.


[1]Dybvig and Hieb described a notion of "rest variable", which is not quite like a Common Lisp or Scheme rest parameter.

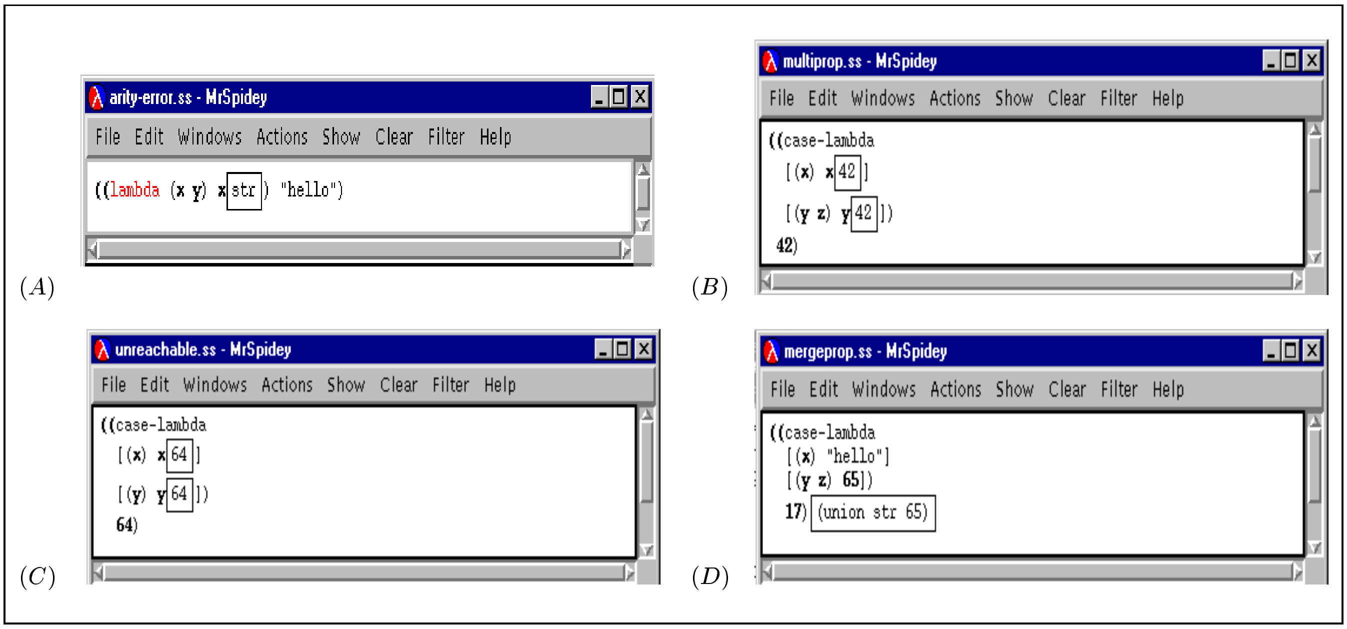


Figure 1: MrSpidey mishandles `case-lambda`.

SBA. Section 9 presents related and future work. Finally, in Section 10, we offer conclusions.

## 2 Limitations of MrSpidey

In this section, we catalog the ways in which the existing MrSpidey does an unsatisfactory job with the analysis of `case-lambda`. In Figure 1, we show the results of MrSpidey's analysis of four programs. The boxes contain type information about their adjacent expressions. A type may be a definite constant, such as a particular number.

*Propagation despite arity errors.* When a procedure is applied to an incorrect number of arguments, MrSpidey propagates data through as many formal arguments as possible. Figure 1-(A) shows MrSpidey's analysis of a procedure of two arguments applied to one argument.[2] At run-time, the value of the actual argument never reaches the bound `x`, though MrSpidey suggests otherwise.

*Propagation through multiple clauses.* MrSpidey propagates values of actual arguments through all clauses of a `case-lambda`. Figure 1-(B) shows a `case-lambda` with two clauses applied to some arguments. Even though the actual argument flows at run-time through just the first clause, MrSpidey shows the actual argument flowing through the other clause as well.[3]

*Propagation through unreachable clauses.* MrSpidey propagates information through unreachable clauses of a `case-lambda`. Because ordering of clauses in a `case-lambda` is significant, only the first of multiple clauses with the same number of arguments will receive data. Figure 1-(C) shows the application of a `case-lambda` with two clauses, both of which take a single argument. MrSpidey propagates the actual argument through both clauses, though data flows only through the first clause at run-time.

*Merging of clause return values.* Symmetrically, MrSpidey merges values returned by all clauses of a `case-lambda`. Figure 1-(D) shows that the result of applying a `case-lambda` with two clauses is the union of the clause results, though only one clause is ever evaluated.

*Spurious errors.* MrSpidey's usefulness as a static debugger is compromised when data is shown to flow to locations that it cannot actually reach. Figure 2 shows a program for which MrSpidey claims a possible arity error, though there is none. MrSpidey reasons that the `lambda` may flow to the formal parameter `f` in the second clause in the `case-lambda`, where it could be misapplied. From the program text, it is clear that the `lambda` flows only through the first clause.

Our modified analysis remedies each fault identified here.

## 3 MrSpidey theory

We review Flanagan's formalism for analyzing the $\lambda$-calculus with constants and special forms for **cons**, **car**, and **cdr**. In this language, `lambda` terms are labeled.

*Set expressions* are defined by the grammar

$$\tau ::= \alpha \mid c \mid \texttt{pair} \mid \texttt{dom}(\alpha) \mid \texttt{rng}(\alpha) \mid \texttt{car}(\alpha) \mid \texttt{cdr}(\alpha)$$

where $\alpha$ is a *set variable* and `pair` is a token. We may also write $\alpha$, $\beta$, and $\gamma$ for set variables. The metavariable $c$ represents constants, including term language constants and `lambda` labels. The forms `dom`, `rng`, `car` and `cdr` are *selectors*. Of these, only `dom` is *contravariant*; the others are *covariant*. We use $\sigma$ as a metavariable for selectors. A *constraint* is an inequality on set expressions of the form $\tau \leq \tau'$. Constraints indicate the flow of data. For example, the constraint $c \leq \alpha$ means that the constant $c$ flows into the expression associated with the set variable $\alpha$.

Flanagan's set-based analysis proceeds by phases. The first phase is *constraint derivation*, performed by a pass over the program's abstract syntax tree. For each subexpression, this phase associates a set variable with the subexpression and generates some constraints according to constraint

[2] A `lambda` expression is treated as a `case-lambda` expression with a single clause.

[3] MrSpidey can use its "if-splitting" feature to restrict flow through `cond` clauses. For example, in `(cond ((string? x) E) ...)`, MrSpidey can deduce that `x` is a string in `E` and not a string in other clause bodies.

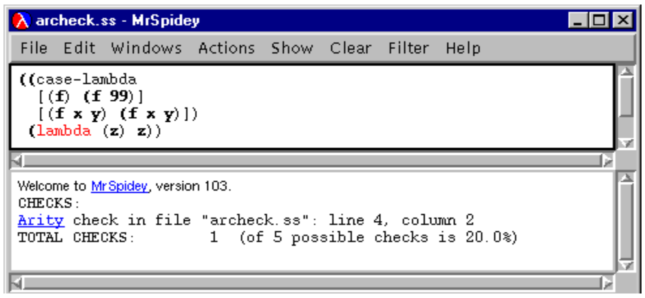


Figure 2: Spurious arity check.

$$\Gamma[x \mapsto \beta] \vdash x : \alpha, \{\beta \leq \alpha\} \qquad (\textsc{var})$$

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

$$\frac{\Gamma[x \mapsto \alpha'] \vdash M : \beta, C'}{\Gamma \vdash (\lambda x.M)^\ell : \alpha, C' \cup C} \qquad (\textsc{lambda})$$

$$\text{where } C = \left\{ \begin{array}{c} \ell \leq \alpha \\ \texttt{dom}(\alpha) \leq \alpha' \\ \beta \leq \texttt{rng}(\alpha) \end{array} \right\}$$

$$\frac{\Gamma \vdash M_i : \beta_i, C_i \quad i \in [1..2]}{\Gamma \vdash (M_1\ M_2) : \alpha, C_1 \cup C_2 \cup C} \qquad (\textsc{app})$$

$$\text{where } C = \left\{ \begin{array}{c} \beta_2 \leq \texttt{dom}(\beta_1) \\ \texttt{rng}(\beta_1) \leq \alpha \end{array} \right\}$$

$$\frac{\Gamma \vdash M_i : \beta_i, C_i \quad i \in [1..2]}{\Gamma \vdash (\textbf{cons}\ M_1\ M_2) : \alpha, C_1 \cup C_2 \cup C} \qquad (\textsc{cons})$$

$$\text{where } C = \left\{ \begin{array}{c} \texttt{pair} \leq \alpha \\ \beta_1 \leq \texttt{car}(\alpha) \\ \beta_2 \leq \texttt{cdr}(\alpha) \end{array} \right\}$$

$$\frac{\Gamma \vdash M : \beta, C}{\Gamma \vdash (\textbf{car}\ M) : \alpha, C \cup \{\texttt{car}(\beta) \leq \alpha\}} \qquad (\textsc{car})$$

$$\frac{\Gamma \vdash M : \beta, C}{\Gamma \vdash (\textbf{cdr}\ M) : \alpha, C \cup \{\texttt{cdr}(\beta) \leq \alpha\}} \qquad (\textsc{cdr})$$

Figure 3: MrSpidey constraint derivation.

derivation rules. Next, a *propagation* phase combines constraints using constraint propagation rules to generate new constraints, effectively mimicking the flow of data through a program. Then, a set of values is computed for each program point. From such a set, a type can be constructed.

Figure 3 shows the constraint derivation rules for the $\lambda$-calculus with constants, **cons**, **car**, and **cdr**. The judgements in Figure 3 are of the form

$$\Gamma \vdash M : \alpha, C$$

where

- $\Gamma$ is an environment from term variables to set variables,
- $M$ is a term,
- $\alpha$ is a set variable, and
- $C$ is a set of constraints.

Let us provide intuition for some of the rules. The VAR rule says that data flows into a variable from its binding variable, a formal parameter. For the bracketed constraints in the LAMBDA rule, we have

- $\ell \leq \alpha$: a procedure label flows into the set variable for the procedure;
- $\texttt{dom}(\alpha) \leq \alpha'$: whatever flows into the domain of the procedure flows into its formal parameter, and
- $\beta \leq \texttt{rng}(\alpha)$: the result of the procedure body flows into the range of the procedure.

There are similar explanations for the other constraints in Figure 3.

Figure 4 shows the constraint propagation rules. In the TRANS-CONST rule, we use the predicates **const?**, **label?**, and **token?** to detect constants, procedure labels, and tokens. The difference between covariant and contravariant selectors shows up in the propagation rules COVARIANT-PROP and CONTRAVARIANT-PROP. The $\textbf{selector}^+$ predicate holds when its argument is `rng`, `car`, or `cdr`; the $\textbf{selector}^-$ predicate holds only for `dom`. These propagation rules follow the presentation in Flanagan [9], with some simplification and notational changes. These rules are repeatedly applied until no new constraints are added.

The full details of MrSpidey's constraint solution and type reconstruction algorithms are beyond the scope of this paper, but we attempt here to convey their essence. See Flanagan's dissertation for details [9]. For a subterm with associated set variable $\alpha$, the set $\{c \mid c \leq \alpha\}$ describes the constants that may be the result of evaluating the subterm. If we have the constraint $\texttt{pair} \leq \alpha$, then the term may evaluate to a pair, and $\{\beta \mid \beta \leq \texttt{car}(\alpha)\}$ is the set of set variables that may flow into the **car** of such a pair. The sets of values for these set variables provide the actual values. We compute the solutions to **cdr**'s and procedure ranges in similar fashion. Procedure domains require a slightly more complex calculation due to the contravariance of the `dom`

$$\frac{\tau \leq \alpha \quad \alpha \leq \beta \quad \begin{array}{l}\textbf{const?}(\tau) \vee \\ \textbf{label?}(\tau) \vee \\ \textbf{token?}(\tau)\end{array}}{\tau \leq \beta} \quad (\textsc{trans-const})$$

$$\frac{\alpha \leq \sigma(\gamma) \quad \sigma(\gamma) \leq \beta}{\alpha \leq \beta} \quad (\textsc{trans-sel})$$

$$\frac{\alpha \leq \sigma(\beta) \quad \textbf{selector}^{+}\textbf{?}(\sigma) \quad \beta \leq \gamma}{\alpha \leq \sigma(\gamma)} \quad (\textsc{covariant-prop})$$

$$\frac{\sigma(\alpha) \leq \beta \quad \textbf{selector}^{-}\textbf{?}(\sigma) \quad \alpha \leq \gamma}{\sigma(\gamma) \leq \beta} \quad (\textsc{contravariant-prop})$$

Figure 4: MrSpidey constraint propagation.

selector.

From the sets of values associated with set variables, we can construct types. For example, let $\alpha_M$ be the set variable associated with a term $M$. Suppose that constraint propagation produces the constraints $\texttt{pair} \leq \alpha_M$, $\beta_1 \leq \texttt{car}(\alpha_M)$, $57 \leq \beta_1$, $\beta_2 \leq \texttt{cdr}(\alpha_M)$, and $\texttt{null} \leq \beta_2$. Then we can conclude that $M$ has type (**cons** 57 `null`).

What is missing from the existing formalism? In the language to be analyzed, all procedures have one clause with one parameter, and there are no rest parameters. In his dissertation [9, Appendix E.3], Flanagan indicates that a procedure of more than one argument is modeled by considering that procedure to take one argument, which becomes bound to a list of actual arguments at its application sites. The values in the list are distributed to the formal arguments by pulling out elements of the list. Because all clauses of a `case-lambda` are considered to have a single argument, the arities of the clauses are not considered, and that list is propagated to all clauses. Similarly, the results of all `case-lambda` clauses are merged into application results. The TRANS-SEL rule in Figure 4 controls the propagation of data into formal parameters (when the selector involved is `dom`) and out of procedures (when the selector is `rng`). Hence, to improve the analysis, we need to focus on the mechanism in that rule.

## 4 Handling case-lambda

In this section, we show how to analyze programs containing `case-lambda` but without rest parameters. We will show how to add rest parameters in Section 5.

Because the run-time clause selection in `case-lambda` depends on the number of actual arguments, our analysis keeps track of clause arities. Whether a clause is selected depends not only on the number of arguments it may accept, but also on the number of arguments accepted by preceding clauses. Therefore, our notion of arity is somewhat unusual. In order to define this notion, we need

**Definition 1** *An* interval *is a pair* $[n, m]$, *where* $n$ *and* $m$ *are nonnegative integers.*

An interval indicates the number of arguments a clause accepts. Without rest parameters, the lower and upper bounds on the interval are the same. We use $\mathcal{I}$ as a metavariable for intervals.

**Definition 2** *An* arity *is a pair whose first element is an interval, and whose second element is a list of intervals, possibly empty.*

The first element of an arity indicates the number of arguments accepted by a clause. The second element puts the clause in context, by listing the intervals associated with preceding clauses. We write $a$ for a typical arity.

We augment Flanagan's `dom` and `rng` selectors by annotating them with various information. The same selector has different kinds of annotations, depending on where it is generated. For constraints generated at `case-lambda` instances, selectors get arity annotations; for constraints generated at applications, selectors carry interval and number-of-argument information. Hence there are two forms each of annotated `dom` and `rng` selectors.

In particular, for `dom` selectors, the two forms are

- $\texttt{dom}_i^a$, where $a$ is an arity and $i$ is an argument index in a clause parameter list, and
- $\texttt{dom}_{i,n}^{\mathcal{I}}$, where $\mathcal{I}$ is an interval, $i$ is an argument index in an application argument list, and $n$ is the total number of arguments.

For `rng` selectors, the forms are

- $\texttt{rng}^a$, where $a$ is an arity, and
- $\texttt{rng}_n^{\mathcal{I}}$, where $\mathcal{I}$ is an interval, and $n$ is the total number of arguments at an application site.

Consider $\texttt{dom}_2^a(\alpha)$; this set expression represents the flow into the second argument of a `case-lambda` clause with arity $a$. Similarly, $\texttt{dom}_{1,3}^{[3,3]}(\alpha)$ represents the flow into the first argument of a procedure that flows into an application site with three actual arguments. The selector $\texttt{rng}^a(\alpha)$ represents the flow out of a `case-lambda` clause with arity $a$. The set expression $\texttt{rng}_3^{[3,3]}(\alpha)$ represents the value returned by a procedure at an application site with three actual arguments.

Figure 5 gives revised constraint generation rules. The rules for constants, variables, **car** and **cdr** are unchanged.

For the constraints in the CASE-LAMBDA rule, arities are computed as follows. For each clause $i$, assign it an interval $[n_i, n_i]$, where $n_i$ is the number of formal arguments for that clause. Then, for each clause, assign it the arity $a_i = ([n_i, n_i], (\mathcal{I}_{i-1}, \ldots, \mathcal{I}_1))$, where $\mathcal{I}_j$ is the interval assigned to the $j^{th}$ clause.

In the APP rule, the selectors are annotated with intervals as well as a separate annotation for the number of arguments. The three numbers in the annotation are all the same, but that will change when we consider rest arguments in the next section.

Note that these rules have the essential form of those in Figure 3, except that

- each `case-lambda` clause generates constraints,
- each application argument generates constraints, and
- the selectors are annotated.

$$\frac{\Gamma[x_{i,j} \mapsto \alpha_{i,j}] \vdash M_i : \beta_i, C_i \quad i \in [1..m], j \in [1..n_i]}{\begin{array}{l}\Gamma \vdash (\textbf{case-}\lambda \\ \qquad ((x_{1,1} \ldots x_{1,n_1})\ M_1) \\ \qquad\qquad \vdots \\ \qquad ((x_{m,1} \ldots x_{m,n_m})\ M_m))^{\ell} : \alpha, \displaystyle\bigcup_{i \in [1..m]} C_i \cup C\end{array}} \quad (\textsc{case-lambda})$$

$$\text{where } C = \left\{ \left. \begin{array}{c} \ell \leq \alpha \\ \mathtt{dom}_j^{a_i}(\alpha) \leq \alpha_{i,j} \\ \beta_i \leq \mathtt{rng}^{a_i}(\alpha) \end{array} \right\} \begin{array}{l} i \in [1..m], \\ j \in [1..n_i] \end{array} \right\}$$

$$\frac{\Gamma \vdash M_i : \beta_i, C_i \quad i \in [1..n]}{\Gamma \vdash (M_0\ \ldots\ M_n) : \alpha, \displaystyle\bigcup_{i \in [1..n]} C_i \cup C} \quad (\textsc{app})$$

$$\text{where } C = \left\{ \begin{array}{l} \beta_i \leq \mathtt{dom}_{i,n}^{[n,n]}(\beta_0) \quad i \in [1..n] \\ \mathtt{rng}_n^{[n,n]}(\beta_0) \leq \alpha \end{array} \right\}$$

Figure 5: Revised constraint derivation rules.

$$\frac{\alpha \leq \mathtt{dom}_{i,s}^{[n,m]}(\beta) \quad \mathtt{dom}_i^{a}(\beta) \leq \gamma \quad (s, [n,m]) \models a}{\alpha \leq \gamma} \quad (\textsc{trans-dom})$$

$$\frac{\alpha \leq \mathtt{rng}^{a}(\beta) \quad \mathtt{rng}_s^{[n,m]}(\beta) \leq \gamma \quad (s, [n,m]) \models a}{\alpha \leq \gamma} \quad (\textsc{trans-rng})$$

$$\frac{\alpha \leq \mathtt{rng}^{a}(\beta) \quad \beta \leq \gamma}{\alpha \leq \mathtt{rng}^{a}(\gamma)} \quad (\textsc{rng-prop})$$

$$\frac{\mathtt{dom}_i^{a}(\alpha) \leq \beta \quad \alpha \leq \gamma}{\mathtt{dom}_i^{a}(\gamma) \leq \beta} \quad (\textsc{dom-prop})$$

Figure 6: Revised propagation rules.

The selector annotations are used in the revised propagation rules in Figure 6, in particular, in the rules TRANS-DOM and TRANS-RNG. The unchanged rules TRANS-CONST, TRANS-SEL, and COVARIANT-PROP are omitted. The COVARIANT-PROP rule is only used to propagate the **car** and **cdr** selectors. We no longer need the CONTRAVARIANT-PROP rule. The rule TRANS-SEL no longer handles `dom` and `rng` selectors. The core idea is to propagate values through a `case-lambda` clause only when the number of actual arguments matches the number expected by that clause, and does not match the number expected by any preceding clause. This idea is captured by the following satisfaction relations used in the TRANS-DOM and TRANS-RNG rules. For intervals, we have

**Definition 3** $[n,m] \models [p,q]$ *iff* $n = m = p = q$.

The satisfaction relation in the propagation rules involves an interval, a number representing a number of actual arguments, and an arity. That relation is defined by:

**Definition 4** $(s, [n,m]) \models ([p,q], (\mathcal{I}_1, \ldots, \mathcal{I}_t))$ *iff*

- $s \in [p,q]$,
- $[n,m] \models [p,q]$, *and*
- $\forall i \in [1..t],\ s \notin \mathcal{I}_i$

The first two requirements assure that a particular clause can handle the number of arguments given; the last one makes sure that no preceding clause can do so.

## 5 Analysis of rest parameters

The introduction of rest parameters requires additional constraints, which need to account for the uncertainties associated with such arguments. With a rest parameter, a clause accepts some number of required arguments, but may take more. So for a particular clause, we cannot be certain how many arguments it will be applied to. Moreover, when deriving constraints at an application site, we do not yet know the arity of selected clauses in procedures that flow to that site. Therefore, our constraints need to account for all possibilities.

With the advent of rest parameters, we continue to generate all the constraints as described in the last section. We

$$\frac{\Gamma \vdash M_i : \beta_i, C_i \quad i \in [1..n]}{\Gamma \vdash (M_0 \ \dots \ M_n) : \alpha, \ \bigcup_{i \in [1..n]} C_i \cup C} \quad (\textsc{app})$$

$$\text{where } C = \left\{ \begin{array}{ll} \mathtt{rng}_n^{[i,\omega]}(\beta_0) \leq \alpha & i \in [0..n] \\ \left. \begin{array}{r} \beta_i \leq \mathtt{car}(\alpha_i) \\ \alpha_{i+1} \leq \mathtt{cdr}(\alpha_i) \\ \mathtt{pair} \leq \alpha_i \end{array} \right\} & i \in [1..n] \\ \mathtt{null} \leq \alpha_{n+1} & \\ \alpha_{i+1} \leq \mathtt{dom}_{i+1,n}^{[i,\omega]}(\beta_0) & i \in [0..n] \\ \beta_j \leq \mathtt{dom}_{j,n}^{[i,\omega]}(\beta_0) & i \in [0..n],\ j \in [1..i] \end{array} \right\}$$

Figure 7: Additional constraints for rest parameters.

revise the definition of intervals (Definition 1) to allow intervals of the form $[n, \omega]$, where $n$ is a nonnegative integer and $\omega$ is a special symbol. The calculation of arities changes slightly: a clause with $n$ required arguments and a rest parameter is assigned the interval $[n, \omega]$. As before, the arity of a clause is a pair consisting of its assigned interval and a list of intervals from preceding clauses. Because intervals have changed, we modify slightly the satisfaction relation on interval pairs from Definition 3:

**Definition 5** $[n, m] \models [p, q]$ *iff*

- $n = m = p = q \neq \omega$, *or*
- $n = m$ *and* $p = q = \omega$

The other satisfaction relation, from Definition 4, now makes use of this new definition.

In Figure 7, we show just the new constraints required. For the language we are now considering, which includes `case-lambda`, **cons**, **car**, and **cdr**, the derivation rules are the VAR, CONST, CONS, CAR, and CDR rules from Figure 3; the CASE-LAMBDA rule from Figure 6, and the APP rules in Figures 6 and 7.

The CASE-LAMBDA rule is unchanged: the new calculation of arities handles the uncertainty associated with individual clauses. All the complications appear in the new constraints for the APP rule.

Consider an application with $n$ actual arguments. A selected clause in a procedure that flows to this site, if that clause has a rest parameter, may take between zero and $n$ required arguments. A procedure in which all clauses have more than $n$ required arguments results in an arity error. We cannot know exactly how many arguments an incoming procedure requires, so we account for each possibility. Therefore, we generate a constraint of the form $\mathtt{rng}_n^{[i,\omega]}(\beta_0) \leq \alpha$ for each $i$ between zero and $n$. These constraints represent flow out of selected `case-lambda` clauses.

Next, consider flow into the required arguments of a selected clause with rest parameters. For each $i$ from zero to $n$, and for each $j$ from one to $i$, we generate a constraint of the form:

$$\beta_j \leq \mathtt{dom}_{j,n}^{[i,\omega]}(\beta_0)$$

Here, $i$ is a particular number of required arguments, $j$ is a position within those arguments, and $n$ is the number of actual arguments. These constraints represent flow into the required parameters of clauses with rest parameters.

Finally, we need to account for flow into rest arguments, which are bound to lists. Suppose a selected clause has exactly $n$ required arguments. Then at run-time, the rest argument becomes bound to the empty list. Hence we have the pair of constraints

$$\begin{array}{c} \mathtt{null} \leq \alpha_{n+1} \\ \alpha_{n+1} \leq \mathtt{dom}_{n+1,n}^{[n,\omega]}(\beta_0) \end{array}$$

Suppose a selected clause takes fewer than $n$ required arguments. Regardless of the number, we always generate the constraints

$$\begin{array}{rl} \beta_1 & \leq \mathtt{car}(\alpha_1) \\ \alpha_2 & \leq \mathtt{cdr}(\alpha_1) \\ & \vdots \\ \beta_n & \leq \mathtt{car}(\alpha_n) \\ \alpha_{n+1} & \leq \mathtt{cdr}(\alpha_n) \end{array}$$

The effect of these constraints is to flow lists of varying lengths into the $\alpha_i$. The $\alpha_i$'s represent possible list flows into rest arguments. For instance, $\alpha_1$ receives a list of length $n$, while $\alpha_n$ receives a list of length one. Then we handle each of the lists that may flow into the rest argument by generating constraints of the form

$$\alpha_{i+1} \leq \mathtt{dom}_{i+1,n}^{[i,\omega]}(\beta_0)$$

for each $i$ from zero to $n-1$. Again, $i$ is a particular number of required arguments. Therefore, the rest parameter receives a list of length $n - i$.

The `pair` token is used by our type reconstruction algorithm to flag pairs. In Section 3, it appeared in the *cons* rule. Here, we generate the constraints

$$\mathtt{pair} \leq \alpha_i$$

for each $i$ from one to $n$. The effect is to propagate the token to the rest argument, but only in case it may become bound to a nonempty list.

We have built a prototype implementation using the new derivation and propagation rules. For each program we showed in Figures 1 and 2, the prototype remedies the problem identified with MrSpidey. For the program in Figure 1-(A), there is no flow through the bound $x$. For the program in Figure 1-(B), there is flow only through the bound $x$, and not through the bound $y$. For the program in Figure 1-(C), there is flow only through the bound variable in the first clause, $x$, but not through the bound variable in the second clause, $y$. For the program in Figure 1-(D), only the return value from the first clause shows up in the flow for the application. In the prototype, the program in Figure 2 does not signal an arity error. Despite these improvements, we argue in the next section, the modified analysis is unsatisfactory.

## 6 Complexity

It is not enough for our analysis to be sound – it must be easily computable. Now, the usual formulations of monovariant SBA claim that it may be done in time cubic in the size of programs [1]. Because there do not appear to be better bounds without imposing restrictions on programs, this complexity is known as the "cubic bottleneck" [12]. Unfortunately, our modified version of Flanagan's SBA exceeds this bound.

### 6.1 MrSpidey's analysis

It is easy to see that the time upper bound on Flanagan's original analysis can be no better than that for graph transitive closure, which can be computed in time cubic in the number of graph nodes [4, Section 26.2]. Deriving the constraints in Figure 3 takes time linearly bounded by the size of a program. For the propagation phase, the rules `trans-const` and `trans-sel` in Figure 4 are ordinary transitive rules. Hence, closing under these two rules does have a cubic-time upper bound. The other two rules in Figure 4 are of a different character, so the actual complexity might be higher. As we shall show, the complexity is cubic, in fact.

We describe now an algorithm that can be used to close the constraints under the propagation rules in Figure 4. The algorithm has a cubic-time upper bound. As each constraint is generated, check whether it matches a premise in a propagation rule, a constant-time operation. Because there are $O(n)$ set expressions, there are $O(n^2)$ possible constraints. Note that each propagation rule has two premises. If the constraint matches a propagation rule premise, find all constraints that match the other premise. There are $O(n)$ many of these constraints. To see this, suppose the rule involved is `trans-const`, and we have the constraint $c \leq \alpha$. So the other premise in the rule is matched by constraints of the form $\alpha \leq \beta$. The left-hand side for eligible constraints is fixed to be $\alpha$, so there are $O(n)$ many candidate set variables for the right-hand side. Similar considerations apply to the other rules. Each eligible constraint can be found in constant time by maintaining lookup tables from set expressions to their lower and upper bounds. If the constraint in the consequent does not exist in the pool of constraints, a constant-time check, add it. The only non-constant factors in this algorithm are the quadratic bound on the number of constraints and the linear bound on the number of eligible constraints. Therefore, the MrSpidey analysis does have a cubic-time upper bound.

### 6.2 Annotated selectors

When we add annotations to selectors, the number of possible set expressions becomes much larger, raising the complexity of both the derivation and propagation phases. First consider what happens when adding just arities for multiple arguments, without rest parameters. The derivation phase still creates a linear number of constraints. Although the CASE-LAMBDA rule in Figure 5 contains a "doubly-nested loop" for constraints with the `dom` selector, there is only one such constraint for each formal parameter. Again, the derivation time is dominated by the propagation time.

In order to obtain the time complexity for the propagation phase in the presence of annotated selectors, we again look at the number of possible constraints and the time for the work to be done when a constraint matches a premise in a propagation rule.

Because `case-lambda` parameter lists and application argument lists may be proportional to the size of the whole program, the number of different annotated `dom` selectors is linear in the size of programs (see Figure 5). For each set variable $\alpha$, then, we now have a *linear* number of possible set expressions containing $\alpha$. Hence the number of possible set expressions is quadratic in the size of the program. Considering just the syntax of constraints, the number of possible constraints is cubic, because every set constraint derived or deduced from the propagation rules has a set variable as its lower or upper bound.

The number of constraints actually produced by the derivation and the propagation rules is only quadratic, as follows. The number of constraints containing only set variables, constants, labels, and the `pair` token is quadratic, because we have only a linear number of each of these items. The only other constraints are those with a selector applied to a set variable on one side, and a set variable on the other. By the derivation rules in Figures 3 and 5, we start with a linear number of such constraints. The only rules that can create new such constraints are COVARIANT-PROP in Figure 4 for the **car** and **cdr** selectors and RNG-PROP and DOM-PROP in Figure 6. In the rules COVARIANT-PROP and RNG-PROP, there is a premise of the form $\alpha \leq \sigma(\beta)$ and the added constraint is of the form $\alpha \leq \sigma(\gamma)$. So $\alpha$ and $\sigma$ appear in the premise and in the added constraint, playing the same syntactic roles in both. We start with $O(n)$ many such $\alpha$ and $\sigma$ pairs, and the propagation rules do not increase their number. There are $O(n)$ many set variables to play $\beta$, the other syntactic role in those rules. So after propagation, there are $O(n^2)$ many constraints of the form $\alpha \leq \sigma(\beta)$. A similar argument holds for the DOM-PROP rule.

Next, we wish to obtain the time needed when a constraint matches a propagation rule premise. As mentioned above, that time is related to the length of the list of constraints eligible to match the other premise in the rule. In the presence of annotated selectors, the number of such constraints eligible to match the other premise has an $O(n)$ bound. This bound arises directly from the syntax of constraints for the rules TRANS-CONST, TRANS-SEL and COVARIANT-PROP. For the other rules, those in Figure 6, we must consider the number of constraints actually produced. We will show that for each such rule, the number of eligible constraints has an $O(n)$ bound.

Consider the rule TRANS-DOM. Suppose we have a constraint matching the first premise, $\alpha \leq \mathtt{dom}_{i,s}^{[n,m]}(\beta)$. As we showed above, there can be at most a linear number of $\sigma$ and $\gamma$ pairs appearing in constraints of the form $\sigma(\beta) \leq \gamma$. So for a given $\beta$, there are at most a linear number of constraints of the form $\mathtt{dom}_i^a(\beta) \leq \gamma$. On the other hand, suppose we have a constraint matching the second premise. From the CASE-LAMBDA rule in Figure 5, there are $O(n)$ many constraints of the form of the first premise produced during the derivation phase, and no new constraints of this form are created during propagation. A similar argument holds when considering the TRANS-RNG rule.

Now consider the rule RNG-PROP. If we have a constraint matching the first premise, then clearly there is a linear bound on the number constraints matching the second premise. Suppose we have a constraint of the form $\beta \leq \gamma$, matching the second premise. As we have shown, there can be at most a linear number of $\alpha$ and $\sigma$ pairs in constraints of the form $\alpha \leq \sigma(\beta)$. For a given $\beta$, then, there is a linear bound on the number of constraints matching the first premise. A similar argument holds for the DOM-PROP rule.

We have shown that there is an $O(n^2)$ bound on the number of constraints, and for each such constraint, an $O(n)$ bound on the number of eligible constraints when matching premises in the propagation rules. For the rules TRANS-DOM and TRANS-RNG, which involve the satisfaction relation, the lists contained in arities add a linear factor. When we check whether a constraint already exists, we need to compute its hash value. That computation has a linear bound, because constraints may contain arities in selector annotations. Combining all these factors, we see that the algorithm has a worst-case time bound of $O(n^5)$.

### 6.3 Rest parameters

If we add in the constraints for rest parameters (Figure 7), the number of constraints produced by the derivation phase becomes quadratic in the size of the program. Nonetheless, the total number of constraints after the propagation phase still has a quadratic upper bound. The constraints involving `dom` and `rng` introduced in Figure 7 do not propagate those selectors to new constraints. For the other constraints in Figure 7, the syntax of constraints imposes a quadratic bound on the number of constraints produced from them during propagation. The number of eligible constraints for rule matches retains a linear bound in this case. Again, we need to consider the linear bounds on checking the satisfaction relation and computing hash values. Therefore, even when we add the constraints for rest parameters, the algorithm has a worst-case time bound of $O(n^5)$.

That time bound is undesirably high. But by using a different analysis, described in the following section, we can compute essentially the same information asymptotically faster.

## 7 Eliminating selectors

In Flanagan's original SBA and our revision, `dom` and `rng` selectors are used to hook up actual arguments with formal parameters, and procedure bodies with applications. Each piece of data flows through a selector at these critical points. But we can eliminate selectors by choosing a more straightforward mechanism for directing flow through formal parameters and from procedure bodies.

An ordinary "closure analysis" SBA can handle `case-lambda` and rest parameters. Figure 8 presents the constraints for such an analysis. As before, procedures are labeled; we now label all other subterms. In Figure 8, we have omitted labels where they are not significant. Each such label $\ell$ has an associated set $\phi_\ell$. A set $\phi$ consists of:

- labels, and
- **cons**es whose first part is a label and whose second part is either a label or another **cons** element.

All **cons** elements have finite length. We use the notation

$$(\textbf{list}\ \ell_1\ \ldots\ \ell_n)$$

to indicate

$$(\textbf{cons}\ \ell_1\ (\textbf{cons}\ldots\ (\textbf{cons}\ \ell_n\ \textbf{nil})\ \ldots))$$

where **nil** is a constant representing the empty list. In Figure 8, we use $L$ to represent a possibly-empty list.

The constraints in Figure 8 are similar to those usually presented for closure analysis of the lambda calculus [15]. Our selector-oriented SBA handles constants and forms for list construction and list projection, so we add constraints to handle those. Of course, we have `case-lambda` instead of `lambda`. The constraints account for that difference by determining which clause is selected at an application, and propagating information through the correct clause. The notation $\mathbf{req}(\ell, n)$ indicates the number of required arguments when the `case-lambda` labeled $\ell$ is applied to $n$ arguments. With these changes, the form of these constraints is the same as for closure analysis of the lambda calculus. As for such closure analyses, there are at most a quadratic number of constraints. This form of constraints can be solved in cubic time [15].

In this form of SBA, dataflow paths between actual arguments and formal parameters, and between clause bodies and applications are established just once. Procedure flow is modeled by the flow just of labels, without requiring the flow of selectors. Despite these simplifications, the information computed by the CA-SBA is effectively the same as for the selector-oriented SBA. While we have not yet devised an algorithm to reconstruct friendly types from the flow information, our intuition is that it will be simpler than the algorithm used by MrSpidey.

## 8 Empirical results

While the worst-case bounds mentioned in Section 6 do not necessarily mean bad performance in practice, it is clear that selector annotations make the problem harder than expected for SBA. To verify our expectations, we ran our annotated selector prototype, MrSpidey, and our CA-SBA prototype on some test programs. For the CA-SBA prototype, we used a variation on the implementation technique for constraint solving described in [15]. MrSpidey runs as an add-on tool in DrScheme [8]. For our tests, we ran the two prototypes directly in MzScheme, the evaluator that underlies DrScheme. The tests were run on a Sun Enterprise 450 with four processors and two gigabytes of main memory. In MrSpidey, types are reconstructed from flow information on user demand for particular subexpressions; hence, we have not considered that time in our comparisons. In all cases, the CA-SBA implementation ran significantly faster than the other two, and with a much lower asymptotic complexity, as we describe in the following section. While adding other analysis techniques and other terms will certainly slow it down, these results are encouraging.

We have not been able to show that the bounds given above for the annotated-selector analysis are tight bounds. But we are able to show that for a particular class of examples, the algorithm for the annotated-selector analysis is nearly cubic, much worse than the other two implementations.

Consider the results in Figure 9. The numbers indicate milliseconds of process time, with garbage-collection times subtracted. The programs s200, s400, and so on contain procedures of a single argument that call one another in a linear chain, where the number indicates how many procedures there are in the chain. For this series of tests, both the annotated selector and CA-SBA versions are asymptotically faster than MrSpidey. The programs m200, m400 and so on are similar, except that the procedures take multiple arguments. Introducing multiple arguments slows down the annotated-selector version somewhat, although it is still asymptotically better than MrSpidey. Multiple arguments also yield a slowdown for the CA-SBA, although it is less than for the annotated-selector version. These results demonstrate that for some programs, at least, our annotated-se-

| **Term** | Constraints |
|---|---|
| $c^{\ell}$ | $\{\ell\} \subseteq \phi_{\ell}$ |
| $x^{\ell}$ | $\phi_{\ell'} \subseteq \phi_{\ell}$ where $\ell'$ labels $x$'s binder |
| **case-**$\lambda$ $((x_{1,1} \ldots x_{1,n_1})\ M_1)$ $\ldots$ $((x_{m,1} \ldots x_{m,n_m})\ M_m))^{\ell}$ | $\{\ell\} \subseteq \phi_{\ell}$ |
| $(M_0^{\ell_0}\ M_1^{\ell_1}\ \ldots\ M_n^{\ell_n})^{\ell}$ | $\ell' \in \phi_{\ell_0} \Rightarrow \begin{cases} \phi_{\ell_j} \subseteq \phi_{\ell'_j}\ 1 \leq j \leq \mathbf{req}(\ell', n) \\ (\mathbf{list}\ \ell_{\mathbf{req}(\ell',n)+1}\ \ldots\ \ell_n) \subseteq \phi_{\ell'_m} \text{ if } m = \mathbf{req}(\ell', n) + 1 \\ \phi_{\ell_b} \subseteq \phi_{\ell} \end{cases}$ if $\ell'$ labels a `case-lambda` whose first clause that matches $n$ arguments has parameters $x_1^{\ell'_1}, \ldots, x_m^{\ell'_m}$ and body $M^{\ell_b}$ |
| $(\mathbf{cons}\ M_1^{\ell_1}\ M_2^{\ell_2})^{\ell}$ | $\{(\mathbf{cons}\ \ell_1\ \ell_2)\} \subseteq \phi^{\ell}$ |
| $(\mathbf{car}\ M^{\ell'})^{\ell}$ | $(\mathbf{cons}\ \ell''\ L) \in \phi_{\ell'} \Rightarrow \{\ell''\} \subseteq \phi_{\ell}$ |
| $(\mathbf{cdr}\ M^{\ell'})^{\ell}$ | $(\mathbf{cons}\ \ell''\ L) \in \phi_{\ell'} \Rightarrow \{L\} \subseteq \phi_{\ell}$ |

Figure 8: Closure analysis style SBA.

| Test | s200 | s400 | s800 | s1200 | s1600 | m200 | m400 | m800 | m1200 | m1600 |
|---|---|---|---|---|---|---|---|---|---|---|
| MrSpidey | 1430 | 3967 | 12833 | 31707 | 47673 | 1967 | 4673 | 14830 | 32350 | 51673 |
| Annotated | 2222 | 4260 | 9248 | 13246 | 20306 | 4569 | 8929 | 20479 | 31239 | 40824 |
| CA-SBA | 858 | 1651 | 3479 | 5273 | 8789 | 1337 | 2614 | 5834 | 9536 | 12176 |
| Ann/MrS | 1.55 | 1.07 | 0.72 | 0.42 | 0.43 | 2.32 | 1.91 | 1.38 | 0.97 | 0.79 |
| CA-SBA/MrS | 0.60 | 0.42 | 0.27 | 0.17 | 0.18 | 0.68 | 0.56 | 0.39 | 0.29 | 0.24 |

Figure 9: Procedure chain tests.

lector algorithm has better asymptotic behavior than MrSpidey.

While the procedure chain tests indicate that the annotated-selector implementation can be competitive with MrSpidey for some programs, another set of stress tests demonstrate its weaknesses. Consider the results in Figure 10. The stress test programs have the form:

```
(define f
  (case-lambda
    [(a) a]
    [(a b) a]
    [(a b c) a]
    [(a b c d) a]
    [(a b c d e) a]))

((f (f (f (f (f f))))) f f f f f)
```

We varied the number of clauses for `f` and the number of applications. In these test programs, the number of clauses is relatively large, and the results of the clause bodies travel a relatively long way. Clearly, the annotated-selector implementation performs much worse than the other two on these tests. We can estimate the exponent for the asymptotic complexity of the implementations on this class of programs by taking the logarithms of the times and the number of nodes. In Figure 10, the last column gives the apparent polynomial exponent for the asymptotic complexity, considering the two largest tests. We calculate the exponent with

$$(log\ t_2 - log\ t_1)/(log\ n_2 - log\ n_1)$$

where $t_1$, $t_2$ are the times and $n_1$, $n_2$ are the number of nodes. For this class of programs, the CA-SBA implementation takes just over a linear amount of time, while the annotated-selector version takes nearly cubic time. The asymptotic complexity of MrSpidey falls in between. Another way to view these relative complexities is given in Figure 11, which shows a log/log graph of the running times for each implementation against the number of program nodes.

What extra work is the annotated-selector algorithm doing that raises its complexity? There are two sources of redundant computation in this framework. First, when a procedure flows to a call site, not only is its label propagated, but also its associated selectors. In Flanagan's original framework, that additional propagation was a constant overhead, because the number of selectors was fixed. With the multiplication of selectors, the selector propagation overhead multiplies as well. Second, in order to establish data paths from actual arguments to formal parameters and from

| Num nodes | 113 | 393 | 848 | 1478 | 2283 | 3263 | Exp |
|---|---|---|---|---|---|---|---|
| MrSpidey | 190 | 757 | 1813 | 7260 | 12653 | 29870 | 2.40 |
| Annotated | 1440 | 21690 | 142100 | 610540 | 2070980 | 5928560 | 2.94 |
| CA-SBA | 50 | 147 | 297 | 497 | 780 | 1130 | 1.04 |
| Ann/MrS | 7.58 | 28.67 | 78.36 | 84.10 | 163.67 | 198.48 | |
| SBA/MrS | 0.26 | 0.19 | 0.16 | 0.07 | 0.06 | 0.04 | |

Figure 10: Stress tests.

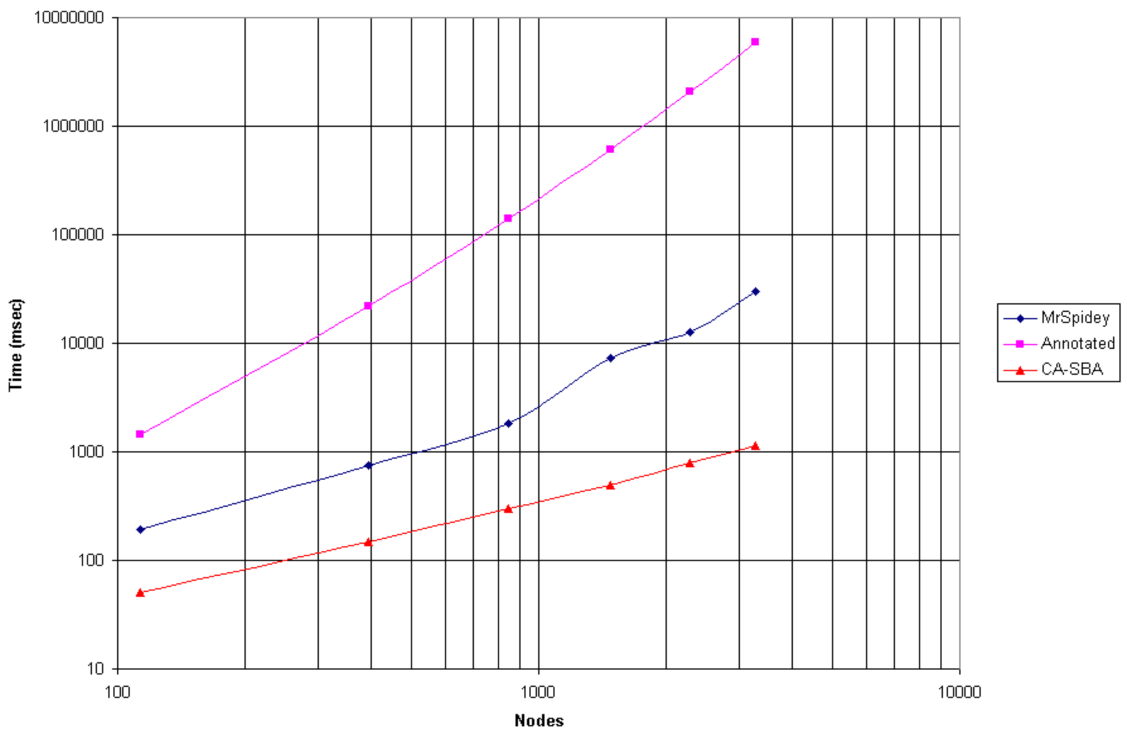


Figure 11: Analysis times, plotted log-log.

procedure clause bodies to applications, we need to search for matching selector pairs. For each candidate selector in that search, we compute whether the satisfaction relation holds. This search is redundant because the data paths can be directly determined from the syntax of procedures.

## 9 Related and future work

We began with Flanagan's theoretical foundations and implementation work for MrSpidey [9, 10]. There are numerous papers on set-based analysis. See [1] for an overview and for pointers to the literature.

The `lambda*` construct, essentially the same as `case-lambda`, was described by Dybvig and Hieb [5].

Heintze and McAllester describe a linear-time algorithm for analyzing ML programs with bounds on the size of types for subexpressions [11]. Their system $LC$ uses *dom* and *ran* constructs that are syntactically similar to Flanagan's `dom` and `rng` selectors, but the two analyses are otherwise quite different. The *dom* and *ran* constructs may be applied to expressions that themselves contain *dom* and *ran*, while `dom` and `rng` may be applied only to set variables. More significantly, the $LC$ system does not include transitive rules, which allows their system to escape the cubic-time bottleneck. Unlike our analyses, the $LC$ system is not concerned with procedures of multiple arguments, because it assumes that all procedures are curried. While our CA-SBA may require cubic time, there are no restrictions on programs to achieve that result.

To our knowledge, there has been no previous attempt to describe set-based analysis for `case-lambda`, nor for Lisp or Scheme rest arguments. Dzeng and Haynes describe a type reconstruction mechanism for an ML-like language with variable-arity procedures [6]. Their language, $\mathrm{ML}^{va}$, allows such procedures only in `let`-bindings, because variable arities are a form of polymorphism. Aiken *et al.* describe a type inference system in which conditional types handle propagation through multi-way `case` expressions – a somewhat different problem than dealing with `case-lambda` [2].

As described here, the closure-analysis style SBA only handles `case-lambda`'s whose program text is known. The analysis needs to be extended to handle separate analysis and other Scheme features. We have begun work on extending the analysis to handle primitives described only by types. Each use of a primitive generates new constraints, giving a limited notion of polyvariance. We believe that this type specification technique can be extended to handle procedures that flow between modules, yielding a true separate analysis.

## 10 Conclusions

We have shown that Flanagan's selector-based framework for SBA can be extended to handle `case-lambda` as well as rest parameters. Unfortunately, the analysis becomes too expensive. Managing the annotations makes it difficult to implement, as well. An ordinary closure-analysis style SBA gives similar results and is straightforward to implement.

For these reasons, we have decided to abandon work using the existing MrSpidey framework. We have begun work on a new static debugger based on the closure analysis framework. The new debugger promises to be significantly faster as well as more precise than MrSpidey.


### Acknowledgements

Kevin Charter first proposed annotating `dom` and `rng` selectors with arity information. Thanks to Cormac Flanagan for discussions about the MrSpidey implementation and our results. Matthias Felleisen and Jamie Raymond offered valuable comments on drafts.


### References


[1] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[2] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*, pages 163–173, 1994.

[3] Cadence Research Systems. Chez Scheme User's Guide. URL: `http://www.scheme.com/csug/`, 1998.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, MA/New York, 1990.

[5] R. K. Dybvig and R. Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3:229–244, 1990.

[6] H. Dzeng and C. T. Haynes. Type reconstruction for variable-arity procedures. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 239–249, 1994.

[7] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A progamming environment for scheme. *Journal of Functional Programming*, 2001. To appear.

[8] C. Flanagan. *MrSpidey: Static Debugger Manual*. Rice University, 1995.

[9] C. Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, May 1997.

[10] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. on Programming Languages and Systems*, 21(2):369–415, Feb. 1999.

[11] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. In *Proc. 1997 ACM Conference on Programming Language Design and Implementation (PLDI '97)*, pages 261–272, 1997.

[12] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS '97)*, pages 342–351, 1997.

[13] R. Kelsey, W. Clinger, and Jonathan Rees, eds. Revised[5] Report on the Algorithmic Language Scheme, Feb. 1998.

[14] J. Palsberg. Closure analysis in constraint form. *Proc. ACM Trans. on Programming Languages and Systems*, 17(1):47–62, Jan. 1995. "Preliminary version appeared in Proc. CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (LNCS 787), pages 276–290, April 1994.".

[15] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. Wiley Professional Computing. Wiley, Chichester, 1994.

[16] PLT. *MzScheme: Language Reference Manual*. Rice University, 2000. Version 103.

[17] PLT. *PLT MrSpidey: Static Debugger Manual*. Rice University, 2000. Version 103.