INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
RICE UNIVERSITY

Selector-based versus Conditional-constraint-based Value-flow Analysis of Programs

by

Philippe Meunier

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Masters of Science

APPROVED, THESIS COMMITTEE:

Matthias Felleisen
Professor of Computer Science, Chair

Keith Cooper,
Professor of Computer Science

John Mellor-Crummey
Senior Faculty Fellow

Paul Steckler
Senior Research Scientist, Northeastern University

Houston, Texas
February, 2002
Selector-based versus Conditional-constraint-based Value-flow Analysis of Programs

Philippe Meunier

Abstract

MrSpidey, a program debugger for PLT Scheme, infers the flow of values in a program. It uses Flanagan's selector-based analysis framework. Unfortunately, due to limitations of that framework, the debugger often flags potential errors where none exists. In particular, it is too conservative when analyzing $n$-ary functions, functions with rest arguments, and arity-overloaded functions (case-lambda). Flanagan's analysis can be extended to give more precise results, but at the cost of a high running time. We therefore conclude that this framework is not well suited to analyzing functions in real-world programming languages.

To overcome the limitations of Flanagan's framework, we develop an alternative based on Palsberg and Schwartzbach's conditional constraint rules. After scaling the analysis to the full R5RS Scheme language (adding primitives using types, multiples values, imperativeness, and generative structures), experiments show that it infers value sets as precisely as the extended selector-based analysis and runs significantly faster.
Acknowledgements

I could not have earned a Masters degree in Computer Science without the support from many people.

Matthias Felleisen, the PLT master slave driver, provided ideas, direction and kept me focused on actually getting the job done. The mere existence of this thesis is proof of his dedication to educating lazy students.

Paul Steckler has been my testing ground for wacky ideas for many many months. Much of the work presented in here was done in collaboration with him. His research experience has certainly been invaluable in designing an analysis that works.

Robby Findler has patiently endured my never ending list of “I still don’t understand…” questions about DrScheme’s tool interface and MzScheme’s syntax system. It would have been impossible for me to extend the analysis beyond just handling a toy language without his help.

Mitch Wand suggested to look at the conditional-constraint-based analysis when annotated selectors did not deliver the expected performance. His suggestion proved to be a very good one.

Finally, Jamie, John, and Paul Graunke provided the friendly working environment that made all this possible.

Many thanks to all of you.
9.1.2 Type Rules .................................................. 38
9.1.3 Extension and Interface with the Term Analysis .......... 45
9.2 Multiple Values .................................................. 46
9.3 Other Language Constructs ................................... 47
9.4 Putting it all Together .......................................... 49

10 Related and future work ........................................ 53

11 Conclusions ....................................................... 56

Bibliography .......................................................... 57
Illustrations

2.1 MrSpidey mishandles case-lambda. ............................. 6
2.2 Spurious arity check. ........................................... 7

3.1 MrSpidey constraint derivation. ............................... 10
3.2 MrSpidey constraint propagation. .............................. 12

4.1 Revised constraint derivation rules. .......................... 16
4.2 Revised propagation rules. ...................................... 17

5.1 Additional constraints for rest parameters. .................. 19

7.1 Closure analysis style SBA. ................................... 28

8.1 Analysis times, plotted log-log. ............................... 33

9.1 Type to constraints transformation rules. ...................... 39
9.1 Type to constraints transformation rules (continued). ........ 40
9.2 Actual to formal argument matching for functions applied inside
primitives. .......................................................... 41
9.3 Example analysis of apply3. ................................. 44
9.4 Example analysis of apply3+. ................................. 44
9.5 Analysis times for extended analysis, plotted log-log. ....... 52
Chapter 1

Introduction

Static analysis of programs encompasses a whole range of techniques used during many stages of the software development process, from compiler optimization, to debugging, to security analysis, to soft-typing. One important technique is value-flow analysis. It computes the set of values to which an expression might evaluate at run-time and determines where these values originate.

MrSpidey is a static value-flow analyzer for PLT Scheme [20]. It is based on the formalism developed by Heintze [12] and Flanagan [10, 11] and analyzes nearly all of PLT Scheme. Unfortunately, due to limitations of the formalism, the implementation sometimes propagates values in an overly conservative manner. As a consequence, MrSpidey often flags potential errors where none exists.

One of the most obvious symptoms of MrSpidey conservatism is when analyzing functions. Flanagan's analysis framework handles only functions of one argument. Scheme, on the other hand, provides a variety of function forms, including quite complex ones such as \texttt{case-lambda} with rest arguments.

A \texttt{case-lambda} expression is like a \texttt{lambda} expression, except that there may be several clauses in a \texttt{case-lambda} procedure, each with its own parameter list and body. When such a procedure is applied, one of the clauses is selected based on the number of actual arguments, and the corresponding body is executed. If none of the clause argument lists matches the number of actual arguments, Scheme signals an arity error. It is essentially the same as the \texttt{lambda*} construct introduced by Dybvig and Hieb [6].

The purpose of \texttt{case-lambda} is to provide overloading by arity. Using it greatly
simplifies the definition of functions that have to execute different code depending on the number of arguments they were passed. In PLT Scheme it is also used to implement opt-lambda, which allows the programmer to specify default values for function arguments, à la C++. Because of its usefulness, case-lambda will replace lambda as the core functional form used in PLT Scheme.

In both PLT Scheme and Chez Scheme, a case-lambda clause parameter list may have a rest parameter, just as for lambda in Scheme [15], which allows the clause to receive any number of extra actual arguments, packed in a list.* Rest arguments are similar to C's varargs feature [16] or Python's * identifier function argument syntax [22].

In this thesis, we describe our attempts to deal with case-lambda in static program analyses, and how these attempts led us to abandon Flanagan's theoretical framework and MrSpidey. Our first analysis is an extension of Flanagan's formalism for MrSpidey, to handle case-lambda and rest parameters. Flanagan's set-based analysis (SBA) is based on selectors (car, cdr, dom, and rng), that are used to select components of compound values (pairs and functions) flowing through expressions. The matching of selectors from procedure definitions and application sites controls the flow of values into and out of procedures.

MrSpidey can already analyze case-lambda, though Flanagan does not provide a formal treatment of that part of its analysis. Since the underlying framework only handles simple functions of one argument, the implementation conservatively propagates values through all clauses of case-lambda and even propagates them in the presence of arity errors. As a result, MrSpidey also shows values flowing through formal parameters, including rest parameters, in unused clauses.

In our modification of Flanagan's framework, we annotate selectors with arity and argument-position information, which assures that values flow into and out of

*Dybvig and Hieb described a notion of "rest variable", which is not quite like a Common Lisp or Scheme rest parameter.
appropriate case-lambda clauses, and also allow us to analyze rest arguments. As we shall show, a framework based on annotated selectors improves upon the existing MrSpidey framework, giving far more precise results.

Although our approach yields good results, its time cost is too great. Our insight is that the selector-based analysis imposes two time burdens. First, the analysis framework requires that 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 that are matched according to argument position and arity annotations. Obtaining these pairs requires finding candidate selectors and checking the annotations for each candidate.

We then consider a closure-analysis version of SBA (CA-SBA). It uses Palsberg and Schwartzbach's conditional constraints [18] instead of selectors and thus does not exhibit the problems of the Flanagan framework. The analysis based on this new framework can provide similar results as the annotated-selectors one and compute them with a lower running time. We also prove that this new framework can be extended to handle the entire R5RS Scheme language [15], plus generative structure definitions, without effects on the running time.

With those insights, we conclude that the selector-based approach to SBA is not suitable for analysis of a Scheme implementation with interesting extensions. A new static debugger for PLT Scheme should use a CA-SBA instead of the annotated-selector approach. We plan to integrate the new static debugger, called MrFlow, in the upcoming version 200 release of DrScheme [8].

The thesis is organized as follows. We begin in Chapter 2 by presenting limitations of the MrSpidey framework. In Chapter 3, we review Flanagan's account of set-based analysis with selectors. Next, in Chapter 4, we describe our extension to Flanagan's system for case-lambda programs without rest arguments. In Chapter 5, we add rest arguments to the analysis. In Chapter 6, we present and discuss the complexity of the resulting analysis. In Chapter 7, we present a closure analysis style SBA
that handles case-lambda and rest parameters. In Chapter 8, we give empirical results, comparing MrSpidey, our modified selector analysis, and a closure analysis style SBA. In Chapter 9, we extend the closure style analysis framework to handle more language constructs and show that the running time of the extended analysis is still good. Chapter 10 presents related and future work. Finally, in Chapter 11, we offer conclusions.
Chapter 2

Limitations of MrSpidey’s Framework

MrSpidey’s theoretical framework handles only functions of one argument (see Chapter 3). Since MrSpidey analyzes a realistic language in which functions can have multiple arguments, rest arguments, and multiple clauses, MrSpidey has to conservatively approximate these features to make them fit its analysis framework. In this chapter, we catalog the ways in which these conservative approximations makes MrSpidey less useful than it could be as a static debugger.

In Figure 2.1, we show the results of MrSpidey’s analysis for four programs. The boxes contain type information about the left adjacent expressions. A type may be a definite constant, such as a particular number, or a set, such as "str", indicating the set of all strings. The results of the analysis can be explained as follows:

- Propagation despite arity errors. MrSpidey’s analysis framework does not address the issue of functions with multiple arguments. MrSpidey therefore simulates multiple arguments using lists and lists selectors. As a consequence, when a procedure is applied to an incorrect number of arguments, MrSpidey conservatively propagates data through as many formal arguments as possible. Figure 2.1-(A) shows MrSpidey’s analysis of a procedure of two arguments applied to one argument. At run-time, the value of the actual argument never reaches the bound occurrence of x, though MrSpidey suggests otherwise. Despite the conservative propagation, MrSpidey still correctly detects the arity error and underlines the lambda.

*A lambda expression is treated as a case-lambda expression with a single clause.
Figure 2.1: MrSpidey mishandles case-lambda.

- Propagation through multiple clauses. Likewise, since MrSpidey's framework does not handle case-lambda, the implementation conservatively propagates values of actual arguments through all clauses of a case-lambda. Figure 2.1-(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.

- *Propagation through unreachable clauses.* For the same reason, MrSpidey also conservatively 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 receives data at run-time. Figure 2.1-(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 values flow 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 2.1-(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.

![Figure 2.2: Spurious arity check.](image)

MrSpidey's usefulness as a static debugger is compromised when values are shown to flow to locations that they cannot actually reach. Figure 2.2 shows a program for which MrSpidey claims a possible arity error, though there is none. Because its analysis framework does not handle multiple clauses, MrSpidey conservatively reasons
that the $\texttt{lambda}$ may flow to the formal parameter $f$ in the second clause in the case-
$\texttt{lambda}$, where it could be misapplied. From the program text, it is clear that the
$\texttt{lambda}$ can flow only through the first clause.

The goal of our research is to create an analysis that correctly deals with real-world parameter passing mechanisms such as case-$\texttt{lambda}$ and rest arguments. To understand the problems MrSpidey faces when analyzing these features, we must first understand the theoretical framework underpinning MrSpidey's selector-based analysis.
Chapter 3

MrSpidey Theory

We now review Flanagan’s theoretical framework for analyzing programs. We first present the programming language to be analyzed, then present a language of constraints, and finally explain how to analyze a program using such constraints.

The programming language we are considering in this chapter is the lambda calculus with constants and basic list operations, defined by the following grammar:

\[ M ::= x \mid c \mid (\lambda x. M)^t \mid (M M) \mid (\text{cons } M \ M) \mid (\text{car } M) \mid (\text{cdr } M) \]

The metavariable \( x \) represents variables, and the metavariable \( c \) represents basic constants. In this language, lambda abstractions are labeled for the analysis’s sake. The \text{cons}, \text{car}, and \text{cdr} list operations are special forms (not primitives).

Set expressions are defined by the grammar

\[ \tau ::= \alpha \mid c \mid \text{pair} \mid \text{dom}(\alpha) \mid \text{rng}(\alpha) \mid \text{car}(\alpha) \mid \text{cdr}(\alpha) \]

where \( \alpha \) is a set variable. A fresh set variable is associated with each term, and is going to represent the set of values that the term might evaluate to at run-time. We may also write \( \alpha, \beta, \) and \( \gamma \) for set variables. The metavariable \( c \) represents constants, including term language constants and lambda labels. The token \text{pair} is used to represent the flow of lists. The forms \text{dom}, \text{rng}, \text{car} and \text{cdr} are selectors, to select the components of complex values flowing through expressions. The selectors \text{car} and \text{cdr} are used to select the first and second components of pairs, and the selectors \text{dom} and \text{rng} are used to select the domain and range of functions. Of these, only \text{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 τ ≤ τ'. Constraints indicate the flow of values. For example, the constraint c ≤ α means that the constant c flows into the expression labeled with the set variable α.

Flanagan’s set-based analysis consists of two phases. The first phase is constraint derivation. It is 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 derivation rules. Next, a propagation phase combines constraints using constraint propagation rules to generate new constraints, effectively mimicking the flow of values through a program. Then, a set of values is computed for each program point. From such a set, a type can be constructed.

\[
\Gamma[x \mapsto \beta] \vdash x : \alpha, \{\beta \leq \alpha\} \quad \text{(VAR)}
\]

\[
\Gamma \vdash c : \alpha, \{c \leq \alpha\} \quad \text{(CONST)}
\]

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

\[
\text{where } C = \begin{cases} 
\ell \leq \alpha \\
\text{dom}(\alpha) \leq \gamma \\
\beta \leq \text{rng}(\alpha)
\end{cases}
\]

\[
\text{(LAMBDA)}
\]

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

\[
\text{where } C = \begin{cases} 
\beta_2 \leq \text{dom}(\beta_1) \\
\text{rng}(\beta_1) \leq \alpha
\end{cases}
\]

\[
\text{(APP)}
\]

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

\[
\text{where } C = \begin{cases} 
\text{pair} \leq \alpha \\
\beta_1 \leq \text{car}(\alpha) \\
\beta_2 \leq \text{cdr}(\alpha)
\end{cases}
\]

\[
\text{(CONS)}
\]

\[
\frac{\Gamma \vdash M : \beta, C}{\Gamma \vdash (\text{car } M) : \alpha, C \cup \{\text{car}(\beta) \leq \alpha\}}
\]

\[
\text{(CAR)}
\]

\[
\frac{\Gamma \vdash M : \beta, C}{\Gamma \vdash (\text{cdr } M) : \alpha, C \cup \{\text{cdr}(\beta) \leq \alpha\}}
\]

\[
\text{(CDR)}
\]

Figure 3.1: MrSpidey constraint derivation.
Figure 3.1 shows the constraint derivation rules for our programming language. The judgments in Figure 3.1 are of the form

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

where

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

An environment like $\Gamma[x \mapsto \alpha]$ represents the extended environment $\Gamma$ that maps the program variable name $x$ to the set variable $\alpha$ and all other variables $y$ to $\Gamma(y)$. A rule like

$$\Gamma' \vdash N : \beta, C' \quad \frac{\Gamma \vdash M : \alpha, C \cup C'}{\Gamma \vdash M : \alpha, C \cup C'}$$

means that, to analyze the term $M$ in the environment $\Gamma$, we must first analyze the term $N$ in the environment $\Gamma'$. The result of analyzing $N$ is the set variable $\beta$ representing $N$ and the set of constraints $C'$. The result of analyzing $M$ is then the fresh set variable $\alpha$ and the set of constraints $C \cup C'$, with $C$ presumably relating $\alpha$ to $\beta$.

Let us provide intuition for some of the rules. The VAR rule says that values flow 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;
- $\text{dom}(\alpha) \leq \gamma$: whatever value flows into the domain of the procedure with label $\alpha$ flows into its formal parameter labeled $\gamma$, and
\[
\begin{align*}
\text{const?}(\tau) \lor \\
\text{label?}(\tau) \lor \\
\tau \leq \alpha \quad \alpha \leq \beta & \quad \text{token?}(\tau) \\
\alpha \leq \sigma(\gamma) \quad \sigma(\gamma) \leq \beta & \quad \text{TRANS-CONST} \\
\alpha \leq \sigma(\gamma) & \quad \text{TRANS-SEL} \\
\alpha \leq \sigma(\beta) \quad \text{selector}^{+?}(\sigma) \quad \beta \leq \gamma & \quad \text{COVARIANT-PROP} \\
\sigma(\alpha) \leq \beta \quad \text{selector}^{-?}(\sigma) \quad \alpha \leq \gamma & \quad \text{CONTRAVARIANT-PROP}
\end{align*}
\]

Figure 3.2: MrSpidey constraint propagation.

- \( \beta \leq \text{rng}(\alpha) \): the result of the procedure body flows into the range of the procedure.

There are similar explanations for the other constraints in Figure 3.1.

Figure 3.2 shows the constraint propagation rules used in the second phase of the analysis. In the \text{trans-const} rule, we use the predicates \text{const?}, \text{label?}, and \text{token?} to detect constants, procedure labels, and tokens. The difference between covariant and contravariant selectors shows up in the propagation rules \text{covariant-prop} and \text{contravariant-prop}. The \text{selector}^{+} predicate holds when its argument is \text{rng}, \text{car}, or \text{cdr}; the \text{selector}^{-} predicate holds only for \text{dom}. These propagation rules follow Flanagan's presentation [10], with some simplification and notational changes. These rules are repeatedly applied until no new constraints are added. Since no new set variable is created during propagation, this phase always terminates.

The full details of MrSpidey's constraint solution and type reconstruction algorithms are beyond the scope of this thesis, but we attempt here to convey their essence. See Flanagan's dissertation [10] for details. 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 $\text{pair} \leq \alpha$, then the term may evaluate to a pair, and $\{\beta \mid \beta \leq \text{car}(\alpha)\}$ is the set of set variables that may flow into the $\text{car}$ of such a pair. The sets of values for these set variables provide the actual values. We compute the solutions to $\text{cdr}$'s and procedure ranges in similar fashion. Procedure domains require a slightly more complex calculation due to the contravariance of the $\text{dom}$ 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 $\text{pair} \leq \alpha_M$, $\beta_1 \leq \text{car}(\alpha_M)$, $57 \leq \beta_1$, $\beta_2 \leq \text{cdr}(\alpha_M)$, and $\text{null} \leq \beta_2$. Then MrSpidey predicts that $M$ will produce at runtime a value with type $(\text{cons} \ 57 \ \text{null})$. Some extra care must be taken to properly reconstruct recursive types.

What is missing from the existing formalism? In the lambda calculus, all procedures have one clause with one parameter, and there are no rest parameters. Flanagan's dissertation [10, Appendix E.3] indicates that a procedure of more than one argument is analyzed by considering it as a procedure of one argument, where the argument becomes bound to a list of actual arguments at its application sites. The values in the list are then distributed to the formal arguments by selecting elements of the list. Because all clauses of a $\text{case-lambda}$ 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 $\text{case-lambda}$ clauses are merged into application results. The $\text{TRANS-SEL}$ rule in Figure 3.2 controls the propagation of data into formal parameters (when the selector involved is $\text{dom}$) and out of procedures (when the selector is $\text{rng}$). Hence, to improve the analysis, we need to focus on the propagation mechanism embodied in that rule.
Chapter 4

Analyzing case-lambda in the Selector Framework

In this chapter, we show how to extend Flanagan's framework to analyze programs containing case-lambda but without rest arguments. We will show how to add rest parameters in Chapter 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 first to define intervals.

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 \(I\) 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 additional 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

- dom$^a_i$, where $a$ is an arity and $i$ is an argument index in a clause parameter list, and

- dom$^T_{i,n}$, where $T$ 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

- rng$^a$, where $a$ is an arity, and

- rng$^T_n$, where $T$ is an interval, and $n$ is the total number of arguments at an application site.

Consider dom$^a_2(\alpha)$; this set expression represents the flow into the second argument of a case-lambda clause with arity $a$. Similarly, dom$^{[3,3]}_{1,3}(\alpha)$ represents the flow into the first argument of a procedure of three arguments that flows into an application site with three actual arguments. The selector rng$^a(\alpha)$ represents the flow out of a case-lambda clause with arity $a$. The set expression rng$^{[3,3]}_3(\alpha)$ represents the value returned by a procedure of three arguments at an application site with three actual arguments.

Figure 4.1 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 $a_i = ([n_i, n_i], (I_{i-1}, \ldots, I_1))$, where $I_j$ 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 chapter.
\[
\frac{\Gamma[x_{i,j} \mapsto \alpha_{i,j}] \vdash M_i : \beta_i, C_i \ i \in [1..m], j \in [1..n_i]}{\Gamma \vdash \text{(case-\lambda)}}
\]

\[
((x_{1,1} \ldots x_{1,n_1}) M_1) \ldots ((x_{m,1} \ldots x_{m,n_m}) M_m) \hat{\ell} : \alpha, \bigcup_{i \in [1..m]} C_i \cup C
\]

where \(C = \left\{ \begin{array}{l}
\ell \leq \alpha \\
\operatorname{dom}_i^\alpha(\alpha) \leq \alpha_{i,j} \\
\beta_i \leq \operatorname{rng}_i^\alpha(\alpha)
\end{array} \right\}
\quad i \in [1..m], \ j \in [1..n_i]
\]

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

where \(C = \left\{ \begin{array}{l}
\beta_i \leq \operatorname{dom}_i^{[n,n]}(\beta_0) \ i \in [1..n] \\
\operatorname{rng}_i^{[n,n]}(\beta_0) \leq \alpha
\end{array} \right\}\)

Figure 4.1: Revised constraint derivation rules.

Note that these rules have the essential form of those in Figure 3.1, except that

- each clause of a case-\texttt{lambda} generates constraints,
- each argument of an application generates constraints, and
- the selectors are annotated.

The selector annotations are used in the revised propagation rules in Figure 4.2, in particular, in the rules\texttt{TRANS\text{-}DOM} and\texttt{TRANS\text{-}RNG}. The unchanged rules\texttt{TRANS\text{-}CONST}, \texttt{TRANS\text{-}SEL}, and\texttt{COVARIANT\text{-}PROP} are omitted. The\texttt{COVARIANT\text{-}PROP} rule is only used to propagate the \texttt{car} and \texttt{cdr} selectors. We no longer need the general \texttt{CONTRAVARIANT\text{-}PROP} rule since its role is taken over by the more specific \texttt{DOM\text{-}PROP} rule. The rule\texttt{TRANS\text{-}SEL} no longer handles \texttt{dom} and \texttt{rng} selectors. The core idea is to propagate values through a case-\texttt{lambda} clause only when the number of
\[
\begin{align*}
\frac{\alpha \leq \text{dom}_{i,s}^{[n,m]}(\beta) \quad \text{dom}^a(\beta) \leq \gamma \quad (s, [n, m]) \models a}{\alpha \leq \gamma} & \quad \text{(TRANS-DOM)} \\
\frac{\alpha \leq \text{rng}^a(\beta) \quad \text{rng}_s^{[n,m]}(\beta) \leq \gamma \quad (s, [n, m]) \models a}{\alpha \leq \gamma} & \quad \text{(TRANS-RNG)} \\
\frac{\alpha \leq \text{rng}^a(\beta) \quad \beta \leq \gamma}{\alpha \leq \text{rng}^a(\gamma)} & \quad \text{(RNG-PROP)} \\
\frac{\text{dom}^a(\alpha) \leq \beta \quad \alpha \leq \gamma}{\text{dom}^a(\gamma) \leq \beta} & \quad \text{(DOM-PROP)}
\end{align*}
\]

Figure 4.2: Revised propagation rules.

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

- \(s \in [p, q],\)

- \([n, m] \models [p, q], \text{ and}\)

- \(\forall i \in [1..t], \ s \not\in \mathcal{I}_i\)

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.
Chapter 5

Analysis of Rest Parameters in the Selector Framework

The introduction of rest parameters requires additional constraints, to account for the uncertainties associated with such parameters. With a rest parameter, a clause takes some number of required arguments, but may accept 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 introduction of rest parameters, we continue to generate all the constraints as described in the last chapter. We 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 5.1, 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 3.1; the CASE-LAMBDA rule from Figure 4.2, and the APP rules in Figures 4.2 and 5.1.

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 \( \text{rng}_{n}^{i}[\omega](\beta_{0}) \leq \alpha \) 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:

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

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

\[
\begin{align*}
\text{null} & \leq \alpha_{n+1} \\
\alpha_{n+1} & \leq \text{dom}_{n+1,n}^{[n,\omega]}(\beta_0)
\end{align*}
\]

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

\[
\begin{align*}
\beta_1 & \leq \text{car}(\alpha_1) \\
\alpha_2 & \leq \text{cdr}(\alpha_1) \\
& \vdots \\
\beta_n & \leq \text{car}(\alpha_n) \\
\alpha_{n+1} & \leq \text{cdr}(\alpha_n)
\end{align*}
\]

The purpose of these constraints is to model the flow of lists of varying lengths into the fresh set variables \( \alpha_i \). 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

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

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 Chapter 3, it appeared in the CONS rule. Here, we generate the constraints

\[ \text{pair} \leq \alpha_i \]
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 2.1 and 2.2, the prototype remedies the problem identified with MrSpidey. For the program in Figure 2.1-(A), there is no flow through the bound $x$. For the program in Figure 2.1-(B), there is flow only through the bound $x$, and not through the bound $y$. For the program in Figure 2.1-(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 2.1-(D), only the return value from the first clause shows up in the flow for the application. In the prototype, the program in Figure 2.2 does not signal an arity error. Despite these improvements, we argue in the next chapter, the modified analysis is unsatisfactory.
Chapter 6

Performance

It is not enough for our analysis to be sound – it must also perform well, since we expect it to be used in an interactive tool. 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” [14]. Unfortunately, as best as we can tell, our modified version of Flanagan’s SBA can exceed 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 [5, Section 26.2]. Deriving the constraints in Figure 3.1 takes time linearly bounded by the size of a program. For the propagation phase, the rules \texttt{trans-const} and \texttt{trans-se1} in Figure 3.2 are ordinary transitive rules. Hence, closing under these two rules does have a cubic-time upper bound. The other two rules in Figure 3.2 are of a different character, so the actual complexity might be higher. As we shall show, the complexity is, in fact, cubic.

We describe an algorithm that can be used to close the constraints under the propagation rules in Figure 3.2. The algorithm has a cubic-time upper bound. As each constraint is generated, it checks 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 if done using a hash table, add the constraint to the pool. 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 4.1 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 4.1). 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.

In fact, 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 3.1 and 4.1, we start with a linear number of such constraints. The only rules that can create new such constraints are `COVARIANT-PROP` in Figure 3.2 for the `car` and `cdr` selectors and `RNG-PROP` and `DOM-PROP` in Figure 4.2. 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 4.2, 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, $\alpha \leq \text{dom}^{[n,m]}_{i,s}(\beta)$. 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 $\text{dom}^{a}_i(\beta) \leq \gamma$. On the other hand, suppose we have a constraint matching the second premise. From the CASE-LAMBDA rule in Figure 4.1, 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 5.1), the number of constraints produced by the derivation phase becomes quadratic in the size of the pro-
gram. Nonetheless, the total number of constraints after the propagation phase still has a quadratic upper bound. The constraints involving \texttt{dom} and \texttt{rng} introduced in Figure 5.1 do not propagate those selectors to new constraints, since the selectors do not appear on the proper side of the constraints to trigger any propagation rule. For the other constraints in Figure 5.1, 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 $O(n^5)$.

A time bound of $O(n^5)$ is undeniably high, especially since experience with Mr-Spidey shows that large programs might actually require $O(n^3)$ time. By using a different analysis, described in the following chapter, we can compute essentially the same information asymptotically faster.
Chapter 7

Eliminating Selectors

In Flanagan's original SBA and in our revision of his analysis, \texttt{dom} and \texttt{rng} selectors are used to hook up actual arguments with formal parameters, and procedure bodies with applications. Every value flows through a selector at these critical points. Fortunately, 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, based on the one of Palsberg and Schwartzbach [18], can handle \texttt{case-lambda} and rest parameters. Figure 7.1 presents the constraints for such an analysis, for the lambda calculus with \texttt{case-lambda} and rest arguments. As before, procedures are labeled; we now label all other subterms. Each label has an associated set $\varphi(\ell)$ of labels and compound labels, that represent the values that flow into or originate at the term labeled with $\ell$. Compound labels represent the flow of complex values like pairs and functions. We write them as

$$(\text{cons } \ell_1 \ \ell_2) \text{ and } (\text{case-\lambda } ((\ell_1 \ldots) \ \ell_1) \ldots).$$

Constraints are of the three forms

\[
\begin{align*}
C^m & := \ell' \in \varphi(\ell) \\
C^i & := \varphi(\ell') \subseteq \varphi(\ell) \\
C^c & := \text{if } C^m \text{ then } C_1^i, \ldots, C_n^i
\end{align*}
\]

The judgments in Figure 7.1 are of the form

$$\Gamma \vdash M^\ell : C$$

where

- $\Gamma$ is an environment from term variables to labels,
\[
\Gamma (x) = \ell \\
\frac{\Gamma \vdash x^\ell : \{\varphi (\ell') \subseteq \varphi (\ell)\}}{(\text{VAR})}
\]
\[
\Gamma \vdash c^\ell : \{\ell \in \varphi (\ell)\}
\]

\[
\Gamma \cup \{x_{ij} : \ell_{ij} \mid 1 \leq j \leq r_i\} \vdash E_i^{\ell_i} : C_i^f \quad 1 \leq i \leq n
\]

\[
\Gamma \vdash \text{case}-\lambda \left( [\{x_{i1}^{\ell_{11}} \ldots x_{ir_i}^{\ell_{ir_i}}\} E_1^{\ell_1}] \ldots [\{x_{n1}^{\ell_{n1}} \ldots x_{nr_n}^{\ell_{nr_n}}\} E_n^{\ell_n}] \right)^\ell : \\
\{\text{case}-\lambda \left( ((\ell_{11} \ldots \ell_{r_1}) \ell_1) \ldots ((\ell_{n1} \ldots \ell_{nr_n}) \ell_n) \right) \in \varphi (\ell)\} \cup_{1 \leq i \leq n} C_i^f
\]

\[
\Gamma \vdash E_i^{\ell_i} : C_i^f \quad 0 \leq i \leq n
\]

\[
\Gamma \vdash (E_0^{\ell_0} E_1^{\ell_1} \ldots E_n^{\ell_n})^{\ell} : \\
\begin{cases}
\text{if (case}-\lambda \left( ((\ell_{11} \ldots \ell_{r_1}) \ell_1) \ldots ((\ell_{n1} \ldots \ell_{nr_n}) \ell_n) \right) \in \varphi (\ell_0) \\
\text{then if } \exists k, r \in \mathcal{N}, q \in \mathcal{N} \cup \{\omega\} : n \in \text{arity}((\ell_{k1}^{\ell'} \ldots \ell_{kr_k}^{\ell'})) = [r, q] \\
\text{and } \forall j \in [1, k-1] : n \not\in \text{arity}((\ell_{j1}^{\ell'} \ldots \ell_{jr_j}^{\ell'})) \\
\text{then } \forall j \in [1, r] : \varphi (\ell_j) \subseteq \varphi (\ell_{kj}^{'}) \\
\varphi (\ell_{kj}^{'}) \subseteq \varphi (\ell) \\
\text{if } q = \omega \text{ then } \\
\{\text{cons } \ell_{r+1}(\text{cons } \ldots (\text{cons } \ell_n \ell_{\text{null}}) \ldots)\} \subseteq \varphi (\ell_{kr+1}^{'})
\end{cases}
\]

\[
\text{else } \text{arity-error}
\]

\[
\bigcup_{0 \leq i \leq n} C_i^f
\]

\[
\Gamma \vdash E_1^{\ell_1} : C_1^f \quad \Gamma \vdash E_2^{\ell_2} : C_2^f
\]

\[
\Gamma \vdash (\text{cons } E_1^{\ell_1} E_2^{\ell_2})^{\ell} : \{\text{cons } \ell_1 \ell_2 \in \varphi (\ell)\} \cup C_1^f \cup C_2^f
\]

\[
\Gamma \vdash E^\ell : C^f
\]

\[
\Gamma \vdash (\text{car } E^\ell)^{\ell} : \\
\begin{cases}
\text{if } \{\text{cons } \ell_1 \ell_2 \in \varphi (\ell')\} \\
\text{then } \varphi (\ell_1) \subseteq \varphi (\ell)
\end{cases} \cup C^f
\]

\[
\Gamma \vdash E^\ell : C^f
\]

\[
\Gamma \vdash (\text{cdr } E^\ell)^{\ell} : \\
\begin{cases}
\text{if } \{\text{cons } \ell_1 \ell_2 \in \varphi (\ell')\} \\
\text{then } \varphi (\ell_2) \subseteq \varphi (\ell)
\end{cases} \cup C^f
\]

---

Figure 7.1 : Closure analysis style SBA.

- $M^{\ell}$ is a labeled term, and
- $C^f$ is a set of constraints.

The rules can be simply explained as follows. The VAR rule, for example, says that labels flowing into a variable binder also flow into the variable's references. The
CASE-LAMBDA rule creates a case-\(\lambda\) compound label, containing the labels for the case-lambda's arguments and bodies, that simulates the flow of the case-lambda function. The APP rule uses a conditional constraint, where a finite number of new constraints might be added, as labels and compound labels flow from set to set. The conditional constraint means that, if a case-lambda function flows into the operator term of the application, and if the case-lambda has a clause with the right arity, then

- the actual arguments of the application flow into the formal arguments of the selected clause of the case-lambda,

- the values produced by the body of that clause flow out to be the result of the application.

If the selected clause has a rest argument (the "if \(q = \omega\)" test in the APP rule), a list of the remaining actual arguments is created, to flow into the rest argument. Note that the conditional constraint used in APP rule has a syntax slightly more complicated than the general syntax we gave for conditional constraints. This is because we have to find a clause with the right arity before creating the flows.

Because this form of SBA uses conditional constraints, flows between actual and formal arguments, and between clause bodies and applications are established without having to go through a search for the dom and rng selectors corresponding to the function being applied, and without having to find among these selectors the ones that match the number of actual arguments. Also, procedure flow is modeled by the flow of just one compound label, 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.

The constraints in Figure 7.1 are similar to those usually presented for closure analysis of the lambda calculus [18], with, of course, case-lambda instead of lambda. With this change, the form of the 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 therefore be solved in cubic time [18].

Reconstructing friendly types from the flow information is also simpler in the case of the conditional-constraint based analysis than it is in the selector style SBA case. For a given term $M$ labeled with $\ell$, $\varphi(\ell)$ gives the set of labels and compound labels representing the values that $M$ might have at run-time. Labels in the set directly correspond to constants, the type of which can easily be computed from their value, while compound labels, like $(\text{cons } \ell_1 \ell_2)$ or $(\text{case-}\lambda ((\ell_{i1} \ldots \ell_{im}) \ell_1) \ldots ((\ell_{n1} \ldots \ell_{nm}) \ell_n))$, represent compound values, the type of which can be recursively reconstructed in the same manner. As in the case of the selector-based SBA, some care still has to be taken to reconstruct recursive types correctly.
Chapter 8

Empirical Results

While the worst-case bounds mentioned in Chapter 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 stress tests. For the CA-SBA prototype, we used a variation on the graph-based implementation technique for constraint solving described by Palsberg and Schwartzbach [18]. MrSpidey runs as an add-on tool in DrScheme [9]. 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 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.

We have not been able to show that the quintic time bound given in Chapter 6 for the annotated-selector analysis is a tight bound. 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.

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

Table 8.1: Procedure chain tests.
Consider the results in Table 8.1. 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-selector 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 Table 8.2. 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)
```
Figure 8.1: Analysis times, plotted log-log.

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 running times and the number of nodes in the abstract syntax tree. In Table 8.2, the last column gives the apparent polynomial exponent for the asymptotic complexity, considering the two largest tests. We calculate the exponent with

$$\frac{\log t_2 - \log t_1}{\log n_2 - \log n_1}$$

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 8.1, which shows a log/log graph of the running times for each implementation against the number of nodes in the abstract syntax tree.

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, the analysis not only propagates its label, but it propagates 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 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.
Chapter 9

Extending the Analysis

While the results from the previous chapter prove that the closure analysis style SBA has the potential to perform much better than the selector based ones, we also have to show that the CA-SBA can be extended to handle a more practical language and still have a good performance. The following sections explain how to add primitives in a systematic way, how to handle other language constructs and mutation, and how to analyze expressions that return multiple values. Finally, we look at the performance of such an extended analysis.

9.1 Primitives

Our analysis so far has handled primitives like \texttt{cons}, \texttt{car}, and \texttt{cdr} in an ad-hoc fashion, by treating each of them as a special form. Since we ultimately want to extend our analysis to the whole of the Scheme language, we need to devise a method to analyze primitives that is general enough to cover most of the primitives we might want to add, but simple enough that adding new primitives does not involve deep modifications of the code of the analyzer.

9.1.1 Type Language

One simple way to describe the flow behavior of primitives is to specify a type for them. We can then define rules to translate a given type into a set of constraints.

A simple type language can be as follows:
• the set \( T^C \) of concrete types is:

\[
T^C := \begin{array}{l}
\text{base-type} \\
T \\
\bot \\
\alpha \\
(\text{cons } T^C T^C) \\
(\text{rec-type } ((\alpha T^C)\ldots) T^C) \\
(\text{case-\(\lambda\) } (T^C\ldots \rightarrow T^C)\ldots) \\
(\text{union } T^C T^C\ldots)
\end{array}
\]

with the convention that \( \alpha \) represents type variables used to name types, and base-type represents the usual atomic types like string, integer, etc.

• the set \( T \) of types is then defined as:

\[
T := \begin{array}{l}
T^C \\
ad \\
(\text{cons } T T) \\
(\text{rec-type } ((\alpha T)\ldots) T) \\
(\text{case-\(\lambda\) } (T\ldots \rightarrow T)\ldots) \\
(\text{union } T T\ldots)
\end{array}
\]

where \( a \) is a flow variable representing an analysis-time flow.

• the set \( T^S \) of type schemes is defined as:

\[
T^S := T \\
(\text{forall } ((a T^C)\ldots) T)
\]

The purpose of flow variables is to simulate the flow of values inside primitives. For example, the unary plus is a function from number to number. A simple type for it could be \((\text{number } \rightarrow \text{number})\), but using such a type would make the analysis lose some information when applying plus to, say, an integer. The type of the result of the application would be number, instead of the more precise type integer. Instead, we give to the unary plus the type scheme \((\text{forall } ((a \text{ number}) (a \rightarrow a)))\), which means that, in our analysis, the type of the result of the application of unary plus will be whatever type plus is given as argument, as long as that type is a subtype of number.

\(^*\text{(case-\(\lambda\) (number } \rightarrow \text{number}))\) to be more precise.
Note that using a flow variable in a type implies that a function with that type does not modify the value bound at run-time to the parameter corresponding to the flow variable. This is similar to the kind of type-based theorems proved by Wadler [23].

Another example is the composition of the cons and car primitives. If cons is given the type \((T \rightarrow (\text{cons } T T)) \) and car the type \(((\text{cons } T T) \rightarrow T)\), then the expression \((\text{car } (\text{cons } 1 \text{"foo"}))\) will have the type \(T\). On the other hand, if we give to cons and car the type schemes \((\forall a. b \rightarrow (\text{cons } a b))\) and \((\forall c. \text{cons } c T \rightarrow c)\), respectively, then, for the same expression, the type integer of the first argument of cons in the expression \((\text{car } (\text{cons } 1 \text{"foo"}))\) will flow through the flow variable \(a\) inside the cons primitive, then through the flow variable \(c\) inside the car primitive to become the type of the whole expression.

Because a flow variable represents a value flow, it is unidirectional. A flow variable can therefore only be used once in contravariant position in a function type, and can be used in covariant position only if it also appears in contravariant position. Note that flow variables are different from type variables, which are only used to name types in order to create recursive types. So a type scheme like \((\forall a. (a \rightarrow a))\) does not have the same meaning as the type \((\text{rec-type } (\alpha \rightarrow \alpha))\). The former is the type of the identity function, the latter is nonsense because a type variable like \(\alpha\) defined in covariant position cannot be used in contravariant position, and vice-versa. Note also that, even though the identity function has the type specification \((\forall a. (a \rightarrow a))\), from a subtyping point of view any instance of the identity function still has a type that is a subtype of \((\bot \rightarrow T)\), not \((T \rightarrow T)\). The \(T\) type specified for \(a\) in the type scheme for the identity is only used as a filter for what is allowed to flow through the flow variable \(a\). Note finally that, while our notation for type schemes resembles the one used for bounded quantification of types, we do not do type inference and we do not do any subtyping of type schemes. Therefore undecidability results for bounded quantification [19] do not apply to our analysis.
9.1.2 Type Rules

Given the type language of the previous section, we can now define the rules to transform a type specification into a set of constraints. Since flow variables are unidirectional, we need to define two separate judgments, depending on whether we are analyzing a type in co- or contravariant position. The two judgments are of the form

$$\Delta, \Sigma \vdash^{\circ}_{T} i^{S} : \ell, C$$

where

- $\Delta$ is an environment from flow variables to labels,
- $\Sigma$ is an environment from type variables to labels,
- $\circ$ is either $+$ or $-$,
- $t^{S}$ is a type scheme,
- $\ell$ is a label, and
- $C$ is a set of set constraints.

In Figure 9.1, we use $\vdash^{+}_{T}$ for the covariant judgment and $\vdash^{-}_{T}$ for the contravariant judgment. We use $\ell^{\text{new}}$ to mean a fresh label.

The rules can be explained as follows. The FLOW SCHEME+ rule creates a new label for each flow variable in the type scheme considered and adds these labels to the $\Delta$ environment. These labels will be used later in the FLOW VAR+ and FLOW VAR- rules. The types $t^{S}_{i}$ that appear in the consequent of the FLOW SCHEME+ rule are not used here. The analyzer remembers these types as a side effect of applying the rule and uses them later in a separate post-analysis phase to check for additional type errors. A separate subtyping system similar to the one described by Amadio and Cardelli [3] is used for that purpose.
Figure 9.1: Type to constraints transformation rules.

The **CONS+** rule is similar to the **CONS** rule from Figure 7.1, while its contravariant counterpart **CONS-** is equivalent to merging the **CAR** and **CDR** rules from the same figure.
Figure 9.1: Type to constraints transformation rules (continued).

The two rec-type rules behave in a way similar to the Scheme letrec construct: the types of the clauses and the type of the rec-type body are analyzed in an environment extended with new labels for all the type variables defined by the recursive type.
Figure 9.2: Actual to formal argument matching for functions applied inside primitives.
The \texttt{CASE-LAMBDA+} rule behaves like the \texttt{CASE-LAMBDA} rule of Figure 7.1: when the type of a primitive specifies that the primitive returns a function, a compound label is created that will simulate the flow of said function.

Symmetrically, the \texttt{CASE-LAMBDA-} rule behaves like the \texttt{APP} rule of Figure 7.1. This is simply because the \texttt{CASE-LAMBDA-} rule simulates the application of a function inside a primitive such as \texttt{map}. There are two major differences between the \texttt{CASE-LAMBDA-} and the \texttt{APP} rule, though. The first difference is that, since we only have two function specifications (the one coming from the primitive type being analyzed, and the one coming from the compound label flowing into \texttt{\ell_{new}}), and no actual arguments, we have to do the arity matching for each and every clause of the analyzed type instead of just once. Hence the loop over \(i\) in the rule. The second difference is that, since we have, again, two function specifications, clauses in both specifications can now have rest arguments. This means that the "if \(q = \omega\)" part of the \texttt{APP} rule from Figure 7.1 has to be replaced by a more complicated test that takes into account all the possible combinations of normal and rest arguments, and creates the corresponding flows. The different combinations are graphically illustrated in Figure 9.2. In each case, squares represent (the set of values of labels associated with) arguments. The top row of squares represents the arguments of the function flowing into the primitive (in the form of a compound label) and whose application has to be simulated. The bottom row of squares represents the arguments of a function whose type specification comes from the primitive type. A square following a dot represents a rest argument. Rectangles represent additional labels inserted by the \texttt{CASE-LAMBDA-} rule, and arrows represent set inclusion constraints created by the analysis. Six cases are then possible.

A. The primitive expects a function of three arguments, and is given a function of three arguments: the flows are direct. This is equivalent to what happens in the \texttt{APP} rule in the simple case.

B. The primitive expects a function of six arguments, and is given a function that requires three normal arguments and has a rest argument. Three direct flows
are created for the first three arguments, and the three remaining ones are packaged into a list that flows into the rest argument of the function. This is equivalent to what happens in the \texttt{APP} rule when the "if }q = \omega" test is true.

C. The primitive expects a function that has three required arguments followed by a rest argument. The function that flows into the primitive expects six arguments. Three direct flows are therefore created, and the rest argument is connected to a series of new splitting labels. These splitting labels are in fact the same kind of labels that are created by the \texttt{CONS-} rule: they take a cons compound label and split it into a car and a \texttt{cdr}. When a list of three labels flows into the rest argument, the list is taken apart, and the different components of the list are then distributed among the three arguments of the function. If a list of less than three labels flows into the rest argument, one of the splitting label will receive the null label signaling the end of the list and flag an arity error. If a list of more than three labels flows into the rest argument, the null label added by the analysis will receive one of the cons label from the list and, again, flag an arity error.

Cases D, E, and F are similar to cases C, A, and B, respectively, but with two rest arguments instead of one.

To illustrate these cases, suppose we want to define a version of the Scheme apply primitive, called \texttt{apply3} that works with functions of exactly three arguments. This new primitive takes four arguments: a function of three arguments, and three arguments to give to the function when it is applied inside the primitive. The code for this primitive could be:

\begin{verbatim}
(define apply3 (lambda (f x y z) (f x y z))
\end{verbatim}

The type for this primitive would be:

\begin{verbatim}
(forall ((a T)(b T)(c T)(d T)) ((a b c \rightarrow d) a b c \rightarrow d)).
\end{verbatim}
The analysis of \texttt{apply3} in an expression like \texttt{(apply3 f 1 2 3)} would then use case A above to produce the kind of constraints represented in Figure 9.3, when given a
function $f$ of three arguments (dashed lines represent constraints created by other parts of the analysis). If we were to define a primitive `apply3+` to work with functions of three or more arguments, the following type could be used:

$$\text{(forall } ((a \ T)(b \ T)(c \ T)(d \ T)(e \ T)) ((a \ b \ c \ . \ e \rightarrow d) \ a \ b \ c \ . \ e \rightarrow d)).$$

If such a primitive were to be given a function $f$ of six arguments in an expression like `(apply3+ f 1 2 3 4 5 6)`, the analysis would then use case C above to produce constraints like the ones represented in figure Figure 9.4 (the additional dashed cons labels and flows are coming from the APP rule).

### 9.1.3 Extension and Interface with the Term Analysis

The type language we have presented so far in this chapter can be easily extended to include other types like vectors, multiple values, or promises. The rules to analyze these new types are similar in spirit to the ones we have seen in the previous section. Once these types have been added, and once basic types have been extended to include a notion of subtyping that takes into account the numeric tower of Scheme [15], we can remove the CONS, CAR, and CDR rules of Figure 7.1 and add the following rule instead:

$$\begin{align*}
\text{lookup-type-of-primitive(prim)} &= t^+ \\
\emptyset, \emptyset \vdash t^+ : \ell', C_f \\
\Gamma \vdash \text{prim} : \{\varphi(\ell') \subseteq \varphi(\ell)\} \cup C_f \\
\text{prim}
\end{align*}$$

We have then a system capable of handling almost all the Scheme primitives defined in R5RS [15], with two restrictions:

- some primitives like `append`, `apply`, and `map` are given quite conservative type specifications. In Scheme, the last argument of `append` might not be a list, and the last argument of `apply` has to be a list. Since we do not have any way in our type language to express the concept of "the last argument" of a function
(a rest argument just represents all the remaining arguments), the type we have
to give to these primitives has to be much more conservative than what it could
be in a slightly more expressive type language. Map has a somewhat similar
problem, in the sense that, while our type language allows for flow variables,
it does not allow for an unknown number of flow variables to be defined. But
to analyze map accurately, we would need one flow variable for each list that is
given to it. So again, we have to make do with a conservative approximation.

- primitives like set-car! that imperatively mutate their arguments cannot be
  expressed in our type language either. And unlike append, apply, and map, it
  is not possible to create in our type language a conservative approximation of
  their type. In such cases, we have to revert to ad-hoc rules in our analysis.

9.2 Multiple Values

Scheme expressions can return several values at the same time, in parallel, by using the
values primitive. In Section 9.1.3 we claimed that extending the type language with
multiple values was straightforward, and it is. We can define a new type constructor
(values T . . .) and analyze it in a manner analogous to what we already do for pairs.
But doing so transforms multiple parallel values into a single first class value (a tuple)
that can flow in and out of functions like any other value. Scheme [15], on the other
hand, only allows multiple values to flow out of expressions, not into. So simply
adding multiple values to our type language makes the analysis unsound. To restore
soundness, we have to modify all the rules of Figure 7.1 to prevent multiple values
from flowing into expressions. This means that we can no longer use set inclusion in
the constraints corresponding to in-flows. Instead, we have to use a restricted version
\( \subseteq_v \) of set inclusion that filters out multiple values and flags them as errors. Out-flows
can still using normal set inclusion.

With these modifications, the Scheme primitive call-with-values can then be
given the type scheme \( \text{forall } ((a \rightarrow \mathbb{T}) (b \leftarrow)) ((\rightarrow (\text{values } a)) ((\text{rest } a) \rightarrow b) \rightarrow b)) \), where "rest" is used to specify a rest argument. The primitive takes as first argument a thunk that returns multiple values. Since \( \text{values } a \) appears in the whole type in contravariant position, the tuple representing the multiple values is taken apart, and the different values flow into the flow variable \( a \), as a list. This list flows into the covariant type \( \text{rest } a \), which means that, according to case C of Figure 9.2, the elements of the list are distributed among the arguments of the function that is given as second argument to \text{call-with-values}. The result of that second function is then the result of the whole expression.

9.3 Other Language Constructs

It is straightforward to extend the CA-SBA analysis to handle other Scheme language constructs like \text{if}, \text{let}, \text{letrec}, and \text{begin}.

Two related language constructs that are slightly more difficult to deal with are \text{define} and \text{set}!. We want programs like:

\[
\begin{align*}
  &\text{(define } x \ 1) \\
  &\text{(define } f \ (\lambda ) \ x)) \\
  &\text{(f); } 1 \text{ should be in the result} \\
  &\text{(define } x \ 2) \\
  &\text{(f); } 2 \text{ should be in the result, even though } f \text{ is analyzed only once}
\end{align*}
\]

and

\[
\begin{align*}
  &\text{(define } x \ 1) \\
  &x \ ; \ \text{expect } 1 \\
  &\text{(define } f \ (\lambda ) \ (\lambda ) \ (\text{set! } x \ 2))) \\
  &x \ ; \ \text{expect } 1 \\
  &\text{(f)} \\
  &x \ ; \ \text{expect } 1 \\
  &\text{(lambda } () \ ((\text{f})) \\
  &x \ ; \ \text{expect } 1 \\
  &\text{((f))} \\
  &x \ ; \ \text{expect } 2
\end{align*}
\]
to be analyzed in a sound and accurate manner.

One simple way to analyze these program would be to have all possible values assigned to \( x \) flow into all possible uses of \( x \). While sound, such an analysis would be overly conservative.

As a first step toward a more precise analysis of these programs, we can, for the first program, extend all the rules of Figure 7.1 to include a top level environment and add two simple rules for define and top level lookups. For the second program, we can add another simple rule for \texttt{set!} that works for both top level and lexical variables. The top level environment (plus the lexical environment in the case of the \texttt{set!} rule) can either be imperatively updated by the rules for define and \texttt{set!}. Alternatively, all rules can pass the environment around (in and out) with the define and \texttt{set!} rules creating new environments as necessary.

We need then to be able to delay the lookup of top level references (as \( x \) in the first program) and the application of \texttt{set!} (as in the second program) when those occur inside functions. Finally, we want to force the top level lookups and \texttt{set!} applications each time, but only when, the function wrapping them is applied (but not applied inside a function that is itself not applied, as is the case in the first application \((f))\) of the second program).

One way to simulate delayed lookups and \texttt{set!} applications when inside a lambda is to again extend the \( \vdash \) judgment of Figure 7.1, to be of the form

\[
\Gamma, \Theta \vdash M^t, f, S : C, S'
\]

where \( S \) and \( S' \) are sets of labeled terms that list the top level variables and the \texttt{set!} applications for which the analysis needs to be delayed, if any, and \( f \) is a boolean flag indicating whether the term \( M \) is inside a lambda or not (\( \Theta \) here is the top level environment we described succinctly in the previous paragraph). The top level lookup rule and the rule for \texttt{set!} can then augment \( S \) with \( M \) when the flag is true, and analyze \( M \) as usual when the flag is false. This simulates the delaying part. The \texttt{CASE-LAMBDA} rule can take the result \( S' \) of analyzing the body of the function and
annotate with it the case-\( \lambda \) compound label it creates. When that compound label later flows into an application, the conditional constraint of the APP rule can then extract the set \( S' \) from the compound label, and re-analyze all the terms in the set on the fly. This creates the forcing part. Annotating case-\( \lambda \) compound labels with \( S' \) is reminiscent of what effects systems [17] do to specify side effects of expressions.

Instead of delaying and forcing, another possible way to implement a precise analysis would be to name and lift all lambdas to the top level, and being careful about using top level variables that have not been defined yet. While we experimented with this approach, it turned out to make the analysis even more complicated than the previous solution.

9.4 Putting it all Together

We have added to the prototype implementation discussed in Chapter 8 the primitive type analysis described in Figure 9.1. Primitives that mutate their arguments (\texttt{set-car!}, \texttt{vector-set!}, etc...) were added as a special case of the term analysis, in a way that still made them first class values. This turned out not to be too complex, since Scheme mutators tend to all have similar interfaces, making a fair amount of abstraction possible.

Primitives in our implementation are analyzed in a polyvariant manner (1CFA [21]) by analyzing each use of a primitive independently of any other use. User-defined functions, on the other hand, are analyzed monovariantly (0CFA), so they are analyzed only once, regardless of how many times they are used in a program. This is a tradeoff between accuracy of the analysis and running time.

Multiple values were then added to the analysis, as per Section 9.2. A fair amount of the analyzer's code had to be modified to use the restricted \( \subseteq_v \) set inclusion. Some quick tests indicated that having to filter out multiple values on all in-flows could add up to 25\% in the running time of the analysis for programs like the ones used for testing in Chapter 8.
The `define` and `set!` forms were added as explained in Section 9.3, and the remaining R5RS language constructs were added as well.

Although structures are not part of the R5RS definition of Scheme, our ultimate goal is to develop an analysis that can handle the whole of the PLT version of the Scheme language. We therefore decided to add generative structures to our analysis. Structures in PLT Scheme are generative in the sense that a function like

```
(define f (lambda () (define-struct foo (a b c))))
```

will return a new structure type `foo` each time the function `f` is applied. Each new structure type `foo` will be incompatible with any previously defined type `foo`. In addition to being generative, structures also have to have a notion of structure hierarchy, whereby a structure type can extend a parent structure type with new members. This notion of hierarchy is used in PLT Scheme to implement, among other things, object-oriented classes.

Adding structures to our analysis proved to be quite difficult. Dealing with the notion of hierarchy was easily done, mainly by extending the analyzer's subtype checker. The main problem with structures was that `define-struct` is not a language form in PLT Scheme, but a macro that expands into the application of several primitives. These primitives are responsible for creating the structure type and the related structure accessors and mutators. They are therefore higher-order. Also, accessors and mutators have to know which structure type they are defined for, so they have to contain some hidden information about their related structure type. Having one `define-struct` expand into several independent primitives that deal with information-hiding functions meant that our analysis had to process all these primitives at once, to make sure generativity problems could be reliably detected (i.e. trying to apply an accessor for one structure type `foo` to a structure of another type `foo`).

The generativity of structures also made it necessary to fake polymorphism for the `make-struct-type` primitive. This primitive is used to create all structure types,
<table>
<thead>
<tr>
<th>Num nodes</th>
<th>113</th>
<th>393</th>
<th>848</th>
<th>1478</th>
<th>2283</th>
<th>3263</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended</td>
<td>31</td>
<td>190</td>
<td>288</td>
<td>498</td>
<td>839</td>
<td>1231</td>
<td>1.07</td>
</tr>
</tbody>
</table>

Table 9.1: Stress tests.

so analyzing it polyvariantly like any other primitive would have meant losing generativity when a define-struct appeared inside a lambda. The define-struct would have been analyzed only once, when analyzing the lambda, and only one structure type foo would have been created overall.

In the end, these two problems were solved by analyzing define-struct in an extremely ad-hoc way. As a consequence, the analysis is currently sound when the program uses the high-level define-struct macro, but might break if the programmer makes direct use of the low-level make-struct-type primitive.

Finally, the subtyping part of the analysis was extended with a rule to subtype case-lambda with rest arguments. This rule is an extension of the normal subtyping rule for functions (i.e. were arguments are subtyped in a contravariant manner), adapted to handle the six different cases illustrated in Figure 9.2.

The performance of the extended analyzer, called MrFlow, was then measured using the same stress tests as in Chapter 8, on the same machine. The results appear in Table 9.1, and are plotted in Figure 9.5 along with the results from Chapter 8.

As can be seen from the figure, extending the analysis does not make the running time any worse when processing programs that do not make use of the extra features added to the analysis. Some tests we ran on real (non-synthetic) programs that heavily use these extra features (mainly the additional language constructs and primitives) seem to show that the conditional-constraint based analysis still performs much better than the selector based one.† More tests with real programs are in order.

†Except for the post-analysis type checking of primitives, since, unlike the old MrSpidey analysis, the new analysis does not yet minimize types (simplification of unions and folding of recursive types) before checking them, making this checking phase currently very slow.
Figure 9.5: Analysis times for extended analysis, plotted log-log.

Finally, it should be noted that Figure 9.5 gives a somewhat misleading idea of the comparative performance of the basic CA-SBA analysis of Figure 7.1 and of the extended CA-SBA. While testing of both analysis was CA-SBA on the same machine, the basic one was implemented on top of DrScheme version 103, while the extended one was implemented on top of DrScheme version 200. Version 200 is using a different parsing system and a different internal representation of terms, which made the basic analysis run 30% to 50% faster when ported to version 200 of DrScheme. This gain in speed was subsequently lost when extending the basic CA-SBA analysis, a fair amount of the loss being attributable to handling multiple values, as described in Section 9.2.
Chapter 10

Related and future work

We began with Flanagan's theoretical foundations and implementation work for Mr-Spidey [10, 11]. His work is itself based on the one of Heintze [12]. The CA-SBA analysis we then presented is based on the work of Palsberg and Schwartzbach [18]. There are numerous other papers on set-based analysis. See [1] for an overview and for pointers to the literature.

The \texttt{lambda*} construct, essentially the same as \texttt{case-lambda}, was described by Dybvig and Hieb [6].

Heintze and McAllester describe a linear-time algorithm for analyzing ML programs with bounds on the size of types for subexpressions [13]. Their system \textit{LC} uses \texttt{dom} and \texttt{ran} constructs that are syntactically similar to Flanagan's \texttt{dom} and \texttt{rng} selectors, but the two analyses are otherwise quite different. The \texttt{dom} and \texttt{ran} constructs may be applied to expressions that themselves contain \texttt{dom} and \texttt{ran}, while \texttt{dom} and \texttt{rng} may be applied only to set variables. More significantly, the \textit{LC} system does not include transitive rules, which allows their system to escape the cubic-time bottleneck. Unlike our analyses, the \textit{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 \texttt{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 [7]. Their language, ML$^{ua}$, allows such procedures only in \texttt{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].

Flanagan [10, Appendix E.5] describes a notion of type schemas that is similar to the one we defined in Chapter 9. Flanagan’s transformation rules use of course selectors, and can handle pairs, unions, recursive types, and functions of one argument. We are not aware of any other work explaining how to transform a type specification into a constraint system. Flanagan's system and ours were created independently.

We believe that we will be able to use our type specification system to provide a true separate analysis of modules. By making the type language available to the user, programmers will be able to specify types for the module interfaces. Imported functions from modules will then be processed like primitives. Making the type language visible to the user will also allow programmers to specify types for non-exported functions, which should help the analysis to verify more program invariants, and flag type errors at the user function level, instead of at the primitive level.

Our type system, while currently expressive enough to be able to handle most of the R5RS primitives [15], will have to be extended to deal better with primitives like append, apply, and map. Also, non-R5RS primitives like printf will have to be added. This means adding dependent typing to our type system [4]. Analyzing primitives like make-struct-type, using a type specification only, should become possible if we add type constructs that explicitly specify polymorphism or polyvariance.

Another feature we will want to add is flow sensitivity inside if constructs. In the expression (if (number? x) x x), the possible values that x might take in the "then" branch should be restricted to being numbers. Similarly, we will also want flow sensitivity across primitives. If, in a program, analyzing an expression like (car some-list) raises an error because the list might be empty, the same error should be not be raised again if the same list is applied to car or cdr further down the program. These features have been implemented by Flanagan in MrSpidey, and proved to be
useful in decreasing the number of spurious errors.

Finally, we will have to prove the soundness of our analysis. This might be done by extending Wand and Williamson's proof for the lambda calculus [24], although no work has been done yet in this regard.
Chapter 11

Conclusions

We have shown that Flanagan's selector-based framework for SBA can be extended to handle case-\texttt{lambda} as well as rest parameters. Unfortunately, the analysis becomes too expensive. Managing the annotations makes it difficult to implement, as well. A closure analysis style SBA gives similar results and is straightforward to implement. It can also be easily extended to handle the whole Scheme language.

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, MrFlow, promises to be significantly faster as well as more precise than MrSpidey.
Bibliography


