Selectors Make Set-Based Analysis Too Hard

Philippe Meunier, Robert Bruce Findler†, Paul Steckler and Mitchell Wand

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

†*Department of Computer Science*
University of Chicago
Chicago, IL 60637
robby@cs.uchicago.edu

Abstract.

A set-based program analysis establishes constraints between sets of abstract values for all expressions in a program. Solving the system of constraints produces a conservative approximation to the program's runtime flow of values.

Some practical set-based analyses use explicit selectors to extract the relevant values from an approximation set. For example, if the analysis needs to determine the possible return values of a procedure, it uses the appropriate selector to extract the relevant component from the abstract representation of the procedure.

In this paper, we show that this selector-based approach complicates the constraint solving phase of the analysis too much and thus fails to scale up to realistic programming languages. We demonstrate this claim with a full-fledged value flow analysis for `case-lambda`, a multi-branched version of lambda. We show how both the theoretical underpinnings and the practical implementation become too complex. In response, we present a variant of set-based closure analysis that computes equivalent results in a much more efficient manner.

1. Introduction

In 1990, Dybvig and Hieb introduced a variable-arity form of `lambda` called `lambda*` [6]. This form generalizes `lambda` by allowing multiple clauses, each with its own parameter list and its own body. When such a procedure is applied, one of the clauses is selected based on the number of actual arguments, and the corresponding body executed. If the number of actual arguments does not match the number of formal argument of some clause, an arity error is signaled. A variation of this construct, called `case-lambda`, was added to Chez Scheme [5] and later to PLT Scheme [12]. In both implementations, a `case-lambda` clause parameter list may also have a *rest* parameter, just as for `lambda` in Scheme [17].[1] DrScheme [8], the graphical development environment based on PLT Scheme, uses `case-lambda` extensively. It is used in particular to define many class methods of the object-oriented graphi-

[1] Dybvig and Hieb described a notion of "rest variable", which is not quite like a Common Lisp or Scheme rest parameter since these rest variables cannot be directly referenced like other parameters, save for special positions in applications.

00: 1–31, 2005.

cal framework on which DrScheme itself is based, where `case-lambda` allows a form of dynamic dispatch based on the number of arguments.

In this paper, we describe an extension of the formalism developed by Flanagan [10, 11] for the MrSpidey static debugger [9], to analyze `case-lambda` and rest parameters. The set-based analysis (SBA) used by Flanagan is derived from Heintze's [13, 14]. For each expression in a program it computes a set of abstract values that conservatively approximate the values that the expression might evaluate to at runtime. These sets are then used to statically detect possible bugs in the program (e.g. using a number where a string is expected) which are then reported by MrSpidey back to the user.

The analysis computes those sets of abstract values by creating constraints between them that simulate the runtime flow of values between the corresponding expressions in the program. The constraints use explicit *selectors* like `dom`, `rng`, `car`, and `cdr` to choose data flowing through expressions. For example, the `rng` selector chooses the ranges of procedures that flow through an expression; the `car` selector obtains the first elements of lists that flow through the expression. Selectors also control the flow of data into and out of procedures at application sites.

MrSpidey analyzes nearly all of PLT Scheme, including `case-lambda`, though Flanagan does not provide a formal treatment of that aspect of its analysis. Unfortunately, the existing implementation is too conservative and propagates values through *all* clauses of `case-lambda`, including unused ones, and even propagates them in the presence of arity errors. As a consequence, MrSpidey often flags errors where none exist, which limits its usefulness as a static debugger.

As part of our goal to develop a practical soft-typing system for the full PLT Scheme programming language, based on a better debugger, we investigated extensions to Flanagan's analysis to support `case-lambda` more precisely. In our modification of Flanagan's analysis, we annotate selectors with new arity and argument-position information, which ensures that data flows into and out of appropriate `case-lambda` clauses. As we shall show, this approach greatly improves upon the precision of the existing MrSpidey implementation. Selector annotations also allow us to analyze rest arguments.

Unfortunately, while our new extension of Flanagan's approach yields sound results, its time cost is too great. Solving the constraints to determine the sets of abstract values requires computing the transitive closure of the constraints. Explicit selectors impose two burdens on that computation. First, the presence of the many selectors associated with procedures means there are many more nodes in the constraint graph. Second, computing the transitive closure of some of the constraints

means matching selector pairs according to argument position and arity annotations. Obtaining these pairs requires searching through a set of candidate selectors and checking the annotations for each candidate, another computationally intensive process. With those insights, we conclude that, while Flanagan's selector approach to SBA is suitable for analysis of most of Scheme, it is not suitable for analysis of Scheme with `case-lambda`.

Fortunately, an ordinary closure analysis style SBA (CA-SBA) like Palsberg's [19], which does not use selectors, can be extended to provide similar analysis results with a lower asymptotic time upper bound. Therefore, the next static debugger for PLT Scheme will use a CA-SBA instead of the annotated-selector approach.

The paper is organized as follows. We motivate our work in Section 2 by presenting some limitations of Flanagan's selector-based approach when analyzing `case-lambda`. In Section 3, we review Flanagan's account of set-based analysis with explicit selectors. Next, we describe in Section 4 a new extension to Flanagan's system to analyze `case-lambda` programs without rest arguments, then add rest arguments to the analysis in Section 5. In Section 6, we discuss the high asymptotic complexity of the resulting annotated selector analysis. In Section 7, we show how to extend Palsberg's closure analysis style SBA to handle `case-lambda` and rest parameters. In Section 8, we give empirical results, comparing MrSpidey, our modified selector analysis, and the closure analysis style SBA. These experiments show that the annotated-selector approach is too complex in practice as well, while the CA-SBA one performs satisfactorily. Section 9 presents related and future work. Finally, in Section 10 we offer conclusions.

2. Limitations of MrSpidey

In this section, we catalog the ways in which the existing MrSpidey analysis of `case-lambda` is unsatisfactory. In Figure 1, we show the results of MrSpidey's analysis of four simple programs that capture the essence of the limitations of MrSpidey. The boxes contain type information about their adjacent expressions. For example Figure 1-(A) shows that the variable `x` has a string type. A type may also be a definite constant, such as a particular number.

Propagation despite arity errors. When a procedure is applied to an incorrect number of arguments, MrSpidey propagates data through as many formal arguments as possible. Figure 1-(A) shows MrSpidey's analysis of a procedure of two arguments applied to one argument.[2]

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

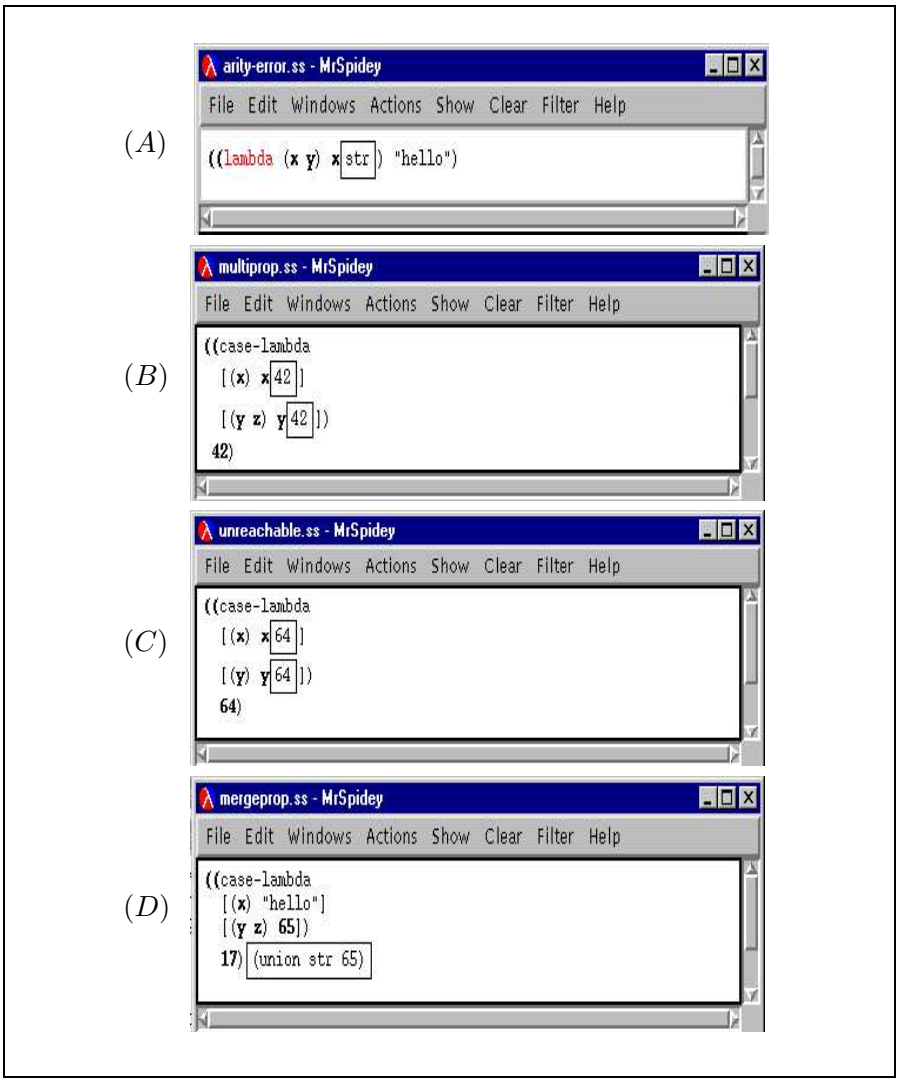

Figure 1. MrSpidey mishandles `case-lambda`.

At run-time, the value of the actual argument never reaches the bound `x`, though MrSpidey suggests otherwise by computing a string type for the variable.

Propagation through multiple clauses. MrSpidey propagates values of actual arguments through all clauses of a `case-lambda`. Figure 1-(B) shows a `case-lambda` with two clauses applied to one argument. 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.

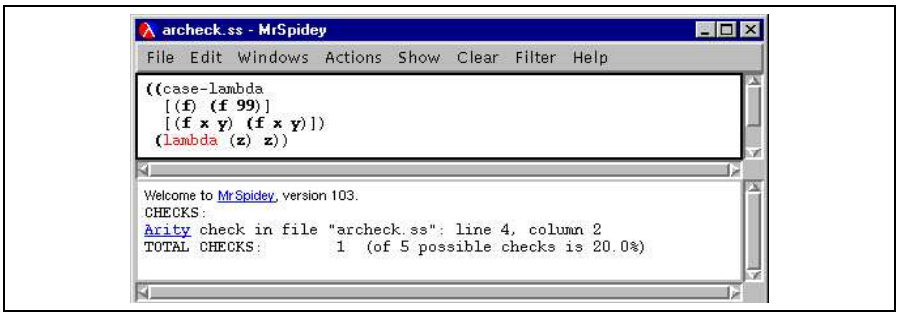

Figure 2. Spurious arity check.

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

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

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

Our modified analysis remedies each fault identified here.

3. MrSpidey theory

We review Flanagan's formalism for analyzing an extended version of the λ-calculus. This formalism is derived from the one by Heintze [13, 14]. Unlike the control-flow analysis of, say, Palsberg [19] or Shivers [22], Flanagan uses explicit selectors in constraints to connect the formal arguments and body of a function to the actual arguments and result of an application, and to connect the actual components of a pair to the results of the pair projections.

The language we analyze is the λ-calculus extended with constants and special forms for **cons**, **car**, and **cdr**. In our language `lambda` terms are labeled:

$$E ::= x \mid c \mid (\lambda x.E)^{\ell} \mid (E\ E) \mid (\mathbf{cons}\ E\ E) \mid (\mathbf{car}\ E) \mid (\mathbf{cdr}\ E)$$

Set expressions representing sets of abstract values are defined by the grammar

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

where α is a *set variable*, representing a program expression, and `pair` is a token. We also write β and γ for set variables. The metavariable c represents term language constants. The forms `dom`, `rng`, `car` and `cdr` are *selectors*. Of these, only `dom` is *contravariant*; the others are *covariant*.[3] We use σ as a metavariable for selectors. A *constraint* is an inequality on set expressions of the form $\tau \leq \tau'$. Constraints indicate the flow of data. For example, the constraint $c \leq \alpha$ means that the constant c flows into the expression associated with the set variable α.

Following Heintze, Flanagan's set-based analysis proceeds by phases. The first phase is *constraint derivation*, performed by a pass over the program's abstract syntax tree. For each subexpression, this phase associates a set variable with the subexpression and generates some constraints according to constraint derivation rules. Next, a *propagation* phase combines constraints using constraint propagation rules to generate new constraints, effectively mimicking the flow of data through a program. Then, a set of values is computed for each program point. From such a set, a type can be constructed.

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

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

where

Γ is an environment mapping term variables to set variables,

E is an expression,

α is a fresh set variable representing the expression E, and

C is a set of constraints.

[3] A constructor like `car` is covariant because, as the set represented by α grows, so does the set represented by $\mathtt{car}(\alpha)$. Conversely, `dom` is contravariant because $\mathtt{dom}(\alpha)$ decreases as α increases, corresponding to the fact that, as the number of possible functions represented by α increases, fewer values can be accepted by all of these functions.

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

$$\Gamma \vdash c : \alpha, \{c \leq \alpha\} \qquad \text{(CONST)}$$

$$\frac{\Gamma[x \mapsto \alpha'] \vdash E : \beta, C'}{\Gamma \vdash (\lambda x.E)^{\ell} : \alpha, C' \cup C} \qquad \text{(LAMBDA)}$$
$$\text{where } C = \left\{ \begin{array}{c} \ell \leq \alpha \\ \texttt{dom}(\alpha) \leq \alpha' \\ \beta \leq \texttt{rng}(\alpha) \end{array} \right\}$$

$$\frac{\Gamma \vdash E_i : \beta_i, C_i \quad i \in [1,2]}{\Gamma \vdash (E_1\ E_2) : \alpha, C_1 \cup C_2 \cup C} \qquad \text{(APP)}$$
$$\text{where } C = \left\{ \begin{array}{c} \beta_2 \leq \texttt{dom}(\beta_1) \\ \texttt{rng}(\beta_1) \leq \alpha \end{array} \right\}$$

$$\frac{\Gamma \vdash E_i : \beta_i, C_i \quad i \in [1,2]}{\Gamma \vdash (\textbf{cons}\ E_1\ E_2) : \alpha, C_1 \cup C_2 \cup C} \qquad \text{(CONS)}$$
$$\text{where } C = \left\{ \begin{array}{c} \texttt{pair} \leq \alpha \\ \beta_1 \leq \texttt{car}(\alpha) \\ \beta_2 \leq \texttt{cdr}(\alpha) \end{array} \right\}$$

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

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

Figure 3. MrSpidey constraint derivation (judgements of the form $\Gamma \vdash E : \alpha, C$).

Let us provide intuition for some of the rules. The VAR rule says that when analyzing a reference to a variable x, a fresh set variable α is used to represent that reference. If the variable's binding is represented in the environment by the set variable β then a new constraint $\beta \leq \alpha$ is created to simulate the flow of values from the binder to the reference.[4] For the bracketed constraints in the LAMBDA rule, we have

[4] The reason for creating a new set variable α instead of just returning the variable β corresponding to the binder is to be able to distinguish the binder from all its references and references from each other. This in turns allows MrSpidey to properly draw arrows showing the flow of values between binders and references. It is not necessary for the analysis proper.

$$\frac{\tau \leq \alpha \quad \alpha \leq \beta \quad \begin{array}{l}\mathbf{const?}(\tau) \vee \\ \mathbf{label?}(\tau) \vee \\ \mathbf{token?}(\tau)\end{array}}{\tau \leq \beta} \quad \text{(TRANS-CONST)}$$

$$\frac{\alpha \leq \sigma(\gamma) \quad \sigma(\gamma) \leq \beta}{\alpha \leq \beta} \quad \text{(TRANS-SEL)}$$

$$\frac{\alpha \leq \sigma(\beta) \quad \mathbf{selector^{+}?}(\sigma) \quad \beta \leq \gamma}{\alpha \leq \sigma(\gamma)} \quad \text{(COVARIANT-PROP)}$$

$$\frac{\sigma(\alpha) \leq \beta \quad \mathbf{selector^{-}?}(\sigma) \quad \alpha \leq \gamma}{\sigma(\gamma) \leq \beta} \quad \text{(CONTRAVARIANT-PROP)}$$

Figure 4. MrSpidey constraint propagation.

$\ell \leq \alpha$: a procedure label ℓ representing the procedure expression flows into the set variable α representing the possible values for the procedure, simulating the fact that a `lambda` expression evaluates to a lambda value;

$\texttt{dom}(\alpha) \leq \alpha'$: whatever flows into the domain of the procedure flows into its formal parameter, and

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

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

Figure 4 shows the constraint propagation rules. In the TRANS-CONST rule, we use the predicates **const?**, **label?**, and **token?** to detect constants, procedure labels, and tokens. The difference between covariant and contravariant selectors shows up in the propagation rules COVARIANT-PROP and CONTRAVARIANT-PROP. The $\mathbf{selector}^{+}$ predicate holds when its argument is `rng`, `car`, or `cdr`; the $\mathbf{selector}^{-}$ predicate holds only for `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.

Note that for a given function represented by a set variable α the set expressions $\texttt{dom}(\alpha)$ and $\texttt{rng}(\alpha)$ do not directly correspond to any term in the analyzed program. In the LAMBDA rule the formal argument is represented by α' and the body of the function is represented by β. Similarly in the APP rule the actual argument and the application's

result are represented by β_2 and α. The role of the set expressions $\texttt{dom}(\alpha)$ and $\texttt{rng}(\alpha)$ is to flow along with the function itself until they reach the operator position of an application. At that point the two expressions are used through the TRANS-SEL rule (along with the two `dom` and `rng` set expressions generated in the APP rule) to establish the connection between the actual and formal arguments and between the body of the function and the result of the application. Similarly when α represents a pair, the set expressions $\texttt{car}(\alpha)$ and $\texttt{cdr}(\alpha)$ are used to connect the components of the pair (represented by β_1 and β_2 in rule CONS) to the result of extracting one or the other of the components (represented by α in rules CAR and CDR).

The full details of MrSpidey's constraint solution and type reconstruction algorithms are beyond the scope of this paper, but we attempt here to convey their essence. See Flanagan's dissertation for details [10]. For a subterm with associated set variable α, the set $\{c \mid c \leq \alpha\}$ describes the constants that may be the result of evaluating the subterm. If we have the constraint $\texttt{pair} \leq \alpha$, then the term may evaluate to a pair, and $\{\beta \mid \beta \leq \texttt{car}(\alpha)\}$ is the set of set variables that may flow into the **car** of such a pair. The sets of values for these set variables provide the actual values. We compute the solutions to **cdr**'s and procedure ranges in similar fashion. Procedure domains require a slightly more complex calculation due to the contravariance of the `dom` selector.

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

What is missing from the existing formalism? In the language to be analyzed, all procedures have one clause with one parameter, and there are no rest parameters. These restrictions on procedures are imposed by the use of the simple `dom` and `rng` selectors. Therefore, in his dissertation [10, Appendix E.3], Flanagan indicates that a procedure of more than one argument is modeled by considering that procedure to take one argument, which becomes bound to a list of actual arguments at its application sites. The values in the list are distributed to the formal arguments by pulling out elements of the list. Because all clauses of a `case-lambda` are considered to have a single argument, the arities of the clauses are not considered, and that list is propagated to all clauses. Similarly, the results of all `case-lambda` clauses are merged into application results. The TRANS-SEL rule in Figure 4 controls the propagation of data into formal parameters (when the selector involved is `dom`) and out of procedures (when the selector is `rng`). Hence, to improve the analysis, we need to focus on the mechanism in that rule.

4. Handling case-lambda

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

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

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

An interval indicates the number of arguments a clause accepts. Without rest parameters, the lower and upper bounds on the interval are the same and could be collapsed into a single number. This will change when we add rest parameters in the next section though. We use $\mathcal{I}$ as a metavariable for intervals.

DEFINITION 2. *An* arity *is a non-empty list of intervals.*

The first element of an arity indicates the number of arguments accepted by a clause. The remaining elements put the clause in context, by listing the intervals associated with preceding clauses. Arities are computed as follows: for each clause i of a function, assign it an interval $[n_i, n_i]$, where n_i is the number of formal arguments for that clause. Then, for each clause i, assign it the arity $a_i = ([n_i, n_i], (\mathcal{I}_{i-1}, \ldots, \mathcal{I}_1))$, where $\mathcal{I}_j$ is the interval assigned to the j^{th} clause. For example, the following function:

```
(define f
  (case-lambda
    [(x y) 1]
    [(x y z) 2]))
```

has two clauses, the first one with arity $([2, 2])$ and the second one with arity $([3, 3], [2, 2])$. We use a as a metavariable for arities.

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

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

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

For `rng` selectors, the forms are

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

Consider $\texttt{dom}_2^a(\alpha)$; this set expression represents the flow into the second argument of a `case-lambda` clause with arity a. Similarly, $\texttt{dom}_{1,3}^{[3,3]}(\alpha)$ represents the flow into the first argument of a procedure of three arguments (as indicated by the $[3,3]$ interval) when that procedure flows into an application site with three actual arguments. The selector $\texttt{rng}^a(\alpha)$ represents the flow out of a `case-lambda` clause with arity a. The set expression $\texttt{rng}_3^{[3,3]}(\alpha)$ represents the value returned by a procedure of three arguments (as indicated by the $[3,3]$ interval) when that procedure is applied to three actual arguments. All these annotations are going to be necessary to ensure that, for a given application, values will flow in and out of only the right `case-lambda` clause.

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

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

These revised rules have essentially the same form as those in Figure 3, except that

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

The selector annotations are used in the revised propagation rules in Figure 6, in particular, in the rules TRANS-DOM and TRANS-RNG. The unchanged rule TRANS-CONST is omitted. The unchanged rules TRANS-SEL and COVARIANT-PROP are omitted as well but have now a more restricted scope: the COVARIANT-PROP rule is now only used to

$$\frac{\Gamma[x_{i,j} \mapsto \alpha_{i,j}]_{j\in[1,n_i]} \vdash E_i : \beta_i, C_i \quad i \in [1,m]}{\Gamma \vdash \begin{pmatrix} \texttt{case-lambda} \\ [(x_{1,1} \ldots x_{1,n_1})\ E_1] \\ \vdots \\ [(x_{m,1} \ldots x_{m,n_m})\ E_m] \end{pmatrix}^{\ell} : \alpha,\ \bigcup_{i\in[1,m]} C_i \cup C} \quad (\textsc{case-lambda})$$

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

(where a_i is the arity of the i^{th} clause of the `case-lambda`)

$$\frac{\Gamma \vdash E_i : \beta_i, C_i \quad i \in [0,n]}{\Gamma \vdash (E_0\ \ldots\ E_n) : \alpha,\ \bigcup_{i\in[0,n]} C_i \cup C} \quad (\textsc{app})$$

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

Figure 5. Revised constraint derivation rules (judgements of the form $\Gamma \vdash E : \alpha, C$).

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

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

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

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

Figure 6. Revised propagation rules.

propagate the `car` and `cdr` selectors and the rule TRANS-SEL no longer handles `dom` and `rng` selectors. We no longer need the CONTRAVARIANT-PROP rule. The core idea is to propagate values through a `case-lambda` clause only when the number of actual arguments matches the number expected by that clause, and does not match the number expected by

any preceding clause. This idea is captured by the following satisfaction relations used in the TRANS-DOM and TRANS-RNG rules.

For intervals we only have to check whether the number of actual arguments matches the number expected by a given clause, so the $\models$ relation is simply:

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

This definition will change in the next section when we introduce rest arguments.

The satisfaction relation in the TRANS-DOM and TRANS-RNG propagation rules involves a number representing a number of actual arguments in an application, an interval representing what kind of clause can handle that number of arguments, and an arity representing an actual clause of the function that might be applied. That relation is defined by (where $\in$ is the usual mathematical interval membership test):

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

$s \in [p, q]$,

$[n, m] \models [p, q]$, *and*

$\forall i \in [1, t],\ s \notin \mathcal{I}_i$

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

Note that, as before, all the new annotated `dom` and `rng` selectors do not correspond to any term in the program. Their role is to flow along with their corresponding function until an application is reached, at which point the TRANS-DOM and TRANS-RNG rules will use them to establish connections between actual and formal arguments, including possible rest arguments, and between clause bodies and application results.

5. Analysis of rest parameters

The introduction of rest parameters requires additional constraints, which need to account for the uncertainties associated with such arguments. With a rest parameter, a clause accepts some number of required arguments, but may take more. For example the function

```
(define f
  (case-lambda
    [(x y . z) z]))
```

can be applied to two or more arguments. When applied, all the actual arguments given to the function after the first two are gathered in a list that is bound to the rest argument `z`. We cannot, therefore, be certain how many arguments a particular function clause will be applied to. Moreover, when deriving constraints at an application site, we do not know the arity of selected clauses in the 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 section. We revise the definition of intervals (Definition 1) to also allow intervals of the form $[n, \omega]$, where n is a nonnegative integer and ω is a special symbol representing an arbitrarily large number of arguments. 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 nonempty list consisting of its assigned interval and the 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*

$n = p$ *and* $m = q = \omega$

The other satisfaction relation, from Definition 4, does not change, except to use this new definition.

In Figure 7, we show just the new constraints required. For the language we are now considering, which includes `case-lambda`, **cons**, **car**, and **cdr**, the derivation rules are the VAR, CONST, CONS, CAR, and CDR rules from Figure 3; the CASE-LAMBDA rule from Figure 5, and the APP rules in Figures 5 and 7.

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

Consider an application site with n actual arguments and a procedure that flows to this site. If a selected clause of that procedure has a rest parameter then that clause 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

$$\frac{\Gamma \vdash E_i : \beta_i, C_i \quad i \in [0,n]}{\Gamma \vdash (E_0 \ \ldots \ E_n) : \alpha, \ \bigcup_{i\in[0,n]} C_i \cup C} \quad (\textsc{app})$$

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

Figure 7. Additional constraints for rest parameters (judgement of the form $\Gamma \vdash E : \alpha, C$).

$\mathtt{rng}_n^{[i,\omega]}(\beta_0) \leq \alpha$ for each i between zero and n. These constraints represent flow out of selected `case-lambda` clauses.

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

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

Here, i is a particular number of required arguments, j is a position within those arguments, and n is the number of actual arguments. These constraints represent flow into the required parameters of clauses with rest parameters. Note that when the number i of required parameters is zero, no flow into required parameters exists (i.e. there is no j between one and i) so we do not have to consider that case.

Finally, we need to account for flow into rest arguments, which are bound to lists. Since we do not know the arity of selected clauses in procedures that flow to the application site, we do not know how many arguments are required by the clauses, and thus we do not know the lengths of the lists of actual arguments that should flow into the rest arguments. Therefore we need to consider all possibilities.

First suppose a selected clause has exactly n required arguments. Then at run-time, the rest argument becomes bound to the empty list. Hence we have the pair of constraints

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

where `null` is one of the constants in our language, evaluating to the empty list.

Second suppose a selected clause takes fewer than n required arguments. Since we do not know the exact number of arguments required by the clause, we always generate the constraints

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

The effect of these constraints is to model the flow of lists of varying lengths into the α_i's. For instance, α_1 receives a list of length n, while α_n receives a list of length one. We then generate constraints of the form

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

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

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

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

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

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

6. Complexity

It is not enough for our analysis to be sound and accurate, it must be easily computable. A simple set-based flow analysis based on the transitive closure of set constraints (a form of monovariant SBA for shallow patterns [18]) can 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" [16]. Unfortunately, our modified version of Flanagan's SBA far exceeds this bound.

6.1. MrSpidey's analysis

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

We now describe an algorithm that can be used to close the constraints under the propagation rules in Figure 4. The algorithm has a cubic-time upper bound and is as follows. As each constraint is generated, check whether it matches a premise in a propagation rule. Both the constraint generation and the check are constant-time operations. 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 the premise of a propagation rule, find all constraints that match the other premise. There are $O(n)$ such 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 α, 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 that map set expressions to their lower and upper bounds. If the constraint in the consequent does not exist in the pool of constraints, a constant-time check, add it. The only non-constant factors in this algorithm are the quadratic bound on the number of constraints and

the linear bound on the number of eligible constraints. Therefore, the MrSpidey analysis does have a cubic-time upper bound.

6.2. Annotated selectors

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

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

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

there are $O(n^2)$ many constraints of the form $\alpha \leq \sigma(\beta)$. A similar argument holds for the DOM-PROP rule.

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

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

Now consider the rule RNG-PROP. If we have a constraint matching the first premise, then clearly there is a linear bound on the number of 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 α and σ pairs in constraints of the form $\alpha \leq \sigma(\beta)$. For a given β, then, there is a linear bound on the number of constraints matching the first premise. A similar argument holds for the DOM-PROP rule.

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

6.3. Rest parameters

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

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

7. Eliminating selectors

In Flanagan's original set-based analysis (SBA) and our revision, `dom` and `rng` selectors are used to hook up actual arguments with formal parameters, and procedure bodies with applications. Creating these flows requires propagating numerous selectors from function definitions to application sites, and checking them against the selectors corresponding to these applications. Instead we can eliminate selectors by choosing a more straightforward mechanism for directing flow through formal parameters and from procedure bodies.

Palsberg's ordinary "closure analysis" SBA (CA-SBA) [19] can be extended in a straightforward manner to handle `case-lambda` and rest parameters. Figure 8 presents the constraints for such an analysis. As before, procedures are labeled; we now label all other subterms. Each such label ℓ has an associated set $\varphi(\ell)$, which can contain:

> labels, representing the flow of program constants, or
>
> ($cons$ ℓ ℓ) compound labels, representing the flow of pairs, and containing two labels for the two parts of the pair represented, or
>
> ($case$-λ $((\ell \ldots)$ $\ell) \ldots$) compound labels, representing the flow of `case-lambda` abstractions, and containing a label for each parameter and body for each clause of the `case-lambda` represented.

$$\Gamma[x \mapsto \ell'] \vdash x^{\ell} : \{\varphi(\ell') \subseteq \varphi(\ell)\} \qquad \text{(VAR)}$$

$$\Gamma \vdash c^{\ell} : \{\ell \in \varphi(\ell)\} \qquad \text{(CONST)}$$

$$\frac{\Gamma[x_{i,j} \mapsto \ell_{i,j}]_{j \in [1,n_i]} \vdash E_i^{\ell_i} : C_i^f \quad i \in [1,m]}{\Gamma \vdash \begin{pmatrix} \texttt{case-lambda} \\ [(x_{1,1}^{\ell_{1,1}} \ldots x_{1,n_1}^{\ell_{1,n_1}})\ E_1^{\ell_1}] \\ \vdots \\ [(x_{m,1}^{\ell_{m,1}} \ldots x_{m,n_m}^{\ell_{m,n_m}})\ E_m^{\ell_m}] \end{pmatrix}^{\ell} : \bigcup_{i \in [1,m]} C_i^f \cup C} \qquad \text{(CASE-LAMBDA)}$$

$$\text{where } C = \left\{ \begin{pmatrix} \textit{case-}\lambda\ ((\ell_{1,1} \ldots \ell_{1,n_1})\ \ell_1) \\ \vdots \\ ((\ell_{m,1} \ldots \ell_{m,n_m})\ \ell_m) \end{pmatrix} \in \varphi(\ell) \right\}$$

$$\frac{\Gamma \vdash E_i^{\ell_i} : C_i^f \quad i \in [0,n]}{\Gamma \vdash (E_0^{\ell_0} E_1^{\ell_1} \ldots E_n^{\ell_n})^{\ell} : \bigcup_{i \in [0,n]} C_i^f \cup C} \qquad \text{(APP)}$$

$$\text{where } C = \left\{ \begin{array}{l} \begin{pmatrix} \textit{case-}\lambda\ ((\ell'_{1,1} \ldots \ell'_{1,n_1})\ \ell'_1) \\ \vdots \\ ((\ell'_{m,1} \ldots \ell'_{m,n_m})\ \ell'_m) \end{pmatrix} \in \varphi(\ell_0) \Rightarrow \\ \left\{ \begin{array}{l} \exists\, k \in [1,m] : n \in \mathcal{I}_k \wedge \forall j \in [1,k-1] : n \notin \mathcal{I}_j \Rightarrow \\ \left\{ \begin{array}{l} \varphi(\ell'_k) \subseteq \varphi(\ell) \\ \forall j \in [1,r] : \varphi(\ell_j) \subseteq \varphi(\ell'_{k,j}) \\ q = \omega \Rightarrow (\textit{cons}\ \ell_{r+1} \ldots (\textit{cons}\ \ell_n\ \ell_{null}) \ldots) \subseteq \varphi(\ell'_{k,r+1}) \\ \text{where } [r,q] = \mathcal{I}_k \end{array} \right. \end{array} \right. \end{array} \right\}$$

(where $\mathcal{I}_i$ is the interval for the i^{th} clause of the *case-*λ)

$$\frac{\Gamma \vdash E_1^{\ell_1} : C_1^f \quad \Gamma \vdash E_2^{\ell_2} : C_2^f}{\Gamma \vdash (\textbf{cons}\ E_1^{\ell_1}\ E_2^{\ell_2})^{\ell} : \{(\textit{cons}\ \ell_1\ \ell_2) \in \varphi(\ell)\} \cup C_1^f \cup C_2^f} \qquad \text{(CONS)}$$

$$\frac{\Gamma \vdash E^{\ell'} : C^f}{\Gamma \vdash (\textbf{car}\ E^{\ell'})^{\ell} : \{(\textit{cons}\ \ell_1\ \ell_2) \in \varphi(\ell') \Rightarrow \varphi(\ell_1) \subseteq \varphi(\ell)\} \cup C^f} \qquad \text{(CAR)}$$

$$\frac{\Gamma \vdash E^{\ell'} : C^f}{\Gamma \vdash (\textbf{cdr}\ E^{\ell'})^{\ell} : \{(\textit{cons}\ \ell_1\ \ell_2) \in \varphi(\ell') \Rightarrow \varphi(\ell_2) \subseteq \varphi(\ell)\} \cup C^f} \qquad \text{(CDR)}$$

Figure 8. Closure analysis style SBA (judgements of the form $\Gamma \vdash E : C$).

The constraints in Figure 8 are similar to those usually presented for closure analysis of the lambda calculus as described by Palsberg [19] and in essence implement Shivers' 0-CFA [22]. Our selector-oriented SBA handles constants and forms for list construction and list projection, so we add constraints to handle those. Of course, we have `case-lambda` instead of `lambda`. The APP rule accounts for that difference. For every *case*-λ compound label in operator position the rule uses the information directly present in the label to precisely determine the first clause that can accept all the actual arguments of the application (something MrSpidey's selector based analysis could not do). Then it creates a flow from that clause's body to the application and from the actual arguments to the formal required parameters of the clause. Finally, all the remaining arguments, if any, flow as a list into the rest parameter of the clause, if it has one. All the other clauses are ignored.

With these changes, the form of these constraints is the same as for the closure analysis of the lambda calculus. The analysis generates at most a quadratic number of constraints; this set of constraints can be solved in cubic time [20]. Soundness of the analysis can likewise be proved by extending a soundness proof for the analysis of the lambda calculus [23].

The implementation of the analysis represents terms as nodes in a graph. Graph edges represent the flow of values from one term to another. New edges are created when *cons* compound labels flow into nodes representing the argument of **car** or **cdr**, or when *case*-λ compound labels flow into nodes representing the operator position of an application. This simulates the implications in the CAR, CDR, and APP rules in Figure 8. These new edges in turn trigger new propagations of labels and compound labels, which might further trigger the creation of new edges. The implementation checks that a label never flows twice through the same edge. This prevents labels from flowing around cycles in the graph forever, thereby ensuring that the analysis reaches a fixed point where no further propagation or edge creation are possible.

The main practical difference between this form of SBA and the selector-based one is that the runtime flow of procedures and pairs is modeled by the flow just of compound labels, without requiring the flow of multiple explicit selectors accompanying them. In essence all the information for a given procedure or pair that was spread among selectors in MrSpidey's analysis is now directly available in the compound label corresponding to that procedure or pair, rendering selectors useless. Also the use of implications in constraints simplifies the search for the right procedure clause when a procedure flows into the operator position of an application. Once the right clause is found, edges between actual and formal arguments and between clause body and application

result can all be created at once, instead of requiring the satisfaction relation to be checked for each selector separately as is the case in the selector-based analyses.

These differences greatly simplify the theoretical exposition of the analysis, compared to the annotated selector based one, as well as the implementation. The machinery necessary to generate the numerous selectors from Figure 5 and Figure 7 can be removed from the CA-SBA implementation. The fairly large amount of hashtable-based code used in the implementation of the propagation rules from Figure 4 and Figure 6 can be removed as well. Together these differences result in an implementation of the CA-SBA analysis that is considerably simpler than the implementations of the selector-based analyses.

Despite these simplifications the information computed by the CA-SBA is effectively the same as for the selector-oriented SBA. Types can then be reconstructed from this information using a simple recursive algorithm.

8. Empirical results

While the worst-case bounds mentioned in Section 6 do not necessarily mean bad performance in practice, it is clear that selector annotations make the problem harder than expected for SBA. To verify our expectations, we ran our annotated selector prototype, MrSpidey, and our CA-SBA prototype on some test programs. MrSpidey runs as an add-on tool in DrScheme [9] while the two prototypes run directly in MzScheme[5]. The analyzers were tested on one processor (900MHz UltraSPARC-III+, 8 MB of cache) of a Sun Fire 280R with two 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 below.

The two prototypes were developed for exploratory purposes and the language they analyze is quite small (the lambda calculus plus basic Scheme constants, `case-lambda`, as well as top-level definitions and references). This is in contrast with MrSpidey which analyzes the whole of the PLT Scheme language. Comparing running times between MrSpidey and the two prototypes is therefore tricky. While expanding the prototypes to analyze other kinds of terms would certainly slow

[5] This allows for the comparison of running times, since MzScheme is also the evaluator that underlies DrScheme, but not for the easy comparison of memory usage, since the memory used by MrSpidey is not easily separable from the memory used by DrScheme itself.

Table I. Procedure chain tests (milliseconds).

Test	MrSpidey	Annotated	CA-SBA	Ann/MrS	CA-SBA/MrS
s200	690	760	210	1.10	0.30
s400	1890	1520	440	0.80	0.23
s800	5700	3070	920	0.54	0.16
s1200	11750	4650	1390	0.40	0.12
s1600	19270	6210	1860	0.32	0.10
m200	1090	2070	330	1.90	0.30
m400	2450	3860	600	1.58	0.24
m800	7010	7660	1260	1.09	0.18
m1200	14560	11350	1930	0.78	0.13
m1600	22060	15590	2610	0.71	0.12

them down compared to MrSpidey, the good results of the CA-SBA one are still encouraging.

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

Consider the results in Table I. The numbers indicate milliseconds of processing 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 example the program s4 would look like:

```
(define f1 (lambda (x1) 42))
(define f2 (lambda (x2) (f1 x2)))
(define f3 (lambda (x3) (f2 x3)))
(define f4 (lambda (x4) (f3 x4)))
(f4 0)
```

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 (i.e. m4 would look very much like s4 above, except that every function would take a fixed randomly chosen number of arguments between zero and nine). 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 can have a better performance than MrSpidey. This in turn shows that the constraint solver used by the annotated-selector prototype compares reasonably well with the one used by MrSpidey.[6]

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

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

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

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

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

where t_1, t_2 are the times and n_1, n_2 are the number of nodes. For this class of programs, the CA-SBA implementation takes just over linear time, while the annotated-selector version takes just over cubic time. The asymptotic complexity of MrSpidey falls in between. Another way to view these relative complexities is given in Figure 9, which shows a

[6] MrSpidey actually performs better than the annotated selector prototype for the smaller programs in Table I. This is probably because the annotated selector prototype uses a more complex but asymptotically more efficient constraint solver than MrSpidey. As a result MrSpidey behaves better for small programs, but cannot compete with the near-linear running time of the annotated selector prototype for bigger ones.

Table II. Stress tests (milliseconds).

Num nodes	MrSpidey	Annotated	CA-SBA	Ann/MrS	SBA/MrS
113	90	410	10	4.56	0.11
393	440	6770	30	15.39	0.07
848	1140	49050	70	43.03	0.06
1478	4370	224240	130	51.31	0.03
2283	8240	808630	200	98.13	0.02
3263	13840	2473990	300	178.76	0.02
Exponent	1.45	3.13	1.14		

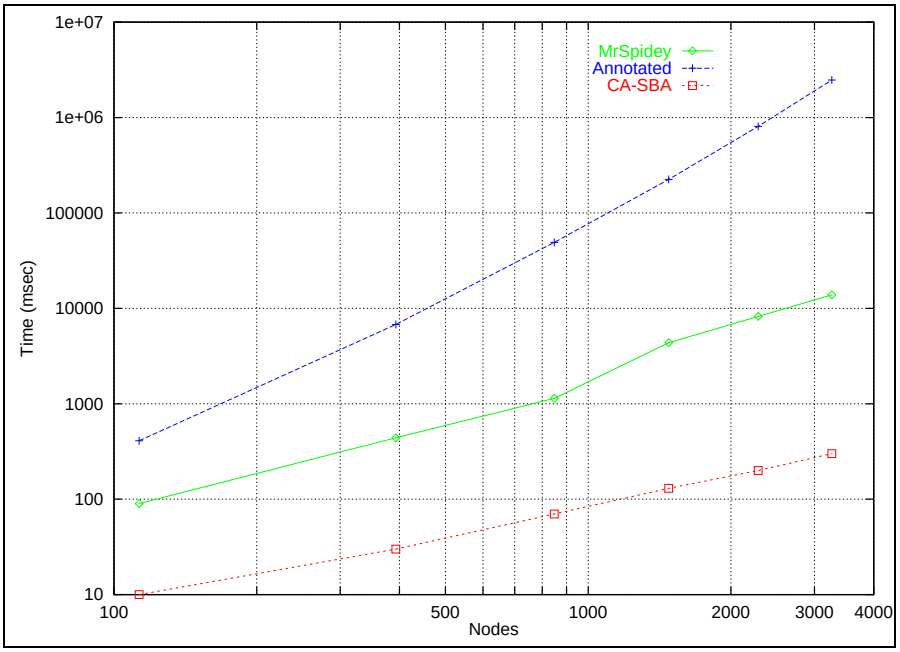

Figure 9. Analysis times, plotted log-log.

log-log graph of the running times for each implementation against the number of nodes.

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

well. Second, in order to establish data paths from actual arguments to formal parameters and from 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.

Note that the examples presented in this section as well as in Section 1 are very simple. As such they are not meant to simulate real programs but merely to illustrate the strengths and weaknesses of the different analyses when handling `case-lambda`. As a reference point day to day experience with MrSpidey shows that it can handle real programs of up to a few thousand lines of code in a reasonable amount of time. Given the good running times exhibited by the CA-SBA, we expect it to be able to handle programs large enough that precisely analyzing the biggest files in DrScheme's code base should be possible.

Note also that given the simplicity of our test programs above, a straightforward syntactic analysis would be enough to determine for each application which function and which clause in that function would be applied at runtime. In general one could run such a linear-time syntactic analysis first to handle the simpler cases, then run any of the three analyses tested above to take care of the remaining more complicated cases where the origin of the applied function is not as immediately obvious. The overall analysis time would most likely decrease, with the annotated-selector analysis probably benefiting the most from the reduction in the number of applications to analyze, since it is the analysis with the worst running time among the three. It is unlikely that this would be sufficient to make the annotated-selector analysis be able to compete with the CA-SBA analysis though, especially since the CA-SBA analysis would also benefit to a certain extent from the presence of the syntactic analysis. We also believe that in practice such a syntactic analysis would be of limited use: in DrScheme for example the main use of `case-lambda` is to implement class methods that require some form of dynamic dispatch based on the number of arguments they receive. Instantiation of these classes and invocations of these methods are usually so removed from each other and from their definitions that a syntactic analyzer like DrScheme's Syntax Checker cannot usually connect the `case-lambda` definitions to their uses.

9. Related and future work

We began with Flanagan's theoretical foundations and implementation work for MrSpidey [10, 11]. Flanagan's framework is directly derived

from the one by Heintze [13, 14]. There are numerous other papers on set-based analysis. In particular Palsberg [19], Shivers [22], and Sestoft [21] have all used set-based flow analyses to statically compute properties of untyped higher-order functional languages. See Aiken's introduction to set-based analysis [1] for an overview of the field and for additional pointers to the literature. Cousot and Cousot [4] also describe a general framework in which these analyses can be modeled.

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

Heintze and McAllester describe a quadratic-time algorithm for analyzing ML programs with bounds on the size of types for subexpressions [15]. Their system LC uses *dom* and *ran* constructs that are syntactically similar to Flanagan's `dom` and `rng` selectors, but the two analyses are otherwise quite different. The *dom* and *ran* constructs may be applied to expressions that themselves contain *dom* and *ran*, while `dom` and `rng` may be applied only to set variables. More significantly, the LC system does not include transitive rules, which allows their system to escape the cubic-time bottleneck while still computing precise results, provided the program has bounded types. The LC algorithm can also be adapted to run in linear time if only partial results are necessary. Unlike our analyses, the LC system is not concerned with procedures of multiple arguments, because it assumes that all procedures are curried. While our CA-SBA may require cubic time, there are no type restrictions on programs to achieve that result.

To our knowledge, there has been no previous attempt to describe a set-based analysis for `case-lambda`, nor for Lisp or Scheme's rest arguments. Dzeng and Haynes [7] describe a type reconstruction mechanism for an ML-like language with variable-arity procedures. The type reconstruction for these procedures is based on the use of infinitary tuples. Those tuples, in turn, could be implemented in ML using an enhanced tuple and tuple matching syntax. The worst case complexity would then be the usual exponential one for type inference in ML.

Aiken *et al.* [2] describe a type inference system in which conditional types handle propagation through multi-way pattern-based `case` expressions. For a given `case` expression the system checks every clause to see if the type of its pattern can match the type of the tested expression. The result types of all the clauses with a matching pattern are then unioned to become the type of the whole `case` expression. Since each clause pattern in a `case` expression is tested in isolation of the other clauses, the system cannot determine precisely which clause might be selected at runtime. Hence the need for a union. This is probably not too much of a problem in practice since each `case` expression is restricted to

have only pairwise disjoint patterns (unlike our `case-lambda` construct which can have clauses with overlapping numbers of arguments).

The closure analysis style SBA described here only handles `case-lambda`'s whose program text is known. The analysis needs to be extended to handle primitives for which the code is unknown. We have therefore begun work on extending the analysis to handle primitives described only by types. These types are used to generate sets of constraints that simulate the behavior of the corresponding primitive when that primitive appears in the analyzed program. This approach works well for most primitives in R5RS Scheme [17], with a few exceptions. For example the `map` primitive requires that it be given exactly as many lists as the mapped function takes arguments. Expressing this dependency would require a dependent type, which is beyond what our type language can currently express. The `map` primitive is therefore conservatively approximated by handling only arguments up to a fixed number and flagging a possible error beyond that limit. Similarly, detecting arity errors when using the `apply` primitive requires knowing the length of the list given as `apply`'s last argument. When statically determining that length is not possible the analysis will conservatively flag a possible arity error. This is similar to what MrSpidey does for such primitives, and the same technique could be used as well for the annotated selector analysis.

We are also currently working on extending this kind of type specification technique to handle flows between modules, using the type specifications to simulate imported procedures. This approach would yield a true separate analysis.

10. Conclusions

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

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

Acknowledgements

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

References

1. Aiken, A.: 1999, 'Introduction to Set Constraint-Based Program Analysis'. *Science of Computer Programming* **35**, 79–111.
2. Aiken, A., E. L. Wimmers, and T. K. Lakshman: 1994, 'Soft Typing with Conditional Types'. In: *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. pp. 163–173.
3. Cormen, T. H., C. E. Leiserson, and R. L. Rivest: 1990, *Introduction to Algorithms*. Cambridge, MA/New York: MIT Press/McGraw-Hill.
4. Cousot, P. and R. Cousot: 1977, 'Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints'. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 238–252.
5. Dybvig, R. K.: 1998, *Chez Scheme User's Guide*. Cadence Research Systems.
6. Dybvig, R. K. and R. Hieb: 1990, 'A New Approach to Procedures with Variable Arity'. *Lisp and Symbolic Computation* **3**, 229–244.
7. Dzeng, H. and C. T. Haynes: 1994, 'Type Reconstruction for Variable-Arity Procedures'. In: *Proc. ACM Conf. on Lisp and Functional Programming*. pp. 239–249.
8. Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen: 2002, 'DrScheme: A Progamming Environment for Scheme'. *Journal of Functional Programming* **12**(2), 159–182.
9. Flanagan, C.: 1995, 'MrSpidey: Static Debugger Manual'. Rice University.
10. Flanagan, C.: 1997, 'Effective Static Debugging via Componential Set-Based Analysis'. Ph.D. thesis, Rice University.
11. Flanagan, C. and M. Felleisen: 1999, 'Componential Set-Based Analysis'. *ACM Trans. on Programming Languages and Systems* **21(2)**, 369–415.
12. Flatt, M.: 2000, 'MzScheme: Language Reference Manual'. Rice University. Version 103.
13. Heintze, N.: 1992, 'Set Based Program Analysis'. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.
14. Heintze, N.: 1994, 'Set-based analysis of ML programs'. In: *Proceedings of the 1994 ACM conference on LISP and functional programming*. pp. 306–317.
15. Heintze, N. and D. McAllester: 1997a, 'Linear-time Subtransitive Control Flow Analysis'. In: *Proc. 1997 ACM Conference on Programming Language Design and Implementation (PLDI '97)*. pp. 261–272.
16. Heintze, N. and D. McAllester: 1997b, 'On The Cubic Bottleneck in Subtyping and Flow Analysis'. In: *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS '97)*. pp. 342–351.
17. Kelsey, R., W. Clinger, and J. Rees, eds.: 1998, *Revised^5 Report on the Algorithmic Language Scheme*.

18. McAllester, D. and N. Heintze: 1997, 'On the Complexity of Set-Based Analysis'. In: *1997 International Conference on Functional Programming*.
19. Palsberg, J.: 1995, 'Closure analysis in constraint form'. *Proc. ACM Trans. on Programming Languages and Systems* **17**(1), 47–62.
20. Palsberg, J. and M. I. Schwartzbach: 1994, *Object-Oriented Type Systems*, Wiley Professional Computing. Chichester: Wiley.
21. Sestoft, P.: 1988, 'Replacing Function Parameters by Global Variables'. Master's thesis, DIKU, Univ. of Copenhagen.
22. Shivers, O.: 1991, 'The semantics of Scheme control-flow analysis'. In: *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. pp. 190–198.
23. Wand, M. and G. B. Williamson: 2002, 'A Modular, Extensible Proof Method for Small-step Flow Analyses'. In: D. Le Métayer (ed.): *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Proceedings*, Vol. 2305 of *Lecture Notes in Computer Science*. pp. 213–227.