Modular Set-Based Analysis from Contracts

Philippe Meunier
College of Computer and Information Science, Northeastern University
meunier@ccs.neu.edu

Robert Bruce Findler
Department of Computer Science, University of Chicago
robby@cs.uchicago.edu

Matthias Felleisen
College of Computer and Information Science, Northeastern University
matthias@ccs.neu.edu

Abstract

In PLT Scheme, programs consist of modules with contracts. The latter describe the inputs and outputs of functions and objects via predicates. A run-time system enforces these predicates; if a predicate fails, the enforcer raises an exception that blames a specific module with an explanation of the fault.

In this paper, we show how to use such module contracts to turn set-based analysis into a fully modular parameterized analysis. Using this analysis, a static debugger can indicate for any given contract check whether the corresponding predicate is always satisfied, partially satisfied, or (potentially) completely violated. The static debugger can also predict the source of potential errors, i.e., it is sound with respect to the blame assignment of the contract system.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.2.4 [*Software / Program Verification*]: Programming by contract

General Terms Languages, Reliability, Verification.

Keywords Static Debugging, Set-based Analysis, Modular Analysis, Runtime Contracts.

1. Modules, Contracts, and Static Debugging

A static debugger helps programmers find errors via program analyses. It uses the invariants of the programming language to analyze the program and determines whether the program may violate one of them during execution. For example, a static debugger can find expressions that may dereference null pointers. Some static debuggers use lightweight analyses, e.g., Flanagan et al.'s MrSpidey [11] relies on a variant of set-based analysis [10, 16, 21]; others use a deep abstract interpretation, e.g., Bourdoncle's Syntox [4]; and yet others employ theorem proving, e.g., Detlefs et al.'s ESC [7].

Experience with static debuggers shows that they work well for reasonably small programs. Using MrSpidey, we have routinely debugged or re-engineered programs of 2,000 to 5,000 lines of code in PLT Scheme. Flanagan has successfully analyzed the core of the interpreter, dubbed MrEd [13], a 40,000 line program. Existing static debuggers, however, suffer from a monolithic approach to program analysis. Because their analyses require the availability of the entire program, programmers cannot analyze their programs until they have everyone else's modules.

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

Over the past few years, we have added a first-order module system to PLT Scheme [12] and have equipped the module system with a contract system [8]. A contract is roughly a predicate on the inputs and outputs of (exported) functions, including object methods and higher-order functions. The contract system monitors the contracts during program execution. If a module violates a contract, the contract system pinpoints the guilty party and issues an explanatory message.

This paper makes five contributions to static debugging and software contracts. *First*, it explains how to construct a modular static debugger for programs with contracts, using those contracts in a dual role: one as a source of abstract values and one as a sink for abstract values. *Second*, we prove that our contract-based, whole-program analysis computes its results in a modular manner. That is, our contract-aware set-based analysis produces the same predictions for a given point in the program regardless of whether it analyzes the whole program or just the surrounding module. *Third*, for any given contract check, the system indicates whether the corresponding predicate is always satisfied, partially satisfied, or completely violated. *Fourth*, the static debugger can also predict the source of potential errors, i.e., it is sound with respect to the blame assignment of the contract system. *Fifth*, the analysis is parameterized over both a predicate approximation relation and a predicate domain function.

2. Overview

The paper presents a model of a modular static debugger. The model consists of two parts: a runtime contract system and a set-based analysis for modules with contracts. A correctness theorem ties the two parts together. Figure 1 provides an overview of these three pieces in graphical form. The vertical column on the left represents the runtime contract system. A contract compiler translates a collection of modules and a main expression into a suitably annotated form. During execution, which we naturally model via a reduction system, the contract system keeps track of the contract obligations; if something goes wrong it blames a specific module.

The first horizontal row of Figure 1 depicts the analysis process, which consists of three stages. First, it partitions the program into module-like pieces by lifting expressions with contract annotations out of the main program. Second, the resulting collection of program pieces is analyzed with a parameterized set-based analysis. This step yields both sets of abstract values and sets of potential errors, including explanations that blame the guilty party; we call the latter *blame sets*. Third, the former are summarized as set-of-values descriptions, dubbed *types*.

The rest of the grid in Figure 1 explains our proof technique for the correctness theorem. Since each reduction step creates a complete program, the correctness proof can proceed via subject reduction. We re-apply the analysis after each reduction step. The proof then shows that the reductions preserve the types and the blame

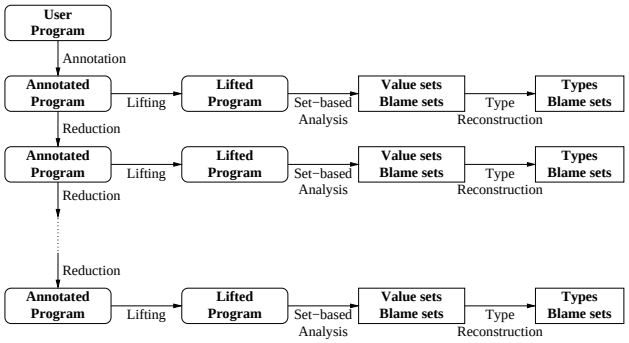

Figure 1. Contract system and analysis overview.

sets. It follows that the predictions of the analysis are conservative. Our desire to use a subject reduction proof heavily influences the details of the reduction semantics and the analysis rules.

Finally, after we establish the soundness of the analysis, we also explain precisely what we mean with "modular analysis" and state and prove a precise theorem.

3. Contract Calculus

In the first subsection, we recall the basics of Findler and Felleisen's contract system [8] with an example. In the second subsection, we introduce our surface syntax and internal syntax of programs. In the third subsection, we explain the translation from surface syntax into internal syntax.

3.1 Sample Contracts, Sample Blame

Let us first illustrate the module and contract system at work. Figure 2 shows an excerpt from our library for preparing figures (including Figure 2 itself). The **Find** module provides a family of functions that find the positions of pictures inside other pictures. Each of these functions accepts a main picture and a secondary picture inside the main picture; each produces a pair of integers indicating where the secondary picture occurs in the outer picture. For example, *ct-find* identifies the *c*enter *t*op coordinates of the embedded picture. The **Connect** module exports a function that accepts two of the functions in **Find** and produces a function that adds an arrow between sub-pictures. Finally, the **Composition** module combines the two other modules, i.e., it instantiates *connect* with *cb-find* and *ct-find*.

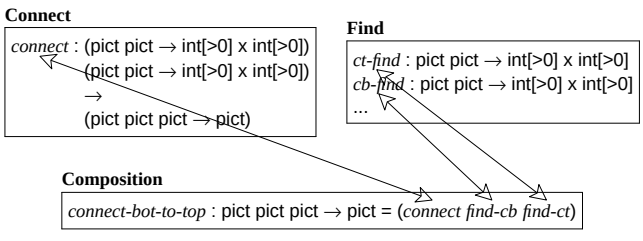

Figure 2. Example modules.

The arrows between the modules indicate which contracts bind which parties. First, consider the connections between **Composition** and **Find**. The contract on *ct-find* dictates that it should only receive pictures and produce integers larger than zero. Accordingly, if **Composition** passes to *ct-find* values other than pictures, it is to be blamed for the contract violation; similarly, if **Find** returns negative integers, it is to be blamed. But, **Composition** does not invoke the functions. Instead, it passes them to **Connect** and that interaction is governed by the contract between **Connect** and **Composition**. Thus, when *connect* invokes its argument functions, it too must call them on pictures and it too expects non-negative integers.

Now imagine that *ct-find* in **Find** returns negative numbers. This failure is only discovered when *connect* in **Connect** applies *ct-find* to two pictures. To determine which party is guilty, the monitoring code must trace the connections between the modules back to **Find** to blame *ct-find*. While computing the backtrace is obvious in this example, higher-order functions (and objects) can greatly obscure the connections in large programs where it is especially important to find the guilty party.

3.2 User Syntax and Annotated Syntax

Figure 3 specifies the surface syntax of our model language, where f is a module-defined variable, n is a number, and x is a lexical variable. To create a manageable model, we make several simplifying assumptions. First, since Findler and Felleisen [8]'s model explains contracts in a typed context, we omit types here because they would only clutter our work with unnecessary details. Second, each module defines and exports a single variable along with a contract; the defined variable stands for a value; it is uniquely named throughout the program; and it is automatically visible everywhere. Third, programs are closed terms and consist of a sequence of modules followed by a single expression. Fourth, the test part of an if0 expression can return any value; the "then" branch is evaluated if this value is 0.

The language of contracts uses just four kinds of constructs: one construct for validating that a value is an integer, which shows how the model deals with basic types; one construct for validating any value; a third construct for validating that a value is a function; and a fourth construct to use arbitrary expressions as contracts. For example the "positive integer" contracts in Figure 2 restrict the int contract in our surface syntax. Each occurrence of int[$>$0] would be expressed as (pred positive?) in the surface syntax, assuming the predicate positive? had been defined somewhere. Unlike arrow contracts, pred is *not* a constructor that contains other contracts; it uses plain expressions to create a contract.

$$
\begin{array}{lcl}
P & ::= & E \mid MP \\
M & ::= & (\text{module } f\ C\ V) \\
V & ::= & n \mid (\lambda x.E) \\
E & ::= & V \mid x \mid f \mid (E\ E) \mid (\text{if0 } E\ E\ E) \\
C & ::= & \text{int} \mid \text{any} \mid (C \rightarrow C) \mid (\text{pred } E)
\end{array}
$$

Figure 3. Surface syntax.

$$
\begin{array}{lcl}
P & ::= & E \mid MP \\
M & ::= & (\text{module } f^{\beta}\ V)^{\ell} \\
V & ::= & n^{\ell}_{E\ldots} \mid (\lambda x^{\beta}.E)^{\ell}_{E\ldots} \\
 & & \mid ((C \dashrightarrow C)^{\ell\ell'}_{f} \Leftarrow V)^{\ell_c} \\
E & ::= & V \mid x^{\beta} \mid f^{\beta} \mid (E\ E)^{\ell} \mid (\text{if0 } E\ E\ E)^{\ell} \\
 & & \mid (C \Leftarrow E)^{\ell} \mid (\text{blame } f\ S)^{\ell} \mid \varepsilon^{\ell} \\
C & ::= & \text{int}^{\ell\ell'}_{f} \mid \text{any}^{\ell\ell'}_{f} \mid (C \rightarrow C)^{\ell\ell'}_{f} \\
 & & \mid (C \dashrightarrow C)^{\ell\ell'}_{f} \mid \langle E\ E\ C \rangle^{\ell\ell'}_{f} \\
S & ::= & \mathcal{O} \mid \mathcal{R}
\end{array}
$$

Figure 4. Annotated syntax.

A program in the surface syntax is ill-suited for monitoring contracts and for analyzing them. We therefore elaborate such programs into the internal syntax of Figure 4. This syntax contains labeled versions of all syntactic phrases—β for labels on variables and ℓ for all others—and new forms that are better suited for our soundness proof than those of the surface syntax.

The major new expression forms are $(C \Leftarrow E)$ and (blame f S). The former evaluates the expression E to a value and checks whether the value satisfies the contract C. The latter form aborts the program and blames a specific module or the main expression (μ) for violating a contract. Such violations have two possible severity levels: $\mathcal{R}$ed for violating a basic integer or arrow contract, and $\mathcal{O}$range for violating a user-provided predicate. Integers and closures have extra subscript annotations to represent contract predicates that they have satisfied. The ε form is a technical device.

The additional contract forms are $(C \dashrightarrow C)_f$ and $\langle E\ E\ C \rangle_f$. We refer to the first as a "blessed" arrow contract and the second as a contract triple. A blessed arrow contract denotes a partially validated contract. It is used when the run-time system has confirmed that a value is a procedure but has yet to confirm that the procedure satisfies the domain and range checks. The contract triple replaces the (pred E) contract. Its first expression turns the predicate into a runtime check; its second expression is the predicate; and the last part is the contract that describes the domain of the predicate. The first is used with the semantics and the soundness proof; the second and third are necessary for the analysis proper.

Consider the following example:

$$\begin{array}{l}(\text{module } f\ (\text{int}\rightarrow\text{int})\ (\lambda x.x))\\(f\ 3)\end{array}$$

The annotation of this program yields the following:

$$\begin{array}{l}(\text{module } f^{\beta_1}\ (\lambda x^{\beta_2}.x^{\beta_2})^{\ell_\lambda})^{\ell_f}\\(((\text{int}_{\mu}^{\ell_1\ell_2}\rightarrow\text{int}_{f}^{\ell_3\ell_4})_{f}^{\ell_5\ell_6} \Leftarrow f^{\beta_1})^{\ell_c}\ 3^{\ell_n})^{\ell_a}\end{array}$$

In the annotated program, each subexpression (except for variables) has a unique label; each contract has two unique labels and a module name (or μ). Furthermore, the reference to the module variable f is wrapped with a contract check that ensures the module satisfies its contract.

3.3 Annotation

The rules of Figure 5 define the annotation process. The main goal is to annotate every expression with a unique label (except for variables) and every contract with two unique labels and a module name. These annotations are required by the analysis: the label on an expression represents the abstract values of that expression; the two labels on a contract represent the contract in its two roles as both a source (first label) and a sink (second label) of abstract values; and the module name on a contract is used to assign blame when the analysis detects a violation of that contract.

The judgement for annotating programs is of the form

$$\vdash^{\mathsf{a}}_{\mathsf{p}} p \rightsquigarrow p'$$

where p is the original program and p' is the annotated version. The PROGRAM rule builds two environments Δ and Γ, the first one mapping module names to contracts and the second one mapping variables to labels.

The judgement for modules is of the form

$$\Delta, \Gamma \vdash^{\mathsf{a}}_{\mathsf{m}} m \rightsquigarrow m'$$

where m' is the annotated version of module m. The MODULE rule removes the contract on the defined module variable and annotates the rest of the module. The remaining rules add the contract to references of the module variable.

The judgement for expressions is of the form

$$\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e \rightsquigarrow e'$$

where f is the name of the module (or μ) in which expression e appears and e' is the annotated version of e. Variable references share their label with their respective binder (rules VAR and MODVAR).

$$\frac{\Delta, \Gamma \vdash^{\mathsf{a}}_{\mathsf{m}} m_i \rightsquigarrow m_i' \quad \Delta, \Gamma, \mu \vdash^{\mathsf{a}}_{\mathsf{e}} e \rightsquigarrow e' \quad \text{where } \Delta \stackrel{def}{=} [f_i \mapsto c_i, \ldots] \text{ and } \Gamma \stackrel{def}{=} [f_i \mapsto \beta_i, \ldots] \quad \text{given } m_i = (\text{module } f_i\ c_i\ v_i)}{\vdash^{\mathsf{a}}_{\mathsf{p}} m_i\ \ldots\ e \rightsquigarrow m_i'\ \ldots\ e'}\ (\textsc{Program})$$

$$\frac{\Gamma(f) = \beta \quad \Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} v \rightsquigarrow v'}{\Delta, \Gamma \vdash^{\mathsf{a}}_{\mathsf{m}} (\text{module } f\ c\ v) \rightsquigarrow (\text{module } f^{\beta}\ v')^{\ell}}\ (\textsc{Module})$$

$$\frac{}{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} n \rightsquigarrow n^{\ell}}\ (\textsc{Int}) \qquad \frac{\Delta, \Gamma[x \mapsto \beta], f \vdash^{\mathsf{a}}_{\mathsf{e}} e \rightsquigarrow e'}{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} (\lambda x.e) \rightsquigarrow (\lambda x^{\beta}.e')^{\ell}}\ (\textsc{Lam})$$

$$\frac{\Gamma(x) = \beta}{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} x \rightsquigarrow x^{\beta}}\ (\textsc{Var}) \qquad \frac{\Gamma(g) = \beta \quad \Delta(g) = c \quad \Delta, \Gamma, g, f \vdash^{\mathsf{a}}_{\mathsf{c}} c \rightsquigarrow c'}{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} g \rightsquigarrow (c' \Leftarrow g^{\beta})^{\ell}}\ (\textsc{ModVar})$$

$$\frac{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e_1 \rightsquigarrow e_1' \quad \Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e_2 \rightsquigarrow e_2'}{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} (e_1\ e_2) \rightsquigarrow (e_1'\ e_2')^{\ell}}\ (\textsc{App})$$

$$\frac{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e_0 \rightsquigarrow e_0' \quad \Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e_1 \rightsquigarrow e_1' \quad \Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e_2 \rightsquigarrow e_2'}{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} (\text{if0 } e_0\ e_1\ e_2) \rightsquigarrow (\text{if0 } e_0'\ e_1'\ e_2')^{\ell}}\ (\textsc{If0})$$

$$\frac{}{\Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} \text{int} \rightsquigarrow \text{int}_{f}^{\ell\ell'}}\ (\textsc{IntC}) \qquad \frac{}{\Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} \text{any} \rightsquigarrow \text{any}_{f}^{\ell\ell'}}\ (\textsc{AnyC})$$

$$\frac{\Delta, \Gamma, g, f \vdash^{\mathsf{a}}_{\mathsf{c}} c_d \rightsquigarrow c_d' \quad \Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} c_r \rightsquigarrow c_r'}{\Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} (c_d \rightarrow c_r) \rightsquigarrow (c_d' \rightarrow c_r')_{f}^{\ell\ell'}}\ (\textsc{ArrowC})$$

$$\frac{\Delta, \Gamma, f \vdash^{\mathsf{a}}_{\mathsf{e}} e \rightsquigarrow e' \quad \Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} \mathcal{D}_{\Delta}((\text{pred } e)) \rightsquigarrow c' \quad e'' \stackrel{def}{=} \mathcal{F}(e', \mathit{lab}^{+}(c'), f)}{\Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} (\text{pred } e) \rightsquigarrow \langle e''\ e'\ c' \rangle_{f}^{\ell\ell'}}\ (\textsc{PredC})$$

Figure 5. Annotation judgments.

Additionally, references to module variables are wrapped with a contract check for the contract that was associated with the variable's definition (rule MODVAR). Module variables that are not referenced in a program are therefore not checked against their contract, i.e., putting contracts on dead code has no effect.

Finally, the judgement for contracts is of the form

$$\Delta, \Gamma, f, g \vdash^{\mathsf{a}}_{\mathsf{c}} c \rightsquigarrow c'$$

where c' is the annotated version of the contract c. The two module names f and g represent the two parties that agreed to the contract c. One is the name of the module variable that uses c in its contract; the other is the name of the module where that variable is used. Which of f and g corresponds to which of those two names varies. The two names switch positions when the annotation process traverses a domain position in a functional contract (rule ARROWC). The rules ensure that every part of a contract that appears in contravariant position is annotated with the name of the module currently analyzed. This mirrors Findler and Felleisen [8]'s rule for assigning blame in the presence of higher-order functions.

Annotating contracts is otherwise straightforward, except that contracts of the form (pred e) are translated into triples of the form

$$\langle \mathcal{F}(e', \mathit{lab}^{+}(c'), f)\quad e'\quad c' \rangle_{f}^{\ell\ell'}$$

according to rule PREDC:

- The expression e' is the annotated version of e;

- The contract c' is the annotated version of $\mathcal{D}_\Delta((\text{pred } e))$. The function $\mathcal{D}_\Delta$ computes an approximation of the domain of predicate e and represents it as a contract. By construction, that contract does not contain any sub-contracts of the form $(\text{pred } E)$ and can therefore be used as a simple contract that approximates the complex predicate e.
- $\mathcal{F}(e', \mathit{lab}^+(c'), f)$ generates boilerplate code that represents the application of the predicate to a value in a schematic manner. The lab^+ function returns the first one of the two labels of its contract argument.

The creation of a triple is necessary for the analysis, which needs to know the program's syntax, especially e' and c'. It uses these terms to determine whether a contract violation is partial—orange: a value satisfies the simple contract c' but not the extra predicate e'—or full—red: a value does not even satisfy the contract c'.

The creation of the boilerplate code is only needed for the soundness proof, which is based on the preservation of labels and that no new labels are introduced throughout the reduction process. Since the analysis requires labels on all expressions, the reductions must not introduce terms that do not re-use existing labels. The boilerplate code and its labels are therefore generated during the annotation phase so that it can be used at an opportune time during the reduction process.

Let's take a closer look at the actual code:

$$\mathcal{F}(e, \ell, f) \stackrel{\mathit{def}}{=} (\text{if0 } (e\ \varepsilon^{\ell})^{\ell_0}\ \varepsilon^{\ell}\ (\text{blame } f\ \mathcal{O})^{\ell_1})^{\ell_2}$$

with ℓ_0 through ℓ_2 fresh. The εs are (non-variable) placeholders for expressions with the same label; they are never evaluated directly. Specifically, ε stands either for a runtime value (during the reduction process) or for a contract representing an abstract value (during the analysis).

From the runtime perspective, the code means that a predicate represented by e is applied to the runtime value represented by ε and the result is checked by the if0 expression. If the predicate does not accept the runtime value, then the if0 expression reduces to a blame expression. The severity of the contract violation is orange, since a user-provided contract is broken. If the predicate accepts the runtime value, the runtime value is simply returned through the second ε expression.

From the analysis perspective, the same code means that a predicate represented by e is applied to the abstract values flowing into ε and the result is checked by the if0 expression. The analysis then conservatively assumes that both branches of the if0 can be taken at runtime and therefore makes the abstract values flow out of the ε expression in the "then" branch and adds the name f to the blame set of ℓ_1 in the "else" branch.

The role of c' in the generated triple is to act as an abstract value simulating the set of all possible values that might satisfy the predicate e' at runtime. A conservative approximation of this set is the domain of the predicate itself, which is computed by $\mathcal{D}_\Delta$ (Fig. 6). Since we do not want to represent the domain of a predicate using another predicate, the function $\mathcal{D}_\Delta$ needs to approximate the domain of a predicate with a contract that uses only the int and $\rightarrow$ contract constructors. The only interesting cases in that definition are therefore the first two:

- If $\mathcal{D}_\Delta$ is applied to a contract of the form $(\text{pred } f)$ (where f is a module variable name), f is looked up in the contract environment Δ; the resulting contract is itself processed by $\mathcal{D}_\Delta$ to recursively eliminate all the pred forms from it; and, if the resulting contract is an arrow contract, the domain of that arrow contract is returned. If the resulting contract is not an arrow contract, then the program is trying to use as a predicate an expression that is not a function.[1] That kind of program is simply rejected by the annotator.
- If $\mathcal{D}_\Delta$ is applied to a contract of the form $(\text{pred } e)$, $\mathcal{D}_\Delta$ returns any. In this case, an expression proper is used as a predicate. It is the programmer's responsibility to ensure that the expression evaluates to a function and that this function can accept any value as input.

$$\begin{array}{ll}
\mathcal{D}_\Delta((\text{pred } f)) & \stackrel{\mathit{def}}{=} c_d \text{ when } \mathcal{D}_\Delta(\Delta(f)) = (c_d \rightarrow c_r) \\
\mathcal{D}_\Delta((\text{pred } e)) & \stackrel{\mathit{def}}{=} \text{any} \\
\mathcal{D}_\Delta(\text{int}) & \stackrel{\mathit{def}}{=} \text{int} \\
\mathcal{D}_\Delta(\text{any}) & \stackrel{\mathit{def}}{=} \text{any} \\
\mathcal{D}_\Delta((c_d \rightarrow c_r)) & \stackrel{\mathit{def}}{=} (\mathcal{D}_\Delta(c_d) \rightarrow \mathcal{D}_\Delta(c_r))
\end{array}$$

Figure 6. Predicate domain function.

Consider for example the following program fragment:

$$\begin{array}{l}
(\text{module } \mathit{prime?}\ (\text{int}\rightarrow\text{int})\ \ldots) \\
(\text{module } f\ (\text{pred } \mathit{prime?})\ 3) \\
f
\end{array}$$

The annotated version has this general form (with many annotations omitted for clarity):

$$\begin{array}{l}
(\text{module } \mathit{prime?}^{\beta_1}\ \ldots) \\
(\text{module } f^{\beta_2}\ 3) \\
(\langle(\text{if0 } (\mathit{prime?}^{\beta_1}\ \varepsilon^{\ell})\ \varepsilon^{\ell}\ (\text{blame } f\ \mathcal{O}))\quad \mathit{prime?}\quad \text{int}_f^{\ell\ell'}\rangle \Leftarrow f^{\beta_2})
\end{array}$$

The annotated code checks the variable reference f^{β_2} against a contract triple. The first part of the triple is an if0 expression that simulates applying the *prime?* predicate to a value and checking whether the predicate is satisfied or not. The second part of the triple is the (name of the) predicate itself. The third part is a basic integer contract that approximates the *prime?* predicate; i.e., to be a prime number, a given value has to be an integer. That integer contract is the result of computing the domain of the *prime?* predicate using $\mathcal{D}_\Delta$ applied to the *prime?* predicate's own contract $(\text{int}\rightarrow\text{int})$. The resulting int contract is then annotated to get the $\text{int}_f^{\ell\ell'}$ contract used in the triple. That contract shares its first label ℓ with the ε^{ℓ} expressions in the if0 part of the triple.

Once a program has been completely annotated it can then be either reduced to a value (if it has one) or analyzed. The two processes are the subject of the next two sections.

4. Reduction Rules

Figure 7 defines the reduction semantics for annotated programs. The goal of the process is to reduce the main expression to a value in the module context. The relation $\longrightarrow$ is the one-step reduction; the set of evaluation contexts for expressions is:

$$\mathcal{E} \stackrel{\mathit{def}}{=} [\,] \mid (\mathcal{E}\ e)^{\ell} \mid (v\ \mathcal{E})^{\ell} \mid (\text{if0 } \mathcal{E}\ e\ e)^{\ell} \mid (C \Leftarrow \mathcal{E})^{\ell}$$

Expression evaluation contexts do not include contexts for contracts and in particular not for contract triples. Expressions inside a contract triple are only evaluated after the surrounding contract check has been reduced. The grammar for annotated programs guarantees that contracts never show up outside a contract check.

[1] In an actual static debugger the function $\mathcal{D}_\Delta$ would also check that there are no reference loops among the contracts for predicates (e.g. trying to define the contract for a predicate using the predicate itself). We omit this check here to simplify our model.

$$
\begin{array}{lcll}
((\lambda x^{\beta}.e)^{\ell_\lambda}\ v^{\ell_v})^{\ell_a} & \longrightarrow & e[v^{\ell_v}/x^{\beta}] & \textsc{subst} \\
(n^{\ell_n}\ v^{\ell_v})^{\ell_a} & \longrightarrow & (\text{blame } \lambda\ \mathcal{R})^{\ell_a} & \textsc{app-error} \\
(\text{if0}\ 0^{\ell_0}\ e_1\ e_2)^{\ell} & \longrightarrow & e_1 & \textsc{if0-true} \\
(\text{if0}\ v^{\ell_v}\ e_1\ e_2)^{\ell} & \longrightarrow & e_2 & \textsc{if0-false} \\
(\text{any}_f^{\ell\ell'} \Leftarrow v^{\ell_v})^{\ell_c} & \longrightarrow & v^{\ell} & \textsc{any} \\
(\langle e_1\ e_2\ \text{any}_f^{\ell\ell'}\rangle_f^{\ell^+\ell^-} \Leftarrow v_{e\ldots}^{\ell_v})^{\ell_c} & \longrightarrow & e_1[v_{e\ldots e_2}^{\ell}/\varepsilon^{\ell}] & \textsc{any-trip} \\
(\text{int}_f^{\ell\ell'} \Leftarrow n^{\ell_n})^{\ell_c} & \longrightarrow & n^{\ell} & \textsc{int-int} \\
(\text{int}_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \longrightarrow & (\text{blame } f\ \mathcal{R})^{\ell'} & \textsc{int-lam} \\
(\langle e_1\ e_2\ \text{int}_f^{\ell\ell'}\rangle_f^{\ell^+\ell^-} \Leftarrow n_{e\ldots}^{\ell_n})^{\ell_c} & \longrightarrow & e_1[n_{e\ldots e_2}^{\ell}/\varepsilon^{\ell}] & \textsc{int-trip-int} \\
(\langle e_1\ e_2\ \text{int}_f^{\ell\ell'}\rangle_f^{\ell^+\ell^-} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \longrightarrow & (\text{blame } f\ \mathcal{R})^{\ell'} & \textsc{int-trip-lam} \\
((c_1 \rightarrow c_2)_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \longrightarrow & ((c_1 \dashrightarrow c_2)_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \textsc{lam-lam} \\
((c_1 \rightarrow c_2)_f^{\ell\ell'} \Leftarrow n^{\ell_n})^{\ell_c} & \longrightarrow & (\text{blame } f\ \mathcal{R})^{\ell'} & \textsc{lam-int} \\
(\langle e_1\ e_2\ (c_1 \rightarrow c_2)_f^{\ell\ell'}\rangle_f^{\ell^+\ell^-} \Leftarrow \vec{v}_{e\ldots}^{\ell_v})^{\ell_c} & \longrightarrow & e_1[((c_1 \dashrightarrow c_2)_f^{\ell\ell'} \Leftarrow \vec{v}_{e\ldots e_2}^{\ell_v})^{\ell_c}/\varepsilon^{\ell}] & \textsc{lam-trip-lam} \\
(\langle e_1\ e_2\ (c_1 \rightarrow c_2)_f^{\ell\ell'}\rangle_f^{\ell^+\ell^-} \Leftarrow n^{\ell_n})^{\ell_c} & \longrightarrow & (\text{blame } f\ \mathcal{R})^{\ell'} & \textsc{lam-trip-int} \\
(((c_1 \dashrightarrow c_2)_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c}\ w^{\ell_w})^{\ell_a} & \longrightarrow & (c_2 \Leftarrow (\vec{v}^{\ell_v}\ (c_1 \Leftarrow w^{\ell_w})^{lab^+(c_1)})^{lab^-(c_2)})^{lab^+(c_2)} & \textsc{split}
\end{array}
$$

Figure 7. Reduction rules.

The module context becomes relevant in only one situation:

$$
\begin{array}{lll}
 & \ldots(\text{module } f^{\beta}\ v)^{\ell}\ldots\mathcal{E}[f^{\beta}] & \\
\longrightarrow & \ldots(\text{module } f^{\beta}\ v)^{\ell}\ldots\mathcal{E}[v] & \textsc{lookup}
\end{array}
$$

The LOOKUP rule replaces a reference to a module variable with its value. Since all module-defined variable references are wrapped with contract checks during the annotation phase, a contract check now surrounds the value v.

In Figure 7 we use n to represent runtime integers, $\vec{v}$ to represent functions or functions with any number of blessed arrow contract checks wrapped around them, and v and w to represent any values whatsoever. When necessary we write $v_{e\ldots}$ for a value v that satisfies all the predicates e, etc. Finally to simplify the exposition we decide that a blame redex in any context reduces the entire program in one step to just that expression, whereupon reduction stops.

The SUBST rule is the usual β_v relation for function calls. Substitution replaces both the variable x and its label β with the value v and its label ℓ_v. The IF0-TRUE and IF0-FALSE rules are also the usual ones for conditional expressions. The APP-ERROR rule blames the programmer (represented as λ) when the program attempts to use an integer as a function, i.e., when the programmer does not violate a contract but abuses the programming language.[2]

The rest of the reduction rules concern contract checking:

- The ANY rule shows a contract check that checks nothing. The check reduces to the tested value. Importantly, the label ℓ on any becomes the label on v. The reason is that in the analysis, label ℓ acts as an abstract value source for the contract $\text{any}_f^{\ell\ell'}$. The reduction rule thus guarantees that the value v has the same label as the abstract source it replaces, which is the key to the relevant step in the soundness proof of the analysis.
- The ANY-TRIP rule is similar to the previous one, except that it deals with a triple. The rule takes the boilerplate code from the first part of the triple and replaces the ε expressions with the value v, again after an appropriate label change on v. The result of these substitutions is code that checks whether the value satisfies the triple's predicate or not. The expression e_2 does not play any active role during the reduction but is added to the set of predicates satisfied by v (again for the purpose of the soundness proof).
- The INT-INT and INT-LAM rules check that a given value is an integer. If it is, the INT-INT behaves just like the ANY one. If it is not, the INT-LAM blames the appropriate module. The label of the blame expression is the second label on the contract: ℓ' acts as an abstract value sink during the analysis. The severity level of the contract violation is red since a basic contract has been broken.
- INT-TRIP-INT and INT-TRIP-LAM correspond to INT-INT and INT-LAM but cope with triples. When the tested value is an integer, the evaluation of the triple requires a substitution to occur, similarly to what happens in the ANY-TRIP rule. In the INT-TRIP-LAM rule the color of the violation is again red since a basic contract has been broken. In essence the contract system is able to show that the value $\vec{v}$ does not satisfy the predicate e_2 simply by looking at the contract int that approximates e_2.
- LAM-LAM, LAM-INT, LAM-TRIP-LAM, and LAM-TRIP-INT correspond to the rules INT-INT, INT-LAM, INT-TRIP-INT, and INT-TRIP-LAM, respectively. The only difference is the presence of blessed arrows in the LAM-LAM and LAM-TRIP-LAM rules: once a value has been checked to be a function, we still need to check that the function's argument or the function's result do not break their respective parts of the contract. It is impossible to check these contracts now because the function might be applied only much later [8]. Hence, the two rules LAM-LAM and LAM-TRIP-LAM introduce a blessed arrow contract check around the function, indicating that the arrow check has succeeded but that the argument and result of the function still remain to be checked. If the function already had blessed arrow contract checks wrapped around it, it now has one more.
- The SPLIT rule breaks a blessed arrow contract into its domain and range contracts. It distributes those to the actual argument of the function and to the result of the whole application, respectively. This is how a higher-order contract is, step by step,

[2] This check is representative of the language designer's power to restrict primitive operations (such as function application, array indexing, etc.) Put differently, it represents the implicit contract between the programmer and the language designer.

transformed into a series of flat contracts [8]. When a function has multiple blessed arrow contract checks wrapped around it, this rule also ensures that the multiple domain contracts are checked outside-in and the multiple range contracts are checked inside-out. This in turn ensures that blame is correctly assigned when one of the domain or range contracts is violated. Since one contract check is replaced by two smaller ones and all expressions have to be labeled, there is seemingly a need for more labels in the contractum than in the redex. However by using the lab^+ and lab^- functions we can share labels between the appropriate terms and avoid the introduction of fresh labels, which would break the soundness proof of the analysis.

Together the annotation and reduction processes ensure that a contract check is always present at the interface between expressions that come from different modules, regardless of how far the reduction process has progressed. This invariant is essential for the modularity of the analysis.

5. The Analysis

Due to contracts, our analysis problem differs from the usual one. As a dynamic element, contracts add new behavior to programs. If a contract fails, the execution stops and the system issues a blame assignment. As a static element, contracts guarantee basic properties about the values that flow out of them; i.e., each contract separates a program into two pieces: those that send values into the contract and those that receive values from the contract. In short, contracts are simultaneously value sources and value sinks, and they naturally partition programs into (analysis) modules.

Based on this insight, we have designed a three-phase analysis. The first step is to lift contract checks out of their context and to leave just a copy of the contracts in their place. The resulting sequence of terms is roughly a modular program with a main expression. In the second step the analysis uses a parameterized algorithm to generate constraints from this program. Finally it produces types from a solution to these constraints.

This makes the analysis applicable to each stage in the reduction process, rendering it well-suited for a subject reduction argument.

5.1 Lifting

The lifting step splits an annotated program at contract boundaries. Each contract check $(c \Leftarrow e)^\ell$ is lifted to the top of the program; the remaining hole in the term is filled with the contract c. The duplication of the contract allows the analysis to separate its two roles. At the bottom of a term, the contract is a source of values, which means the analysis uses only its positive labels. At the top level, it is a value sink; the analysis uses only the negative labels.

Figure 8 illustrates the lifting process with two examples. In the upper left part of the figure, the white triangle represents the primary expression before the reduction process has started. It contains a grey triangle, which is a reference to a module variable. The oval between the two trees represents the module contract. Lifting produces two triangles: the white one, with just the contract where the grey term was located, and the grey one, with the original contract check at its top. Naturally, the grey one is just an (indirect) reference to the module that defines and exports the variable.

The lower row of the figure depicts the main expression after several reduction steps. The reduction steps copy terms and split up contracts. The result is, for example, that a single module reference can turn into numerous embedded terms with contracts. The triangles in the lower left of the figure depict such a term. Imagine that a function body under c_1 has been duplicated and applied once. The small white triangle under c_0 is the actual argument that was substituted into the function body. The lifting step for this reduced program produces four terms.

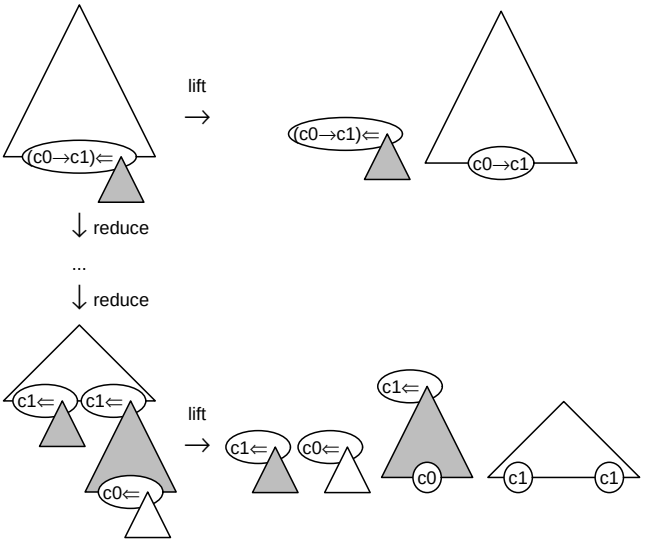

Figure 8. Lifting subtrees.

Figure 9 defines the lifting process. The four judgements are of the form

$$\vdash^{\mathtt{l}}_{\mathtt{t}} t \rightsquigarrow ts$$

where `t` is in `p`, `m`, `e`, and `c` (for programs, modules, expressions, and contracts respectively), t is the term to be lifted, and ts represents the resulting lifted trees.

Most of the rules defining the lifting process are structural rules that simply gather the terms resulting from lifting subterms and push all those terms to the program's top level in the right deterministic order. We omit most of those trivial rules from Figure 9, except for the PROGRAM, MODULE, LAM, and APP ones.

The only rule of interest is CHECK: a contract check $(c' \Leftarrow e')^\ell$ is lifted to the top and a copy c' of the contract takes its place in the tree currently being processed. For simplicity, we ignore the distinction between arrow contracts and their blessed counterparts (rule BARROWC).

Lifting occurs almost everywhere, including inside the first expression of triples (rule TRIPC). Since triples are dissolved during the reduction process and since the resulting expressions contribute to the final result (or blame), the analysis must predict which values flow from the first part of triples. It is unnecessary, however, to lift the third part of the triple because we know from the definition of $\mathcal{D}_\Delta$ that this component never contains any contract checks. The second part of the triple is not lifted either, because the analysis phase of the next section relies on this expression remaining in its original form.

5.2 Analysis

After the lifting step, programs satisfy additional syntactic invariants: see the grammar in Figure 10. This new grammar differs from the one in Figure 4 in three ways: (1) contracts are now expressions; (2) contract checks are no longer expressions and can only appear at the program's top-level, like module definitions; (3) blessed arrow contracts have disappeared.

The purpose of the analysis is to predict (1) the flow of values and (2) potential contract violations and violators. Accordingly, the analysis produces two results: a mapping φ from labels to sets of labels and a mapping ψ from labels to module variable names plus one of two "violation" colors. The former points to values in the program. The latter associates contract labels with modules that might violate the contract. If the associated color is orange ($\mathcal{O}$),

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{m}} m_i \rightsquigarrow es_i \ldots m_i' \quad \vdash^{\mathsf{l}}_{\mathsf{e}} e \rightsquigarrow es \ldots e'}{\vdash^{\mathsf{l}}_{\mathsf{p}} m_i \ldots e \rightsquigarrow es_i \ldots m_i' \ \ldots \ es \ldots e'} \ (\textsc{Program})$$

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{e}} v \rightsquigarrow es \ldots v'}{\vdash^{\mathsf{l}}_{\mathsf{m}} (\text{module } f^{\beta}\ v)^{\ell} \rightsquigarrow es \ldots (\text{module } f^{\beta}\ v')^{\ell}} \ (\textsc{Module})$$

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{e}} e \rightsquigarrow es \ldots e'}{\vdash^{\mathsf{l}}_{\mathsf{e}} (\lambda x^{\beta}.e)^{\ell}_{e_1 \ldots} \rightsquigarrow es \ldots (\lambda x^{\beta}.e')^{\ell}_{e_1 \ldots}} \ (\textsc{Lam})$$

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{e}} e_1 \rightsquigarrow es_1 \ldots e_1' \quad \vdash^{\mathsf{l}}_{\mathsf{e}} e_2 \rightsquigarrow es_2 \ldots e_2'}{\vdash^{\mathsf{l}}_{\mathsf{e}} (e_1\ e_2)^{\ell} \rightsquigarrow es_1 \ldots es_2 \ldots (e_1'\ e_2')^{\ell}} \ (\textsc{App})$$

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{c}} c \rightsquigarrow es_c \ldots c' \quad \vdash^{\mathsf{l}}_{\mathsf{e}} e \rightsquigarrow es \ldots e'}{\vdash^{\mathsf{l}}_{\mathsf{e}} (c \Leftarrow e)^{\ell} \rightsquigarrow es_c \ldots es \ldots (c' \Leftarrow e')^{\ell} c'} \ (\textsc{Check})$$

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{c}} (c_d \rightarrow c_r)^{\ell\ell'}_{f} \rightsquigarrow es}{\vdash^{\mathsf{l}}_{\mathsf{c}} (c_d \dashrightarrow c_r)^{\ell\ell'}_{f} \rightsquigarrow es} \ (\textsc{BArrowC})$$

$$\frac{\vdash^{\mathsf{l}}_{\mathsf{e}} e_1 \rightsquigarrow es_1 \ldots e_1'}{\vdash^{\mathsf{l}}_{\mathsf{c}} \langle e_1\ e_2\ c\rangle^{\ell\ell'}_{f} \rightsquigarrow es_1 \ldots \langle e_1'\ e_2\ c\rangle^{\ell\ell'}_{f}} \ (\textsc{TripC})$$

Figure 9. Lifting judgments.

$$\begin{array}{lcl}
P & ::= & E \mid MP \\
 & & \mid (C \Leftarrow E)^{\ell} P \\
M & ::= & (\text{module } f^{\beta}\ V)^{\ell} \\
V & ::= & n^{\ell}_{E \ldots} \mid (\lambda x^{\beta}.E)^{\ell}_{E \ldots} \\
E & ::= & V \mid x^{\beta} \mid f^{\beta} \mid (E\ E)^{\ell} \mid (\text{if0 } E\ E\ E)^{\ell} \\
 & & \mid C \mid (\text{blame } f\ S)^{\ell} \mid \varepsilon^{\ell} \\
C & ::= & \text{int}^{\ell\ell'}_{f} \mid \text{any}^{\ell\ell'}_{f} \mid (C \rightarrow C)^{\ell\ell'}_{f} \\
 & & \mid \langle E\ E\ C\rangle^{\ell\ell'}_{f} \\
S & ::= & \mathcal{O} \mid \mathcal{R}
\end{array}$$

Figure 10. Analyzed syntax.

a part of the contract has been satisfied; otherwise it is red ($\mathcal{R}$), meaning no part of the contract could be proved to be satisfied.

Our analysis extends and adapts techniques of 0CFA [23, 27] and SBA [10, 16, 21]. It is parameterized over a predicate approximation relation $\sqsubseteq$, and generates conditional constraints on the sets of labels and sets of errors that can show up at any given label. Any pair of mappings from labels to sets of labels and from labels to module variable names that satisfy these constraints is a sound approximation to the actual run-time behavior of the program. A minimal approximation is the solution.

The constraint generation algorithm needs to identify value sources and value sinks in programs. In the grammar for expressions value sources are syntactic values; numbers and abstractions are the only expressions that are sources. A value sink consumes values and triggers computations; applications are the primary value sinks among expressions.

As mentioned before, contracts play the role of both sources and sinks. Contracts that occur as leaves in an expression are sources; contracts inside of top-level checks are sinks. Because of this dual role, contracts have two labels: one represents the contract as a value source and the other as a value sink. Consider

$$\text{int}^{\ell^+\ell^-}_{f} \ .$$

The analysis uses ℓ^+ when it deals with the contract as an integer source and ℓ^- when it deals with it as an integer sink, i.e., for an integer contract check.

The matrix in Table 1 describes the essence of the constraint generation process. It explains how every possible combination of a source and a sink in the entire program generates constraints concerning the flow of values and blame assignment. The entries do not assume anything about the context in which a source or sink occurs. This implies that, for example, expressions inside contract triples are analyzed like any other expression.

The next two subsections explain the meaning of the constraints in Table 1, followed by a subsection presenting a few additional constraints that do not involve source-sink pairs.

5.2.1 Value Flow Constraints

Let us illustrate how to read Table 1 with some key examples. We start with the combination of λ-abstractions and applications because the form of the constraints should be familiar from the analysis of the simple lambda calculus [23]:

Source$\backslash$*Sink*	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$
$(\lambda x^{\beta}.e^{\ell})^{\ell_\lambda}_{e_1 \ldots}$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\beta)$ $\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell) \subseteq \varphi(\ell_a)$

The above specifies the creation of two constraints for every possible pair of an abstraction and an application in the program. The first constraint says that, if the abstraction (labeled ℓ_λ) flows into the application's function position (ℓ_5), then the arguments from the application (ℓ_6) flow into the abstraction's parameter (β). The second constraint has the same antecedent as the first and implies that the value of the abstraction's body (ℓ) flows into the result set for the function application (ℓ_a).

The second example juxtaposes two contracts:[3]

Source$\backslash$*Sink*	$\text{any}^{\ell_5^+\ell_5^-}_{h}$
$(c_g^{\ell_1^+\ell_1^-} \rightarrow c_f^{\ell_2^+\ell_2^-})^{\ell_3^+\ell_3^-}_{f}$	$\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_5^+) \subseteq \varphi(\ell_1^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_2^+) \subseteq \varphi(\ell_5^-)$

The first of those two constraints says that, if values represented by the function contract (labeled with ℓ_3^+) flows into the any check (ℓ_5^-), then that same any—represented as a value source (ℓ_5^+)—flows into the domain part of the arrow contract (ℓ_1^-).

To understand this flow from the any contract to the function's domain contract, remember that any represents the union of all data, including functions from any to any. This means that a value checked against any can turn out to be a function and can then potentially be applied to all sorts of values.[4] Naturally these values flow into the domain position of the arrow contract, which is similar to what happens in the cell that matches function contracts with function contracts in the bottom half of Table 1. The analysis must therefore check for such a possibility and ensure that the domain part of the arrow contract is coherent with receiving all possible values. The same argument for the function's range explains the second constraint above.

Of course, a practical debugger does not directly re-use the $\text{any}^{\ell_5^+\ell_5^-}_{h}$ contract to check the functional contract as well as its domain and range. Instead, it creates a new $(\text{any}^{\ell_5^+\ell_5^-}_{h} \rightarrow \text{any}^{\ell_5^+\ell_5^-}_{h})^{\ell\ell'}_{h}$

[3] To save space, the same cell in Table 1 shares its two constraints with the cell below as well as with the two cells on the right (each of which has itself a third constraint not shared with any other cell).

[4] At an abstract level this is analogous to Henglein's notion of a `Dynamic` $\rightsquigarrow$ (`Dynamic`$\rightarrow$`Dynamic`) coercion [17].

Source$\backslash$*Sink*	$\text{int}_h^{\ell_5^+\ell_5^-}$	$\langle \dots e_5\ \text{int}_h^{\ell_5^+\ell_5^-} \rangle_h^{\ell_6^+\ell_6^-}$	$\text{any}_h^{\ell_5^+\ell_5^-}$	$\langle \dots e_5\ \text{any}_h^{\ell_5^+\ell_5^-} \rangle_h^{\ell_6^+\ell_6^-}$
$n_{e_1\dots}^{\ell_n}$		$\left.\begin{array}{l}\{\ell_n\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{l}\{\ell_n\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\text{int}_f^{\ell_1^+\ell_1^-}$		$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\langle \dots e_1\ \text{int}_f^{\ell_1^+\ell_1^-} \rangle_f^{\ell_2^+\ell_2^-}$		$\left.\begin{array}{l}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{l}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\text{any}_f^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$			$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\langle \dots e_1\ \text{any}_f^{\ell_1^+\ell_1^-} \rangle_f^{\ell_2^+\ell_2^-}$				$\left.\begin{array}{l}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$(\lambda x^\beta.e^\ell)_{e_1\dots}^{\ell_\lambda}$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_5^+)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_5^-)$	$\left.\begin{array}{l}\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$(c_g^{\ell_1^+\ell_1^-}\to c_f^{\ell_2^+\ell_2^-})_f^{\ell_3^+\ell_3^-}$	$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_5^+)\subseteq\varphi(\ell_1^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_5^-)$	
$\langle \dots e_3\ (c_g^{\ell_1^+\ell_1^-}\to c_f^{\ell_2^+\ell_2^-})_f^{\ell_3^+\ell_3^-} \rangle_f^{\ell_4^+\ell_4^-}$				$\left.\begin{array}{l}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ e_3\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$

Source$\backslash$*Sink*	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$	$(c_i^{\ell_7^+\ell_7^-}\to c_h^{\ell_8^+\ell_8^-})_h^{\ell_5^+\ell_5^-}$	$\langle \dots e_5\ (c_i^{\ell_7^+\ell_7^-}\to c_h^{\ell_8^+\ell_8^-})_h^{\ell_5^+\ell_5^-} \rangle_h^{\ell_6^+\ell_6^-}$
$n_{e_1\dots}^{\ell_n}$	$\{\ell_n\}\subseteq\varphi(\ell_5)\Rightarrow\{\langle\lambda,\mathcal{R}\rangle\}\subseteq\psi(\ell_a)$	$\{\ell_n\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$	
$\text{int}_f^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\{\langle\lambda,\mathcal{R}\rangle\}\subseteq\psi(\ell_a)$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$	
$\langle \dots e_1\ \text{int}_f^{\ell_1^+\ell_1^-} \rangle_f^{\ell_2^+\ell_2^-}$			
$\text{any}_f^{\ell_1^+\ell_1^-}$			
$\langle \dots e_1\ \text{any}_f^{\ell_1^+\ell_1^-} \rangle_f^{\ell_2^+\ell_2^-}$			
$(\lambda x^\beta.e^\ell)_{e_1\dots}^{\ell_\lambda}$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_6)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_a)$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_7^+)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_8^-)$ $\left.\begin{array}{l}\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	
$(c_g^{\ell_1^+\ell_1^-}\to c_f^{\ell_2^+\ell_2^-})_f^{\ell_3^+\ell_3^-}$	$\{\ell_3^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_6)\subseteq\varphi(\ell_1^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_a)$	$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_7^+)\subseteq\varphi(\ell_1^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_8^-)$	
$\langle \dots e_3\ (c_g^{\ell_1^+\ell_1^-}\to c_f^{\ell_2^+\ell_2^-})_f^{\ell_3^+\ell_3^-} \rangle_f^{\ell_4^+\ell_4^-}$		$\left.\begin{array}{l}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ e_3\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	

Table 1. Constraints creation for source-sink pairs.

contract on the fly (with ℓ and ℓ' fresh) and uses it to check the domain and range of the function contract. For deeply nested function contracts, the process is repeated recursively thereby creating a witness for each possible contract violation.[5] In essence this process simply makes explicit the sinks for the complex abstract values that flow into $\text{any}_h^{\ell_5^+\ell_5^-}$. The analysis therefore remains sound. Here we forsake this process and re-use the $\text{any}_h^{\ell_5^+\ell_5^-}$ contract and its labels only to simplify the soundness proof.

[5] The debugger must then be careful to re-use the original $\text{any}_h^{\ell_5^+\ell_5^-}$ contract for both the domain and range of the new $(\text{any}_h^{\ell_5^+\ell_5^-}\to\text{any}_h^{\ell_5^+\ell_5^-})_h^{\ell\ell'}$ contract because the use of new any contracts for the domain and range makes the analysis fail to terminate when a function with a recursive type fbws into $\text{any}_h^{\ell_5^+\ell_5^-}$.

5.2.2 Blame Constraints

The third example explains blame assignment:

Source$\backslash$*Sink*	$(c_i^{\ell_7^+\ell_7^-}\to c_h^{\ell_8^+\ell_8^-})_h^{\ell_5^+\ell_5^-}$
$\text{int}_f^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$

It specifies the creation of a single blame set constraint for every possible pair of abstract integer source and arrow contract check in the program. The constraint says that, if the abstract integers represented by ℓ_1^+ flow into the arrow check represented by ℓ_5^-, module h has to be blamed because it might produce integers when only abstractions are expected. The name of the module is tagged with red ($\mathcal{R}$) because it is a complete violation of the contract.

For a partial contract violation, which is tagged with orange ($\mathcal{O}$), consider this entry:

Source$\backslash$*Sink*	$\langle \ldots\ e_5\ \mathrm{int}_h^{\ell_5^+\ell_5^-} \rangle_h^{\ell_6^+\ell_6^-}$
$\langle \ldots\ e_1\ \mathrm{int}_f^{\ell_1^+\ell_1^-} \rangle_f^{\ell_2^+\ell_2^-}$	$\left.\begin{array}{c}\{\ell_1^+\} \subseteq \varphi(\ell_5^-) \\ e_1 \not\sqsubseteq e_5\end{array}\right\} \Rightarrow \{\langle h, \mathcal{O}\rangle\} \subseteq \psi(\ell_5^-)$

The cell specifies the creation of a single blame set constraint for every possible pair of an integer contract triple (viewed as a source) that has an additional predicate e_1 and another triple with an integer contract check that has an additional predicate e_5. The constraint says that, if the abstract integer (ℓ_1^+) flows into the integer check (ℓ_5^-) and if the source predicate e_1 does not imply ($\not\sqsubseteq$) the sink predicate e_5, then the h module variable is blamed for the violation. The "blame" color, however, is orange because the analysis can prove that the abstract values flowing into the contract check are at least always integers. Note that the boilerplate code in the triples plays no explicit role here so we use dots for this code.

The blame constraints in Table 1 always use the name h associated with the sink (or λ when the module violates the language specification), never the name f associated with the source. This makes the analysis consistent with the invariant established via rule ARROWC during the annotation process of Section 3.3. That rule switches the two module variable names used by the $\vdash_{\mathsf{C}}^{\mathsf{a}}$ judgment as it traverses the domain positions in a functional contract. This switch ensures that when an expression is reduced and triggers a contract violation at runtime, blame for that violation is always correctly assigned to the module that originally contained the expression being reduced. The switch also ensures that, at analysis time, the name of the module that originally contained the currently analyzed lifted expression tree is always the name associated with any contract check that is used at the top of that tree.

For example, in the lower left part of Figure 8, the original grey module is always blamed when the reduction process triggers a runtime contract violation in either of the two grey terms. In the lower right corner of the figure the name of the original grey module is always associated with the contract checks at the top of both grey subtrees. By always using the name h associated with such contact checks when assigning blame, the constraints of Table 1 guarantee that the analysis is consistent with the runtime behavior in blaming the original grey module for all contract violations occurring inside a grey term.

This treatment of blame assignment is also consistent with a modular analysis. The analysis completely trusts the contracts at the top and bottom of a lifted expression tree to correctly approximate the outside world, even if analyzing later that outside world might show that assumption to be untrue. Since it trusts the contracts, the analysis can only assign blame to the analyzed expression. While this makes blame assignment look easy, it is really a consequence of a carefully engineered annotation process and lifting phase.

5.2.3 Additional Constraints

Finally, to get the analysis started, we must supplement Table 1 with rules that get the flows initiated for all the value sources. In general, all value sources must have their label included in their own value set. Similarly, each blame set is seeded with the names from blame expressions: see the top two rows of Table 2.

The third row in Table 2 describes the flows from the two branches of an if0 expression to the whole expression. Naturally there are no flows out of the test.

The fourth row explains the analysis of contract checks at the top of the lifted trees. Recall that a contract at the top of a lifted tree simulates the context in which the tree used to occur. Since any given contract can be both a value source or a value sink, the constraint generation algorithm merely connects the outflow of the sub-expression with the inflow of the contract.

Triples such as $\langle e^{\ell_1}\ e^{\ell_2}\ c_f^{\ell\ell'} \rangle_f^{\ell^+\ell^-}$ also need to create value flows. Remember that the third part of a triple—the domain contract derived by the $\mathcal{D}_{\Delta}$ function—shares its label ℓ with ε expressions in the first part of the triple. There is therefore no need to create flows between the first and third parts of the triple. Two flows are still missing, however. First, the result of the first part flows out to be the result of the entire triple. Second, the values that flow into the triple really flow into the ℓ' position of the contract; this guarantees that these in-flowing values are checked against the contract c. See the fifth row of Table 2 for details on these checks.

One interesting aspect of triples is that they are not themselves abstract value sources. What acts as a value source is the predicate-free contract c, which approximates the predicate e in the triple. When c reaches a value sink it is directly checked against the sink if the sink is another predicate-free contract, or it is used as an approximation of e if the sink is another triple.

To be more concrete, consider again the example at the end of Section 3.3. Starting from the contract (pred *prime?*) on the definition of the module variable f the annotation process inserts around the reference to f a contract check with a triple of the form:

$$\langle (\text{if0}\ (\textit{prime?}^{\beta_1}\ \varepsilon^{\ell})\ \varepsilon^{\ell}\ (\text{blame}\ f\ \mathcal{O}))\quad \textit{prime?}\quad \mathrm{int}_f^{\ell\ell'} \rangle$$

When considered as a source the $\mathrm{int}_f^{\ell\ell'}$ contract flows naturally to the ε^{ℓ} expression, out of the if0 one, and then out of the triple because of the first constraint from the fifth row of Table 2. If later that $\mathrm{int}_f^{\ell\ell'}$ contract flows into a simple arrow contract check then a red error occurs. If the $\mathrm{int}_f^{\ell\ell'}$ contract flows into a simple integer contract check then everything is fine. In both cases the analysis has reached a conclusion without ever having to consider the predicate *prime?*, which is the only information the programmer supplied for f's contract. In essence the analysis has computed that, to be a prime, a value must first be an integer. It can then use that knowledge to simplify many of the contract checks.

Similarly if the sink for $\mathrm{int}_f^{\ell\ell'}$ is a triple with a simple arrow contract as its third part, the analysis flags a red error without having to consider either *prime?* or the predicate in the sink triple. It is only when $\mathrm{int}_f^{\ell\ell'}$ flows into a triple with an integer contract as its third part that the analysis has to compare the predicate *prime?* from the source with the predicate from the sink and decide, using the $\not\sqsubseteq$ relation, whether the first implies the second. If not, an orange error is flagged.

Finally, we are left with modules. Initially, a module contributes only its single value to the analysis. The last row in Table 2 therefore adds a constraint that connects the value to the module variable. Since a variable shares its label with all its references, the value thus flows from the variable definition to each reference to a $\Leftarrow$ form that checks the values against the module variable's contract. The analysis thereby ensures that the expression defining the module variable satisfies its own contract.

Once all the constraints have been generated from a program's text, they have to be solved to obtain the solution. This is done using

n^{ℓ} $\text{int}_f^{\ell\ell'}$ $\text{any}_f^{\ell\ell'}$ $(\lambda x^{\beta}.e^{\ell_e})^{\ell}$ $(c_1 \rightarrow c_2)_f^{\ell\ell'}$	$\{\ell\} \subseteq \varphi(\ell)$
$(\text{blame } f\ s)^{\ell}$	$\{\langle f, s\rangle\} \subseteq \psi(\ell)$
$(\text{if0 } e^{\ell_0}\ e^{\ell_1}\ e^{\ell_2})^{\ell}$	$\varphi(\ell_1) \subseteq \varphi(\ell)$ $\varphi(\ell_2) \subseteq \varphi(\ell)$
$(c_f^{\ell\ell'} \Leftarrow e^{\ell_e})^{\ell_c}$	$\varphi(\ell_e) \subseteq \varphi(\ell')$
$\langle e^{\ell_1}\ e^{\ell_2}\ c_f^{\ell\ell'} \rangle_f^{\ell^+\ell^-}$	$\varphi(\ell_1) \subseteq \varphi(\ell^+)$ $\varphi(\ell^-) \subseteq \varphi(\ell')$
$(\text{module } f^{\beta}\ v^{\ell_v})^{\ell}$	$\varphi(\ell_v) \subseteq \varphi(\beta)$

Table 2. Additional constraints.

standard technology for solving Horn constraints. See for example Palsberg and Schwartzbach [25].

5.3 Analysis Parameterization

The analysis is parameterized over the approximation relation $\not\sqsubseteq$ that is used to compare predicates. Intuitively, the relation is a version of (the negation of) observational approximation. Consider $n+1$ predicates $e_1,\ldots, e_n$, and e, and the question of whether the relation $e_1 \ldots e_n \not\sqsubseteq e$ holds or not. Since predicates work on values, this question only makes sense if it is asked for a given abstract value v: if v has satisfied each of the predicates e_i, does v then satisfy e? More formally, we define the $\not\sqsubseteq$ relation as follows: given the predicates $e_1,\ldots, e_n$ and e, we have $e_1 \ldots e_n \not\sqsubseteq e$ if and only if there exists an abstract value v such that $(e_i\ v)$ reduces to 0 for all i and $(e\ v)$ does not reduce to 0.

In practice a static debugger will only analyze an unreduced program, where the relation will always be only of the form $e_1 \not\sqsubseteq e$, but we have to use the multi-predicate version here for the sake of the soundness proof. All the $e_1 \ldots e_n$ and e predicates should be non-lifted expressions, otherwise the $\not\sqsubseteq$ relation might in some cases end up comparing contracts rather than expressions.

Since observational approximation is undecidable, an implementation must use a decidable and conservative version of it. The selection of a decidable relation is a trade-off between the power of the analysis and the time complexity of the relation. Many reasonable choices exist: the vacuous **false** relation; the equality of predicate names; λ-calculi; or general theorem proving à la ESC [7].

In practice a relation based on predicate names and contract combinators is a good choice. DrScheme programmers who use the contract system tend to give names to contract predicates and re-use those names. For complex contracts they use contract combinators. Thus, a DrScheme programmer may introduce a contract (*and/c even? prime?*) and name it `ep`. If other modules use `ep`, the analysis can avoid false positives when the result of an `ep`-generating function flows into the argument of an `ep`-consuming function. This works well even though the analysis itself has no notion of the concept of evenness or primality. The resulting system then is in essence the idea of type qualifiers [14] applied to contracts.

Of course, the analysis is not able to bless an `ep` flowing, say, into a `positive?` contract, but it is at least possible to check that both `ep` and `positive?` are integer-based predicates and flag that second contract in orange rather than red. The orange color means that the analysis has detected that a contract violation has only been a partial one and it can report that information back to the programmer who is using the static debugger.

Put from the point of view of that programmer, the red color means that either an actual violation has been detected or that the analysis has unknowingly reached its own limit (a limit inherent to the core value flow analysis). The orange color means that either an actual violation has been detected or that the analysis has knowingly reached its own limit. That is, in the orange case the analysis has detected that the $\not\sqsubseteq$ relation is not capable of proving the desired property, either because the property is wrong or because the relation is too weak to prove it, while in the red case the analysis simply concludes that the property is wrong. From the point of view of the programmer then, getting rid of an orange false-positive requires using a stronger $\not\sqsubseteq$ relation, while getting rid of a red false-positive requires changing the core of the analysis in Table 1.

The analysis is also parameterized over the $\mathcal{D}_{\Delta}$ function (Fig. 6) used in the annotation process. Looking once more at the example at the end of Section 3.3, we see that $\mathcal{D}_{\Delta}$ approximates the *prime?* predicate with an int contract. If that int contract flows from the contract triple into an int check elsewhere in the program then Table 1 tells us that everything is fine. If instead we weaken the $\mathcal{D}_{\Delta}$ function to approximate *prime?* with an any contract, that any contract now flows from the triple into the same int check and Table 1 tells us a red error is flagged. This shows that choosing a reasonably precise $\mathcal{D}_{\Delta}$ function is important for the accuracy of the analysis.[6]

The only way to get rid of the spurious red error stemming from such a weakened $\mathcal{D}_{\Delta}$ function is to modify the cell in Table 1 that has $\langle \ldots e_1\ \text{any}_f^{\ell_1^+\ell_1^-} \rangle_f^{\ell_2^+\ell_2^-}$ as a source and $\text{int}_h^{\ell_5^+\ell_5^-}$ as a sink to make it use the $\not\sqsubseteq$ relation and extend that relation to check whether $e \not\sqsubseteq \text{int}$. After all, even though $\mathcal{D}_{\Delta}$ poorly approximates *prime?*, that predicate by itself mathematically ensures that all values satisfying it are integers, so there is no reason to flag a red error if $\not\sqsubseteq$ is strong enough to compensate for $\mathcal{D}_{\Delta}$'s weakness. In general, the less accurate the $\mathcal{D}_{\Delta}$ function is in approximating predicates, the more work the $\not\sqsubseteq$ relation has to do to prevent the appearance of false-positives.

Obviously the parameterization of the analysis over $\mathcal{D}_{\Delta}$ and $\not\sqsubseteq$ has a strong influence on the analysis's total running time. There is no limit to how complex $\mathcal{D}_{\Delta}$ and $\not\sqsubseteq$ can be. Outside of those two specific running times, the constraints created from the core of the analysis in Table 1 can still be solved in time proportional to the cube of the size of the *lifted* program [25] in the worst case. Remember that the annotation process duplicates contracts, and in fact it can do so a linear number of times if there is a linear number of module variable references in the program. If a given module variable has a linear number of references and its contract is itself linear in the size of the original program, the size of the lifted program is then quadratic in the size of the original program in the worst case, and the total running time of the constraint solving part of the analysis is then proportional to the sixth power of the size of the original program. In practice contracts have a constant size so the programmer is unlikely to ever experience this worst case analysis time.

5.4 Type Reconstruction

Given the solution φ of the set constraints for value flows, we can create a type-like description of value sets for each node in the program. Specifically, for a given mapping φ and label ℓ, the two functions in Figure 11 reconstruct a (recursive) type specification. It is those types that the static graphical debugger presents to the programmer together with the blame sets.

The $\mathcal{R}^{\varphi}$ function computes the set of all reachable labels from a label ℓ. The $\mathcal{T}^{\varphi}$ function then uses these labels as the names of types to construct a (potentially) recursive type for ℓ. The reconstruction itself is straightforward. A set of labels corresponds to a union; an empty set corresponds to dead code or an expression that never returns a result. A label on an integer or integer contract

[6] Weakening $\mathcal{D}_{\Delta}$ does not make any difference for the runtime contract system because the reduction rules always check all the predicates no matter how weak the approximations that $\mathcal{D}_{\Delta}$ computes are.

corresponds to an integer type. A label on an any contract corresponds to an any type. Last but not least, a label on an abstraction or arrow contract corresponds to a function type. The surrounding rec type constructor takes accounts for the binding of labels for the function's argument and result types.

We are not concerned here with the readability of types. Hence, we skip any simplification steps for the reconstructed types. The types are, however, useful for the formulation of the analysis soundness theorem and its proof.

$$\mathcal{R}^{\varphi}(\ell) \stackrel{def}{=} \{\ell\} \cup \mathcal{R}_u^{\varphi}(\ell)$$

$$\mathcal{R}_u^{\varphi}(\ell) \stackrel{def}{=} \bigcup_{\ell_i \in \varphi(\ell)} \mathcal{R}_t^{\varphi}(\ell_i)$$

$$\mathcal{R}_t^{\varphi}(\ell) \stackrel{def}{=} \begin{cases} \{\ell\} \text{ if } n^{\ell} \text{ or } \mathrm{int}_f^{\ell\ell'} \text{ or any}_f^{\ell\ell'} \\ \{\ell\} \cup \mathcal{R}^{\varphi}(\ell_1) \cup \mathcal{R}^{\varphi}(\ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \text{ or } (c_g^{\ell_1'\ell_1 \to c_f^{\ell_2\ell_2'}})_f^{\ell\ell'} \end{cases}$$

$$\mathcal{T}^{\varphi}(\ell) \stackrel{def}{=} (\text{rec } ([\ell_i\ \mathcal{T}_u^{\varphi}(\ell_i)]_{\ell_i \in \mathcal{R}^{\varphi}(\ell)} \ldots)\ \ell)$$

$$\mathcal{T}_u^{\varphi}(\ell) \stackrel{def}{=} (\text{union } \mathcal{T}_t^{\varphi}(\ell_i)_{\ell_i \in \varphi(\ell)} \ldots)$$

$$\mathcal{T}_t^{\varphi}(\ell) \stackrel{def}{=} \begin{cases} \text{int if } n^{\ell} \text{ or } \mathrm{int}_f^{\ell\ell'} \\ \text{any if any}_f^{\ell\ell'} \\ (\ell_1 \to \ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \text{ or } (c_g^{\ell_1'\ell_1 \to c_f^{\ell_2\ell_2'}})_f^{\ell\ell'} \end{cases}$$

Figure 11. $\mathcal{R}^{\varphi}$ and $\mathcal{T}^{\varphi}$ functions.

6. Soundness

We adapt Wand and Williamson's proof technique [30] to prove the soundness of our analysis. Let $[\![p]\!]$ be the set of constraints that the analysis generates when given the lifted program p, and let $\models$ denote implication between sets of constraints: for two sets of constraints A and A', we have $A \models A'$ if and only if every solution of A is a solution of A'.

Given this machinery, an adaptation of Wand and Williamson's soundness theorem for our analysis is as follows.

Theorem 1. *For a given annotated program p, let $p' \stackrel{\text{def}}{=} m' \ldots e^{\ell'}$ be such that $\vdash_{\mathsf{P}}^{\mathsf{1}} p \rightsquigarrow p'$. Then:*

- *p reduces to $m \ldots v^{\ell}$ and $[\![p']\!] \models \{\ell\} \subseteq \varphi(\ell')$;*
- *p reduces to $(\text{blame } \pi\ s)^{\ell}$ and $[\![p']\!] \models \{\langle \pi, s \rangle\} \subseteq \psi(\ell)$;*
- *or p reduces forever;*

where π indicates the party to blame for the violation (either a module variable name like f, μ for the main expression, or λ for the user) and s indicates the severity of the violation ($\mathcal{O}$ or $\mathcal{R}$).

The proof follows the one by Wand and Williamson, extended to handle blame sets.

Intuitively, our analysis conservatively predicts the runtime behavior of the program. If the program terminates normally by returning a value then the analysis correctly predicts the label on that value. If the program terminates abnormally because of a contract violation then the analysis conservatively predicts the violation, its severity, and the module that is to be blamed for it.

While necessary, the theorem above is not quite enough. It shows that, if the program reduces to a value, the analysis correctly predicts the label on that value. This does not automatically mean that the analysis predicts the value itself; after all, the label on a given value changes every time the value crosses a contract boundary. Indeed, one of the invariants of the reduction rules from Figure 7 is that a value that successfully goes through a contract check always acquires the label that was on that contract (seen as an abstract value source).

What we want is a strengthening of the theorem that tells us something about values and types. Fortunately, contracts ensure that types are preserved as values cross contract boundaries. For example, when the analysis encounters the expression

$$(\mathrm{int}_f^{\ell\ell'} \Leftarrow 3^{\ell_n})^{\ell_c},$$

the analysis predicts that the result is an integer with label ℓ. In this case we obtain 3^{ℓ} after just one reduction step (INT-INT)

Using this insight, we can state and prove an improved correctness theorem:

Theorem 2. *For a given annotated program p, let $p' \stackrel{\text{def}}{=} m' \ldots e^{\ell'}$ be such that $\vdash_{\mathsf{P}}^{\mathsf{1}} p \rightsquigarrow p'$. Then:*

- *p reduces to $m \ldots v^{\ell}$ and $[\![p']\!] \models \mathcal{T}^{\varphi}(\ell) \leq \mathcal{T}^{\varphi}(\ell')$,*
- *p reduces to $(\text{blame } \pi\ s)^{\ell}$ and $[\![p']\!] \models \{\langle \pi, s \rangle\} \subseteq \psi(\ell)$;*
- *or p reduces forever.*

where $\leq$ is subtyping between recursive types [3, 18] and π and s have the same meaning as before.

Proof Sketch. We adapt Wand and Williamson's technique as follows for this proof. Take the set of constraints $[\![p']\!]$. Replace every constraint of the form $\varphi(\ell) \subseteq \varphi(\ell')$ with a constraint of the form $\mathcal{T}^{\varphi}(\ell) \leq \mathcal{T}^{\varphi}(\ell')$. Now prove the entailment and type preservation properties for these sets of constraints using Wand and Williamson's technique and the fact that all contract checking reductions in Figure 7 ensure that types are preserved when a value crosses a contract boundary. □

7. Modularity

Conventionally, an analysis is called modular if it is applied to a module and a description of the rest of the world. That is, the approach assumes that a modular analysis is what an analysis applied to a module is. In contrast, we have formulated the analysis in terms of the *entire* program, and we now prove that it is modular, i.e. that a lifted tree of a program can be analyzed in isolation of the rest of the program.

Theorem 3. *Given an annotated program p, let p' be such that $\vdash_{\mathsf{P}}^{\mathsf{1}} p \rightsquigarrow p'$. Consider a single lifted tree t' in p'. Consider the minimal solution $\varphi_{p'}$ of $[\![p']\!]$ and its restriction $\varphi_{p'/t'}$ to the labels that occur in t'. Consider also the minimal solution $\varphi_{t'}$ of $[\![t']\!]$. Then $\varphi_{p'/t'}$ and $\varphi_{t'}$ are the same.*

In other words, analyzing a lifted tree (either a module or a lifted expression) in isolation of the rest of the program produces the same results as analyzing the whole program and then looking at the results for just that tree. This is true regardless of how many times the program has already been reduced.

Proof Sketch. A direct consequence of the lemma below. We consider minimal solutions because all other pairs of solutions are incomparable in general. □

To show that module contracts are complete descriptions of the program context, we prove that abstract values cannot flow between any two lifted trees during the constraint solving phase.

Lemma. *Given an annotated program p, let p' be such that $\vdash_{\mathsf{P}}^{\mathsf{1}} p \rightsquigarrow p'$. Then for two different lifted trees t and t' that are in p', the only labels ℓ in t and ℓ' in t' such that $[\![p']\!] \models \varphi(\ell) \subseteq \varphi(\ell')$ are labels where $\ell = \ell' = \beta$ with $t = (\text{module } f^{\beta}\ v^{\ell_v})^{\ell_m}$ and $t' = (c_f^{\ell^+\ell^-} \Leftarrow f^{\beta})^{\ell_c}$.*

Intuitively, the lemma says that the analysis propagates only values from modules to occurrences of contracted module names. That is,

from a module variable binder to a reference that is wrapped with a contract check. Of course, such flows do not break modularity in practice because they merely mean that the module's value is checked against its own contract. That such checks create a seemingly intertree flow is an artifact of our lifting function. A practical implementation simply propagates the module's value directly into the check without going through the variable reference. This is in fact what happens as soon as the LOOKUP rule has been used.[7]

Proof Sketch. A close look at the syntax of Figure 10 shows that intertree flows can only occur in the following three cases: (1) across the same contract seen as a sink at the top of a lifted tree and as a source at the bottom of another tree; (2) through an ε^{ℓ} expression that shares its label with another expression in another tree; or (3) from a lexical or module-defined variable binder in one tree to a reference to the same variable in another tree.

(1) All contracts are tagged with two labels. The first one is used when the contract is seen as an abstract value source, the second one when the contract is seen as a sink. Tables 1 and 2 are defined, however, in such a way that no abstract value ever flows into a source contract (apart from the abstract value represented by that contract itself) or flows out of a sink contract. Leaking values across contracts is therefore impossible.

(2) By construction the ε^{ℓ} expressions initially occur only inside triples. Furthermore, they share their labels with the contract in the same triple and nothing else. The triple's boilerplate code can only have contract checks inside the predicate expression in the test part of the if0 expression (Sec. 3.3). Lifting judgments therefore may only affect that part of the boilerplate code. Hence, the two ε^{ℓ} expressions and the contract with the same label all remain in the same triple after lifting. There is thus no possibility for values to flow from one tree to another through ε^{ℓ} expressions.

(3a) Similarly, the binder and all the references for a given lexical variable always remain inside the same tree. By construction contracts are initially only on module-defined variables. No reduction rule, including the SPLIT rule, ever introduces a contract between a binder and one of its references. The lifting function therefore never separates binder and references into two different trees. Leaks through lexical variables are thus impossible, too.

(3b) Module variables are the only remaining mechanism for intertree value propagation. Recall (Sec. 3.3) that the annotation phase wraps all module variable references with a contract check:

$$\begin{array}{l}(\text{module } f\ c\ v)\\ \ldots f \ldots\end{array}$$

becomes

$$\begin{array}{l}(\text{module } f^{\beta}\ v^{\ell_v})^{\ell_m}\\ \ldots (c_f^{\ell^+\ell^-} \Leftarrow f^{\beta})^{\ell_c} \ldots\end{array}$$

Now the lifting function lifts all contract checks to the top so that after lifting, the annotated code above is split into three trees:

$$\begin{array}{l}(\text{module } f^{\beta}\ v^{\ell_v})^{\ell_m}\\ (c_f^{\ell^+\ell^-} \Leftarrow f^{\beta})^{\ell_c}\\ \ldots c_f^{\ell^+\ell^-} \ \ldots\end{array}$$

And in fact, the analysis of this code propagates the value v^{ℓ_v} in the first tree to f^{β} in the module and afterwards to the reference f^{β} in the contract check.

In short, this last part validates that intertree flows are possible from a module variable definition to a contract check for just this variable. No other kind of flow is possible through module variables because by construction all contract checks are initially on module variable references. Such references can only disappear by being substituted for their bound value (LOOKUP rule in Figure 7), which then makes the second lifted tree in the example above independent of the first one. □

[7] Putting the contract checks on the module variable binders rather than on each module reference would make the analysis monovariant in such values. As it stands, it is naturally polyvariant in module values [32].

8. Implementation

We have created a proof-of-concept static debugger based on our analysis. It implements the annotation phase of Section 3.3, and the lifting, constraints generation, and type reconstruction phases described in Section 5. We use simple name equality to implement the $\not\sqsubseteq$ relation. In that implementation abstract value sets are represented as nodes in a graph. Simple inclusion constraints between value sets such as the ones in Table 2 are represented as direct edges between nodes. Conditional constraints like the ones in Table 1 are represented as special edges that create new direct edges whenever their condition becomes true. Solving the constraint is then a simple matter of computing the transitive closure of the graph, which can be done in cubic worst case time in the size of the graph. Constraints for blame sets are handled in a similar manner.

```
(module i int 0)
(i 1)
```

Figure 12. Example program with red error.

Figure 12 shows the result of using our debugger on a toy program consisting of a single module and a main expression. The main expression is highlighted and underlined in red because it is trying to apply the integer `i` as if it were a function. The error message (not shown) blames λ, the programmer of the main expression. This example corresponds to the cell in Table 1 that has an integer $n^{\ell_n}_{e_1\ldots}$ as source and an application $(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$ as sink. Thanks to DrScheme's syntax object system, the error highlighting is done in terms of the user's original program, not in terms of the lifted one, which remains internal to the debugger.

```
(module prime? (int -> int)
  (lambda x . 1))

(module p (pred prime?) 4)

p
```

Figure 13. Example program with orange error.

Our second screenshot in Figure 13 shows an orange error. We define a predicate `prime?` that accepts integers as input. Actually implementing a primality test is not our concern here so we simply defined `prime?` as a function which we know never violates `prime?`'s own contract. Next we define the variable `p` and use the `prime?` predicate just defined to promise that `p` is a prime number. We then use that integer in the main expression. The debugger colors the `prime?` predicate in orange, because, while it can prove that the number `4` is an integer just as the `prime?` predicate expects, it cannot prove that `4` is actually a prime number as promised. The error message blames `p`. This example corresponds to the cell in Table 1 that has an integer $n^{\ell_n}_{e_1\ldots}$ as a source and a triple $\langle \ldots e_5\ \text{int}^{\ell_5^+\ell_5^-}_h \rangle^{\ell_6^+\ell_6^-}_h$ as a sink. Here $e_1 \ldots$ is empty so $e_1 \ldots \not\sqsubseteq e_5$ is vacuously true.

Our final example in Figure 14 shows a use of the $\not\sqsubseteq$ relation. As in the previous example we define a predicate `prime?` and a

```
(module prime? (int -> int)
  (lambda x . 1))

(module p (pred prime?) 4)

(module f ((pred prime?) -> int)
  (lambda y . y))

(f p)
```

Figure 14. Example program with no second `prime?` error.

prime number `p`. As before the debugger signals an orange error because `p` might not actually be a prime number. We also define a function `f`, which acts as a sink for prime numbers, and then give `p` as input to `f`. Notice that, even though the debugger has discovered that `p` might not be a prime number, it does not signal any error when giving `p` to `f`. The debugger is able to tell that, if the value of `p` passes `p`'s contract check at runtime, then it also passes `f`'s domain contract. Even though the debugger does not understand the concept of primality, it does use the name-based $\not\sqsubseteq$ relation to check that the contract on `p` matches the contract on the domain of `f` and consequently does not signal an error. This behavior corresponds to the cell in Table 1 that has a triple $\langle \ldots e_1 \text{ int}_f^{\ell_1^+ \ell_1^-} \rangle_f^{\ell_2^+ \ell_2^-}$ as source and another triple $\langle \ldots e_5 \text{ int}_h^{\ell_5^+ \ell_5^-} \rangle_h^{\ell_6^+ \ell_6^-}$ as sink. Since e_1 and e_5 are both `prime?`, the relation $e_1 \not\sqsubseteq e_5$ is not satisfied, the constraint $\{\langle h, \mathcal{O} \rangle\} \subseteq \psi(\ell_5^-)$ is thus not triggered, and the debugger does not highlight the `prime?` predicate in `f`'s contract. This also shows that the orange contract violation for the body of `p` does not influence the analysis of the uses of `p` elsewhere; *the analysis is modular.* Finally, notice that after flowing through `f`'s body a prime number does not trigger `f`'s int range contract check. The analysis correctly recognizes primes as integers, since the domain for the `prime?` predicate itself is int, which is what $\mathcal{D}_\Delta$ computes.

9. Related Work

Probst [26], Flanagan and Felleisen [10], and Fähndrich and Aiken [2] develop set-based analyses for module-like components in (higher-order) object-oriented and functional languages. All three approaches rely on a variation of the same basic technique. Their analysis generates separate constraint sets for each module, simplifies them using various heuristics, stores the resulting sets for later use, and eventually combines all the necessary sets together to get the solution for a specific module. While this form of analysis clearly helps programmers who wish to explore a large set of modules in an incremental manner, it does not qualify as a truly modular analysis. Without the entire program around, a programmer cannot start the analysis.

Tang and Jouvelot [28] present a technique that uses type and effect information, possibly coming from module signatures, to extend an abstract interpretation to support separate analysis. They use 1CFA as an example for their technique, though it can be applied to any abstract interpretation. While this analysis truly qualifies as modular, it only considers contracts as value sources, never as value sinks, and therefore cannot check module definitions against their own contracts. Worse, because errors are impossible in their language the analysis comes without any blame assignment, which we consider a centerpiece of contract monitoring.

Cousot and Cousot [6] formalize a framework for modular abstract interpretation and consider several solutions, including the idea of programmer-specified interfaces. For this case they provide general conditions relating the analysis and the interfaces so that the analysis is sound. We conjecture that our approach is a special case of this framework, but we have no proof for this conjecture. We chose to develop our own model and soundness proof so that we could cope with the blame analysis properly.

Much work has also been done on modules in the context of Hindley-Milner type systems[19, 20, 22]. The most obvious difference between such type systems for modules and our analysis is the restricted set of program properties that can be expressed within the type language, while contracts can use the full power of the expression language to describe any possible property. This comes at the price of $\not\sqsubseteq$ being undecidable.

Identifying the source of type errors in ML-like languages is notoriously difficult [29]. Since we use a flow analysis, our graphical debugger can easily trace values back to their source when a contract violation occurs [11]. The closest equivalent is Haack and Wells's type error slicing system [15]. Extending that system to handle module signatures is described as future work, however.

There is a general equivalence between polyvariant flow analyses and type systems with intersection and union types [24, 31]. Our system is polyvariant (in the sense that the contract for a given module variable is duplicated and re-analyzed at each reference of that variable) but it is doubtful that an equivalent static type system exists, due to the presence of predicates in our contract language. Systems with intersection and union types also usually do not consider the problem of modularity. Wells et al. indicate that their λ^{CIL} calculus could possibly serve as the basis for a modular compilation system [31] but do not elaborate on that point.

Other systems [1, 5, 9, 17] have investigated the combination of static types and dynamic checks to ensure program correctness. Flanagan's hybrid type checker [9] is closest to our system. His type checker is parameterized over a three-valued subtyping judgement, which is similar in spirit to the parameterization of our analysis over the approximation relation. Flagging a red error in our analysis then parallels rejecting a program in his type system, and flagging an orange error parallels inserting a dynamic check.

Both our contract language and Flanagan's type language include predicates. The type $x : B.t$ denotes in his language the set of values of base type B that satisfy the predicate t. The user must therefore specify both B and t. On our system the user only specifies the predicate t and we use the function $\mathcal{D}_\Delta$ to automatically approximate B. In both systems two predicates are compared only once their base types (the third parts of the corresponding contract triples in our case) have proved to match. Flanagan's type language also includes dependent function types, whereas our model does not yet include Findler and Felleisen's dependent contracts [8].

While Flanagan does not examine the question of modules, it should be easy to add them to his language by using his types as interface specifications. The way he assigns blame is based on the work by Findler and Felleisen, as is ours.

10. Future Work

As it is, Table 1 is only partially parameterized over the $\not\sqsubseteq$ relation. Five cells in the table have orange blame constraints that do not use the relation in their antecedent. If $\not\sqsubseteq$ is extended to handle relations of the form $c \not\sqsubseteq e$ then those five cells can be modified to depend on the relation. Symmetrically, if $\not\sqsubseteq$ is extended to handle relations of the form $e \not\sqsubseteq c$ (e.g. to prove that *prime?* mathematically implies int) then five cells with red blame constraints can be modified to become orange blame constraints that rely on $\not\sqsubseteq$. Table 1 will then be fully parameterized over the $\not\sqsubseteq$ relation.

Our model of a static debugger also needs to be extended to cover some of the most commonly used contract combinators (*and/c*, *or/c*, etc.) used in DrScheme's contract system.

11. Conclusion

The paper shows how a program analysis can exploit module contracts to produce sound approximations of the value flows in a program in a fully modular manner. Moreover that analysis is parameterizable. To understand the exact design trade-offs, we plan to include a full-fledged implementation with a future release of DrScheme. Then experimentation by practicing programmers will help us understand how a practical contract system and a modular analysis should work together.

Acknowledgments

We thank Mitchell Wand as well as the anonymous POPL reviewers for their valuable comments on several drafts of this paper. The authors also acknowledge the support of the National Science Foundation for this research.

References

[1] Abadi, M., L. Cardelli, B. Pierce and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[2] Aiken, A. S. and M. Fähndrich. Making set-constraint based program analyses scale. Technical Report CSD-96-917, University of California, Berkeley, September 1996.

[3] Amadio, R. M. and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] Bourdoncle, F. Abstract debugging of higher-order imperative languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55, 1993.

[5] Cartwright, R. and M. Fagan. Soft typing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[6] Cousot, P. and R. Cousot. Modular static program analysis, invited paper. In Horspool, R., editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, pages 159–178, Grenoble, France, April 6—14 2002. LNCS 2304, Springer, Berlin.

[7] Detlefs, D. L., K. R. M. Leino, G. Nelson and J. B. Saxe. Extended static checking. Technical Report 159, Compaq SRC Research Report, 1998.

[8] Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[9] Flanagan, C. Hybrid type checking. In *Proceedings of the symposium on Principles of Programming Languages*, 2006. In this volume.

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

[11] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. *ACM SIGPLAN Notices*, 31(5):23–32, 1996.

[12] Flatt, M. Composable and compilable macros: You want it *when?* In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[13] Flatt, M., R. B. Findler, S. Krishnamurthi and M. Felleisen. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, September 1999.

[14] Foster, J. S., M. Fähndrich and A. Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[15] Haack, C. and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Programming*, 50:189–224, 2004.

[16] Heintze, N. Set-based analysis of ml programs. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 306–317, New York, NY, USA, 1994. ACM Press.

[17] Henglein, F. Dynamic typing. In *Proceedings of the 4th European Symposium on Programming*, pages 233–253, London, UK, 1992. Springer-Verlag.

[18] Hosoya, H., J. Vouillon and B. C. Pierce. Regular expression types for xml. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 11–22. ACM Press, 2000.

[19] Leroy, X., D. Doligez, J. Garrigue, D. Rémy and J. Vouillon. The Objective Caml system – documentation and user's manual, 2005.

[20] MacQueen, D. B. Modules for Standard ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 198–207, New York, 1984. ACM Press.

[21] Meunier, P., R. B. Findler, P. A. Steckler and M. Wand. Selectors make set-based analysis too hard. *Higher Order and Symbolic Computation*, 2005. To appear.

[22] Milner, R., M. Tofte, R. Harper and D. Macqueen. *The Definition of Standard ML - Revised*. MIT Press, Cambridge, MA, USA, 1997.

[23] Palsberg, J. Closure analysis in constraint form. *Proc. ACM Trans. on Programming Languages and Systems*, 17(1):47–62, Jan. 1995.

[24] Palsberg, J. and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 197–208, New York, NY, 1998.

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

[26] Probst, C. W. Modular control flow analysis for libraries. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 165–179, London, UK, 2002. Springer-Verlag.

[27] Shivers, O. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26(9), pages 190–198, New Haven, CN, June 1991.

[28] Tang, Y. M. and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In Hagiya, M. and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 224–243. Springer, Berlin, Heidelberg, 1994.

[29] Wand, M. Finding the source of type errors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–43, 1986.

[30] Wand, M. and G. B. Williamson. A modular, extensible proof method for small-step flow analyses. In Métayer, D. L., editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 213–227, Berlin, 2002. Springer-Verlag.

[31] Wells, J. B., A. Dimock, R. Muller and F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 12(3):183–227, May 2002.

[32] Wright, A. K. and S. Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, 20(1):166–207, 1998.