GRADUATE SCHOOL APPROVAL RECORD

NORTHEASTERN UNIVERSITY
Graduate College of Computer and Information Science

Dissertation Title:	Modular Set-Based Analysis from Contracts
Author:	Philippe Meunier
Department:	Computer Science

Approved for Dissertation Requirements of the Doctor of Philosophy Degree

Dissertation Committee

Matthias Felleisen	Date
Mitchell Wand	Date
Karl Lieberherr	Date
Robert Bruce Findler	Date
Cormac Flanagan	Date

Head of Department

Larry Finkelstein	Date

Graduate School Notified of Acceptance

Director of the Graduate School	Date

Copy Deposited in Library

Signed	Date

DEPARTMENTAL APPROVAL RECORD

NORTHEASTERN UNIVERSITY
Graduate College of Computer and Information Science

Dissertation Title:	Modular Set-Based Analysis from Contracts
Author:	Philippe Meunier
Department:	Computer Science

Approved for Dissertation Requirements of the Doctor of Philosophy Degree

Dissertation Committee

Matthias Felleisen	Date
Mitchell Wand	Date
Karl Lieberherr	Date
Robert Bruce Findler	Date
Cormac Flanagan	Date

Head of Department

Larry Finkelstein	Date

Graduate School Notified of Acceptance

Director of the Graduate School	Date

MODULAR SET-BASED ANALYSIS FROM CONTRACTS

A dissertation presented by

Philippe Meunier

to the Faculty of the Graduate School of the College of Computer Science
and Information Science of Northeastern University in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts
May, 2006

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 dissertation, 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.

The result is a static debugger for checking modular programs with contracts, that is both sound and useful to programmers.

Contents

List of Figures

List of Tables

Chapter 1

Modules, Contracts, and Static Debugging

Few things are more frustrating for computer users than to see their data disappear in front of their eyes, followed by a cryptic error message informing them that something just went horribly wrong with the software they were using. Such occurrences are frequent enough that regularly saving one's work has become second nature for many users. Most software packages released today probably contain many such undetected errors that might later be triggered by unsuspecting users. Detecting bugs before releasing software has therefore become a major goal of software engineering.

Several approaches are used to detect bugs during the software development process. Currently the most common one is testing: unit testing, integration testing, system testing, along with systematic regression testing,

etc. While testing is and always will remain an essential part of software development, the testing space is generally so huge that bugs that occur only infrequently are difficult to detect.

Another approach is to change the software development process itself. Design reviews, code reviews, pair programming are for example all advocated by proponents of extreme programming. Other strategies try to emphasize automatic code generation from high level specifications. While this leads to great improvements in software design and reliability, none of those approaches ensure the creation of bug-free software packages.

A third approach to bug detection is to use formal methods to try to prove the correctness of software code. From sound type systems to theorem provers, such formal systems have been available for a long time, but the adoption of these advanced systems has been slow, due to both their inherent complexity as well as their sometimes poor running times. Since the formal approach is the only one that can guarantee the absence of bugs, or at least the absence of some classes of bugs, we believe efforts should be made to make formal methods more accessible to the software developers. This work is therefore focused on both the theoretical and practical aspects of using static analyses to help programmers create more reliable software. In particular we concentrate on the creation of a static program debugger that is at the same time powerful, reasonably fast, and easy to use.

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 [21] relies on a variant of set-based analysis [20, 30, 43]; others use a deep abstract interpretation, e.g., Bourdoncle's Syntox [7]; and yet others employ theorem proving, e.g., Detlefs et al.'s ESC [16] and its ESC/Java successor [22].

Experience with static debuggers shows that they work well for reasonably small programs. Using MrSpidey, some DrScheme users 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 DrScheme interpreter, dubbed MrEd [26], 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.

Over the past few years, PLT developers have added a first-order module system to PLT Scheme [24] and have equipped the module system with a contract system [18]. 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 dissertation makes five contributions to static debugging and software contracts. *First*, we explain how to construct a static debugger for modular 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 the contract-based, whole-program analysis computes its results in a modular manner. That is, the 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 also predicts the source and violation level of potential errors, i.e., it is sound with respect to the blame assignment of the contract system (though it is not complete, so false-positives are possible). *Fifth*, the analysis is parameterized over a predicate approximation relation and a theorem prover.

Chapter 2

Overview

In each of the following chapters we present a model of a static debugger. Each successive chapter introduces new complexities to the model, first starting with the lambda calculus, then adding modules and simple contracts, and finally adding unrestricted contracts. The models always consist of two parts: a dynamic part consisting of a runtime system and a static part consisting of a set-based analysis. In each chapter a correctness theorem ties the two parts together. Figure 2.1 provides an overview in graphical form of how these three pieces—the runtime system, the analysis, and the theorem—always combine. The vertical column on the left represents the runtime system. A compiler translates a program into a suitably annotated form. Execution is then defined via a reduction system.

The first horizontal row of Figure 2.1 depicts the analysis process, which usually will consists of three stages. First, it will partition the program into

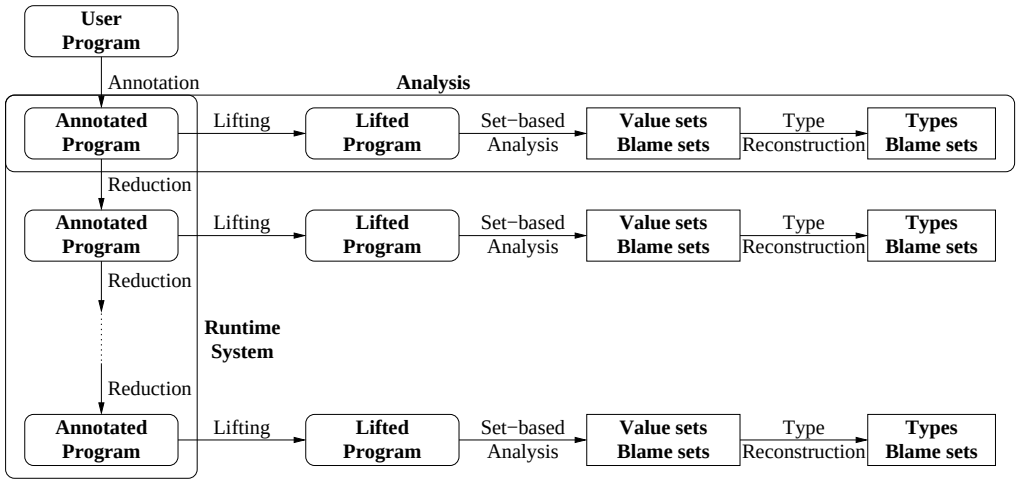

Figure 2.1: Runtime system and analysis overview.

module-like pieces by lifting expressions with contract annotations out of the main program (this stage will not appear in the simplest model we present in the next chapter). Second, the resulting collection of program pieces is analyzed with a 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 is summarized as set-of-values descriptions, dubbed *types*.

The rest of the grid in Figure 2.1 explains the proof technique we use in each chapter to prove the correctness of the analysis. Since each reduction step creates a complete program, the correctness proof proceeds via subject reduction. We re-apply the analysis after each reduction step. The proof of the soundness theorem then shows that the reductions preserve the types and the blame sets. It follows that the predictions of the analysis are conservative.

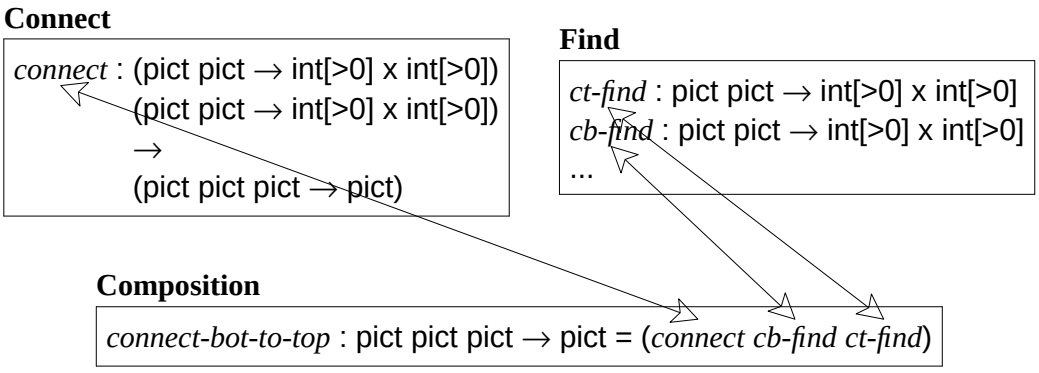

Figure 2.2: Example modules.

2.1 Sample Contracts, Sample Blame

Before we present our models, let us first illustrate the module and contract system at work to give an idea of the kind of problems we want to solve. Figure 2.2 shows an excerpt from our library for preparing figures (including Figure 2.2 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*.

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.

As our debugger models get more and more complex, we will therefore have to ensure that the analyses can always correctly predict contract violations and who is to be blamed for them. In the next chapter, as a warmup, we first consider the problem of analyzing the lambda calculus.

Chapter 3

The Lambda Calculus

We begin by recalling the basics of value-flow analysis for the untyped lambda calculus. Subsequent chapters will roughly follow the same structure as this one, first presenting the syntax for the calculus, then reduction rules, followed by the analysis proper, theorems about the analysis, a study of its complexity, and finally discussing related work.

3.1 The Calculus

In the first subsection, we introduce our surface syntax and internal syntax for an extended untyped lambda calculus. In the second subsection, we explain the translation from surface syntax into internal syntax.

$$\begin{array}{rcl} V & ::= & n \mid (\lambda x.E) \\ E & ::= & V \mid x \mid (E\ E) \mid (\text{if0}\ E\ E\ E) \end{array}$$

Figure 3.1: Surface syntax for the lambda calculus.

$$\begin{array}{rcl} V & ::= & n^{\ell} \mid (\lambda x^{\beta}.E)^{\ell} \\ E & ::= & V \mid x^{\beta} \mid (E\ E)^{\ell} \mid (\text{if0}\ E\ E\ E)^{\ell} \\ & & \mid (\text{blame}\ \lambda\ \mathcal{R})^{\ell} \end{array}$$

Figure 3.2: Annotated syntax for the lambda calculus.

3.1.1 User Syntax and Annotated Syntax

Figure 3.1 specifies the surface and internal syntaxes for expressions in the untyped lambda calculus with integers and if0 expressions. We use n for an integer, and x for a lexical variable. Values are either integers or functions. We make the simplifying assumption that the test part of an if0 expression can return any value; the "then" branch is evaluated if this value is 0. From this grammar for expressions we then define programs as closed expressions.

A program in the surface syntax is ill-suited for analysis. We therefore elaborate such programs into the internal syntax of Figure 3.2. This syntax contains labeled versions of all syntactic phrases: β for labels on variables and ℓ for all others. It also contains a new form (blame λ $\mathcal{R}$) that aborts the program, blames the programmer (represented by the λ symbol) for violating a constraint of the lambda calculus itself, and colors the corresponding code in red ($\mathcal{R}$).

Consider the following example:

$$((\lambda x.x)\ 3)$$

The annotation of this program yields the following:

$$((\lambda x^{\beta_2}.x^{\beta_2})^{\ell_\lambda}\ 3^{\ell_n})^{\ell_a}$$

In the annotated program, each subexpression (except for variables) has a unique label.

3.1.2 Annotation Process

The rules of Figure 3.3 define the annotation process. The goal is to annotate every expression with a unique label (except for variables, where binder and references for a given variable all share the same label).[1] These labels are required by the analysis: a label on an expression represents the abstract values of that expression.

The annotation judgement is of the form

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

where e' is the annotated version of e. Variable references share their label with their respective binder (rules VAR and MODVAR).

[1]The annotation rules of Figure 3.3 would have to pass around some state, such as a counter, to ensure that labels are indeed unique. We omit such state here for clarity.

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

$$\frac{\Gamma(x) = \beta}{\Gamma \vdash^{\mathsf{a}}_{\mathsf{e}} x \rightsquigarrow x^{\beta}}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash^{\mathsf{a}}_{\mathsf{e}} e_1 \rightsquigarrow e_1' \qquad \Gamma \vdash^{\mathsf{a}}_{\mathsf{e}} e_2 \rightsquigarrow e_2'}{\Gamma \vdash^{\mathsf{a}}_{\mathsf{e}} (e_1\ e_2) \rightsquigarrow (e_1'\ e_2')^{\ell}}\ (\textsc{App})$$

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

Figure 3.3: Annotation judgments for the lambda calculus.

$$\begin{array}{lll}
((\lambda x^{\beta}.e)^{\ell_\lambda}\ v^{\ell_v})^{\ell_a} & \longrightarrow e[v^{\ell_v}/x^{\beta}] & \textsc{subst} \\
(n^{\ell_n}\ v^{\ell_v})^{\ell_a} & \longrightarrow (\text{blame } \lambda\ \mathcal{R})^{\ell_a} & \textsc{app-error} \\
(\text{if0}\ 0^{\ell_0}\ e_1\ e_2)^{\ell} & \longrightarrow e_1 & \textsc{if0-true} \\
(\text{if0}\ v^{\ell_v}\ e_1\ e_2)^{\ell} & \longrightarrow e_2 & \textsc{if0-false}
\end{array}$$

Figure 3.4: Reduction rules for the lambda calculus.

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.

3.2 Reduction Rules

Figure 3.4 defines the reduction semantics for annotated programs. The goal of the process is to reduce the expression to a value. The relation $\longrightarrow$ is the one-step reduction; the set of evaluation contexts for expressions is:

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

In Figure 3.4 we use n to represent runtime integers and v to represent any value whatsoever. 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. With this in mind, the reduction rules are then as follows.

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 abuses the programming language. 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.

3.3 The Analysis

The analysis we present for our extended untyped lambda calculus is a set-based analysis [3, 49, 30, 45, 20] based on Shivers's 0-CFA [50]. The analysis is designed to be applicable at each stage in the reduction process, rendering it well-suited for a subject reduction argument.

For each expression in the analyzed program, it computes a conservative approximation of the set of possible values and set of possible errors that the expression might evaluate to at runtime. This is done in two phases: first constraints are generated from the annotated version of the program, relating the various flows of values in the program. Any solution to the constraints is a conservative approximation of the runtime behavior. In practice though the analysis finds a minimal conservative solution by computing the closure of the constraints. From such a solution, the second phase reconstructs a type-like description that can be displayed to the user. These two phases are described more in detail in the next two sections.

3.3.1 Constraints Generation

The purpose of the analysis is to predict (1) the flow of values and (2) potential errors. Accordingly, the analysis produces two results: a mapping φ from labels to sets of labels and a mapping ψ from labels to offender and severity. The former points to values in the program. The latter indicates who is to be blamed for the error (only, for now, the programmer, represented by λ, for violating a constraint of the lambda calculus itself) and which color should be used to highlight the offending code in the static debugger (red, represented as $\mathcal{R}$).

The analysis 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 error culprit and severity

$Source \diagdown Sink$	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$
n^{ℓ_n}	$\{\ell_n\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R} \rangle\} \subseteq \psi(\ell_a)$
$(\lambda x^{\beta}.e^{\ell})^{\ell_\lambda}$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\beta)$ $\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell) \subseteq \varphi(\ell_a)$

Table 3.1: Constraints creation for the simple lambda calculus.

that satisfy these constraints is a sound approximation to the actual runtime behavior of the program. A minimal sound approximation is the solution.

The constraint generation algorithm needs to identify value sources and value sinks in programs. In the grammar of Figure 3.1 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 main value sink in our language.

The matrix in Table 3.1 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.

Let us explain how to read Table 3.1. The first constraint in the table specifies the creation of a single blame set constraint for every possible pair of integer source and application in the program. The constraint says that, if an integer (represented by ℓ_n) flows into the operator position (represented by ℓ_5) of an application (represented by ℓ_a), then the programmer (represented

by λ) has to be blamed for the error and the application (represented by ℓ_a) has to be highlighted in red ($\mathcal{R}$).

Next we have the combination of λ-abstractions and applications. The table 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 operator 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).

Additional Constraints

Finally, to get the analysis started, we must supplement Table 3.1 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 expression acts as an error source: see the top two rows of Table 3.2.

The last row in Table 3.2 describes the flows from the two branches of an if0 expression to the whole expression. Naturally there are no flows out of the test since if0 expressions act as (trivial) sinks for the values flowing out of their tests.

Once all the constraints have been generated from a program's text, they have to be solved to obtain the solution. This can be done using standard technology for solving Horn constraints. See for example Palsberg and

$n^\ell \quad (\lambda x^\beta.e^{\ell_e})^\ell$	$\{\ell\} \subseteq \varphi(\ell)$
(blame λ $\mathcal{R}$)$^\ell$	$\{\langle\lambda, \mathcal{R}\rangle\} \subseteq \psi(\ell)$
(if0 e^{ℓ_0} e^{ℓ_1} e^{ℓ_2})$^\ell$	$\varphi(\ell_1) \subseteq \varphi(\ell)$ $\varphi(\ell_2) \subseteq \varphi(\ell)$

Table 3.2: Additional constraints for the simple lambda calculus.

Schwartzbach [47]. Of course only computing the mapping φ actually requires a solving phase, since no constraint in Table 3.1 involves flows between blame sets.

3.3.2 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 3.5 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 corresponds to an integer type and a label on an abstraction corresponds to a function type. The surrounding rec type constructor takes accounts

$$
\begin{aligned}
\mathcal{R}^{\varphi}(\ell) &\stackrel{def}{=} \{\ell\} \cup \mathcal{R}_u^{\varphi}(\ell) \\
\mathcal{R}_u^{\varphi}(\ell) &\stackrel{def}{=} \textstyle\bigcup_{\ell_i \in \varphi(\ell)} \mathcal{R}_t^{\varphi}(\ell_i) \\
\mathcal{R}_t^{\varphi}(\ell) &\stackrel{def}{=} \begin{cases} \{\ell\} \text{ if } n^{\ell} \\ \{\ell\} \cup \mathcal{R}^{\varphi}(\ell_1) \cup \mathcal{R}^{\varphi}(\ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \end{cases} \\
\mathcal{T}^{\varphi}(\ell) &\stackrel{def}{=} (\text{rec } ([\ell_i \ \mathcal{T}_u^{\varphi}(\ell_i)]_{\ell_i \in \mathcal{R}^{\varphi}(\ell)} \ldots) \ \ell) \\
\mathcal{T}_u^{\varphi}(\ell) &\stackrel{def}{=} (\text{union } \mathcal{T}_t^{\varphi}(\ell_i)_{\ell_i \in \varphi(\ell)} \ldots) \\
\mathcal{T}_t^{\varphi}(\ell) &\stackrel{def}{=} \begin{cases} \text{int if } n^{\ell} \\ (\ell_1 \rightarrow \ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \end{cases}
\end{aligned}
$$

Figure 3.5: Type reconstruction for the lambda calculus.

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 [40, 11] for the reconstructed types.

3.4 Soundness

We adapt Wand and Williamson's proof technique [54] to prove the soundness of our analysis. Let $[\![e]\!]$ be the set of constraints that the analysis generates when given the annotated expression e, 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 expression* $e^{\ell'}$*, either:*

- e *reduces to* v^{ℓ} *and then* $[\![e]\!] \models \{\ell\} \subseteq \varphi(\ell')$*,*
- *or* e *reduces to* $(\mathit{blame}\ \lambda\ \mathcal{R})^{\ell}$ *and then* $[\![e]\!] \models \{\langle \lambda, \mathcal{R} \rangle\} \subseteq \psi(\ell)$*,*
- *or* e *reduces forever.*

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 value. If the program terminates abnormally because of a runtime error then the analysis conservatively predicts the error, its location (as represented by ℓ), and its severity.

The proof follows the one by Wand and Williamson, extended to handle integers, if0 expressions, and blame sets. It proceeds in three steps.

First, for two expressions e and e' such that $e \longrightarrow e'$, define a relation \$ in such a way that it relates sub-expressions of e' and their labels to the corresponding sub-expressions of e and their labels, based on the effect the reduction relation has on the shape of e to get e'. This step relies on the fact that the reduction relation in Figure 3.4 does not introduce new labels and rearranges existing labels in a specific way.

Second, show that, for two expression e and e' such that $e \longrightarrow e'$, the value set (blame set, respectively) of every sub-expression in e' is a subset of the value set (blame set) of the \$-related sub-expression in e. This in essence shows that the reduction relation can only make the value set (blame

set) of an expression become smaller, which is at the heart of the soundness theorem.

Third, based on the previous step, show that, for two expression e and e' such that $e \longrightarrow e'$, if φ (ψ, respectively) satisfies the set of constraints $[\![e]\!]$, then φ (ψ) is also a solution of the set of constraints $[\![e']\!]$. This gives us a preservation lemma. Combining that preservation lemma with a simple progress lemma gives us the theorem above.

3.5 Analysis Complexity

Generating constraints from the source code of the program is easily done by traversing the program's abstract syntax tree, which takes an amount of time linear in the size of the program.

Once all the constraints have been generated they have to be solved to obtain the minimal solution (the minimal fixed point for the constraints). This part of the analysis is in fact the most time-consuming one. The overall complexity of the analysis is therefore highly dependent on the way constraints are represented in the analyzer and on the algorithm used to solve them. In practice representing value sets as nodes in a graph and constraints between value sets as edges in the graph works well. Computing the minimal solution amounts then to computing the closure of the graph. Such a simple set-based flow analysis based on the transitive closure of set constraints (a form of monovariant SBA for shallow patterns [41]) can be done in time cubic

in the size of programs [3]. While some programs can be analyzed in almost linear time [43], there does not appear to be better bounds without imposing restrictions on the programming language analyzed. This complexity is known as the "cubic bottleneck" [32] and is one of the main reasons why a modular analysis is desirable when debugging big programs.[2]

Once a solution has been computed, reconstructing the type for a given expression requires a recursive traversal of the computed abstract value sets to determine the set of labels reachable from the label for that expression. This traversal is necessary because, as Heintze observed [31], the type information we seek is not explicitly available in the computed value sets but is rather collectively encoded in those sets. Because the graph resulting from the transitive closure computation contains many subgraphs that are reachable from many different nodes, and because of the additional potential existence of many cycles in that graph (resulting from the presence of recursive data structures or recursive functions in the analyzed program) a naive type reconstruction algorithm could take exponential time in the size of the graph. A slightly less naive algorithm that uses memoization can reconstruct a type in time linear in the size of the graph though, and hence in time linear in the size of the original program. Since there are at most a linear number of labels, the type reconstruction phase then takes at most a quadratic time. In practice though those types are only used by a graphi-

[2] A sound solution that assigns the abstract value "any" to all value sets can be computed in linear time, but such overly conservative non-minimal solution is of little value to the user of a static debugger.

cal static debugger for display to the user. Their computation can therefore even be done on request, rather than at once. Regardless, the complexity of the analysis is dominated by the computation of the minimal solution to the constraints. The whole analysis has therefore a cubic worst-cast running time $\mathcal{O}(n^3)$, where n is the size of the original program.

3.6 Related Work

The analysis we have just presented for our extended untyped lambda calculus is a set-based analysis [3, 49, 30, 45, 20] based on Shivers's 0-CFA [50]. Cousot and Cousot [13] show that such set-based analyses are special cases of their abstract interpretation framework [12].

There is a general equivalence between polyvariant flow analyses and type systems with intersection and union types [31, 46, 55]. The system we have presented so far is monovariant, in the sense that functions are analyzed only once, even when used multiple times. It is fairly straightforward to extend the analysis to k-CFA [50], by keeping track of the different applications a given function flows through before being applied, or to instead use Agesen's cartesian product algorithm [2], which in both cases will make the analysis polyvariant [51].

Identifying the source of type errors in ML-like languages is notoriously difficult [53]. Since we use a flow analysis, our graphical debugger can easily trace values back to their source when a contract violation occurs [21]. The

closest equivalent is Haack and Wells's type error slicing system [29], which uses fairly complex annotations to basically compute the same information as we do.

Chapter 4

Modules and Simple Contracts

The cubic bottleneck for the running time of the analysis we described in the previous chapter makes it in practice difficult to analyze programs larger than a few thousand lines. The analysis is also a whole program analysis, making it impossible to analyze the different parts of a program in isolation of each other.

To solve these problem we now present a new analysis for a lambda calculus extended with modules. We use a runtime contract system similar to the one described by Findler and Felleisen [18] at the interface of modules. The analysis then extracts from these runtime contracts enough static information to still compute precise results.

In this chapter we restrict ourselves to a contract language that contains only integer and arrow contracts. This allows us to introduce the machinery necessary for a modular analysis, before we consider more complex contracts in the next chapter.

4.1 Contract Calculus

As before, we introduce in the first subsection our surface syntax and internal syntax of programs with modules and simple contracts. In the second subsection, we explain the translation from surface syntax into internal syntax, which is more complex than in the case of the simple lambda calculus.

4.1.1 User Syntax and Annotated Syntax

Figure 4.1 specifies the surface syntax of our lambda calculus with modules and contracts, 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's model for contracts [18] explains them 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.

As indicated before, the language of contracts is limited to just two kinds of constructs: one construct for validating that a value is an integer, which shows how the model deals with basic types, and one construct for validating that a value is a function.

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

Figure 4.1: Surface syntax for the lambda calculus with modules and simple contracts.

$$
\begin{array}{lcl}
P & ::= & E \mid MP \\
M & ::= & (\text{module } f^{\beta}\ V)^{\ell} \\
V & ::= & n^{\ell} \mid (\lambda x^{\beta}.E)^{\ell} \\
 & & \mid ((C \hat{\rightarrow} C)^{\ell\ell'}_{f} \Leftarrow V)^{\ell_c} \\
E & ::= & V \mid x^{\beta} \mid f^{\beta} \mid (E\ E)^{\ell} \mid (\text{if0 } E\ E\ E)^{\ell} \\
 & & \mid (C \Leftarrow E)^{\ell} \mid (\text{blame } L\ \mathcal{R})^{\ell} \\
C & ::= & \text{int}^{\ell\ell'}_{f} \mid (C \rightarrow C)^{\ell\ell'}_{f} \\
 & & \mid (C \hat{\rightarrow} C)^{\ell\ell'}_{f} \\
L & ::= & f \mid \mu \mid \lambda
\end{array}
$$

Figure 4.2: Annotated syntax for the lambda calculus with modules and simple contracts.

Again we need to elaborate such programs into the internal syntax of Figure 4.2 for the purpose of the analysis. As before the syntax contains labeled versions of all syntactic phrases—β for labels on lexical and module variables and one or two labels ℓ for all others. The major new expression form is $(C \Leftarrow E)$. It evaluates the expression E to a value and checks whether the value satisfies the contract C. Blame expressions can now use the name of the variable defined in a module (or μ for the main expression) to blame that specific module when a contract violation is detected.

The annotated grammar also has a new contract form $(C \hat{\rightarrow} C)_f$. We refer to it as a "blessed" arrow contract. It 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.[1]

Consider the following example:

$$(\text{module } f \ (\text{int}\rightarrow\text{int}) \ (\lambda x.x))$$
$$(f \ 3)$$

The annotation of this program yields the following:

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

In the annotated program, each subexpression (except for variables) has a unique label; each contract has two unique labels and a module name (or

[1] For reduction purposes blessed arrows could be replaced with eta expansion. However we will later see that this would break the modularity of the analysis by creating free variables during the lifting phase of Section 4.3.1. See Footnote 3.

μ). Furthermore, the reference to the module variable f is wrapped with a contract check that ensures the module satisfies its contract.

4.1.2 Annotation Process

The rules of Figure 4.3 define the annotation process for our language with modules and simple contracts. Unlike expressions, every contract is annotated with two unique labels and a module name. These annotations are required by the analysis: 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

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

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

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

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

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

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

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

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

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

$$\frac{}{\Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathsf{c}} \text{int} \rightsquigarrow \text{int}^{\ell\ell'}_{f}} \text{ (IntC)} \qquad \frac{\begin{array}{c}\Delta,\Gamma,g,f \vdash^{\mathsf{a}}_{\mathsf{c}} c_d \rightsquigarrow c_d' \\ \Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathsf{c}} c_r \rightsquigarrow c_r'\end{array}}{\Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathsf{c}} (c_d \rightarrow c_r) \rightsquigarrow (c_d' \rightarrow c_r')^{\ell\ell'}_{f}} \text{ (ArrowC)}$$

Figure 4.3: Annotation judgments for the lambda calculus with modules and simple contracts.

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 for the main expression) 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). 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 [18]'s rule for assigning blame in the presence of higher-order functions. Annotating contracts is otherwise straightforward.

As before, 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.

4.2 Reduction Rules

Figure 4.4 defines the reduction semantics for annotated programs with contract checks. 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:

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

$$
\begin{array}{lll}
((\lambda x^{\beta}.e)^{\ell_\lambda}\ v^{\ell_v})^{\ell_a} & \longrightarrow e[v^{\ell_v}/x^{\beta}] & \textsc{subst} \\
(n^{\ell_n}\ v^{\ell_v})^{\ell_a} & \longrightarrow (\text{blame } \lambda\ \mathcal{R})^{\ell_a} & \textsc{app-error} \\
(\text{if0 } 0^{\ell_0}\ e_1\ e_2)^{\ell} & \longrightarrow e_1 & \textsc{if0-true} \\
(\text{if0 } v^{\ell_v}\ e_1\ e_2)^{\ell} & \longrightarrow e_2 & \textsc{if0-false} \\
(\text{int}_f^{\ell\ell'} \Leftarrow n^{\ell_n})^{\ell_c} & \longrightarrow n^{\ell} & \textsc{int-int} \\
(\text{int}_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \longrightarrow (\text{blame } f\ \mathcal{R})^{\ell'} & \textsc{int-lam} \\
((c_1 \rightarrow c_2)_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \longrightarrow ((c_1 \hat{\rightarrow} c_2)_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} & \textsc{lam-lam} \\
((c_1 \rightarrow c_2)_f^{\ell\ell'} \Leftarrow n^{\ell_n})^{\ell_c} & \longrightarrow (\text{blame } f\ \mathcal{R})^{\ell'} & \textsc{lam-int} \\
(((c_1 \hat{\rightarrow} c_2)_f^{\ell\ell'} \Leftarrow \vec{v}^{\ell_v})^{\ell_c}\ w^{\ell_w})^{\ell_a} & \longrightarrow (c_2 \Leftarrow (\vec{v}^{\ell_v}\ (c_1 \Leftarrow w^{\ell_w})^{\mathcal{L}^+(c_1)})^{\mathcal{L}^-(c_2)})^{\mathcal{L}^+(c_2)} & \textsc{split-arrow}
\end{array}
$$

Figure 4.4: Reduction rules for the lambda calculus with modules and simple contracts.

Expression evaluation contexts do not include contexts for contracts, which are syntax, not values. The grammar for annotated programs guarantees that contracts never show up outside a contract check.

The module context becomes relevant in only one situation:

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

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 4.4 we use n to represent again runtime integers, $\vec{v}$ to represent this time functions or functions with any number of blessed arrow contract

checks wrapped around them, and v and w to represent any values whatsoever. Again a blame redex in any context reduces the entire program in one step to just that expression, whereupon reduction stops.

The first four rules in Figure 4.4 are as in the previous chapter. The rest of the reduction rules concern contract checking:

- The INT-INT and INT-LAM rules check that a given value is an integer. If it is, the check reduces to the tested value. Importantly, the label ℓ on the int contract becomes the label on n. The reason is that in the analysis, label ℓ acts as an abstract value source for the contract $\mathrm{int}_f^{\ell\ell'}$. The reduction rule thus guarantees that the value n 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. If the check for an integer fails, the INT-LAM blames the appropriate module using the module variable annotation from the int contract. The label of the blame expression is the second label on the contract: ℓ' acts as an abstract value sink during the analysis and the reduction rule thus guarantees preservation.

- LAM-LAM and LAM-INT correspond to the rules INT-INT and INT-LAM, respectively. The only difference is the presence of a blessed arrow in the LAM-LAM rule: 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 or even not at all [18]. Hence, the rule introduces 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-ARROW 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, transformed into a series of flat contracts [18]. 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 $\mathcal{L}^+$ and $\mathcal{L}^-$ functions (which extract the first and second label of a contract, respectively) 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 we now present.

4.3 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 result is a sequence of terms made of a mix of modules and expressions. The second and third step can then proceed in a way that parallels the analysis described in the previous chapter: generate constraints, both from expressions and contracts, and then produce types from a solution to these constraints. We now describe those three phases.

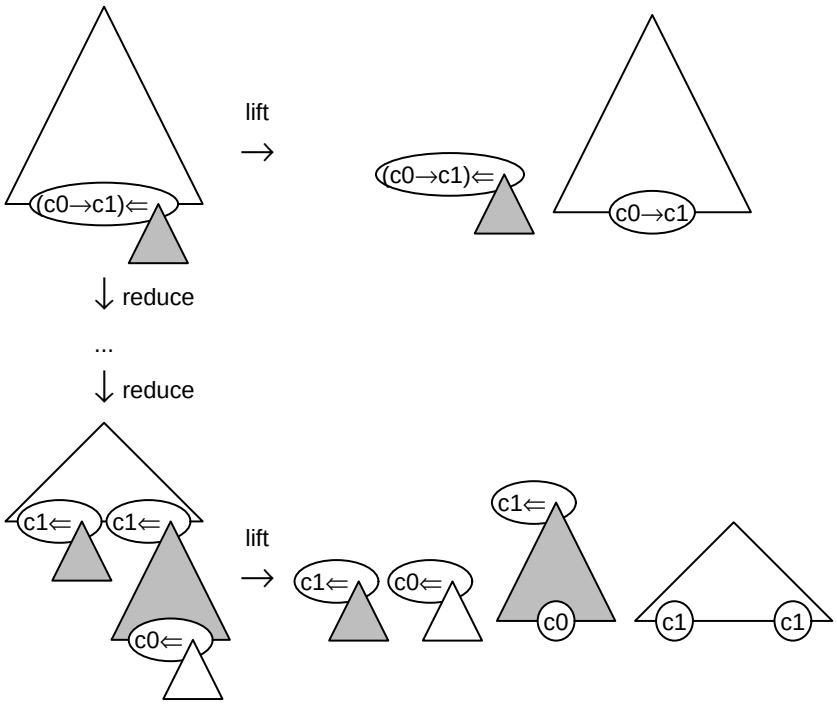

Figure 4.5: Lifting subtrees.

4.3.1 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 4.5 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.

For a second example, look again at the program in Figure 2.2. Recall that **Find** and **Connect** are separate modules and that **Composition** is the main expression. The module variable references to *connect*, *cb-find*, and *ct-find* in the main (*connect find-cb find-ct*) expression are annotated with their respective contract during the initial annotation phase. Once we have the program in the internal syntax, the lifting step described above removes these three module variable references, lifting them to the top level, and replaces them in the main expression with the corresponding contracts.

The main expression can now be analyzed without having to look at the definitions of *connect*, *cb-find*, and *ct-find* in the other two modules.

If the annotated program is reduced an interesting situation will eventually occur: the contract on *connect* will be a blessed arrow as will be the contracts on both *find-cb* and *find-ct*. At that point the next reduction will split *connect*'s blessed arrow contract and place *connect*'s first argument subcontract on the already annotated *find-cb*, place *connect*'s second argument subcontract on the already annotated *find-ct*, and place *connect*'s result subcontract on the application. The result is that both *find-cb* and *find-ct* will have two contracts on top of them: their own and the one coming from the argument part of *connect*'s contract (note that this is allowed by the grammar for annotated programs in Figure 4.2). Lifting this program will then result in two small trees (one for *find-cb* and one for *find-ct*), which will each have a contract at the top, a contract at the bottom, and no expression in between! This makes sense: it is the way we will make sure that *find-cb* and *find-ct* can indeed be given as arguments to *connect*, but ensuring so while only analyzing the main expression.

Figure 4.6 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 are 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.

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).

4.3.2 Constraints Generation

After the lifting step, programs satisfy additional syntactic invariants: see the grammar in Figure 4.7. This new grammar differs from the one in Figure 4.2 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.

Again, the analysis has to produce two results: a mapping φ from labels to sets of labels, as before, and a mapping ψ from labels to error culprit (that include now module names) and severity (still only red, for now).

To do so, the constraint generation algorithm again needs to identify value sources and value sinks in the analyzed program and to consider all the possible combinations, regardless of where the sources and sinks appear in the program (i.e. constraints should be created even when sources and sinks appear in different modules). As mentioned before, contracts play the role of both sources and sinks. Contracts that occur as leaves in an expression are

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{m}} m_i \rightsquigarrow es_i \ldots m_i' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e \rightsquigarrow es \ldots e'}{\vdash^{\mathtt{l}}_{\mathtt{p}} m_i \ldots e \rightsquigarrow es_i \ldots m_i' \ \ldots \ es \ldots e'} \text{ (Program)}$$

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

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} n^{\ell}_{e\ldots} \rightsquigarrow n^{\ell}_{e\ldots}} \text{ (Int)} \quad \frac{\vdash^{\mathtt{l}}_{\mathtt{e}} e \rightsquigarrow es \ldots e'}{\vdash^{\mathtt{l}}_{\mathtt{e}} (\lambda x^{\beta}.e)^{\ell}_{e_1\ldots} \rightsquigarrow es \ldots (\lambda x^{\beta}.e')^{\ell}_{e_1\ldots}} \text{ (Lam)}$$

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} x^{\beta} \rightsquigarrow x^{\beta}} \text{ (Var)} \qquad \frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} f^{\beta} \rightsquigarrow f^{\beta}} \text{ (ModVar)}$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{e}} e_1 \rightsquigarrow es_1 \ldots e_1' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e_2 \rightsquigarrow es_2 \ldots e_2'}{\vdash^{\mathtt{l}}_{\mathtt{e}} (e_1\ e_2)^{\ell} \rightsquigarrow es_1 \ldots es_2 \ldots (e_1'\ e_2')^{\ell}} \text{ (App)}$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{e}} e_0 \rightsquigarrow es_0 \ldots e_0' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e_1 \rightsquigarrow es_1 \ldots e_1' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e_2 \rightsquigarrow es_2 \ldots e_2'}{\vdash^{\mathtt{l}}_{\mathtt{e}} (\text{if0 } e_0\ e_1\ e_2)^{\ell} \rightsquigarrow es_0 \ldots es_1 \ldots es_2 \ldots (\text{if0 } e_0'\ e_1'\ e_2')^{\ell}} \text{ (If0)}$$

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} (\text{blame } f\ s)^{\ell} \rightsquigarrow (\text{blame } f\ s)^{\ell}} \text{ (Blame)}$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{c}} c \rightsquigarrow es_c \ldots c' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e \rightsquigarrow es \ldots e'}{\vdash^{\mathtt{l}}_{\mathtt{e}} (c \Leftarrow e)^{\ell} \rightsquigarrow es_c \ldots es \ldots (c' \Leftarrow e')^{\ell} c'} \text{ (Check)}$$

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{c}} \text{int}^{\ell\ell'}_{f} \rightsquigarrow \text{int}^{\ell\ell'}_{f}} \text{ (IntC)}$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{c}} c_d \rightsquigarrow es_d \ldots c_d' \qquad \vdash^{\mathtt{l}}_{\mathtt{c}} c_r \rightsquigarrow es_r \ldots c_r'}{\vdash^{\mathtt{l}}_{\mathtt{c}} (c_d \rightarrow c_r)^{\ell\ell'}_{f} \rightsquigarrow es_d \ldots es_r \ldots (c_d' \rightarrow c_r')^{\ell\ell'}_{f}} \text{ (ArrowC)}$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{c}} (c_d \rightarrow c_r)^{\ell\ell'}_{f} \rightsquigarrow es}{\vdash^{\mathtt{l}}_{\mathtt{c}} (c_d \hat{\rightarrow} c_r)^{\ell\ell'}_{f} \rightsquigarrow es} \text{ (BArrowC)}$$

Figure 4.6: Lifting judgments for the lambda calculus with modules and simple contracts.

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

Figure 4.7: Analyzed syntax for the lambda calculus with modules and simple contracts.

Source$\diagdown$*Sink*	$\text{int}_{h}^{\ell_5^+ \ell_5^-}$
n^{ℓ_n}	
$\text{int}_{f}^{\ell_1^+ \ell_1^-}$	
$(\lambda x^{\beta}.e^{\ell})^{\ell_\lambda}$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R}\rangle\} \subseteq \psi(\ell_5^-)$
$(c_g^{\ell_1^+ \ell_1^-} \rightarrow c_f^{\ell_2^+ \ell_2^-})_f^{\ell_3^+ \ell_3^-}$	$\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R}\rangle\} \subseteq \psi(\ell_5^-)$

Source$\diagdown$*Sink*	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$	$(c_i^{\ell_7^+ \ell_7^-} \rightarrow c_h^{\ell_8^+ \ell_8^-})_h^{\ell_5^+ \ell_5^-}$
n^{ℓ_n}	$\{\ell_n\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R}\rangle\} \subseteq \psi(\ell_a)$	$\{\ell_n\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R}\rangle\} \subseteq \psi(\ell_5^-)$
$\text{int}_{f}^{\ell_1^+ \ell_1^-}$	$\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R}\rangle\} \subseteq \psi(\ell_a)$	$\{\ell_1^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R}\rangle\} \subseteq \psi(\ell_5^-)$
$(\lambda x^{\beta}.e^{\ell})^{\ell_\lambda}$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\beta)$ $\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell) \subseteq \varphi(\ell_a)$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_7^+) \subseteq \varphi(\beta)$ $\{\ell_\lambda\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell) \subseteq \varphi(\ell_8^-)$
$(c_g^{\ell_1^+ \ell_1^-} \rightarrow c_f^{\ell_2^+ \ell_2^-})_f^{\ell_3^+ \ell_3^-}$	$\{\ell_3^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\ell_1^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_2^+) \subseteq \varphi(\ell_a)$	$\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_7^+) \subseteq \varphi(\ell_1^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_2^+) \subseteq \varphi(\ell_8^-)$

Table 4.1: Constraints creation for the lambda calculus with modules and simple contracts.

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

$$\mathrm{int}_f^{\ell^+\ell^-}$$

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.

Table 4.1 is similar to Table 3.1, only with more source and sink combinations. We therefore only explain here the constraints in the bottom right cell of the second part of the table.

The first (second, respectively) of those two constraints says that, if an abstract functional value, represented by the arrow contract labeled with ℓ_3^+, flows into a function check, represented by the arrow contract labeled with ℓ_5^-, then the abstract value source from the domain (range) of the function check (functional value), represented by ℓ_7^+ (ℓ_2^+), flows into the abstract value check from the domain (range) of the functional value (function check) represented by ℓ_1^- (ℓ_8^-).

The blame constraints in Table 4.1 always use the name h associated with the sink (or λ when the program 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 4.1.2. That rule switches the two module variable names used

by the $\vdash^{\mathsf{a}}_{\mathsf{c}}$ 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 4.5, 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 4.1 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.

Additional Constraints

Finally, as in the previous chapter, we must supplement Table 4.1 with rules that get the flows initiated for all the value sources. See the top row of Table 4.2, where integer and arrow contracts are treated as abstract value sources.

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.

Initially, a module contributes only its single value to the analysis. The last row in Table 4.2 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, they are solved exactly as described in the previous chapter, by computing the closure of a graph.

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

Table 4.2: Additional constraints for the lambda calculus with modules and simple contracts.

4.3.3 Type Reconstruction

Since the contract language considered in this chapter is quite simple, extending the type reconstruction process to handle those new abstract values is straightforward. See Figure 4.8.

In the previous chapter type reconstruction was only of interest because we wanted our static graphical debugger to use a type-like representation when displaying the results of the analysis to the user. Here, however, these types are also useful for the formulation of the analysis soundness theorem of the next section.

4.4 Soundness

Let $[\![p]\!]$ be the set of constraints that the analysis generates when given the lifted program p, and, sa before, let $\models$ denote implication between sets of constraints: for two sets of constraints A and A', we have $A \models A'$ if and only

$$
\begin{aligned}
\mathcal{R}^{\varphi}(\ell) &\stackrel{def}{=} \{\ell\} \cup \mathcal{R}_u^{\varphi}(\ell) \\
\mathcal{R}_u^{\varphi}(\ell) &\stackrel{def}{=} \textstyle\bigcup_{\ell_i \in \varphi(\ell)} \mathcal{R}_t^{\varphi}(\ell_i) \\
\mathcal{R}_t^{\varphi}(\ell) &\stackrel{def}{=} \begin{cases} \{\ell\} \text{ if } n^{\ell} \text{ or } \mathrm{int}_f^{\ell\ell'} \\ \{\ell\} \cup \mathcal{R}^{\varphi}(\ell_1) \cup \mathcal{R}^{\varphi}(\ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \text{ or } (c_g^{\ell_1' \ell_1} \to c_f^{\ell_2 \ell_2'})_f^{\ell\ell'} \end{cases} \\
\mathcal{T}^{\varphi}(\ell) &\stackrel{def}{=} (\mathrm{rec}\ ([\ell_i\ \mathcal{T}_u^{\varphi}(\ell_i)]_{\ell_i \in \mathcal{R}^{\varphi}(\ell)} \ldots)\ \ell) \\
\mathcal{T}_u^{\varphi}(\ell) &\stackrel{def}{=} (\mathrm{union}\ \mathcal{T}_t^{\varphi}(\ell_i)_{\ell_i \in \varphi(\ell)} \ldots) \\
\mathcal{T}_t^{\varphi}(\ell) &\stackrel{def}{=} \begin{cases} \mathrm{int} \text{ if } n^{\ell} \text{ or } \mathrm{int}_f^{\ell\ell'} \\ (\ell_1 \to \ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \text{ or } (c_g^{\ell_1' \ell_1} \to c_f^{\ell_2 \ell_2'})_f^{\ell\ell'} \end{cases}
\end{aligned}
$$

Figure 4.8: Type reconstruction for the lambda calculus with modules and simple contracts.

if every solution of A is a solution of A'. Given this machinery, an adaptation of Wand and Williamson's soundness theorem for our modular analysis is as follows.

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

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

where π indicates the party to blame for the violation: either a module variable name like f, or μ for the main expression, or λ for a violation by the programmer of a constraint of the lambda calculus itself.

The proof follows the lines of the one described in the previous chapter.

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 4.4 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 theorem above says that the analysis will predict a value with label ℓ as the final result of the program, but we want a more informative theorem that says that the analysis will predict that that result is in fact 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 3. *For a given annotated program p, let $p' \stackrel{\text{def}}{=} m' \ldots e^{\ell'}$ be such that $\vdash^{\mathsf{l}}_{\mathsf{p}} p \rightsquigarrow p'$. Then either:*

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

where $\leq$ *is subtyping between recursive types [5, 34] and* π *has the same meaning as before.*

Proof Sketch. We adapt again 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 type preservation property for these sets of constraints using Wand and Williamson's technique and the fact that all contract checking reductions in Figure 4.4 ensure that types are preserved when a value crosses a contract boundary. □

4.5 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. This makes sense if the analysis is defined compositionally (i.e. if the result of analyzing a term only depends on the results of analyzing the term's subterms). In contrast, we have formulated the analysis in terms of the *entire* program,

and we now have to 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 4. *Given an annotated program p, let p' be such that $\vdash^{\mathtt{l}}_{\mathtt{p}} p \rightsquigarrow p'$. 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 $\vdash^{\mathtt{l}}_{\mathtt{p}} p \rightsquigarrow p'$. 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 $t = (\mathit{module}\ f^{\beta}\ v^{\ell_v})^{\ell_m}$ and $t' = (c^{\ell^+ \ell^-}_{fg} \Leftarrow f^{\beta})^{\ell_c}$.*

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 inter-tree flow is an artifact of our lifting process. 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.[2]

Proof Sketch. A close look at the syntax of Figure 4.7 shows that inter-tree flows can only occur in the following two 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; or (2) 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 4.1 and 4.2 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]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 values exported from modules [56].

(2a) 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-ARROW 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.[3] Leaks through lexical variables are thus impossible, too.

(2b) Module variables are the only remaining mechanism for inter-tree value propagation. Recall (Sec. 4.1.2) 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

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

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

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

[3] This is the invariant that would be broken if the LAM-LAM reduction rule in Figure 4.4 used eta expansion instead of a blessed arrow. See Footnote 1.

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 inter-tree 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 4.4), which then makes the second lifted tree in the example above independent of the first one. □

4.6 Analysis Complexity

The constraints created by the analysis using the rules in Tables 4.1 and 5.3 can still be solved in time proportional to the cube of the size of the *lifted* program [47] in the worst case. Remember though 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. The

worst-case running time for the whole analysis is therefore $\mathcal{O}(n^6)$, where n is 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.

A more interesting question is what happens in the most common case. To answer this we have to define things slightly more formally. Assume first that the modules and main expression of a program are numbered from 1 to m, with m being the number assigned to the main expression. We then use S_i^c to indicate the size of the contract on the module variable defined in module i, and we use S_i^e to indicate the size of the corresponding expression in the definition in module i. The size S_i^m of module i is then roughly $S_i^e + S_i^c$. The main expression does not have a contract so S_m^c is zero. The size S^p of the original program is then $S^p = \sum_{i=1}^{m} S_i^m = \sum_{i=1}^{m} S_i^e + \sum_{i=1}^{m} S_i^c$.

To compute the size of the lifted program we have to take into account the contract copying done by the lifting phase. Define R_{ji} to be the number of times the module variable defined in module j is referenced in module i (R_{mi} is zero for all i, since the main expression does not define any variable). When lifting module i, the lifting process does two things: first it removes from module i the contract on the module variable defined in module i; second, for all possible j (including i itself), it wraps around each reference to the module variable defined in module j a contract check that contains a copy of the contract that was originally on the variable defined in module j. The new size $S_i^{m\prime}$ of module i after lifting is therefore $S_i^{m\prime} = S_i^m - S_i^c + \sum_{j=1}^{m}(R_{ji} S_j^c) = (S_i^e + S_i^c) - S_i^c + \sum_{j=1}^{m}(R_{ji} S_j^c) = S_i^e + \sum_{j=1}^{m}(R_{ji} S_j^c)$.

Let us now make two simplifying assumptions. First we assume that all modules in the original unlifted program had the same sizes, both for their expression part ($S_i^e = S^e$ for all i) and contract part ($S_i^c = S^c$ for all i, ignoring the problem of the main expression). From a practical point of view, S^e and S^c can simply be thought of as averages, though for the benefit of our mathematical treatment here it is more convenient to simply assume that all the modules have the same sizes. The size $S^{m\prime}$ of module i after lifting then becomes $S^{m\prime} = S^e + S^c \sum_{j=1}^{m} R_{ji}$.

Now define the density d of an expression in the original program to be the number of module variable references in that expression divided by the size of the abstract syntax tree for the expression. The density d is therefore a real number between zero and one. If you consider an expression that is only a single module variable reference then the density is exactly one. If you consider an expression that is written in the lambda calculus of Chapter 3 then the density of module variable references in that expression is zero. Our second simplifying assumption is to assume that the density d is a constant throughout the program. In practice we simply expect such density to be relatively constant throughout the program for sufficiently big expressions.

The total number of module variable references that appear in module i can then be computed in two different ways. First, as the sum of the number of references to the module variable from module j that appear in module i: $\sum_{j=1}^{m} R_{ji}$. Second, as the product of the density of module variable references in the original unlifted module i times the size of the original unlifted module

i: $dS_i^e = dS^e$. From this and the above we can deduce that the size of module i after lifting is $S^{m\prime} = S^e + S^c dS^e = S^e(1 + dS^c)$.

Since our analysis is modular, the time $T^{p\prime}$ required to analyze the whole lifted program is the sum of the times required to analyze the different modules, taking into account that all modules have the same size and that a given module can be analyzed in time proportional to the cube of its lifted size: $T^{p\prime} = \sum_{i=1}^{m} T^{m\prime} = mT^{m\prime} = mk_1 S^{m\prime 3} = mk_1(S^e(1 + dS^c))^3 = mk_1 S^{e3}(1 + dS^c)^3$, for some constant k_1.

If we assume S^c to be proportional to S^e, i.e., $S^c = k_2 S^e$, then we have $T^{p\prime} = mk_1 S^{e3}(1 + dS^c)^3 = mk_1 S^{e3}(1 + dk_2 S^e)^3 \approx mk_1 S^{e3}(dk_2 S^e)^3 = mk_3 S^{e6}$ and we find again the sixth power result we discussed at the beginning of this section (making S^c proportional to S^e is the same as making S^c linear in the size of the original program since the size of the original program is mS^e, following our first simplifying assumption above).

As we indicated above, having modules with contracts that have a size linear in the size of the whole program is not likely to be seen in practice. If we therefore assume S^c to be constant we obtain $T^{p\prime} = mk_1 S^{e3}(1 + dS^c)^3 = mk_4 S^{e3}$.

Practical experience with big software projects like DrScheme show that, as the project grows, the number of modules increases steadily with the size of the project while the size of individual modules seldom goes beyond a few thousand lines. Modules that become too big are refactored by programmers to keep the complexity of the code manageable. For example, among the 2088

Scheme modules that are in DrScheme's code base at the time of this writing, only one is longer than 10000 lines. That one module contains in fact only automatically generated data and no code. Only five modules are between 5000 and 9999 lines of Scheme code, one among the five again containing only automatically generated data and two others containing only test cases for other modules. Taking this into account we can use for S^e an upper bound of a few thousand lines and conclude that, for big projects, $T^{p\prime} = mk_5$, for some (big) constant k_5. The running time for the analysis is then linear in the number of modules in the program, which is what we expect from a modular analysis.

Let us contrast this with a hypothetical analysis that uses one label per contract instead of two. In such an analysis abstract values flow across contract boundaries, the analysis therefore can not be done in a modular manner, and the resulting complexity is cubic in the size of the whole program: $T^{p\prime} = k_6(\sum_{i=1}^{m} S^{m\prime})^3 = m^3 k_6 S^{m\prime 3} = m^3 k_6 S^{e3}(1 + dS^c)^3$. If we again consider S^c and S^e to be upper-bounded by constants we then obtain $T^{p\prime} = m^3 k_7$, for some (big) constant k_7. The running time of the analysis then grows as the cube of the number of modules, which makes this hypothetical analysis unrealistic for big projects.

4.7 Related Work

Cousot and Cousot [14] formalize a modular version of their abstract interpretation framework [12] 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, i.e., that our contract language and analysis fulfill their general conditions, 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.

Probst [48], Flanagan and Felleisen [20], and Fähndrich and Aiken [4] 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 [52] 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 1-CFA 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.

The conventional data-flow community has developed its own approaches to the problem of modular analysis for higher-order languages. Chatterjee, Ryder, and Landi [10] describe a symbolic technique for computing data-flows in object-oriented programs in a modular fashion. For each module, their analysis computes a data-flow transfer function that is parameterized over the context. For references to the module, they use the transfer function and a parameterized solution to compute the actual flow. Besson and Jensen [6] describe a variation of this idea. Their analysis generates constraints from object-oriented programs and represents them as clauses in a simple relational query language; the unknowns are represented as predicate symbols. They then simplify these clauses using techniques from logic programming. In the end, both approaches suffer from the same problems as the analyses from Probst, Flanagan and Felleisen, and Fähndrich and Aiken that we discussed above.

Much work has also been done on modules in the context of Hindley-Milner type systems [38, 39, 44, 37]. The power of the system we have

presented so far is roughly equivalent to that of those type systems, though this will change in the next chapter.

Dreyer et al. [17] present a type system for higher-order modules. Our modules are first order only. The MzScheme programming language [23], in which our static graphical debugger is written and which is the ultimate target language for our analysis, has higher-order modules in the form of units [25] but DrScheme's current contract system does not handle units yet and neither does our analysis.

As we have indicated in the previous chapter, there is a general equivalence between polyvariant flow analyses and type systems with intersection and union types [31, 46, 55]. Systems with intersection and union types also usually do not consider the problem of modularity. Since these analysis are based on extending type systems with flow annotations, and since our contract language so far is simple enough to closely resemble types, we estimate that extending those type systems to support first-order modules should not be too difficult. Wells et al. indicate that their λ^{CIL} calculus could possibly serve as the basis for a modular compilation system [55] but do not elaborate on that point. Similarly, Haack and Wells's work on type error slicing [29] describes extending that system to handle module signatures as future work.

Chapter 5

Unrestricted Contracts

The previous chapter introduced a simple contract language based on integer and arrow contracts. While this contract language allowed us to describe the mechanisms necessary to have a modular analysis, namely using contracts as both sources and sinks of abstract values, and having a lifting phase before constraints are generated, programmers often use runtime contracts that state invariants that are far beyond the reach of conventional value-flow analyses or type systems. Therefore our analysis must somehow deal with those.

In this chapter we introduce a new contract form that allows programmers to use any expression as a contract. The analysis handles those complex contracts in two ways. First, whenever possible, it tries to approximate a complex contract with a type-like contract of the kind we have used in the previous chapter, and use this approximation in lieu of the complex contract. The approximation mechanism is based on computing the domain of the

predicate used in the complex contract. Second, in the case where using such an approximation is not enough to establish whether a contract is violated or not, the analysis delegates the proof to a theorem prover. The analysis we now present is therefore parameterized over two things: an approximation function and a theorem prover.

5.1 Contract Calculus

As usual, we introduce in the first subsection our surface syntax and internal syntax of programs with modules and complex contracts. In the second subsection, we explain the translation from surface syntax into internal syntax, which requires the definition of the approximation function we just discussed.

5.1.1 User Syntax and Annotated Syntax

In the user syntax of Figure 5.1, the language of contracts uses two new kinds of constructs: one for validating any value, and one to use arbitrary expressions as contracts. Using the latter we can for example define the "positive integer" contracts used in Figure 2.2. 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 combines 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 5.1: Surface syntax for the lambda calculus with unrestricted contracts.

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

Figure 5.2: Annotated syntax for the lambda calculus with unrestricted contracts.

The annotated syntax of Figure 5.2 is different from the one in the previous chapter in several ways.

First, integers and closures have extra subscript annotations to represent contract predicates that they have satisfied. Such annotations are added only during reductions. In practice a static debugger will only analyze unreduced programs so the analyzed terms will not have such extra annotations. These annotations are required for the soundness proof of the analysis though.

Second, blame expressions now have two possible severity levels when a contract violation is detected: $\mathcal{R}$ed for violating a basic integer or arrow contract, and $\mathcal{O}$range for violating a user-provided predicate.

Third, a new ε expression form is introduced. This is a technical device to be explained shortly.

Fourth, all annotated contracts have now two module name annotations instead of one. The two names represent the two parties that agreed to the contract. Having these two names available is necessary for the proper handling of the new any contract form.

Fifth, the any contracts have an equivalent blessed form aⁿ̂y, similarly to what is done for arrow contracts.

Finally the annotated contract language includes a new form $\langle E\ E\ C \rangle$, which we refer to as a contract triple. Contract triples replace the (pred E) contract form in the unannotated syntax. Its first expression turns the predicate into a runtime check; its second expression is the original predicate; and the last part is the contract's projection 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. The translation from the (pred E) form into contract triples is described in the next section, along with the full annotation process.

5.1.2 Annotation Process

The rules of Figure 5.3 define the full annotation process for our language with modules and complex contracts. The judgements have the same form as in the previous chapter. The major additions to the annotation process are the presence of the complex predicates (rule PredC) and the fact that contracts are now annotated with two module names. As before, these 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. The two names switch positions when the annotation process traverses a domain position in a functional contract (rule ArrowC).

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

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

according to rule PredC:

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Delta,\Gamma,g,f \vdash^{\mathsf{a}}_{\mathtt{c}} c_d \rightsquigarrow c_d' \\ \Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathtt{c}} c_r \rightsquigarrow c_r'\end{array}}{\Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathtt{c}} (c_d \rightarrow c_r) \rightsquigarrow (c_d' \rightarrow c_r')^{\ell\ell'}_{fg}} \text{ (ArrowC)}$$

$$\frac{\begin{array}{c}\Delta,\Gamma,f \vdash^{\mathsf{a}}_{\mathtt{e}} e \rightsquigarrow e' \qquad \Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathtt{c}} \mathcal{D}_{\Delta}((\text{pred } e)) \rightsquigarrow c' \\ e'' \stackrel{def}{=} \mathcal{F}(e', \mathcal{L}^{+}(c'), f)\end{array}}{\Delta,\Gamma,f,g \vdash^{\mathsf{a}}_{\mathtt{c}} (\text{pred } e) \rightsquigarrow \langle e''\ e'\ c' \rangle^{\ell\ell'}_{fg}} \text{ (PredC)}$$

Figure 5.3: Annotation judgments for the lambda calculus with unrestricted contracts.

- 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 (pred E) and can therefore be used as a simple contract that approximates the complex predicate e.
- $\mathcal{F}(e', \mathcal{L}^{+}(c'), f)$ generates boilerplate code that represents the application of the predicate to a value in a schematic manner. The $\mathcal{L}^{+}$ 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 for the first element of the triple 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:

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

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. 5.4). 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, any, 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 (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. That kind of program is simply rejected by the annotator.

- If $\mathcal{D}_\Delta$ is applied to a contract of the form (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.[1]

  The need for the $\mathcal{D}_\Delta$ function to return some type-like contract even in the case where an expression proper is used as a predicate justifies

[1]Both these requirements will be verified by the analysis, because the analysis will check the predicate expression against its domain, as computed by $\mathcal{D}_\Delta$, when it processes the boilerplate code in the triple that results from translating the (pred e) form.

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

Figure 5.4: Predicate domain function.

why we have to introduce the any contract in the same chapter as we introduce unrestricted predicates. Of course we need such a contract to represent the domain of general predicates that can actually be applied to any value, but we also need the any contract to fall back on when we have failed to compute a more precise approximation of the domain of the predicate.

Consider for example the following program fragment:

(module *prime?* (int→int) ...)
(module f (pred *prime?*) 3)
f

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

(module $\mathit{prime?}^{\beta_1}$ ...)
(module f^{β_2} 3)
$(\langle(\text{if0 } (\mathit{prime?}^{\beta_1}\ \varepsilon^{\ell})\ \varepsilon^{\ell}\ (\text{blame } f\ \mathcal{O}))\quad \mathit{prime?}\quad \text{int}^{\ell\ell'}_{f\mu}\rangle \Leftarrow f^{\beta_2})$

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 at least 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 (int→int). The resulting int contract is then annotated to get the $\text{int}^{\ell\ell'}_{f\mu}$ contract used in the triple. That contract shares its first label ℓ with the ε^{ℓ} expressions in the if0 part of the triple.

An interesting problem arises when a predicate, say *prime?*, uses another predicate, say *nat?*, to define the domain part of its own contract:

(module *nat?* (int→int) ...)
(module *prime?* ((pred *nat?*)→int) ...)
(module f (pred *prime?*) 3)
f

In such a case, the $\mathcal{D}_\Delta$ function replaces both uses of the predicates with triples. In each of those two triples, it uses int as the approximation, meaning that, to be a natural number or a prime, a number first has to be an integer. But is then a prime considered a natural number? With the definition of $\mathcal{D}_\Delta$ given here, the answer to that question is no, since once the triples have been created, the relationship between the two *nat?* and *prime?*

predicates is lost: both are now approximated by an int contract. It would nevertheless be easy to extend the definition of $\mathcal{D}_\Delta$ to return a list of predicate approximations (the sequence of predicate domains traversed by $\mathcal{D}_\Delta$ as it computes the current predicate-free approximation). Later making this list of approximations available to the analysis would then make the analysis automatically aware of the fact that a prime number is always a natural number. With the definition of $\mathcal{D}_\Delta$ above, the analysis instead has to ask a theorem prover to prove that *nat?* implies *prime?* (see Section 5.3 below).

Since a predicate can be used in the definition of the contract of another predicate, the function $\mathcal{D}_\Delta$ in an actual static debugger would have to check that there are no reference loops among contracts (e.g. trying to define the contract for a predicate using the predicate itself). We omit this check in our definition here to simplify our model, but checking for such self-referential or mutually-referential contracts would be easy to do by keeping a trace of the module variables that $\mathcal{D}_\Delta$ looks up in the Δ environment.

Note that the $\mathcal{D}_\Delta$ function in Figure 5.4 is only one of many possible definitions for $\mathcal{D}_\Delta$. A simpler definition would be to return any in all cases. That way we would avoid having to do any lookup in the Δ environment. If in addition we modified the ModVar rule to always use an annotated any contract instead of using Δ, then we would have a worst case analysis [14] that does not assume anything about other modules. In that case a module could be analyzed even when the contracts for other module variables are not available.

At the other end of the spectrum, the $\mathcal{D}_\Delta$ function could look at the content of the expression e in the (pred e) form to try to extract from that expression a domain that is more precise than just any. By using a backward analysis [35] the function could try to compute a conservative approximation of the expression's domain (assuming of course that the expression evaluates to a function) and return that approximation to be used in the corresponding contact triple. There is no limit to how complex such a backward analysis could be, as long as it were guaranteed to terminate, though in practice we would want to use an analysis that computes a reasonable approximation in a short time. Using such an analysis would also partially break the modularity of the analysis, since it would require the code of all predicates used in contracts to be available to the $\mathcal{D}_\Delta$ function. The definition of $\mathcal{D}_\Delta$ we give in Figure 5.4 computes a decent approximation of a predicate's domain in linear time in the size of the predicate's contract at the most, does not require access to the predicate's code, and is therefore good enough for our purpose.

How to reduce and analyze triples and any contracts is the subject of the next two sections.

5.2 Reduction Rules

Figure 5.5 defines the full reduction semantics for annotated programs in the presence of triples and any contracts. The set of evaluation contexts for expressions is the same as in the previous chapter:

$$((\lambda x^{\beta}.e)^{\ell_\lambda}\ v^{\ell_v})^{\ell_a} \longrightarrow e[v^{\ell_v}/x^{\beta}] \quad \text{SUBST}$$

$$(n^{\ell_n}\ v^{\ell_v})^{\ell_a} \longrightarrow (\text{blame } \lambda\ \mathcal{R})^{\ell_a} \quad \text{APP-ERROR}$$

$$(\text{if0}\ 0^{\ell_0}\ e_1\ e_2)^{\ell} \longrightarrow e_1 \quad \text{IF0-TRUE}$$

$$(\text{if0}\ v^{\ell_v}\ e_1\ e_2)^{\ell} \longrightarrow e_2 \quad \text{IF0-FALSE}$$

$$(\text{int}^{\ell\ell'}_{fg} \Leftarrow n^{\ell_n})^{\ell_c} \longrightarrow n^{\ell} \quad \text{INT-INT}$$

$$(\text{int}^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} \longrightarrow (\text{blame } f\ \mathcal{R})^{\ell'} \quad \text{INT-LAM}$$

$$(\langle e_1\ e_2\ \text{int}^{\ell\ell'}_{fg}\rangle^{\ell^+\ell^-}_{fg} \Leftarrow n^{\ell_n}_{e\ldots})^{\ell_c} \longrightarrow e_1[n^{\ell}_{e\ldots e_2}/\varepsilon^{\ell}] \quad \text{INT-TRIP-INT}$$

$$(\langle e_1\ e_2\ \text{int}^{\ell\ell'}_{fg}\rangle^{\ell^+\ell^-}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} \longrightarrow (\text{blame } f\ \mathcal{R})^{\ell'} \quad \text{INT-TRIP-LAM}$$

$$((c_1 \to c_2)^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} \longrightarrow ((c_1 \hat{\to} c_2)^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} \quad \text{LAM-LAM}$$

$$((c_1 \to c_2)^{\ell\ell'}_{fg} \Leftarrow n^{\ell_n})^{\ell_c} \longrightarrow (\text{blame } f\ \mathcal{R})^{\ell'} \quad \text{LAM-INT}$$

$$(\langle e_1\ e_2\ (c_1 \to c_2)^{\ell\ell'}_{fg}\rangle^{\ell^+\ell^-}_{fg} \Leftarrow \vec{v}^{\ell_v}_{e\ldots})^{\ell_c} \longrightarrow e_1[((c_1 \hat{\to} c_2)^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v}_{e\ldots e_2})^{\ell_c}/\varepsilon^{\ell}] \quad \text{LAM-TRIP-LAM}$$

$$(\langle e_1\ e_2\ (c_1 \to c_2)^{\ell\ell'}_{fg}\rangle^{\ell^+\ell^-}_{fg} \Leftarrow n^{\ell_n})^{\ell_c} \longrightarrow (\text{blame } f\ \mathcal{R})^{\ell'} \quad \text{LAM-TRIP-INT}$$

$$(\text{any}^{\ell\ell'}_{fg} \Leftarrow n^{\ell_n})^{\ell_c} \longrightarrow n^{\ell} \quad \text{ANY-INT}$$

$$(\text{any}^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} \longrightarrow (\hat{\text{any}}^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c} \quad \text{ANY-LAM}$$

$$(\langle e_1\ e_2\ \text{any}^{\ell\ell'}_{fg}\rangle^{\ell^+\ell^-}_{fg} \Leftarrow n^{\ell_n}_{e\ldots})^{\ell_c} \longrightarrow e_1[n^{\ell}_{e\ldots e_2}/\varepsilon^{\ell}] \quad \text{ANY-TRIP-INT}$$

$$(\langle e_1\ e_2\ \text{any}^{\ell\ell'}_{fg}\rangle^{\ell^+\ell^-}_{fg} \Leftarrow \vec{v}^{\ell_v}_{e\ldots})^{\ell_c} \longrightarrow e_1[(\hat{\text{any}}^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v}_{e\ldots e_2})^{\ell_c}/\varepsilon^{\ell}] \quad \text{ANY-TRIP-LAM}$$

$$(((c_1 \hat{\to} c_2)^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c}\ w^{\ell_w})^{\ell_a} \longrightarrow (c_2 \Leftarrow (\vec{v}^{\ell_v}\ (c_1 \Leftarrow w^{\ell_w})^{\mathcal{L}^+(c_1)})^{\mathcal{L}^-(c_2)})^{\mathcal{L}^+(c_2)} \quad \text{SPLIT-ARROW}$$

$$((\hat{\text{any}}^{\ell\ell'}_{fg} \Leftarrow \vec{v}^{\ell_v})^{\ell_c}\ w^{\ell_w})^{\ell_a} \longrightarrow (\text{any}^{\ell\ell'}_{fg} \Leftarrow (\vec{v}^{\ell_v}\ (\text{any}^{\ell\ell'}_{gf} \Leftarrow w^{\ell_w})^{\ell})^{\ell'})^{\ell} \quad \text{SPLIT-ANY}$$

Figure 5.5: Reduction rules for the lambda calculus with unrestricted contracts.

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

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 reduction rules in Figure 5.5 use the same conventions as the ones in the previous chapters. In addition we write $v_{e\ldots}$ for a value v that satisfies all the predicates e, etc. We use that notation only when strictly necessary for the comprehension of the rules. In addition of the rules in Figure 5.5 we also use the LOOKUP rule from the previous chapter when looking up module variables.

Compared to Figure 4.4, all the new rules in Figure 5.5 are concerned with triples and any contracts.

- The INT-TRIP-INT and INT-TRIP-LAM rules 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. The rule takes the boilerplate code from the first part of the triple and replaces the two ε expressions in that code with the value n, after an appropriate label change on n as usual. The result of the substitution is code that checks whether the integer value satisfies the triple's predicate or not: it applies the predicate to the value, and either returns the value or blames a module for breaking the contract. The expression e_2 does

not play any active role during the reduction but is added to the set of predicates satisfied by n (for the purpose of the analysis and the soundness proof). 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 the behavior of e_2.

- Similarly, the rules LAM-TRIP-LAM and LAM-TRIP-INT correspond to the rules LAM-LAM and LAM-INT when in the presence of triples. The LAM-TRIP-LAM rules wraps a blessed arrow check around the $\vec{v}$ value in exactly the same way as the LAM-LAM rule does. The ε expressions in the boilerplate code are then replaced by the wrapped value in a way similar to what happens in the INT-TRIP-INT rule we described just above.

- The ANY-INT rule shows a contract check that checks nothing. The check reduces to the tested value. As usual, the label ℓ on any becomes the label on n. The ANY-LAM rule is slightly more complicated. It wraps a blessed any contract around the $\vec{v}$ value in the same way that a blessed arrow contract is wrapped around the $\vec{v}$ value in the LAM-LAM rule, and for exactly the same reason: once the ANY-LAM rule has discovered that the value is a functional value, it has to wrap a blessed any contract around it as a way to remember that the argument and

result of the functional value still have to be checked. In essence, since an any contract represents a set of abstract values that includes the set of abstract values represented by arrow contracts, the behavior of the ANY-LAM rule should resemble and encompass the behavior of the LAM-LAM rule.

- The ANY-TRIP-INT and ANY-TRIP-LAM rules are then similar to the ANY-INT and ANY-LAM rules, respectively, with the difference that they deal with triples. Like for all other reduction rule that have triples as redexes and do not have a blame expression as contractum, the two rules are based on substituting checked values (possibly with a blessed contract wrapped around them) in the boilerplate code of the triple.

- Finally, the SPLIT-ANY rule splits the blessed any contracts introduced by the ANY-LAM and ANY-TRIP-LAM rules. Its behavior is similar to the SPLIT-ARROW rule: it breaks a blessed any contract into a domain and range contract and distributes those to the actual argument of the function and to the result of the whole application, respectively. The need to swap the module names on the new any contract that checks the argument w explains why contracts need to be annotated with both module names in this chapter.

  The two new any contract checks are seemingly useless for reduction purposes though. In fact, as far as DrScheme's actual runtime contract system is concerned, all checks against an any contract are non-

operations. Nevertheless, putting these two checks in place at this stage ensures that the analysis presented in the next section will check both that the $\vec{v}$ function can accept any value as input and that the application's context can accept any value returned by the application. It will also ensure that the labels on the w value and on the value returned by the $\vec{v}$ function are replaced by the label ℓ once the two checks have been reduced (otherwise the analysis would be unsound). Our POPL'06 paper [42] has neither a SPLIT-ANY rule nor blessed any contracts, and the reduction semantics described in that paper considers all checks against an any contract as checks that do nothing and simply reduce to the checked value. While this is fine from a runtime point of view, it was the root cause of an unsound analysis.

The reader will have noticed many similarities between rules with contract triples and rules without contract triples. For example between the INT-INT and INT-TRIP-INT rules or between the INT-LAM and INT-TRIP-LAM rules, etc. Such similarities are a general consequence of the fact that a triple-free contract c can equivalently be represented as a triple $\langle \mathcal{F}(e, \ell, f)\ e\ c \rangle$ where e is a vacuous predicate expression like $(\lambda x.0)$ that accepts all values. By modifying the annotation process to convert triple-free contracts into contract triples it would be therefore possible to reduce the number of reduction rules in Figure 5.5. We nevertheless keep those triple-free contracts because they allow for the creation of many simple triple-free type-like contracts that are in a syntax close to the one used both in the previous chapters and in

DrScheme's contract system, and because they are more intuitive to understand than only having triples everywhere.

Another way to reduce the number of reduction rules in Figure 5.5 would be to transform the simple int contract into a triple like $\langle \mathcal{F}(int?, \ell, f)\ int?\ \text{any}\rangle$ that uses an $int?$ expression as a predicate equivalent to the int contract. The check against the any contract would then always succeed and the real integer check would occur inside the boilerplate code using the $int?$ expression. The INT-INT rule would then be subsumed by the ANY-TRIP-INT rule. Or course this strategy would work only for "flat" triple-free contracts like int. Higher-order triple-free contracts like (int→int) could not be transformed this way, since there is no equivalent predicate expression that can check whether a given function is only ever applied to integers and only ever returns integers.

5.3 The Analysis

The analysis now has to handle two news construct: contract triples and any. Like other contracts before, these contracts will act as both sources and sinks of abstract values. Since contract triples each contain as their third part a triple-free contract that approximates the behavior of the corresponding predicate expression, the value-flow analysis will use those approximations whenever possible. If those approximations are not precise enough to decide whether a contract check may result in a contract violation, the value-flow analysis will delegate the decision to a theorem prover. The any contracts

will also need to be handled carefully, since the corresponding abstract values will have to simulate the behavior of any possible value, including functions of any complexity. Before we consider the analysis of those two constructs any further, we first describe in the next section the new triple-aware lifting process.

5.3.1 Lifting

The lifting process for annotated programs with contract triples is essentially the same as in the previous chapter. Each contract check $(c \Leftarrow e)^{\ell}$ is lifted to the top of the program and the remaining hole in the term is filled with the contract c. 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 5.6 defines the complete lifting process. As was the case with blessed arrows, we ignore the distinction between any contracts and their blessed counterparts (rule BAnyC).

Lifting occurs almost everywhere, including inside the first expression of triples (rule TripC). Since triples disappear 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

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

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

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

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} x^{\beta} \rightsquigarrow x^{\beta}} \ (\textsc{Var}) \qquad \frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} f^{\beta} \rightsquigarrow f^{\beta}} \ (\textsc{ModVar}) \qquad \frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} \varepsilon^{\ell} \rightsquigarrow \varepsilon^{\ell}} \ (\textsc{Ref})$$

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

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{e}} e_0 \rightsquigarrow es_0 \ldots e_0' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e_1 \rightsquigarrow es_1 \ldots e_1' \qquad \vdash^{\mathtt{l}}_{\mathtt{e}} e_2 \rightsquigarrow es_2 \ldots e_2'}{\vdash^{\mathtt{l}}_{\mathtt{e}} (\text{if0 } e_0\ e_1\ e_2)^{\ell} \rightsquigarrow es_0 \ldots es_1 \ldots es_2 \ldots (\text{if0 } e_0'\ e_1'\ e_2')^{\ell}} \ (\textsc{If0})$$

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{e}} (\text{blame } f\ s)^{\ell} \rightsquigarrow (\text{blame } f\ s)^{\ell}} \ (\textsc{Blame})$$

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

$$\frac{}{\vdash^{\mathtt{l}}_{\mathtt{c}} \text{int}^{\ell\ell'}_{fg} \rightsquigarrow \text{int}^{\ell\ell'}_{fg}} \ (\textsc{IntC}) \qquad \frac{}{\vdash^{\mathtt{l}}_{\mathtt{c}} \text{any}^{\ell\ell'}_{fg} \rightsquigarrow \text{any}^{\ell\ell'}_{fg}} \ (\textsc{AnyC})$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{c}} c_d \rightsquigarrow es_d \ldots c_d' \qquad \vdash^{\mathtt{l}}_{\mathtt{c}} c_r \rightsquigarrow es_r \ldots c_r'}{\vdash^{\mathtt{l}}_{\mathtt{c}} (c_d \rightarrow c_r)^{\ell\ell'}_{fg} \rightsquigarrow es_d \ldots es_r \ldots (c_d' \rightarrow c_r')^{\ell\ell'}_{fg}} \ (\textsc{ArrowC})$$

$$\frac{\vdash^{\mathtt{l}}_{\mathtt{c}} \text{any}^{\ell\ell'}_{fg} \rightsquigarrow es}{\vdash^{\mathtt{l}}_{\mathtt{c}} \hat{\text{any}}^{\ell\ell'}_{fg} \rightsquigarrow es} \ (\textsc{BAnyC}) \qquad \frac{\vdash^{\mathtt{l}}_{\mathtt{c}} (c_d \rightarrow c_r)^{\ell\ell'}_{fg} \rightsquigarrow es}{\vdash^{\mathtt{l}}_{\mathtt{c}} (c_d \hat{\rightarrow} c_r)^{\ell\ell'}_{fg} \rightsquigarrow es} \ (\textsc{BArrowC})$$

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

Figure 5.6: Lifting judgments for the lambda calculus with unrestricted contracts.

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

Figure 5.7: Analyzed syntax for the lambda calculus with unrestricted contracts.

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.

5.3.2 Constraints Generation

Figure 5.7 defines the syntax of the language we analyze. As before contracts are now expressions and contract checks only appear at the top-level. Blessed arrow contracts and blessed any contracts have disappeared.

Once again, the analysis has to produce two results: a mapping φ from labels to sets of labels, as always, and a mapping ψ from labels to error culprit (module names, μ for the main expression, or λ for a violation by the programmer of a constraint of the lambda calculus itself) and severity (the red or orange color used to highlight the erroneous term).

Source$\diagdown$*Sink*	$\mathrm{int}^{\ell_5^+\ell_5^-}_{hi}$	$\langle \dots\ e_5\ \mathrm{int}^{\ell_5^+\ell_5^-}_{hi} \rangle^{\ell_6^+\ell_6^-}_{hi}$	$\mathrm{any}^{\ell_5^+\ell_5^-}_{hi}$	$\langle \dots\ e_5\ \mathrm{any}^{\ell_5^+\ell_5^-}_{hi} \rangle^{\ell_6^+\ell_6^-}_{hi}$
$n^{\ell_n}_{e_1\dots}$		$\left.\begin{array}{c}\{\ell_n\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{c}\{\ell_n\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\mathrm{int}^{\ell_1^+\ell_1^-}_{fg}$		$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\langle \dots\ e_1\ \mathrm{int}^{\ell_1^+\ell_1^-}_{fg} \rangle^{\ell_2^+\ell_2^-}_{fg}$		$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\mathrm{any}^{\ell_1^+\ell_1^-}_{fg}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$			$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\langle \dots\ e_1\ \mathrm{any}^{\ell_1^+\ell_1^-}_{fg} \rangle^{\ell_2^+\ell_2^-}_{fg}$				$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$(\lambda x^{\beta}.e^{\ell})^{\ell_\lambda}_{e_1\dots}$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_5^+)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_5^-)$ $\left.\begin{array}{c}\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	
$(c^{\ell_1^+\ell_1^-}_{gf}\!\to\! c^{\ell_2^+\ell_2^-}_{fg})^{\ell_3^+\ell_3^-}_{fg}$	$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_5^+)\subseteq\varphi(\ell_1^-)$	
$\langle \dots\ e_3\ (c^{\ell_1^+\ell_1^-}_{gf}\!\to\! c^{\ell_2^+\ell_2^-}_{fg})^{\ell_3^+\ell_3^-}_{fg} \rangle^{\ell_4^+\ell_4^-}_{fg}$			$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_5^-)$ $\left.\begin{array}{c}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ e_3\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	

Table 5.1: Constraints creation for the lambda calculus with unrestricted contracts.

$Source \diagdown Sink$	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$	$(c_{ih}^{\ell_7^+\ell_7^-} \to c_{hi}^{\ell_8^+\ell_8^-})_{hi}^{\ell_5^+\ell_5^-}$	$\langle \dots\ e_5\ (c_{ih}^{\ell_7^+\ell_7^-} \to c_{hi}^{\ell_8^+\ell_8^-})_{hi}^{\ell_5^+\ell_5^-} \rangle_{hi}^{\ell_6^+\ell_6^-}$
$n_{e_1\dots}^{\ell_n}$	$\{\ell_n\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R} \rangle\} \subseteq \psi(\ell_a)$	$\{\ell_n\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R} \rangle\} \subseteq \psi(\ell_5^-)$	
$\mathrm{int}_{fg}^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R} \rangle\} \subseteq \psi(\ell_a)$	$\{\ell_1^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R} \rangle\} \subseteq \psi(\ell_5^-)$	
$\langle \dots\ e_1\ \mathrm{int}_{fg}^{\ell_1^+\ell_1^-} \rangle_{fg}^{\ell_2^+\ell_2^-}$			
$\mathrm{any}_{fg}^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R} \rangle\} \subseteq \psi(\ell_a)$ $\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\ell_1^-)$ $\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_1^+) \subseteq \varphi(\ell_a)$	$\{\ell_1^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{R} \rangle\} \subseteq \psi(\ell_5^-)$ $\{\ell_1^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_7^+) \subseteq \varphi(\ell_1^-)$ $\{\ell_1^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_1^+) \subseteq \varphi(\ell_8^-)$	
$\langle \dots\ e_1\ \mathrm{any}_{fg}^{\ell_1^+\ell_1^-} \rangle_{fg}^{\ell_2^+\ell_2^-}$			
$(\lambda x^{\beta}.e^{\ell})_{e_1\dots}^{\ell_\lambda}$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\beta)$ $\{\ell_\lambda\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell) \subseteq \varphi(\ell_a)$	$\{\ell_\lambda\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_7^+) \subseteq \varphi(\beta)$ $\{\ell_\lambda\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell) \subseteq \varphi(\ell_8^-)$ $\left.\begin{array}{l} \{\ell_\lambda\} \subseteq \varphi(\ell_5^-) \\ e_1 \dots \not\sqsubseteq e_5 \end{array}\right\} \Rightarrow \{\langle h, \mathcal{O} \rangle\} \subseteq \psi(\ell_5^-)$	
$(c_{gf}^{\ell_1^+\ell_1^-} \to c_{fg}^{\ell_2^+\ell_2^-})_{fg}^{\ell_3^+\ell_3^-}$	$\{\ell_3^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\ell_1^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_2^+) \subseteq \varphi(\ell_a)$	$\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \{\langle h, \mathcal{O} \rangle\} \subseteq \psi(\ell_5^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_7^+) \subseteq \varphi(\ell_1^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_2^+) \subseteq \varphi(\ell_8^-)$	
$\langle \dots\ e_3\ (c_{gf}^{\ell_1^+\ell_1^-} \to c_{fg}^{\ell_2^+\ell_2^-})_{fg}^{\ell_3^+\ell_3^-} \rangle_{fg}^{\ell_4^+\ell_4^-}$		$\left.\begin{array}{l} \{\ell_3^+\} \subseteq \varphi(\ell_5^-) \\ e_3 \not\sqsubseteq e_5 \end{array}\right\} \Rightarrow \{\langle h, \mathcal{O} \rangle\} \subseteq \psi(\ell_5^-)$	

Table 5.2: Constraints creation for the lambda calculus with unrestricted contracts (continued).

Tables 5.1 and 5.2 explain 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, the boilerplate code inside contract triples is analyzed like any other expression. To save space, some cells in the tables share some constraints with their neighboring cells.

Let us now explain some of the constraints involving any contracts or contact triples. Our first example involves an any contract:

Source$\diagdown$*Sink*	$\text{any}_{hi}^{\ell_5^+\ell_5^-}$
$(c_{gf}^{\ell_1^+\ell_1^-} \rightarrow c_{fg}^{\ell_2^+\ell_2^-})_{fg}^{\ell_3^+\ell_3^-}$	$\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_5^+) \subseteq \varphi(\ell_1^-)$ $\{\ell_3^+\} \subseteq \varphi(\ell_5^-) \Rightarrow \varphi(\ell_2^+) \subseteq \varphi(\ell_5^-)$

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 abstract values, 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.[2] 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 Table 5.2. 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 $\text{any}_{hi}^{\ell_5^+\ell_5^-}$ contract to check the functional contract as well as its domain and range. Instead, it creates a new $(\text{any}_{hi}^{\ell_5^+\ell_5^-} \rightarrow \text{any}_{hi}^{\ell_5^+\ell_5^-})_{hi}^{\ell\ell'}$ 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.[3] In essence this process simply makes explicit the sinks for the complex abstract values that flow into $\text{any}_{hi}^{\ell_5^+\ell_5^-}$. The analysis therefore remains sound. Here we forsake this process and re-use the $\text{any}_{hi}^{\ell_5^+\ell_5^-}$ contract and its labels only to simplify the soundness proof.

Note that in general this expansion process for any contracts should occur for all non-atomic values. If our language had, say, pairs, then we could have

[2] At an abstract level this is analogous to Henglein's notion of a `Dynamic` $\rightsquigarrow$ (`Dynamic`$\rightarrow$`Dynamic`) coercion [33].

[3] The debugger must then be careful to re-use the original $\text{any}_{hi}^{\ell_5^+\ell_5^-}$ contract for both the domain and range of the new $(\text{any}_{hi}^{\ell_5^+\ell_5^-} \rightarrow \text{any}_{hi}^{\ell_5^+\ell_5^-})_{hi}^{\ell\ell'}$ contract because the use of new any contracts for the domain and range would make the analysis fail to terminate when a function with a recursive type flowed into $\text{any}_{hi}^{\ell_5^+\ell_5^-}$: new any contracts would be created on the fly for ever.

a pair value with functions as its two elements. If such a pair were to flow into an any contract check, the any contract would have to expand into a pair of any contracts, and each of those new any contract would have in turn to expand to handle each of the functions that are the pair's elements.

Our second example, from Table 5.2, handles the symmetric case: when an any contract flows into an arrow contract:

$Source \diagdown Sink$	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$
$\text{any}_{fg}^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \{\langle \lambda, \mathcal{R} \rangle\} \subseteq \psi(\ell_a)$
	$\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_6) \subseteq \varphi(\ell_1^-)$
	$\{\ell_1^+\} \subseteq \varphi(\ell_5) \Rightarrow \varphi(\ell_1^+) \subseteq \varphi(\ell_a)$

A violation of a constraint of the lambda calculus is detected, since the any abstract value might turn out at runtime to be an integer. If instead the runtime value turns out to be a function, than the actual argument from the application will flow into the function's formal argument and the result of the whole application will be the result of the function, whatever value that might be. To conservatively simulate this, the analysis has to make the formal argument represented by ℓ_6 flow back into the any contract, and make that same any contract flow out of the application.

The analysis in our POPL'06 paper correctly created the first constraint out of the three in this second example, but the other two constraints were

simply forgotten. This was not realized at the time because the reduction rules in that paper treated any checks as vacuous checks that always reduced to the value being checked, rather than checks that wrap a blessed any contract around the value when that value turns out to be a function. This was based on the idea that, from the point of view of an actual contract system like the one in DrScheme, any contracts act as useless checks. It was not realized at the time that, while any contract checks are useless from the point of view of the runtime contract checking, they are useful from the point of view of the analysis because, after lifting, the copies of the any contracts left behind by the lifting process will act as abstract value sources and might trigger contract violations in applications elsewhere (even though the same kind of reasoning was behind the need for the two constraints described in the first example above, which was correct in the paper).

The third example explains partial contract violation, which is tagged with the orange color ($\mathcal{O}$). Consider this entry:

$Source \diagdown Sink$	$\langle \ldots\ e_5\ \mathrm{int}_{hi}^{\ell_5^+ \ell_5^-} \rangle_{hi}^{\ell_6^+ \ell_6^-}$
$\langle \ldots\ e_1\ \mathrm{int}_{fg}^{\ell_1^+ \ell_1^-} \rangle_{fg}^{\ell_2^+ \ell_2^-}$	$\left.\begin{array}{l} \{\ell_1^+\} \subseteq \varphi(\ell_5^-) \\ \\ e_1 \not\sqsubseteq e_5 \end{array}\right\} \Rightarrow \{\langle h, \mathcal{O} \rangle\} \subseteq \psi(\ell_5^-)$

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 addi-

tional 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. As in the previous chapter, such blame constraints always use the name h associated with the sink (or λ when the program violates the language specification), never the name f associated with the source, to be consistent with what happens during reductions.

Additional Constraints

Finally, we again have a few extra constraints to get the analysis started. They are described in Table 5.3, and are similar to the ones in Table 4.2, with just one addition to handle contract triples.

Triples such as $\langle e^{\ell_1}\ e^{\ell_2}\ c_{fg}^{\ell\ell'}\rangle_{fg}^{\ell^+\ell^-}$ 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

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

Table 5.3: Additional constraints for the lambda calculus with unrestricted contracts.

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.

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 5.1.2. 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:

$$\langle(\text{if0 } (\mathit{prime?}^{\beta_1}\ \varepsilon^\ell)\ \varepsilon^\ell\ (\text{blame } f\ \mathcal{O}))\quad \mathit{prime?}\quad \text{int}^{\ell\ell'}_{f\mu}\rangle$$

When considered as a source the $\text{int}^{\ell\ell'}_{f\mu}$ 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 5.3. If later that $\text{int}^{\ell\ell'}_{f\mu}$ contract flows into a simple arrow contract check then a red error occurs. If the $\text{int}^{\ell\ell'}_{f\mu}$ 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 $\text{int}^{\ell\ell'}_{f\mu}$ 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 $\text{int}^{\ell\ell'}_{f\mu}$ 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.

5.3.3 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 unreduced programs, where the relation will always be 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 [16].

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 [27] 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 limits (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 limits. 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 Tables 5.1 and 5.2 (e.g. adding context sensitivity, flow sensitivity, etc.)

In the case of an actual error, its color can be changed from orange to red (or vice versa) by moving knowledge from the theorem prover to the value-flow analysis (or vice versa). In practice one then wants to have as much knowledge as possible be present at the value-flow analysis level, since this analysis is likely to be much faster than the theorem prover.[4] This is in fact the whole point of using triple-free predicate approximations to simulate predicate expressions: in most cases those approximations are good enough to allow the value-flow analysis to judge whether an abstract value violates a contract check or not, without having to get the theorem prover involved. The theorem prover is invoked only at specific points in the analysis when the both the abstract value source and abstract value sink involve a predicate and the value-flow analysis is unable to resolve the problem using just approximations.

The analysis is also parameterized over the $\mathcal{D}_\Delta$ function (Fig. 5.4) used in the annotation process. Looking once more at the example at the end of Section 5.1.2, 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 5.1 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

[4] Adding knowledge to the value-flow analysis will probably slightly slow it down because it will then have to consider new kinds of abstract value sources and sinks, but the cost of this extra processing will most likely be small compared to the gain obtained from not using the theorem prover as much as before.

$$
\begin{array}{rcl}
\mathcal{R}^{\varphi}(\ell) & \stackrel{def}{=} & \{\ell\} \cup \mathcal{R}_u^{\varphi}(\ell) \\
\mathcal{R}_u^{\varphi}(\ell) & \stackrel{def}{=} & \bigcup_{\ell_i \in \varphi(\ell)} \mathcal{R}_t^{\varphi}(\ell_i) \\
\mathcal{R}_t^{\varphi}(\ell) & \stackrel{def}{=} & \begin{cases} \{\ell\} \text{ if } n^{\ell} \text{ or } \mathrm{int}_{fg}^{\ell\ell'} \text{ or } \mathrm{any}_{fg}^{\ell\ell'} \\ \{\ell\} \cup \mathcal{R}^{\varphi}(\ell_1) \cup \mathcal{R}^{\varphi}(\ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \text{ or } (c_{gf}^{\ell_1'\ell_1 \to} c_{fg}^{\ell_2\ell_2'})_{fg}^{\ell\ell'} \end{cases} \\
\mathcal{T}^{\varphi}(\ell) & \stackrel{def}{=} & (\mathrm{rec}\ ([\ell_i\ \mathcal{T}_u^{\varphi}(\ell_i)]_{\ell_i \in \mathcal{R}^{\varphi}(\ell)} \ldots)\ \ell) \\
\mathcal{T}_u^{\varphi}(\ell) & \stackrel{def}{=} & (\mathrm{union}\ \mathcal{T}_t^{\varphi}(\ell_i)_{\ell_i \in \varphi(\ell)} \ldots) \\
\mathcal{T}_t^{\varphi}(\ell) & \stackrel{def}{=} & \begin{cases} \mathrm{int} \text{ if } n^{\ell} \text{ or } \mathrm{int}_{fg}^{\ell\ell'} \\ \mathrm{any} \text{ if } \mathrm{any}_{fg}^{\ell\ell'} \\ (\ell_1 \to \ell_2) \text{ if } (\lambda x^{\ell_1}.e^{\ell_2})^{\ell} \text{ or } (c_{gf}^{\ell_1'\ell_1 \to} c_{fg}^{\ell_2\ell_2'})_{fg}^{\ell\ell'} \end{cases}
\end{array}
$$

Figure 5.8: Type reconstruction for the lambda calculus with unrestricted contracts.

check and Table 5.1 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. 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.[5]

5.3.4 Type Reconstruction

Since the flow of triples is simulated by the flow of the triple-free contract that approximates the triple's predicate, triples do not introduce any new kind of

[5]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.

abstract values. The only change in the definition of our type reconstruction process in Figure 5.8 is the addition of the any contracts.

5.4 Soundness

The soundness theorem from the previous chapter remains pretty much unchanged, with just the addition of the new orange blame color.

Theorem 5. *For a given annotated program p, let $p' \stackrel{\text{def}}{=} m' \ldots e^{\ell'}$ be such that $\vdash^{\mathtt{l}}_{\mathtt{p}} p \rightsquigarrow p'$. Then either:*

- *p reduces to $m \ldots v^{\ell}$ and then $[\![p']\!] \models \mathcal{T}^{\varphi}(\ell) \leq \mathcal{T}^{\varphi}(\ell')$,*
- *or p reduces to $(\mathit{blame}\ \pi\ s)^{\ell}$ and then $[\![p']\!] \models \{\langle \pi, s \rangle\} \subseteq \psi(\ell)$,*
- *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), s indicates the severity of the violation ($\mathcal{O}$ or $\mathcal{R}$), and $\leq$ is the subtyping relation between recursive types [5, 34].

The proof of soundness now relies of course on the existence of a proof of soundness for the theorem prover, and on a proof that the $\mathcal{D}_{\Delta}$ function computes a correct approximation of predicates. The constraints from Tables 5.1, 5.2, and 5.3 are still expressed in the form of Horn clauses though, so the same technique from Wand and Williamson [54] can be used to show entailment of sets of constraints across reductions.

5.5 Modularity

Our modularity theorem remains the same as in the previous chapter:

Theorem 6. *Given an annotated program p, let p' be such that $\vdash^{\mathtt{l}}_{\mathtt{p}} p \rightsquigarrow p'$. 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.*

The introduction of any contracts does not change anything to the proof, since, as far as modularity is concerned, any contracts behave in pretty much the same way as int contracts. We have to prove though that the introduction of contract triples does not invalidate our inter-tree flow lemma, on which the theorem above is based:

Lemma. *Given an annotated program p, let p' be such that $\vdash^{\mathtt{l}}_{\mathtt{p}} p \rightsquigarrow p'$. 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 $t = (\mathit{module}\ f^{\beta}\ v^{\ell_v})^{\ell_m}$ and $t' = (c^{\ell^+ \ell^-}_{fg} \Leftarrow f^{\beta})^{\ell_c}$.*

Proof Sketch. The proof remains the same, with the addition of one new possible case for inter-tree flows: through an ε^{ℓ} expression that shares its label with another expression in another tree.

We show that such flows are impossible as follows. 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. 5.1.2). 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. □

5.6 Analysis Complexity

While the core value-flow analysis now generates more constraints than in the previous chapter, it still generates a linear number of them. The complexity of the core value-flow analysis therefore remains the same as described in Section 4.6.

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 $\not\sqsubseteq$ both can be.

In practice though we expect the $\mathcal{D}_{\Delta}$ to be fairly simple and fast, since its only role is to compute a predicate-free approximation of a predicate based on its domain. Figure 5.4 shows a possible definition for $\mathcal{D}_{\Delta}$ that computes a useful approximation in time linear in the size of the contracts traversed, i.e., linear in the total size of contracts in the worst case. Using this definition of $\mathcal{D}_{\Delta}$, computing approximations for all the predicates used as contracts in a program therefore takes a worst-case time that is quadratic in the total

size of all the contracts in the program. It is easy to reduce that worst-case time to being linear in the total size of contracts by memoizing the computed domains.

Unlike the $\mathcal{D}_\Delta$ function, we expect the $\not\sqsubseteq$ relation to have a very high complexity. Nevertheless, the analysis as a whole should still have a decent running time since $\not\sqsubseteq$ is used only at very specific points in the analysis, when comparing predicates. This is in fact the whole point of using the $\mathcal{D}_\Delta$ function: to reduce as much as possible the need for $\not\sqsubseteq$ to analyze predicates by having instead the core value-flow analysis use predicate approximations whenever it can.

In practice which theorem prover to use is going to be determined by which trade-off between precision and complexity is acceptable to users of the debugger. We expect a simple theorem prover based on name equality and basic contract combinators to be enough in most cases. More powerful theorem provers can then be used when the one based on name-equality turns out to be insufficient. In such case, the time complexity can still be managed trough the use of a timer that limits the amount of time theorem provers spend trying to compare predicates.

5.7 Related Work

If the theorem prover used by our analysis can be expressed as an abstract interpretation [12], then the whole analysis is the combination of several ab-

stract interpretations and therefore an abstract interpretation as well: the $\mathcal{D}_\Delta$ function, which statically approximates predicate expressions, is obviously an abstract interpretation; the theorem prover might not be expressible as an abstract interpretation though (e.g. ESC [16] is not sound).

Most of the related works described in the previous chapters do not handle unrestricted types or contracts. Of course our ability to analyze unrestricted contracts comes at the price of the $\not\sqsubseteq$ relation being undecidable in the general case.

Other systems [1, 9, 19, 33] have investigated the combination of static types and dynamic checks to ensure program correctness. Flanagan's hybrid type checker [19] is closest to our system. It is in essence a statically undecidable extension of refinement types [28] that allows for arbitrary predicates. Since his type checker has to handle complex predicates, it 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. His use of a three-valued subtyping judgement, as opposed to our two-valued theorem prover, means that his system has the equivalent of one more error color though: when a contract check has been shown to be fine at the basic type level but has actually been proved to be violated at the higher level. Our system conflates this case (colored orange) with the case when a contract check has been shown to be fine at the basic type level but the higher

level is simply not powerful enough to be able to prove anything beyond that (orange color as well). We could transform our two-valued theorem prover into a three-valued one by asking the theorem prover to always try to prove both $e_1 \ldots e_n \not\sqsubseteq e$ and the negation of that property.

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 refinement predicate t. The user must therefore specify both B and t. In 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 [18].

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.

Chapter 6

Implementation

We have created a proof-of-concept static debugger based on our analysis. It implements the annotation phase of Section 5.1.2, and the lifting, constraints generation, and type reconstruction phases described in Section 4.3. 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 5.3 are represented as direct edges between nodes. Conditional constraints like the ones in Tables 5.1 and 5.2 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.

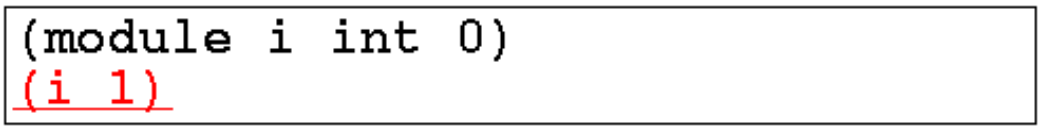

Figure 6.1: Example program with red error.

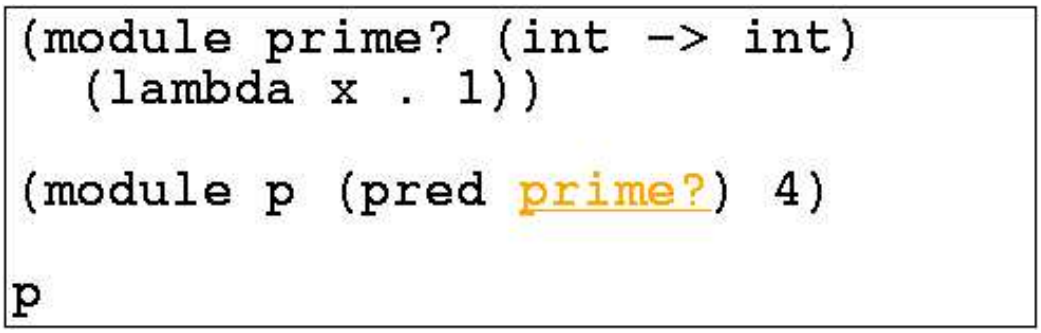

Figure 6.2: Example program with orange error.

Figure 6.1 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 μ, the main expression. This example corresponds to the cell in Table 5.1 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.

Our second screenshot in Figure 6.2 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

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

(module p (pred prime?) 4)

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

(f p)
```

Figure 6.3: Example program with no second `prime?` error.

a function that 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 5.1 that has an integer $n^{\ell_n}_{e_1 \ldots}$ as a source and a triple $\langle \ldots\ e_5\ \mathrm{int}^{\ell_5^+\,\ell_5^-}_{hi} \rangle^{\ell_6^+\,\ell_6^-}_{hi}$ 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 6.3 shows a use of the $\not\sqsubseteq$ relation. As in the previous example we define a predicate `prime?` and a 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 5.1 that has a triple $\langle \ldots\ e_1\ \mathrm{int}^{\ell_1^+\ell_1^-}_{fg}\rangle^{\ell_2^+\ell_2^-}_{fg}$ as source and another triple $\langle \ldots\ e_5\ \mathrm{int}^{\ell_5^+\ell_5^-}_{hi}\rangle^{\ell_6^+\ell_6^-}_{hi}$ 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, illustrating the fact that 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.

Chapter 7

Extending $\not\sqsubseteq$ to Contracts

As it is, Tables 5.1 and 5.2 are only partially parameterized over the $\not\sqsubseteq$ relation.

Consider the following example:

(module *prime?* (any→int) ...)

(module n (pred *prime?*) 3)

(module f (int→int) ...)

(f n)

The predicate *prime?* is defined to work on all values. When this predicate is used to define the contract for n, it is therefore transformed into a triple of the form $\langle \ldots$ *prime?* any$\rangle$. Table 5.1 tells us that when n then flows into the int domain contract of f, a red error is raised, since the any approximation used in the triple means the abstract values flowing out of that triple into the int check might include integers but also functions. There is in fact no

reason to flag a red error here since we know that the *prime?* predicate by itself mathematically ensures that all values satisfying it are integers. If the analysis were able to prove that the *prime?* predicate implies the int contract, then everything would be fine. So far the $\not\sqsubseteq$ relation has only been used to compare predicates to predicates, not predicates to contracts. It is therefore natural to extend that relation to handle contracts so that properties of the form $e \not\sqsubseteq c$ can be checked.[1]

There are five places in Tables 5.1 and 5.2 where constraints can be modified to use that extended version of the $\not\sqsubseteq$ relation. They all correspond to the case when a contract triple of the form $\langle \ldots e \text{ any} \rangle$ acts as source and a contract checks for integers or functions (regardless of whether those checks are part of a triple or not).

Symmetrically, there are cases when extending the $\not\sqsubseteq$ relation to check properties of the form $c \not\sqsubseteq e$ can be useful. Consider the case where an int contract acts as a source and a $\langle \ldots e \text{ int} \rangle$ triple acts as a sink. Table 5.1 shows that this should trigger an orange violation, because the integers flowing into the triple might not fulfill the predicate e. But in some cases the predicate e might be so weak that the simple fact that the in-flowing values are integers might be enough to prove that e is satisfied. In essence the predicate e is then weaker than its contract approximation int that is used as the third part of the triple: e is a vacuous predicate that always accepts all values that

[1] Such a modification then also helps to solve the problem we described when weakening the $\mathcal{D}_\Delta$ function in Section 5.3.3.

are in its domain. While such predicates are not very useful in practice, the $\not\sqsubseteq$ relation can still be extended to handle those cases.

Once again there are five cases in Tables 5.1 and 5.2 where constraint can thus be modified. They all correspond to cases when a triple-free contract source flows into a contract triple for which the contract approximation accepts the in-flowing contract and an orange violation would have been raised because of the predicate in the triple.

Note that the analysis can be modified in such way without requiring any similar change to the reduction rules of Section 5.2, and the resulting system will still be sound. The reason for this is that the only effect these changes can have is to potentially remove some red (for the $e \not\sqsubseteq c$ case) or orange (for the $c \not\sqsubseteq e$ case) false positives. The soundness and modularity theorems are therefore not affected by these changes. The resulting constraints are described in Tables 7.1 and 7.2. The additional constraints necessary for the analysis are the ones already described in Table 5.3.

After extending the $\not\sqsubseteq$ relation from handling properties of the form $e \ldots \not\sqsubseteq e$ to also handle properties of the form $e \not\sqsubseteq c$ and $c \not\sqsubseteq e$, the final question is then whether it is useful to also extend it to check properties of the form $c \not\sqsubseteq c$. Such an extension is doable but unnecessary, however, since comparing directly contracts to contracts is precisely what the core value-flow analysis is supposed to do. The constraints in Tables 7.1 and 7.2 are therefore as fully parameterized over the $\not\sqsubseteq$ relation as possible.

Source$\diagdown$*Sink*	$\text{int}^{\ell_5^+\ell_5^-}_{hi}$	$\langle \dots\ e_5\ \text{int}^{\ell_5^+\ell_5^-}_{hi} \rangle^{\ell_6^+\ell_6^-}_{hi}$	$\text{any}^{\ell_5^+\ell_5^-}_{hi}$	$\langle \dots\ e_5\ \text{any}^{\ell_5^+\ell_5^-}_{hi} \rangle^{\ell_6^+\ell_6^-}_{hi}$
$n^{\ell_n}_{e_1\dots}$		$\left.\begin{array}{c}\{\ell_n\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{c}\{\ell_n\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\text{int}^{\ell_1^+\ell_1^-}_{fg}$		$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ \text{int}\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ \text{int}\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\langle \dots\ e_1\ \text{int}^{\ell_1^+\ell_1^-}_{fg} \rangle^{\ell_2^+\ell_2^-}_{fg}$		$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\text{any}^{\ell_1^+\ell_1^-}_{fg}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$			$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ \text{any}\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$\langle \dots\ e_1\ \text{any}^{\ell_1^+\ell_1^-}_{fg} \rangle^{\ell_2^+\ell_2^-}_{fg}$	$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq\text{int}\end{array}\right\}\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$ $\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\sqsubseteq\text{int}\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$			$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$
$(\lambda x^\beta.e^\ell)^{\ell_\lambda}_{e_1\dots}$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$		$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_5^+)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_5^-)$ $\left.\begin{array}{c}\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	
$(c^{\ell_1^+\ell_1^-}_{gf}\!\to\!c^{\ell_2^+\ell_2^-}_{fg})^{\ell_3^+\ell_3^-}_{fg}$	$\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$		$\left.\begin{array}{c}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ (\dots\!\to\!\dots)\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_5^+)\subseteq\varphi(\ell_1^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_5^-)$	
$\langle \dots\ e_3\ (c^{\ell_1^+\ell_1^-}_{gf}\!\to\!c^{\ell_2^+\ell_2^-}_{fg})^{\ell_3^+\ell_3^-}_{fg} \rangle^{\ell_4^+\ell_4^-}_{fg}$				$\left.\begin{array}{c}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ e_3\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$

Table 7.1: Constraints creation for the extended $\not\sqsubseteq$ relation.

Source$\backslash$*Sink*	$(e^{\ell_5}\ e^{\ell_6})^{\ell_a}$	$(c_{ih}^{\ell_7^+\ell_7^-}\!\!\rightarrow\!\!c_{hi}^{\ell_8^+\ell_8^-})_{hi}^{\ell_5^+\ell_5^-}$	$\langle\dots\ e_5\ (c_{ih}^{\ell_7^+\ell_7^-}\!\!\rightarrow\!\!c_{hi}^{\ell_8^+\ell_8^-})_{hi}^{\ell_5^+\ell_5^-}\rangle_{hi}^{\ell_6^+\ell_6^-}$
$n_{e_1\dots}^{\ell_n}$	$\{\ell_n\}\subseteq\varphi(\ell_5)\Rightarrow\{\langle\lambda,\mathcal{R}\rangle\}\subseteq\psi(\ell_a)$	$\{\ell_n\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$	
$\mathrm{int}_{fg}^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\{\langle\lambda,\mathcal{R}\rangle\}\subseteq\psi(\ell_a)$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$	
$\langle\dots\ e_1\ \mathrm{int}_{fg}^{\ell_1^+\ell_1^-}\rangle_{fg}^{\ell_2^+\ell_2^-}$			
$\mathrm{any}_{fg}^{\ell_1^+\ell_1^-}$	$\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\{\langle\lambda,\mathcal{R}\rangle\}\subseteq\psi(\ell_a)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_6)\subseteq\varphi(\ell_1^-)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_1^+)\subseteq\varphi(\ell_a)$	$\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_7^+)\subseteq\varphi(\ell_1^-)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_1^+)\subseteq\varphi(\ell_8^-)$	
$\langle\dots\ e_1\ \mathrm{any}_{fg}^{\ell_1^+\ell_1^-}\rangle_{fg}^{\ell_2^+\ell_2^-}$	$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5)\\ e_1\not\sqsubseteq(\dots\!\rightarrow\!\dots)\end{array}\right\}\Rightarrow\{\langle\lambda,\mathcal{R}\rangle\}\subseteq\psi(\ell_a)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_6)\subseteq\varphi(\ell_1^-)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_1^+)\subseteq\varphi(\ell_a)$	$\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\not\sqsubseteq(\dots\!\rightarrow\!\dots)\end{array}\right\}\Rightarrow\{\langle h,\mathcal{R}\rangle\}\subseteq\psi(\ell_5^-)$ $\left.\begin{array}{c}\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\\ e_1\sqsubseteq(\dots\!\rightarrow\!\dots)\\ e_1\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_7^+)\subseteq\varphi(\ell_1^-)$ $\{\ell_1^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_1^+)\subseteq\varphi(\ell_8^-)$	
$(\lambda x^{\beta}.e^{\ell})_{e_1\dots}^{\ell_\lambda}$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_6)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_a)$	$\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_7^+)\subseteq\varphi(\beta)$ $\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell)\subseteq\varphi(\ell_8^-)$ $\left.\begin{array}{c}\{\ell_\lambda\}\subseteq\varphi(\ell_5^-)\\ e_1\dots\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	
$(c_{gf}^{\ell_1^+\ell_1^-}\!\!\rightarrow\!\!c_{fg}^{\ell_2^+\ell_2^-})_{fg}^{\ell_3^+\ell_3^-}$	$\{\ell_3^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_6)\subseteq\varphi(\ell_1^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_a)$	$\left.\begin{array}{c}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ (\dots\!\rightarrow\!\dots)\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_7^+)\subseteq\varphi(\ell_1^-)$ $\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\Rightarrow\varphi(\ell_2^+)\subseteq\varphi(\ell_8^-)$	
$\langle\dots\ e_3\ (c_{gf}^{\ell_1^+\ell_1^-}\!\!\rightarrow\!\!c_{fg}^{\ell_2^+\ell_2^-})_{fg}^{\ell_3^+\ell_3^-}\rangle_{fg}^{\ell_4^+\ell_4^-}$		$\left.\begin{array}{c}\{\ell_3^+\}\subseteq\varphi(\ell_5^-)\\ e_3\not\sqsubseteq e_5\end{array}\right\}\Rightarrow\{\langle h,\mathcal{O}\rangle\}\subseteq\psi(\ell_5^-)$	

Table 7.2: Constraints creation for the extended $\not\sqsubseteq$ relation (continued).

Chapter 8

Future Work

Our model of a static debugger needs to be extended to cover some of the most common contract combinators used in DrScheme's contract system. The two simplest ones are and/c and or/c.

When considered as an abstract value sink, the first is easy to add to the analysis by creating special constraints that forward an abstract value to the next contract check in the and/c-ed sequence of contracts whenever the value has passed their own check.

When considered as an abstract value sink as well, the second is more difficult to handle. The problem is to determine which contract should be used to check a given abstract value. For example, imagine that an int abstract value flows into a contract such as (or/c int (int→int)). If the abstract value flows into both components of the or/c, then a contract violation will always be detected. While such a behavior is a sound approximation of

the runtime behavior, it generates many false positives. The solution is to look at the top-level contract constructor for each element of the or/c and based on that decide to which element the incoming int abstract value should go for further checking. Unfortunately there is no guarantee that such top-level contract constructors are unique in the list of or/c-ed contracts. For example, if a functional abstract value flows into a contract like (or/c (int→int) ((int→int)→any)), to which of the two functional contracts should the in-flowing abstract value go? At this point there are only two solutions: fall back to a conservative behavior and make the abstract value flow into both arrow contract checks; or force the debugger's user to merge the two contracts together to transform (or/c (int→int) ((int→int)→int)) into ((or/c int (int→int))→(or/c int any)) to ensure that contracts that are or-ed always have unique top-level constructors. The former solution generates false positives, the latter makes the contract check less precise since it now accepts abstract values like (int→any).

When the two and/c and or/c contract combinators are considered as abstract value sources, the problem is easier: analyzing or/c is done by simply generating abstract values from each of the contracts that are or-ed and making those values flow into a single value set. In the case of and/c, the analysis can again use special constraints so that the smallest value set (presumably coming from the rightmost contract in the and-ed sequence of contracts) is forwarded through the chain of contracts to become the value of the whole combined contract.

In general whether other contract constructors can be implemented will depend very much on how well their semantics can be adapted to a set-based analysis. Anaphoric contracts should be easy to analyze, since they closely correspond to the analysis's idea of a flow between two value sets. A contract constructor like not/c will be easy to analyze when used as a contract check, by simply swapping the two possible outcomes of the check (do nothing or flag a contract violation). It will probably be impossible to analyze it precisely when considered as a value source: the analysis is based on a fixed-point computation that requires value set to grow monotonically. It is therefore impossible when trying to analyze a contract like (not/c int) to take an abstract value like any (representing the universe of all possible abstract values) and somehow remove from it the set of integers represented by int. The only solution will then be to simply conservatively approximate (not/c int) with any, which will give sound but most likely not very precise results. Analyzing other contract constructors like between/c would require either a precise numerical analysis based on abstract interpretation, or a strong theorem prover than can handle full integer arithmetic.

Other programming constructs need to be added to the analysis. Experience with the MrFlow static debugger [43] show that, for example, adding recursive data structures is easy, while adding generative records requires a huge amount of ad-hoc analysis. Analyzing functions with variable arities is relatively easy when using a set-based analysis [43] while analyzing macro code is most likely quite complex, etc. One interesting construct to study is

exceptions. It is an open question whether raising and catching exceptions can be simulated by creating error flows between blame sets (our analysis currently never has any such flows).

Another important area of exploration will be contract inference. The modular analysis currently requires the user to put contracts on module interfaces. By using a backward analysis we expect that the debugger will be able to infer those contracts from the invariants required by the user's code. It is likely that the inferred contracts will contain many invariants that the user does not wish to check for. The debugger will therefore require a contract simplification system, which, using heuristics, will help the user extract from the inferred contract those invariants that are relevant.

Since the analysis is parameterized over a theorem prover, we should be able to use it as an experimental platform to test several provers (e.g., Simplify [15], ACL2 [36]). By varying the respective powers of the core value flow analysis and the theorem provers we will gain experience on the trade-offs between precision and running time for the whole analysis. Practical use of the debugger and feedback from users will then allow us to decide on a theorem prover that best fits our needs.

Work should also be done on using a theorem prover or interactive proof checker to automate as much as possible the soundness proof of the analysis. In fact using a constructive proof of existence of a solution to the analysis's constraints would allow us to have both a proof of correctness and extract from that proof an implementation of the analysis [8].

Chapter 9

Conclusion

Our work 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. The analysis can indicate whether a given contact is always satisfied, partially satisfied, or completely violated. Moreover that analysis is parameterized over both a predicate approximation relation and a theorem prover.

Bibliography

[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] Agesen, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26, London, UK, 1995. Springer-Verlag.

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

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

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

[6] Besson, F. and T. Jensen. Modular class analysis with datalog. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*. Springer, 2003.

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

[8] Cachera, D., T. Jensen, D. Pichardie and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, September 2005.

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

[10] Chatterjee, R., B. G. Ryder and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[11] Considine, J. Efficient hash-consing of recursive types. Technical Report 2000-006, Boston University, January 2000.

[12] Cousot, P. and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[13] Cousot, P. and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 170–181, New York, NY, USA, 1995. ACM Press.

[14] 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.

[15] Detlefs, D., G. Nelson and J. Saxe. Simplify: A theorem prover for program checking, 2003.

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

[17] Dreyer, D., K. Crary and R. Harper. A type system for higher-order modules. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 236–249, New York, NY, USA, 2003. ACM Press.

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

[19] Flanagan, C. Hybrid type checking. In *Proceedings of the symposium on Principles of Programming Languages*, pages 245–256, 2006.

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

[21] 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.

[22] Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[23] Flatt, M. *MzScheme: Language Reference Manual*. Rice University, 2000. Version 103.

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

[25] Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[26] 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.

[27] 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.

[28] Freeman, T. and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

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

[30] Heintze, N. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and functional pro gramming*, pages 306–317. ACM Press, 1994.

[31] Heintze, N. Control-flow analysis and type systems. In *Static Analysis Symposium*, pages 189–206, 1995.

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

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

[34] 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.

[35] Hughes, J. Backwards Analysis of Functional Programs. In Bjørner and Ershov, editors, *IFIP Workshop on Partial Evaluation and Mixed Computation*, 1987.

[36] Kaufmann, M., J. S. Moore and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[37] Leroy, X. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

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

[39] 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.

[40] Mauborgne, L. Improving the representation of infinite trees to deal with sets of trees. In Smolka, G., editor, *European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 275–289. Springer-Verlag, 2000.

[41] McAllester, D. and N. Heintze. On the complexity of set-based analysis. In *1997 International Conference on Functional Programming*, 1997.

[42] Meunier, P., R. B. Findler and M. Felleisen. Modular set-based analysis from contracts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2006.

[43] 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.

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

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

[46] 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.

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

[48] 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.

[49] Sestoft, P. Replacing function parameters by global variables. Master's thesis, DIKU, Univ. of Copenhagen, Oct. 1988.

[50] 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.

[51] Smith, S. F. and T. Wang. Polyvariant flow analysis with constrained types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 382–396, London, UK, 2000. Springer-Verlag.

[52] Tang, Y. M. and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In Hagiya, M. and J. C. Mitchell, editors, *Theo-*

retical Aspects of Computer Software, pages 224–243. Springer, Berlin, Heidelberg, 1994.

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

[54] 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.

[55] 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.

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