Foundations for Behavioral Higher-Order Contracts

A dissertation presented

by

Christos Dimoulas

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

December, 2012

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM

THESIS TITLE: Foundations for Higher-order Behavioral Contracts

AUTHOR: Christos Dimoulas

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Matthias Felleisen — 12 Dec 2012
Thesis Advisor *Date*

Amal Ahmed — 12/12/12
Thesis Reader *Date*

Mitch Wand — 12/12/12
Thesis Reader *Date*

Cormac Flanagan — 12/12/12
Thesis Reader *Date*

GRADUATE SCHOOL APPROVAL:

12/14/12
Director, Graduate School *Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

12/14/2012
Recipient's Signature *Date*

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Abstract

Contracts are a popular mechanism for enhancing the interface of components. In the world of first-order functions, programmers embrace contracts because they write them in a familiar language and easily understand them as a pair of a pre-condition and a post-condition. In a higher-order world, contracts offer the same expressiveness to programmers but their meaning subtly differs from the familiar first-order notion. For instance, it is unclear what the behavior of dependent contracts for higher-order functions or of contracts for mutable data should be. As a consequence, it is difficult to design monitoring systems for such higher-order worlds.

In response to this problem, this dissertation investigates complete monitors, a formal framework for deciding if a contract system is correct. The intuition behind the framework is that a correct contract system should:

- mediate the exchange of values between contracted components
- and blame correctly in case of contract violations.

The framework reveals flaws in the semantics for dependent contracts from the literature and suggests a natural fix. In addition, this dissertation demonstrates the usefulness of the framework for language design with a language with contracts for mutable data and a language that mixes typed and untyped imperative programs. The final contribution is the provably correct design of a novel form of contracts, dubbed options contracts, that mix contract checking with random and probabilistic checking.

Acknowledgments

First and foremost, I am indebted for life to Matthias Felleisen, my advisor, for his guidance, advice and support over the last six years. He taught me by his example the correct way to carry out research and behave to my colleagues. Thank you Matthias for being my teacher and mentor, not in the narrow modern academic sense but in the original life-transforming one. It is a unique privilege to be your student!

I would like to thank Amal Ahmed, Mitch Wand and Cormac Flanagan for serving as my thesis committee members and for their insightful comments on my work. All three of them are also my research collaborators and co-authors and have taught me some of their secrets on the craft of research in programming languages. Moreover, Mitch is the first person I wrote a research paper with.

Many thanks to all my co-authors over the last 6 years for their contributions and insights to my research. Special thanks to Robby Findler and Sam Tobin-Hochstadt. The many discussions with Sam have resulted in some important conclusions about my work. For instance, it was Sam that first noticed the relation between my contract checking and random testing system and business option contracts and thus christened my new contracts option contracts. Robby showed me the ropes of Racket's contract system, shared his experience and insights on contracts and helped a lot with my first POPL paper. He also invited me to participate in his PLT Redex validation project which lead to a second POPL paper for me.

The past and current members of the PRL group at Northeastern created a fertile and engaging research environment. Their feedback to my work is invaluable. Vincent St-Amour, Aaron Turon, Stephen Chang, Asumu Takikawa, Ian Johnson, Jesse Tov and Felix Klock spent a significant amount of time and effort reading my papers, listening to my talks and giving comments that improved my prose and presentations.

This work would not be possible without Racket and its community of users and developers. Many thanks to all of them and to the members of PLT that I collaborated with.

Nikos Papaspyrou, my undergrad advisor at NTUA, was the first person that believed in my potential, introduced me to programming languages and urged me to continue my studies.

Therapon Skotiniotis, Vasilis Koutavas, Vangelis Kanoulas, Dimitris Kanoulas and Alper Okçan offered me their friendship and made Boston feel less far away from home. I am especially thankful for all the discussions and their advice on matters ranging from professional and research problems to personal issues.

My research over the last six years was generously supported by Northeastern University and the US Air Force Office of Scientific Research.

There are no words to describe the impact of Gönen Memişoglu on my life during the last two and half years. She has been always there for me putting up with my erratic behavior, creating a relaxing environment at home and trying to bring my life to a balance. Ordinary descriptions, such as girlfriend or partner, do not capture the importance of her role in my life. Teşekkürler birtanem!

This dissertation is as much my achievement as it is an achievement of my parents, Ersi and Yanni, and my sister, Marieta. Their sacrifices all these years enabled me to chase my dreams. I hope I have made them proud.

Last but not least, thank you God. For everything!

Στους γονείς μου.

Contents

List of Figures

CHAPTER 1

Introduction

Modern software development is a highly distributed and collaborative effort. Programmers write self-standing components to perform basic tasks and others compose components to construct large and complicated systems. As early as 1970 [49], it had become clear that the modularization of software is the only way forward for programmers to keep up with the needs of our world for more and more software products. In fact, software engineers claim that the power of components is so strong that it will lead to a full-blown market of components [64]. During the last decade, the first steps of this vision have become already reality with the emergence of service-oriented programming.

From modules to web services, the notion of a component is loose and it can be materialized in many different ways. No matter what form a component takes, however, it always has an interface, a description of the service it provides to its clients and the restrictions that it imposes [64]. Just like components, interfaces also take many forms with the most popular one being that of types, syntactic descriptions of the structure of the inputs and outputs of a component.

Even though types can express important properties of the values that components exchange, it is often necessary to go beyond their reach to achieve a strong enough interface. For instance, consider a component that provides an encryption

function for strings. A type system can express that the result is a string, but for a sensitive client of the component a deeper correctness guarantee may be necessary, namely that the result can be only decrypted using a secret key. The absence of a system that allows the programmer to state and check the property may result in an incorrectly encrypted string to move from one component to another under the assumption that it is correct. If at some point one component receives the "encrypted" string and realizes that it is flawed, the detection of the source of the error, which goes back to the encryption component, requires the re-construction of the path of the string and may involve dozens of components.

Adding such interfaces such strong guarantees to interfaces naturally leads to the notion of behavioral software contracts. Eiffel [47] injected them to mainstream programming as predicates on the results and outputs of components. Programmers use plain predicates to express behavioral requirements and if one predicate of the contracts fails, they know how to interpret the failure: if it is a failure of a predicate on an argument, they blame the client component; in contrast, if it is a failure of a predicate on a result, they blame the provider component. For our encryption example, a programmer would state the correctness property as a predicate on the result and the contract system would check it immediately when the encryption function delivered its result. Thus the error would be detected fast and accurately. In sum, contracts for such first-order scenarios are easy and intuitive.

Modern software is not only component-based but also higher-order. Objects and closures are the norm in all cutting-edge programming languages and naturally programmers would like to have contracts that can express properties of those values, too. Unlike flat values such as strings, higher-order values challenge the design of contract systems. First, they require a rethinking of contract checking. Validating whether a function meets a contract is incomputable in general and thus breaks the simple checking strategy for the first-order world. Second, the blame story is

more involved than in the first-order case. Functions and objects as first-class values imply that a component may run *unknown* code. Thus the exact meaning of the service and the client of a component is not as simple as in the first-order world where all the dependencies between components are known in advance.

In response, Findler and Felleisen [32] suggest a semantics for checking and assigning blame for contracts for higher-order functions. Their idea, though, is open to discussion about the meaning and correctness of higher-order contracts [6, 33]. Moreover, attempts to transfer the result of Findler and Felleisen to different language features often face difficulties concerning design decisions and correctness [37, 39]. Finally, even if programmers understand the intuitive idea behind Findler-Felleisen contract checking and blame assignment, they no have clear-cut evidence of the guarantees that languages with contracts offer.

Based on this history, I consider it necessary to develop a semantic framework for evaluating the effectiveness of contract checking and the accuracy of blame assignment. This framework should incorporate the intuition of the programmer that a contract system offers effective protection to components with contracts Furthermore in case of contract violations, the contract system should blame the source of the violation.

> This dissertation develops a theoretical foundation for higher-order behavioral contracts, and it shows how to use this foundation to evaluate existing systems and design new ones.

This dissertation consists of seven chapters:

- the following chapter is a short history of contracts and the challenges posed by higher-order contracts,
- chapter 3 presents a formal framework for proving a contract system blame correct [24],

- chapter 4 shows how to strengthen the framework of the chapter 3 to prove that a contract system not only blames correctly but also protects components effectively, a property dubbed complete monitoring [26],

- chapter 5 attests to the applicability of compete monitors in language design with a series of examples [26],

- chapter 6 demonstrates the scalability of our theory via the design of option contracts, a novel contract system that combines contract checking and probabilistic and random checking [25],

- finally, chapters 7 and 8 complete this dissertation with a discussion of related work and concluding remarks.

CHAPTER 2

A Short History of Contracts

2.1 The Origins of Contracts

Eiffel [47] is the first language to offer built-in support for behavioral contracts. When programmers define a class in Eiffel, they can specify pre-conditions and post-conditions on a per method basis along with class invariants.

We can transfer the ideas of Eiffel to a functional world with contracts for first-order functions. For instance, figure 2.1 presents an extract of an interface for a high-school math library's `sqrt` function and its contract in Racket [35].

```
#lang racket

(provide
 (contract-out
  [sqrt
   ;; for some small real ε
   (->d ([input (and/c positive? real?)])
        (result (<= (abs (- (* input input) result)) ε)))]))

(define (sqrt input) ...)
```

Figure 2.1: A first-order function contract

Like in Eiffel, a contract is a pair of assertions: a pre-condition on the flat input of `sqrt` and a post-condition on its `result`. The contract is a dependent one in the sense that it makes the `input` value available in the post-condition. As a result, the contract specifies the `sqrt` function in a rather precise fashion.[1] In Racket, we use the contract combinator `->d` to construct a dependent function contract[2] and **`contract-out`** to attach a contract to an identifier before exporting it from a module.

As the example shows, programmers use intuitive contract combinators, such as `and/c` and `->d`, to compose contracts from ordinary predicates, such as `positive?` and `real?`. The combinators borrow notation from type systems and thus they are familiar to the programmer.[3] The predicates are written in the same language as ordinary programs and hence their meaning is transparent for a programmer. This last feature of contracts also allows programmers to easily scale the precision of contracts.

Despite the complexity of the properties that contracts can describe, the programmer has an intuitive understanding of what the contract system offers, in a first-order world. The creator and the clients of a component can easily extract the component's obligations from a contract. For instance, for the `sqrt` example, the obligations and benefits of the math library and its clients are:

	library's obligation	**client's obligation**
library's benefit		pre-condition
client's benefit	post-condition	

Based on the above information the programmer can safely assume that the contract system enforces the benefits the contract promises. Conversely, if a com-

[1] Eiffel' contracts for methods are also dependent, arguments are visible in the post-condition by default.

[2] Non-dependent function contracts use the contract combinator `->`.

[3] In Eiffel the situation is similar. Contracts are part of the class definition and their pieces are indicated with self-explanatory tags.

ponent does not behave as promised, the contract system has a method for tracking the violation back to specific code.

As far as the contract system itself is concerned, contract checking is simple; the two predicates are checked immediately, the pre-condition at the entrance point and the post-condition at the exit point of the function. In addition, it is trivial to decide which component is to blame in case one of the predicates fails. The client is always responsible for failures of a pre-condition and the provider component is always responsible for failures of a post-condition.

2.2 From First-Order to Higher-Order Contracts

The simplicity of the world of first-order contracts is one of the reasons contracts have become popular among programmers [27, 36, 40, 41, 43, 44, 45, 46, 53, 58, 59]. However, Eiffel, Racket and most of the modern programming languages come with higher-order features such as closures and objects. It is only natural for programmers to ask for contracts that can protect higher-order features. For instance, figure 2.2 shows a contract that a programmer wishes to write for a higher-order derivative operator `d/dx`. The code fragment specifies a function that maps a real-valued function `f` in the interval $(0,1)$ and a real number ε to a real-valued function `fp` on the same interval. A quick look at the contract reveals that the pre-condition and post-condition are function contracts. From Rice's theorem [56], we know that this check is not computable as it is for contracts on first-order functions. Thus the simple first-order approach to contract checking breaks.

Nevertheless, Eiffel does manage to support contracts for its higher-order features to some extent through its nominal contract system, a contract system where each contract is distinct and identifiable through a unique name.[4] Unlike Eiffel,

[4] Eiffel achieves this by associating contracts with its nominal classes.

```
(provide
 (contract-out
  [d/dx
   (-> ([f (-> 0<real<1? 0<real<1?)]
        [ε real?])
       (fp (-> 0<real<1? real?)))]))
```

Figure 2.2: A higher-order function contract

Racket does not have a nominal contract system but, for the sake of the presentation, we use Racket notation to show how Eiffel suggests to handle contracts for higher-order functions. Figure 2.3 shows what the contract from figure 2.2 would look like in a nominal world. In this setting, attaching a contract to a value results in tagging the value as satisfying the contract. In essence, the component that attaches a contract to a value guarantees that the value meets the contract. Functions that carry a contract consume only values tagged with the contract of their pre-condition and have to return a value tagged with the contract of their post-condition. If the first requirement doesn't hold the client is blamed. In case the function breaks the second requirement, the contract system blames the component that provides the function under the contract. For instance, in our example one could apply `d/dx` to `foo` without a contract violation. In contrast an application of `d/dx` to `bar` would lead to a contract error pinpointing the calling context.

Our example already hints at problem that a nominal contract system poses for higher-order programming. A function that has a contract can only accept arguments that are protected by the function's precondition. Thus programmers have to work hard to make sure that values are tagged with the appropriate contract. As a consequence, programs get quickly polluted with contract annotations. One could argue that the contract system should tag values with contracts automatically and relieve the programmer from the pain. However, in a language with first-class functions like Racket, a context can call unknown objects. Hence, blaming the context

```
(define 0<real<1?/0<real<1?  (-> 0<real<1? 0<real<1?))

(define 0<real<1/real?  (-> 0<real<1? real?))

(provide
 (contract-out
  [d/dx
   (-> ([f 0<real?<1?/0<real?<1? ]
        [ε real?])
       (fp 0<real?<1?/real? ))]
  [sigmoid 0<real<1?/0<real<1?]
  [scale 0<real<1?/real?]))
```

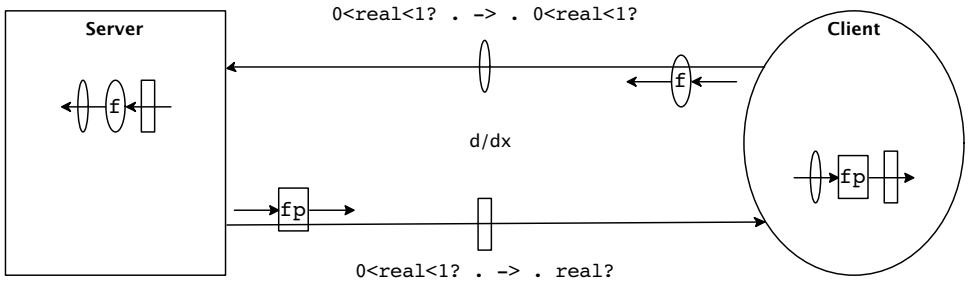
Figure 2.3: A nominal higher-order contract

for misusing a function implies blaming a component that is unaware of its contractual obligation. Thus, unless there is an explicit mechanism for components to declare their contractual responsibilities and expectations, blame does not carry useful information.

Eiffel's tight integration of contracts into classes creates another problem. For an object to be useful in multiple contexts, it has to be able to carry different contracts in different contexts. Since an object inherits the contracts of its class, the requirement implies that programmers define different subclasses for each context and make sure that each method expects objects of the appropriate sub-class. Consider a class with a method with the `0<real<1?/0<real<1?` contract. If we want to pass an object of this class to a method that expects an object with a method with the `0<real<1?/real?` contract, we need to create a sub-class of the original class with the latter contract on the method. In more complicated scenarios that involve more version of the same method, the amount of additions and rewriting in the class hierarchy increases dramatically. In sum, while class subtyping reduces some of the contract annotation overhead, the tight relation between class definitions and contracts leads to code duplication. Hence, even an Eiffel-style nominal contract

system is not the right solution for monitoring the behavior of higher-order features.

Instead, Findler and Felleisen [32] propose a structural approach. Their contract system does not attempt to check contracts on functions immediately. Rather, it decomposes such contracts, wraps arguments and results with the pre-condition and post-conditions respectively and thus delays contract checking for functions until contracts are decomposed to predicates for flat values. The following diagram depicts the proposal of Findler and Felleisen as applied to our *deriv* example:



When the client component uses the deriv function, the contract system intervenes and instead of passing the argument to the `d/dx` function directly, it creates a wrapper proxy value for the argument using the contract's pre-condition. This proxy ensures that the argument receives and returns the kind of real numbers specified in the contract. The contract system uses the post-condition of the contract to treat similarly the result of `d/dx`.

As for blame, the contract system associates to each contract two identifiers, one for the component that attaches the contract to a function and one for the component that imports the function. In our example these names are `server` and `client`, respectively. The first identifier indicates the component responsible for the pre-condition and the second one the component responsible for the post-condition. If these two sub-contracts are predicates for flat values and the corresponding checks fail the contract system blames the responsible party. If they are function contracts, the contract system switches the roles of the two identifiers when it creates the

proxy for the argument. Intuitively this accounts for the fact that the client component is responsible for the result of the argument while the the provider component is the one that is going to use the argument and is responsible for these uses. Our diagram demonstrates the responsible parties for the contract of `d/dx` and its subcontracts with shapes; each contract has the same shape as the component that is responsible for it.

In sum, the Findler-Felleisen system delays contract checking for higher-order functions until a flat value witness is available to check a flat predicate. At the same time, the system maintains and manages meta-information to associate each contract failure to a responsible component to blame. Despite the intuitive nature of the process, the behavior of the contract system is not simple and calls for a rigorous description.

2.3 CPCF: A Language with Contracts

The purpose of this section is to gives a formal description of the Findler-Felleisen system. The starting point of our formal presentation is CPCF, a contract-equipped variant of Plotkin's PCF. PCF is a programming language derived from the simply-typed λ-calculus [55]:

$$
\begin{array}{lrcl}
\textbf{Types} & \tau & = & o \mid \tau \to \tau \\
 & o & = & \mathcal{N} \mid \mathcal{B} \\
\textbf{BaseValues} & b & = & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \texttt{tt} \mid \texttt{ff} \\
\textbf{Values} & v & = & b \mid \lambda x{:}\tau.e \\
\textbf{Terms} & e & = & x \mid v \mid e+e \mid e-e \mid e \wedge e \mid e \vee e \mid \texttt{zero?}(e) \\
 & & \mid & \texttt{if}\ e\ e\ e \mid \mu x:\tau.e \mid e\ e
\end{array}
$$

We assume a simple[5] type system for the language. Figure 2.4 shows the corresponding rules. Types do not play a significant role for the rest of the development, and we omit them entirely unless needed for disambiguation. As usual, programs are closed terms.

$$\frac{\mathtt{n} \in \{...,-1,0,1,...\}}{\mathcal{G} \vdash \mathtt{n} : \mathcal{N}} \qquad \frac{}{\mathcal{G} \vdash \mathtt{tt} : \mathcal{B}} \qquad \frac{}{\mathcal{G} \vdash \mathtt{ff} : \mathcal{B}}$$

$$\frac{\mathcal{G} \vdash e_1 : \mathcal{N} \quad \mathcal{G} \vdash e_2 : \mathcal{N}}{\mathcal{G} \vdash e_1 + e_2 : \mathcal{N}} \qquad \frac{\mathcal{G} \vdash e_1 : \mathcal{N} \quad \mathcal{G} \vdash e_2 : \mathcal{N}}{\mathcal{G} \vdash e_1 - e_2 : \mathcal{N}}$$

$$\frac{\mathcal{G} \vdash e_1 : \mathcal{B} \quad \mathcal{G} \vdash e_2 : \mathcal{B}}{\mathcal{G} \vdash e_1 \wedge e_2 : \mathcal{B}} \qquad \frac{\mathcal{G} \vdash e_1 : \mathcal{B} \quad \mathcal{G} \vdash e_2 : \mathcal{B}}{\mathcal{G} \vdash e_1 \vee e_2 : \mathcal{B}}$$

$$\frac{\mathcal{G} \uplus \{x : \tau_a\} \vdash e : \tau_f}{\mathcal{G} \vdash \lambda x{:}\tau_a.e : \tau_a \to \tau_f} \qquad \frac{\mathcal{G} \uplus \{x : \tau\} \vdash e : \tau}{\mathcal{G} \vdash \mu x{:}\tau.e : \tau}$$

$$\frac{\mathcal{G} \vdash e : \mathcal{N}}{\mathcal{G} \vdash \mathtt{zero?}(e) : \mathcal{B}} \qquad \frac{\mathcal{G} \vdash e : \mathcal{B} \quad \mathcal{G} \vdash e_1 : \tau \quad \mathcal{G} \vdash e_2 : \tau}{\mathcal{G} \vdash \mathtt{if}\ e\ e_1\ e_2 : \tau}$$

Figure 2.4: PCF: typing rules

Next we equip the language with a call-by-value semantics. While *any* formalization of the semantics should work, we rely on reduction semantics [30, 54]. An evaluation context E is a context with the hole in a leftmost-outermost position:

$$\begin{array}{rrcl} \textbf{Ev. Ctxt.} & E & = & [\,] \mid E + e \mid v + E \mid E - e \mid v - E \mid E \wedge e \mid E \vee e \\ & & \mid & \mathtt{zero?}(E) \mid \mathtt{if}\ E\ e\ e \mid E\ e \mid v\ E \end{array}$$

Figure 2.5 shows the reduction rules for PCF. The notions of reduction generate the compatible closure with evaluation contexts, which is the standard reduction relation of the call-by-value PCF calculus.

[5] We conjecture that type polymorphism and type inference are entirely orthogonal to our goal.

$E[\cdots]$	$\longmapsto$	$E[\cdots]$	
$\mathtt{n}_1 + \mathtt{n}_2$	.	$\mathtt{n}$	where $n_1 + n_2 = n$
$\mathtt{n}_1 - \mathtt{n}_2$	.	$\mathtt{n}$	where $n_1 - n_2 = n$
$\mathtt{zero?}(\mathtt{0})$	.	$\mathtt{tt}$	
$\mathtt{zero?}(\mathtt{n})$	.	$\mathtt{ff}$	if $n \neq 0$
$v_1 \wedge v_2$	.	v	where $v_1 \wedge v_2 = v$
$v_1 \vee v_2$	.	v	where $v_1 \vee v_2 = v$
$\mathtt{if\ tt}\ e_1\ e_2$	.	e_1	
$\mathtt{if\ ff}\ e_1\ e_2$	.	e_2	
$\lambda x.e\ v$	.	$\{v/x\}e$	
$\mu x.e$	.	$\{\mu x.e/x\}e$	

Figure 2.5: PCF: reduction semantics

Adding contracts to PCF requires extensions to the language of types and terms:

Types	τ	$=$	$\ldots \mid \mathit{con}(\tau)$
Contracts	κ	$=$	$\mathtt{flat}(e) \mid \kappa \mapsto \kappa \mid \kappa \overset{d}{\mapsto} (\lambda x : \tau.\kappa)$
Terms	e	$=$	$\ldots \mid \mathtt{mon}_l^{l,l}(\kappa, e) \mid \mathtt{error}_l^l \mid \mathtt{check}_l^l(e, v)$
Labels	l	$\in$	$\mathbb{L}$

The result is CPCF, PCF with contracts.

CPCF comes with three kinds of contracts: flat contracts, $\mathtt{flat}(e)$, that specify an assertion via a predicate e; function contracts $\kappa_1 \mapsto \kappa_2$, which compose a pre-condition contract, κ_1, for the argument of a function with a post-condition, κ_2, for its result; and dependent function contracts, $\kappa_1 \overset{d}{\mapsto} (\lambda x : \tau.\kappa_2)$, that are like function contracts except they make the argument visible inside the post-condition κ_2 to allow the expression of properties of that relate the result to the argument of a function. The language integrates contracts and terms with a contract monitor form: $\mathtt{mon}_j^{l,k}(\kappa, e)$. Its purpose is to observe the values that flow from e, dubbed the *server*, to its context, dubbed the *client*, and back and to make sure they satisfy contract κ. The labels on the statement uniquely identify the three components related to

$$\frac{}{\mathcal{G} \vdash \mathtt{error}^k_j : \tau} \qquad \frac{\mathcal{G} \vdash e : \mathcal{B} \quad \mathcal{G} \vdash e_0 : \tau}{\mathcal{G} \vdash \mathtt{check}^k_j(e, e_0) : \tau} \qquad \frac{\mathcal{G} \vdash \kappa : con(\tau) \quad \mathcal{G} \vdash e : \tau}{\mathcal{G} \vdash \mathtt{mon}^{k,l}_j(\kappa, e) : \tau}$$

$$\frac{\mathcal{G} \vdash e : o \to \mathcal{B}}{\mathcal{G} \vdash \mathtt{flat}(e) : con(o)} \qquad \frac{\mathcal{G} \vdash \kappa_1 : con(\tau_1) \quad \mathcal{G} \vdash \kappa_2 : con(\tau_2)}{\mathcal{G} \vdash \kappa_1 \mapsto \kappa_2 : con(\tau_1 \to \tau_2)}$$

$$\frac{\mathcal{G} \vdash \kappa_1 : con(\tau_1) \quad \mathcal{G} \uplus \{x : \tau_1\} \vdash \kappa_2 : con(\tau_2)}{\mathcal{G} \vdash \kappa_1 \overset{d}{\mapsto} (\lambda x : \tau_1.\kappa_2) : con(\tau_1 \to \tau_2)}$$

Figure 2.6: CPCF: typing rules

the monitor: k for the server, l for the client and j for the contract.[6] In case of a contract violation, the contract system uses the labels to issue an error report, $\mathtt{error}^k_j$. The form $\mathtt{check}^k_l(e\,v, v)$ implements the application of the predicate e from a flat contract $\mathtt{flat}(e)$ to a value v. The last two constructs are not available at source code but appear only in intermediate terms during evaluation.[7]

Figure 2.6 shows the typing rules for this language extension. Errors are available at all types. A monitoring term demands that the first position is a contract for a certain type and that the second term is a term at that type. A predicate check has the same type as its second argument if its first argument is a boolean term. Next, a flat contract is just a predicate on base types. In contrast, the two versions of function contracts combine two existing contracts to a contract for a function, restricting both its domain and range types. A program is a term of ground type.

Our syntax ensures that contracts show up in $\mathtt{mon}^{k,l}_j(\kappa, e)$ terms only. Intuitively, this term enforces the contract κ between the server e and its client component. For flat, first-order values, this just means checking that the value of e has the

[6] In an implementation, the server label *is* the name of the module that provides a contract-protected function, the client label *is* the name of the module that imports the function, and the contract label *is* the name of the module that provides the contract.

[7] Enforcing this restriction is straightforward, either with a separation of the syntax in source and intermediate syntax, or with a type-system judgment that excludes programs that contain errors and checks from valid source programs.

desired property. For functions, it requires checking the arguments to the function and its result. Finally, for higher-order functional values, the monitoring requires additional checks on the use of functions as arguments. If this monitoring process discovers that a predicate from an embedded flat contract does not hold, it raises the exception $\texttt{error}_j^k$ to denote that component k violates contract j.

Our formal semantics adapts the system of Findler and Felleisen [32] to the syntax of CPCF. We start with some additional evaluation contexts:

$$\textbf{Ev}.\ \textbf{Ctxt}.\quad E \;=\; \ldots \mid \texttt{mon}_l^{l,l}(\kappa, E) \mid \texttt{check}_l^l(E, v)$$

and the reduction rules for monitors:

$E[\cdots]$	$\longmapsto$	$E[\cdots]$
$\texttt{mon}_j^{k,l}(\texttt{flat}(e), v)$	$\cdot$	$\texttt{check}_j^k(e\ v, v)$
$\texttt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)$	$\cdot$	$\lambda x.\texttt{mon}_j^{k,l}(\kappa_2, v\ \texttt{mon}_j^{l,k}(\kappa_1, x))$
$\texttt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)$	$\cdot$	$\lambda x.\texttt{mon}_j^{k,l}(\kappa_2, v\ \texttt{mon}_j^{l,k}(\kappa_1, x))$

The three reductions use monitors and labels to enforce contracts and report contract violations:

1. A flat contract applies the given predicate to the given value using a predicate check. The check comes with two labels for contract error reporting in case the predicate fails. The party k receives blame for violating contract j.

2. A contract for a function v rewrites into a function that checks the domain contract for the parameter of this function and the range contract for the result of applying v to it.

3. A dependent contract for a function works like a functional contract, but it substitutes the argument for v also into the range contract.

As above, these notions of reduction generate the compatible closure with evaluation contexts, which in turn represents the standard reduction relation of the CPCF calculus. In addition, we need these three rules:

$$\begin{array}{lcl} E\left[\texttt{check}^k_j(\texttt{tt}, v)\right] & \longmapsto & E\left[v\right] \\ E\left[\texttt{check}^k_j(\texttt{ff}, v)\right] & \longmapsto & E\left[\texttt{error}^k_j\right] \\ E\left[\texttt{error}^k_j\right] & \longmapsto & \texttt{error}^k_j \end{array}$$

That is, a `check` reduces to its second argument if the predicate succeeds while it reduces to a contract error if the predicate fails. Finally, if an error shows up in the hole of an evaluation context, the program stops.

From here we get a semantics, i.e., a function mapping programs to answers:

$$\textbf{Answers} \quad a \;=\; v \mid \texttt{error}^k_j \,.$$

We say that a program converges ($e \Downarrow a$) iff $e \longmapsto_* a$; if the answer does not matter, we write ($e \Downarrow$). Conversely, a program diverges ($e \Uparrow$) iff for all e', there exists e'' such that when $e \longmapsto_* e'$, $e' \longmapsto e''$.

2.4 Why (Not) This Contract Error?

As the semantics of CPCF reveals, monitoring higher-order contracts requires a gradual disintegration a higher-order contract to obtain the predicate portions of the contract. In effect, this process delays contract checking until computation creates a witness value of base type that can validate or invalidate one of the flat pieces of a higher-order contract. Thus a contract predicate can fail many computational steps away from the application of a term to the contract to which the predicate belongs, making the relation between the contract, the predicate and the term obscure.

Consequently, it is not always obvious why the contract system signals a contract violation or why it blames a particular party. For instance the following exam-

ple, loosely adapted from Blume and McAllester [6],[8] shows that that higher-order contract monitoring may yield unexpected results:

SerM	κ	*CliM*
$\lambda f.\texttt{if zero?}(f\ 1)\ f\ f$	κ_0	$[\,]\ \lambda y.y\ 2$

where $\kappa_0 = (\texttt{flat}(O?) \mapsto \texttt{flat}(O?)) \mapsto (\mathit{Any} \mapsto \mathit{Any})$

and $O?$ checks if a number is odd and *Any* is $\texttt{flat}(\lambda x.\texttt{tt})$

In this example, the client context *CliM* hosts in its hole the server term *SerM*. The two components are brought together via a monitor expression featuring contract κ_0. Many programmers would expect that since the server does not explicitly abuse its argument and promises only that its result is a function, the server cannot be held responsible for much. But, as the client receives the result of the server and applies it to 2, the monitoring system recognizes a contract failure and raises an exception blaming the server. As far as the the monitoring system is concerned, it must ensure that the identity function that flows from the client to the server is always applied to odd numbers. Even though the server hands this function back to the client as part of the result, it remains responsible for the function's domain. In other words, the server lets its argument flow unprotected into the context and therefore takes the blame for failing to keep its promise to its client that it will always apply its argument to odd numbers.

Similarly, Xu and Peyton Jones [72, p. 2] describe a situation with two contract violations where the monitor discovers only one:

[8] The differences between the original example and the one given here are due to the statically typed nature of our language. More precisely Blume and McAllester [6] use an untyped *Any* predicate that accepts any value as the post-condition contract of the server instead of the $\mathit{Any} \mapsto \mathit{Any}$ function contract. We follow the analysis of Findler and Blume [31] who show that, in this setting, the most interesting interpretation of *Any* is $\mathit{Any} \mapsto \mathit{Any}$.

SerM	κ	*CliM*
$\lambda f.(f\ 1)-1$	$(Any \mapsto \texttt{flat}(PZ?)) \mapsto \texttt{flat}(PZ?)$	$[\,]\ \lambda x.x-1$

where $PZ?$ checks that a number is ≥ 0

First, they agree with the monitoring semantics in that this example should signal an error blaming the server for producing -1, even though its contract promises numbers that are greater or equal to 0. Second, they claim a violation on the client side, namely, that the function $\lambda x.x-1$ does not map arbitrary integers to positive ones or zero.

Here is a modification of the same example without the distracting server-side violation:

SerM	κ	*CliM*
$\lambda f.(f\ 1)+1$	$(Any \mapsto \texttt{flat}(PZ?)) \mapsto \texttt{flat}(PZ?)$	$[\,]\ \lambda x.x-1$

Xu and Peyton Jones [72] insist that the client violates the contract when it hands the server a function that returns the value -1 when given the value 0. They justify their claim with an explicit reference to the Sage contract checking approach [42, 38]. The CPCF monitoring semantics, however, reduces this program to 1 and does not signal a violation.

The above examples illustrate that the mere description of higher-order contract checking as part of the reduction semantics of a language with contracts, such as CPCF, is not sufficient to illuminate the workings of the contract system. The programming language designer, though, needs a robust argument for the behavior of the contract system in order to provide to the programmer a convincing explanation for blame assignment. Otherwise, the blame messages of the language may mislead the programmer who searches for a bug fix. For example, a programmer unaware of the meaning of contracts when encountering our first example might first be surprised by the contract error and then be unable to realize that the right fix is to strengthen the post-condition of κ_0. Thus, it is critical for the designer to

have available precise criteria that determine when blame assignment is correct and a method for checking these criteria for a specific contract systems.

2.4.1 The First Models for Contracts and Why They Fail

The programming languages community immediately recognized the need for a different model of contracts. This alternative model would separate the meaning of contracts from the contracts decomposition and the label bookkeeping of the reduction semantics of Findler and Felleisen.

Blume and McAllester [6] were the first to suggest such a model. They give the meaning of a contract κas the set of values that satisfy the contract, $[\![\kappa]\!]$. In more details,

1. if $\kappa = \texttt{flat}(e)$, $[\![\kappa]\!] = \{v \mid e\ v \Downarrow \texttt{tt} \text{ or } e\ v \not\Downarrow\}$,
2. if $\kappa = \kappa_1 \mapsto \kappa_2$, $[\![\kappa]\!] = \{v \mid \forall\ v_a \text{ such that } v_a \in [\![\kappa_1]\!],\ v\ v_a \in [\![\kappa_2]\!]\}$,
3. if $\kappa = \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$, $[\![\kappa]\!] = \{v \mid \forall\ v_a \text{ such that } v_a \in [\![\kappa_1]\!],\ v\ v_a \in [\![\{v/x\}\kappa_2]\!]\}$.

They show that their quotient definition of the meaning of contracts is sound and complete with respect to a so-called implied semantics of contracts [6, p. 3]:

> A server v satisfies a contract κ iff for all clients $C[\,]$ and labels server such that server does not occur in $C[\,]$, $C[\texttt{mon}^{\text{server,client}}_{\text{contract}}(\kappa, v)] \not\Downarrow \texttt{error}^{\text{server}}_{\text{contract}}$.

Their model helps explain the behavior of our first example; the identity function does not satisfy κ_0, $(\texttt{flat}(O?) \mapsto \texttt{flat}(O?)) \mapsto (\mathit{Any} \mapsto \mathit{Any})$. As a counter-example, consider an argument function

$$f = \texttt{mon}^{k,l}_{j}(\texttt{flat}(O?) \mapsto \texttt{flat}(O?), \lambda x.3).$$

This function clearly satisfies the pre-condition of κ_0, $\texttt{flat}(O?) \mapsto \texttt{flat}(O?)$. It does not, however, satisfy the post-condition $\mathit{Any} \mapsto \mathit{Any}$; even arguments violate

contract j. Thus the identity function fails to be a member of the meaning of κ_0 and the contract system correctly detects it and raises an error blaming the identity function.

Unfortunately, the quotient model is too stringent to predict the behavior of the contract system in the third example. In fact the model claims that the contract system should detect a contract violation. The reason for this discrepancy lies in the implied semantics of contracts. The definition sketches an intractable contract checking mechanism as it requires that no contract error is raised for any client $C[\,]$. In contrast, the actual contract system only considers a particular client, the given context of the server in the program that the language compiler or interpreter evaluates.

To reconcile these two contradicting views on contract checking, Findler et al [33] suggest an alternative model based on projections. Scott [60] defines projections p_k as functions on a Scott domain $\mathbf{P}_\omega$ that have two properties:

- $p_j^k = p_j^k \circ p_j^k$,
- $p_j^k \sqsubseteq \mathbf{1}$.

In this setting, they interpret each contract as a pair of error projections, a pair of projections where the second property has been replaced by $p_j^k \sqsubseteq \mathtt{error}_j^k$. This property implies that a contract does not change the behavior of a value except by possibly raising contract errors.

Intuitively the meaning of each contract is a pair of filter functions: the first discards any arguments that violate the responsibilities of the server and it raises a contract error blaming the server if it is provided with such a value, while the second does the same for the client's responsibilities. From that it is straightforward to extract the meaning of contract monitors: if p_j^k and p_j^l are the two error projections

that correspond to contract κ and $[\![v]\!]$ is the denotation of v in the domain $\mathbf{P}_\omega$, then the meaning of $\mathtt{mon}_j^{k,l}(\kappa, v)$ is the function composition $p_j^k \circ p_j^l \circ [\![v]\!]$.

This model successfully explains the behavior of all our examples. In addition, Findler and Blume [31] show that it can serve as the basis for an efficient implementation of Findler-Felleisen contract checking. Finally, the ordering properties of projections enable the construction of an ordering for contracts. The ordering of contracts sheds light on the meaning of the *Any* contract, the contract that all values satisfy. Findler and Blume argue that there are two possible interpretations for this contract: the top of the ordering or the bottom. In the first case, *Any* is the contract that always succeeds, while in the second it is the contract that always blames the client. The latter interpretation coincides with the one Blume and McAllester assign to *Any*. However Blume and Findler claim that the first interpretation is more useful for pragmatic reasons.

Despite its success, the model of Findler et al. comes with a serious shortcoming. Dependent function contracts are not projections and there is no syntactic restriction that can render them such. Informally, the post-condition of a dependent function contract has access to the argument of the function f that the contract protects. If the argument is also a function, call it s, then the post-condition can explore parts of s that f does not touch, as in the following example which uses functions to encode streams in CPCF:

SerM	κ	*CliM*
$\lambda s.\mathtt{fst}\ s$	κ_0	$[\,]\ \lambda i.i+1$

where $\kappa_0 = (\mathit{Any} \mapsto \mathit{Any}) \overset{d}{\mapsto} (\lambda s.\mathtt{flat}(\lambda r.PZ?\ (\mathtt{fst}\ s) \wedge PZ?\ (\mathtt{fst}\ (\mathtt{rst}\ s))))$

and $\mathtt{fst}$ is $\lambda s.s\ 0$ and $\mathtt{rst}$ is $\lambda s.\lambda i.s\ (i+1)$

The server of the example consumes a stream and looks at its first element, while the client provides to the server a stream where the value of each element is the position of the element plus 1. The contract between the two parties checks that the

stream contains positive numbers and that the result of the server is a stream where the first and the second elements are positive. A quick look at the example reveals that the contract explores more elements of the argument stream than the server. Thus, the composition of the contract and the server accesses a larger portion of the argument than the server alone. Hence, dependent contracts violate the second property of projections and change the behavior of the value they protect beyond raising contract errors.

2.4.2 A Different Semantics for Dependent Contracts

An important side-effect of the early quest for a model for higher-order contracts is that in order to cope with dependent contract in their proofs, Blume and McAllester suggest a new reduction semantics for dependent contracts:

$$E\left[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)\right] \\ \longmapsto E\left[\lambda x.\mathtt{mon}_j^{k,l}(\{\mathtt{mon}_j^{l,k}(\kappa_1, x)/x\}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))\right]$$

The new *picky* rule differs to the original *lax* rule in that the *picky* rule installs the argument in the post-condition with a monitor for the pre-condition. The server label for the latter monitor is the client label of the dependent function contract and client one the server label of the monitor for dependent function contract. The following variant of the example of the preceding sub-section illustrates the difference in contract checking when using the *picky* rule instead of the *lax* one:

SerM	κ	*CliM*
$\lambda s.\mathtt{fst}\ s$	κ_0	$[\,]\ \lambda i.i-1$

where $\kappa_0 = (\mathtt{flat}(PZ?) \mapsto \mathtt{flat}(PZ?)) \overset{d}{\mapsto} (\lambda s.\mathtt{flat}(\lambda r.PZ?\ (\mathtt{fst}\ (\mathtt{rst}\ s)+2)))$

The example consists of a server that consumes a stream and looks at its first element and a client that passes the server a stream where the value of each element is

the position of the element minus 1. The contract that regulates the communication of the two parties requires that the stream argument of the server maps positive numbers to positive numbers and that the result of the server is a stream where the second element incremented by 2 is positive.

Under the *lax* semantics, the example returns 1. The server observes only the first element of the stream that is positive and the contract system does not see any violation. The post-condition does visit the second element of the stream that is not positive. Since the stream is not protected by any contracts in the post-condition, no contract error is signaled. In contrast, under the *picky* semantics, the example produces a contract error blaming the server exactly because of the post-condition accessing the non-positive second element of the stream. The server is held responsible for this violation since the monitor that guards the stream in the post-condition has *SerM* as its client label.

Greenberg et al. [37], who also coined the terms *lax* and *picky*, prove that *picky* raises more contract errors than *lax*. Nevertheless, they do not show whether these errors correspond to actual contract violations or whether they blame the right party. For instance, it is unclear why *SerM* receives blame for the post-condition that breaks the contract of the stream in our example. The lack of a formal criterion for the correctness of blame prohibits to definitely accept or reject the contract error. In fact, in the absence of a provable argument against *lax*, one could argue that it is reasonable to consider that contracts are trusted code and contract monitoring is not necessary when evaluating contract code. Thus *lax* may be preferable to *picky* since it avoids the extra and possibly costly checks and does not assign blame in unclear corner cases like in our example. Until a formal description of the necessary and required properties of a contract system is available, decisions about the correctness of a contract system cannot be based anything more than intuition and speculation. The remaining chapters of this dissertation fill these gaps.

CHAPTER 3

Correct Blame[9]

3.1 Why Do You Blame Me?

The preceding chapter enlightens the steep contrast between contracts in the first-order world and contracts in the higher-order world. On one hand, programmers can easily specify read, write, and interpret first-order contracts. On the other hand, Findler and Felleisen [32]'s introduction of contracts for higher-order features, which is pragmatically necessary in the world of modern programming, raises puzzling, yet practically interesting questions. The efforts to create models for higher-order contracts not only failed to give a definitive answer to the question of the meaning of contracts but in addition gave birth to doubts about the correctness of the contract checking mechanism of Findler and Felleisen.

One particular issue concerns dependent higher-order function contracts. Such contracts come with two distinct semantics in the literature: *lax* and *picky*.

To make this discussion concrete, consider the dependent function contract in figure 3.1. This Racket fragment enhances the example from figure 2.2 in section 2.1 with a dependent post-condition. The post-condition adds that for some number `n` of numbers `x`, the slope of `f` at `x` is within δ of the value of `fp` at `x`. A

[9] This chapter reports work in collaboration with Findler, Flanagan and Felleisen [24].

```
;; for some natural number n and real δ
(->d ([f (-> 0<real<1? 0<real<1?)]
      [ε real?])
     (fp (-> 0<real<1? real?))
     #:post-cond
     (for/and ([i (in-range 0 n)])
       (define x (random-number))
       (define slope
         (/ (- (f (- x ε)) (f (+ x ε)))
            (* 2 ε)))
       (<= (abs (- slope (fp x))) δ)))
```

Figure 3.1: A higher-order, dependent contract

picky interpretation enforces that `f` and `fp` are applied to real numbers and produce such numbers during the evaluation of the post-condition; a *lax* interpretation does not check these specifications. Consequently, the *picky* system may raise a contract error that the *lax* system would not produce. However, it remains unclear if the additional contract errors are due to contract violations and if they blame the correct party. For instance consider the scenario where `random` returns a complex number, a possible result in Racket. In this case the **`#:post-cond`** code applies both functions to arguments that break their pre-conditions. While *lax* does not detect this violation, *picky* blames the `d/dx` module for the violation of `f`'s pre-condition and the client module for the violation of `fp`'s contract. A closer look shows that *picky* uses a consistent strategy to determine blame but ignores the provenance of the faulty value. In practice, this means that the programmer should look for a bug in one of the latter two modules, the one that receives blame, while the actual bug resides in **`#:post-cond`**. In contrast, the *lax* semantics avoids enforcing the contracts in **`#:post-cond`** and does not assign blame in this case, but the program may produce questionable results.

In this chapter, we develop a unified semantic framework that compares the two possible interpretations based on the reduction semantics for CPCF with dependent

contracts. We enrich the framework with the necessary information to track code ownership and contract obligations, two novel technical notions. This enriched framework is used to formalize the following correctness criterion for blame assignment: a contract system should blame a party only if *the party controls the flow or return of values into the particular contract check that fails.* We can prove that the *lax* interpretation satisfies this criterion while *picky* fails to live up to it.

In sum, our work takes a different view on assessing the correctness of higher-order contract checking. Instead of developing a model for contracts and comparing it with the results of contract checking, we develop an intuitive semantic criterion for an essential property of a contract system, namely correct blame assignment, and we use it to review and compare existing contract system.

3.2 CPCF with Ownership and Obligations

While the preceding section illuminates the problems of the *picky* and *lax* contract systems and the difficulty of comparing and deciding the correctness of contract systems in general, it also suggests a new way of thinking about contract violations. The first major insight is that the *picky* system may blame either the server module or the client module when, in fact, the contract itself is flawed. That is, the guilty module is the one that defines the contract, which may be the server, the client, or an independent third party. From here, it is obvious to inspect what a contract monitoring system would do if contracts were a part of the server or a part of the client. Doing so produces the second major insight, namely, that none of these alternatives agrees with the *picky* semantics. The *picky* system decides blame assignment based on a pre-determined notion of contractual obligations for each party but ignores the origin of the value that breaks the contract.

Putting the two insights together suggests that we need an instrumented seman-

tics that (1) for each party, tracks its contract obligations and (2) for each value accounts for its origin. Once a semantics provides this additional information, we can check whether a contract system ever blames a party for violating an obligation even if the party has no control over the value's flow into the contract. In this section, we equip CPCF's semantics with ownership and obligation information, which is maintained across reductions. This way we build, on top of the reduction semantics of CPCF, a system to reason about the responsibilities of the contract parties and the migration and provenance of values that is independent to the label bookkeeping of the monitoring reduction. In the next section, we use this information to state a contract correctness property and to measure how the various monitoring systems fare with respect to this property.

3.2.1 Ownership ...

To model an ownership relationship between parties and code, we extend CPCF with a new construct that relates terms and values to parties:

$$\begin{array}{lrcl} \textbf{Terms} & e & = & \ldots \mid \|e\|^{l} \\ \textbf{Values} & v & = & \ldots \mid \|v\|^{l} \end{array}$$

During reductions, terms and values come with a stack of owners, reflecting transfers from one party to another. The notations $\|e\|^{\overrightarrow{l_n}}$ and $\|e\|^{\overleftarrow{l_n}}$ are short-hands for such stacks, abbreviating $\|\ldots\|e\|^{l_1}\ldots\|^{l_n}$ and $\|\ldots\|e\|^{l_n}\ldots\|^{l_1}$, respectively. Ownership l for an expression means its result is attributed to l. In turn, a value with an ownership l originates from component l or is affected by a traversal through component l. In this setting, $\|\ldots\|v\|^{l_1}\ldots\|^{l_n}$ denotes that v was originally owned by l_1 but then ownership of v was transferred to l_2, etc, and eventually to l_n.

3.2.2 . . . and Obligations

CPCF contracts consist of trees with flat contracts at the leaves. Exploiting the analogy with function types, Findler and Felleisen [32] implicitly decompose these trees into obligations for servers (positive positions) and clients (negative positions). Their semantics tracks this connection via labels; errors use labels to pinpoint contract violators.

Dimoulas and Felleisen [22] use this idea for a static decomposition of contracts into server and client obligations. They define two functions from contracts to contracts that tease out the respective obligations. The one for teasing out server obligations replaces flat contracts in negative positions with $\top = \texttt{flat}(\lambda x.\texttt{tt})$ and then reconstructs the overall contract; analogously, the decomposition map for teasing out client obligations replaces flat contracts in positive positions with $\top$. For instance, consider the following example:

$$\begin{aligned}\Pi^0 \quad &= \quad \texttt{mon}_*^{k,l_o}(\kappa, \lambda f.f\ \lambda x.x)\ \lambda g.g\ 42\\ &\text{where } \kappa \ = ((\texttt{P?} \mapsto \texttt{P?}) \overset{d}{\mapsto} (\lambda f.\texttt{flat}(\lambda x.f\ 0 > -1))) \mapsto \texttt{P?}\\ &\qquad\ \texttt{P?} = \texttt{flat}(\lambda x.x > 0)\end{aligned}$$

The example yields these decompositions:

	$((\texttt{P?} \mapsto \texttt{P?}) \overset{d}{\mapsto} (\lambda f.\texttt{flat}(\lambda x.f\ 0 > -1))) \mapsto \texttt{P?}$
server	$((\top \mapsto \texttt{P?}) \overset{d}{\mapsto} (\lambda f.\top)) \mapsto \texttt{P?}$
client	$((\texttt{P?} \mapsto \top) \overset{d}{\mapsto} (\lambda f.\texttt{flat}(\lambda x.f\ 0 > -1))) \mapsto \top$

Decomposition implies that each flat contract imposes obligations on a specific component, i.e., party to a contract. We modify the syntax of contracts to statically associate flat contracts and owners:

$$\textbf{Contracts} \quad \kappa \ ::= \lfloor \texttt{flat}(\|e\|^l) \rfloor^{\bar{l}} \mid \kappa \mapsto \kappa \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa)$$

In contrast with ownership, obligations come as sets of labels $\bar{l}$, not vectors. After all, there is no need to order obligations or to change them during an evaluation. Of course, a static attribute about a dynamic obligation calls for a way to determine whether such annotations are well-formed.

3.2.3 Well-formed Ownership and Obligations

Only some annotations make sense for a source program. Both ownership and contract monitors specify boundaries and, at the source level, these boundaries should coincide. We therefore introduce a well-formedness judgment to enforce these conditions for source programs. Before doing so, we present the simple typing rules for the two new constructs:

$$\frac{\mathcal{G} \vdash e : o \to \mathcal{B}}{\mathcal{G} \vdash \lfloor \texttt{flat}(e) \rfloor^{\bar{l}} : \mathit{con}(o)} \qquad \frac{\mathcal{G} \vdash e : \tau}{\mathcal{G} \vdash \|e\|^{l} : \tau}$$

Concerning ownership annotations, a CPCF source program may contain those at only two places: in contract monitors and in flat contracts. Since contract monitors establish a boundary between the client component and the server component, we demand an ownership annotation on the server component and that a match of these annotation with the positive label of the monitor. Conversely, the context of such an expression must belong to the client. Finally, flat contracts must come with ownership labels consistent with the surrounding monitor because they are turned into plain code during the evaluation, and the semantics must track where they originated from.

We express this constraint with the well-formedness relation $\Gamma; l \Vdash e$, which says that l "owns" e. Equivalently, l is the owner for the context of e. We equip the well-formedness relation with an environment that records the label of bound variables. With the environment, it becomes possible to check that free variables in

$\boxed{\Gamma; l \Vdash e}$

$$\frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1\ e_2} \qquad \frac{\Gamma \uplus \{x : l\}; l \Vdash e}{\Gamma; l \Vdash \lambda x.e} \qquad \frac{\Gamma \setminus \{x\}; e \Vdash l}{\Gamma; l \Vdash \mu x.e}$$

$$\frac{\Gamma; l \Vdash e_1}{\Gamma; l \Vdash \mathtt{zero?}(e_1)} \qquad \frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2 \qquad \Gamma; l \Vdash e_3}{\Gamma; l \Vdash \mathtt{if}\ e_1\ e_2\ e_3}$$

$$\frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 + e_2} \qquad \frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 - e_2}$$

$$\frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \wedge e_2} \qquad \frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \vee e_2}$$

$$\frac{}{\Gamma; l \Vdash \mathtt{n}} \qquad \frac{}{\Gamma; l \Vdash \mathtt{tt}} \qquad \frac{}{\Gamma; l \Vdash \mathtt{ff}} \qquad \frac{\Gamma(x) = l}{\Gamma; l \Vdash x}$$

$$\frac{\Gamma; k \Vdash e \qquad \Gamma; \{k\}; \{l\}; j \rhd \kappa \qquad k \neq l}{\Gamma; l \Vdash \mathtt{mon}^{k,l}_{j}(\kappa, \|e\|^{k})}$$

Figure 3.2: Ownership coincides with contract monitors

$\boxed{\overline{k}; \overline{l}; j; \Gamma \rhd \kappa}$

$$\frac{\Gamma; \overline{l}; \overline{k}; j \rhd \kappa_1 \qquad \Gamma; \overline{k}; \overline{l}; j \rhd \kappa_2}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \mapsto \kappa_2}$$

$$\frac{\Gamma; \overline{l}; \overline{k} \cup \{j\}; j \rhd \kappa_1 \qquad \Gamma; \overline{k}; \overline{l}; j \rhd \kappa_2}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)}$$

$$\frac{\Gamma; j \Vdash e}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{flat}(\|e\|^{j}) \rfloor^{\overline{k}}}$$

Figure 3.3: Obligations coincide with labels on monitors

terms have the same owner as their context. A closed expression e is a well-formed program if $\varnothing; l_o \Vdash e$ where l_o is the label reserved for the owner of the program.

Figure 3.2 defines the well-formedness judgment. For terms that do not involve monitors and contracts, the definition proceeds with structural recursion, and base values and variables are well-formed under the owner of the construct that binds them. The actual key rule is the one for contract monitor, which we present separately. According to our informal description, a well-formed contract monitor is a boundary between a client and a server, implying this shape for the judgment:

$$\Gamma; l \Vdash \mathtt{mon}_j^{k,l}(\kappa, \|e\|^k)$$

It says that if l owns the context and k is the blame label for the server, then the blame label for the client should be l *and* the wrapped expression e should come with an ownership annotation that connects it to k. Next, e must be well-formed with respect to its owner k, because it may contain additional contract monitors. But even with this antecedent, the well-formedness judgment is incomplete. After all, the contract κ that governs the flow of values between the server and the client contains code and this code must be inspected. Furthermore, we must ensure that all flat contracts within κ are obligations of the appropriate parties including the contract monitor itself, which is represented by the contract label j. We assign to the contract a separate third label so that we can identify it either with the server or the client or a third party without loss of generality. Finally, we require that k and l are distinct so that the monitor corresponds to a proper component boundary.

Putting everything together, we get this well-formedness rule for contract monitors in source programs:

$$\frac{\Gamma; k \Vdash e \qquad \Gamma; \{k\}; \{l\}; j \rhd \kappa}{\Gamma; l \Vdash \mathtt{mon}_j^{k,l}(\kappa, \|e\|^k)}$$

It relies on a secondary well-formedness judgment for contracts, to which we turn next.

Roughly speaking, $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$ says that contract κ is well-formed for sets of positive and negative obligation labels $\overline{k}$ and $\overline{l}$, respectively, and the owner j of the contract monitor that attaches the contract to a boundary. As the definition in figure 3.3 shows, the two sets are swapped for the antecedents of higher-order (dependent) contracts. For the negative positions in the precondition of dependent contract label j is added to indicate that these are also obligations of the owner of the contract monitor. For flat contracts the positive obligation labels must coincide with the obligation labels of the contract.

Note that the well-formedness of flat contracts also enforces an ownership annotation. Specifically, the owner of the context—which, by assumption, is a contract monitoring construct labeled with j—is also the owner of the predicate in the flat contract. The antecedent of the rule recursively uses the well-formedness judgments for ownership to ensure that e itself is well-formed.

3.2.4 Ownership and Obligations Semantics

The final change to the CPCF model concerns the reduction semantics. Specifically, we change the reduction relations so that each reduction step keeps track of ownership rights and obligations. While ownership and obligations do not affect the semantics per se, the information is critical for characterizing the behavior of contract monitoring systems, as we show in the next section.

Our first step is to equip the grammar of evaluation contexts with a parameter that accounts for the owner of the hole. In the parameterized grammar, E^l, of figure 3.4 the parameter l points to the ownership or contract annotation that is closest to the hole of the context. Both contract monitors and ownership annotations indicate component boundaries and thus we consider them positions where ownership changes. Predicate checks are also component boundaries. Their first argument embeds in a component a predicate that is part of a contract. Since the contract

may belong to a different component, checks also indicate ownership transition and we treat them like monitors and ownership annotations.

$$
\begin{array}{lrcl}
\textbf{E. Contexts} & E^{l} & = & F^{l} \\
& E^{l_o} & = & F \\
& F^{l} & = & F^{l}\,e \mid v\,F^{l} \mid F^{l}+e \mid v+F^{l} \\
& & \mid & F^{l}-e \mid v-F^{l} \mid F^{l}\wedge e \mid v\wedge F^{l} \\
& & \mid & F^{l}\vee e \mid v\vee F^{l} \mid \texttt{zero?}(F^{l}) \\
& & \mid & \texttt{if}\,F^{l}\;e\;e \mid \texttt{mon}_{j}^{l,k}(\kappa,F) \\
& & \mid & \texttt{mon}_{j}^{l',k}(\kappa,F^{l}) \mid \|F\|^{l} \mid \|F^{l}\|^{l'} \\
& & \mid & \texttt{check}_{j}^{l}(F,e) \mid \texttt{check}_{j}^{l}(F^{l'},e) \\
& F & = & [\,] \mid F\,e \mid v\,F \mid F+e \mid v+F \\
& & \mid & F-e \mid v-F \mid F\wedge e \mid v\wedge F \\
& & \mid & F\vee e \mid v\vee F \mid \texttt{zero?}(F) \\
& & \mid & \texttt{if}\,F\;e\;e
\end{array}
$$

Figure 3.4: Parameterized evaluation contexts

Evaluation contexts are labeled with the label l_o—the label reserved for the whole program—if they do not contain an ownership or contract constructs on the path from the hole to the root: E^{l_o}.

From now on, all reduction relations assume labeled evaluation contexts. This implies that newly created values are always assigned an owner. For the reduction relations concerning primitive operators, conditionals and predicate checks, the changes are straightforward and summarized in the top part of figure 3.5. For the rules concerning monitors with flat and plain higher-order contracts and their blame assignments, specified in the lower part of the same figure, we also know that they do not need to manipulate any ownership annotations. These reduction rules remain unchanged, modulo the labeled evaluation contexts. The obligation annotation on flat contracts is ignored. For details, see the bottom part of figure 3.5. We add one last simple rule separately:

$$E^{l}[\texttt{error}_{j}^{k}] \longmapsto_{m} \texttt{error}_{j}^{k}$$

Since the act of signaling errors erases the surrounding evaluation context, the format of this rule doesn't fit the table. Note that the context on the right is $[\ ]^{l_o}$ and l may not equal l_o.

$E^l[\cdots]$	$\longmapsto_m$	$E^l[\cdots]$	
$\|\mathtt{n}_1\|^{\vec{k}} + \|\mathtt{n}_2\|^{\vec{l}}$	$\cdot$	$\mathtt{n}$	where $n_1 + n_2 = n$
$\|\mathtt{n}_1\|^{\vec{k}} - \|\mathtt{n}_2\|^{\vec{l}}$	$\cdot$	$\mathtt{n}$	where $n_1 - n_2 = n$
$\mathtt{zero?}(\|\mathtt{0}\|^{\vec{l}})$	$\cdot$	$\mathtt{tt}$	
$\mathtt{zero?}(\|\mathtt{n}\|^{\vec{k}})$	$\cdot$	$\mathtt{ff}$	if $n \neq 0$
$\|v_1\|^{\vec{k}} \wedge \|v_2\|^{\vec{l}}$	$\cdot$	v	where $v_1 \wedge v_2 = v$
$\|v_1\|^{\vec{k}} \vee \|v_2\|^{\vec{l}}$	$\cdot$	v	where $v_1 \vee v_2 = v$
$\mathtt{if}\ \|\mathtt{tt}\|^{\vec{l}}\, e_1 e_2$	$\cdot$	e_1	
$\mathtt{if}\ \|\mathtt{ff}\|^{\vec{l}}\, e_1 e_2$	$\cdot$	e_2	
$\mathtt{check}^k_j(\mathtt{tt}, v)$	$\cdot$	v	
$\mathtt{check}^k_j(\mathtt{ff}, v)$	$\cdot$	$\mathtt{error}^k_j$	
$\mathtt{mon}^{k,l}_j(\kappa_1 \mapsto \kappa_2, v)$	$\cdot$	$\lambda x.\mathtt{mon}^{k,l}_j(\kappa_2, v\, \mathtt{mon}^{l,k}_j(\kappa_1, x))$	
$\mathtt{mon}^{k,l}_j(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l'}}, v)$	.	$\mathtt{check}^k_j((e\ v), v)$	

Figure 3.5: Ownership and obligation propagation

A β_v reduction typically demands several re-alignments with respect to ownership. To start with, the function and the argument may belong to different parties. Furthermore, the context brings together the operand and operator, and the semantics should keep track of this responsibility. Together, the two observations suggest the following relation:

$$E^l[\|\lambda x.e\|^{\vec{l_n}}\ v] \quad \longmapsto_m \quad E^l[\|\{\|v\|^{l\overleftarrow{l_n}}/x\}e\|^{\vec{l_n}}]$$

The relation says that after tagging the value with the ownership label l of the context, the value moves under the ownership annotations of the function. The result is a value whose innermost owner is l and whose outermost owner is l_1 of $\vec{l_n}$:

$$\|v\|^{l\overleftarrow{l_n}}$$

The properly annotated value is then substituted into the body e of the function for its parameter x. The result itself is owned by the same owner as the function.

Put differently, it is best to view function application as a form of communication between two components: the function and its context. The context picks the argument, declares itself its owner, and then passes it to the function. The function accepts the argument, adjusts its ownership, and integrates the result into its body.

Recursion is treated as a special form of function application:

$$E^l[\mu x.e] \quad \longmapsto_m \quad E^l[\{\|\mu x.e\|^l/x\}e]$$

The owner of the context l and user of the recursive function declares itself owner of $\mu x.e$ before substituting it in the body of the recursive function.

All the complexity of tracking ownership is due to dependent function contracts. Consider the simplest variant, *lax*:

$$E\,[\mathtt{mon}^{k,l}_j(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \quad \longmapsto_l \qquad (lax)$$
$$E\,[\lambda y.\mathtt{mon}^{k,l}_j(\underline{\{y/x\}\kappa_2}, v\ \mathtt{mon}^{l,k}_j(\kappa_1, y))]$$

For emphasis, this version of the reduction rule uses y as the parameter of the proxy function on the right hand side. The use of y as parameter demands that we also replace all occurrences of x in κ_2 with y so that when the proxy function is applied, the actual argument is substituted into the dependent range contract; without the substitution, the reduction would create free variables.[10]

Rewriting the *lax* rule in this way reveals that it encodes a masked function application. The problem is that, as discussed above, a function application must add the label of the responsible owner at the bottom of the stack, and this label is not available here. Instead, it is found at the flat leafs of the contract, which—according to the static semantics of the preceding subsection—must come with an ownership

[10]The value v is unaffected by this change of parameters, because we assume the usual hygiene condition [3] for meta-variables.

annotation. The solution is to introduce the substitution function $\{e/^c x\}\kappa$, which copies the ownership label from flat contracts to the substituted term.

With this substitution function in place, it is easy to specify the two variants for the reduction of dependent functional contracts:

$$\frac{E^l[\cdots] \quad \ldots \quad E^l[\cdots]}{\begin{array}{ll} \mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) & \longmapsto_l \\ \qquad \lambda x.\mathtt{mon}_j^{k,l}(\{x/^c x\}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x)) & (\mathit{lax}) \\ \mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v) & \longmapsto_p \\ \quad \lambda x.\mathtt{mon}_j^{k,l}(\underline{\{\mathtt{mon}_j^{l,k}(\kappa_1, x)/^c x\}\kappa_2}, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x)) & (\mathit{picky}) \end{array}}$$

We conclude this section with the definition of the auxiliary substitution function:

$$\begin{array}{rcl} \{e/^c x\}\lfloor \mathtt{flat}(\|e'\|^{l'})\rfloor^{\bar{l}} & = & \lfloor \mathtt{flat}(\{\|e\|^{l'}/x\}\|e'\|^{l'})\rfloor^{\bar{l}} \\ \{e/^c x\}(\kappa_1 \mapsto \kappa_2) & = & \{e/^c x\}\kappa_1 \mapsto \{e/^c x\}\kappa_2 \\ \{e/^c x\}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)) & = & \{e/^c x\}\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2) \\ \{e/^c x\}(\kappa_1 \overset{d}{\mapsto} (\lambda y.\kappa_2)) & = & \{e/^c x\}\kappa_1 \overset{d}{\mapsto} (\lambda y.\{e/^c x\}\kappa_2) \\ & & \text{where } x \neq y \end{array}$$

The definition is total. The redefinition of the contract syntax enforces that flat contracts always have the annotations expected by the domain of the substitution function. This also guarantees that the reduction relations $\longmapsto_m$ are well-defined.

3.3 Defining and Proving Correct Blame

Using ownership and obligation annotations, we can formulate what it means for a contract system to correctly blame a violator. After all, the tracking of ownership and obligations is entirely independent of the contract checking, and it is thus appropriate to use tracking as an independent specification of contract monitoring.

Informally, the contract system should blame a party only if it contributes a value that breaks one of the party's obligations. Put differently, values originating

from one of the parties of the contract should be checked only against flat pieces of the contract for which the party is responsible. Now we can phrase this property in terms of ownership and obligations: when the evaluation reaches a redex that checks a flat contract on a value, then the owner of the value must be the same as the positive party of the monitor and in addition the positive party is included in the obligations of the contract.

Definition 3.3.1 (Blame Correctness)**.** *A contract system m is* blame correct *iff for all terms e_0 such that $\varnothing; l_o \Vdash e_0$, if*

$$e_0 \longmapsto_m^* E^\dagger[\mathtt{mon}_\dagger^{k,\dagger}(\lfloor \mathtt{flat}(e_1) \rfloor^{\bar{l}}, v)],$$

then v is of the form $\|v_1\|^k$ and $k \in \bar{l}$. The identity of the † labels is irrelevant.

The definition says that when the reduction of a well-formed program reaches a state in which it checks a flat contract, then the server (positive) label of the monitor and the ownership label on the value must coincide and, furthermore, the set of obligations for the flat contract must contain this label. Conversely, if these obligations are not met, the monitor may blame *k* for a contract failure even though the party had no control over the flow of *v* into this monitor.

We can prove that the *lax* system is blame correct, while *picky* isn't. The proof of the positive theorem directly follows from a subject reduction theorem for ownership annotations. This latter theorem requires a complex proof, which is the subject of the second subsection. The third subsection explains how to prove the two main theorems. To start with, however, we clarify that ownership annotations and obligations are orthogonal to the underlying semantics.

3.3.1 It is all about Information Propagation

The addition of ownership and obligation annotations does not affect the behavior of any programs. Our revised semantics simply propagates this information

so that it can be used to characterize execution states. In order to formulate this statement, we use the symbol $\overset{cpcf}{\longmapsto}{}^*_m$ for the transitive-reflexive reduction relations of section 2.3 and $\overset{anno}{\longmapsto}{}^*_m$ for the relations of section 3.2.

Proposition 3.3.2. *The following statements hold for $m \in \{l, p\}$.*

1. *Let e be a well-formed CPCF program: $\Gamma; l_o \Vdash e$. Let $\overline{e}$ be the plain CPCF expression that is like e without annotations. If $e \overset{anno}{\longmapsto}{}^*_m e'$, then $\overline{e} \overset{cpcf}{\longmapsto}{}^*_m \overline{e'}$.*

2. *Let $\overline{e}$ be a plain CPCF program. There exists some labeled CPCF program e such that $\Gamma; l_o \Vdash e$. Furthermore, if $\overline{e} \overset{cpcf}{\longmapsto}{}^*_m \overline{e'}$, then $e \overset{anno}{\longmapsto}{}^*_m e'$.*

Proof. By a straightforward bi-simulation argument for the two relations. ∎

3.3.2 Subject Reduction

While $\Gamma; l \Vdash e$ specifies when source programs are well-formed, the reduction semantics creates many expressions that do not satisfy these narrow constraints. For example, a well-formed program contains only monitor terms of the restricted form $\texttt{mon}^{k,l}_j(\kappa, \|e_0\|^k)$. A reduction sequence may contain programs with differently shaped monitors, however. In particular due to the reductions of higher-order contracts, the monitored expressions may be applications, $\texttt{mon}^{k,l}_j(\kappa, \|e_0\|^k\, e_r)$, or variables, $\texttt{mon}^{k,l}_j(\kappa, x)$. Fortunately, such deviations are only temporary. In the case of the application the argument e_r is always well-formed under l and, when it is absorbed by $\|e_0\|^k$, the monitor expression is once again well-formed. In the case of the free variable, we can show that x is always replaced with a value of the form $\|v_0\|^k$, which conforms to the standard form.

In addition, reductions may propagate ownership annotations in positions other than in monitors. This calls for a set of rules that handle ownership annotations in arbitrary positions. Finally, the set of intermediate terms is larger than the one for source terms. Hence, extra rules for errors and predicate checks are required.

To formulate a subject reduction theorem, we must generalize both the judgment for well-formed programs, $\Gamma; l \Vdash e$, and the one for well-formed contracts, $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$.

$\boxed{\Gamma; l \Vdash e}$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1\ e_2} \qquad \frac{\Gamma; l \Vdash e_1}{\Gamma; l \Vdash \mathtt{zero?}(e_1)} \qquad \frac{}{\Gamma; l \Vdash \mathtt{error}^k}$$

$$\frac{\Gamma; j \Vdash e \quad \Gamma; l \Vdash v}{\Gamma; l \Vdash \mathtt{check}^k_j(e, v)} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2 \quad \Gamma; l \Vdash e_3}{\Gamma; l \Vdash \mathtt{if}\ e_1\ e_2\ e_3}$$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 + e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 - e_2}$$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \wedge e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \vee e_2}$$

$$\frac{}{\Gamma; l \Vdash \mathtt{n}} \qquad \frac{}{\Gamma; l \Vdash \mathtt{tt}} \qquad \frac{}{\Gamma; l \Vdash \mathtt{ff}} \qquad \frac{\Gamma(x) = l}{\Gamma; l \Vdash x}$$

$$\frac{\Gamma; k \Vdash e}{\Gamma; l \Vdash \|e\|^k} \qquad \frac{\Gamma; k \vDash e \quad \Gamma; \{k\}; \{l\}; j \rhd \kappa}{\Gamma; l \Vdash \mathtt{mon}^{k,l}_j(\kappa, e)}$$

$\boxed{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa}$

$$\frac{\Gamma; \overline{l}; \overline{k}; j \rhd \kappa_1 \quad \Gamma; \overline{k}; \overline{l}; j \rhd \kappa_2}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \mapsto \kappa_2}$$

$$\frac{\Gamma; \overline{l}; \overline{k} \cup \{j\}; j \rhd \kappa_1 \quad \Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa_2}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)}$$

$$\frac{\Gamma; j \Vdash e \quad \overline{k} \subseteq \overline{k'}}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{flat}(\|e\|^j) \rfloor^{\overline{k'}}}$$

Figure 3.6: Obligations coincide with labels on monitors (2)

The first step is to add a rule for checking expressions that already have an owner:

$$\frac{\Gamma; k \Vdash e}{\Gamma; l \Vdash \|e\|^k}$$

As mentioned, while these terms show up only within monitors in source programs, they flow into many positions during evaluations. Using this new rule, we can check these cases, too. Intuitively, this rule says that a "foreign" term can show up in any "host" context as long as it has an explicit ownership annotation that marks its owner.

Second, we add rules for errors and predicate checks. The rule for errors is straightforward as errors are well-formed under any owner. The rule for predicate checks, though, is complex. It requires that the owner of the first argument of the construct, e_1, is the contract label of the check construct, j, while the owner of the second argument, e_2, is the owner of the context l:

$$\frac{\Gamma; j \Vdash e \qquad \Gamma; l \Vdash v}{\Gamma; l \Vdash \mathtt{check}^k_j(e, v)}$$

All other rules—except those concerning monitors— remain the same. For details, see figure 3.6.

The rule for monitors requires a small change to check monitored intermediate terms. Here is the key rule:

$$\frac{\Gamma; k \vDash e \qquad \Gamma; \{k\}; \{l\}; j \rhd \kappa}{\Gamma; l \Vdash \mathtt{mon}^{k,l}_j(\kappa, e)}$$

To check whether the wrapped expression is well-formed, it delegates to the auxiliary relation $\Gamma; k \vDash e$. The contract is checked as before, though, with an environment.

With $\Gamma; k \vDash e$, we can check the ordinary ownership terms but also applications and variables as introduced during reductions. For function applications, the label

serves as ownership label for the operator and the operand, similar to the standard application rule:

$$\frac{\Gamma; k \Vdash e_1 \qquad \Gamma; k \Vdash e_2}{\Gamma; k \vDash \|e_1\|^k \; e_2}$$

For free variables, the environment serves as the source of the ownership label:

$$\frac{\Gamma(x) = k}{\Gamma; k \vDash x}$$

After all, the variable in this position is going to be replaced by a value via a function application, and the substitution is going to use a value with the specified label. Finally, for a guarded term with an ownership annotation, it suffices to check if it is well-formed with respect to the specified owner. The rule allows for expressions with the appropriate ownership annotation to show up monitors and thus subsumes the condition for well-formed monitors in source code:

$$\frac{\Gamma; k \Vdash e}{\Gamma; k \vDash \|e\|^k}$$

Finally, the rules for well-formed contracts do not change except the one for flat contracts:

$$\frac{\Gamma; j \Vdash e \qquad \overline{k} \subseteq \overline{k'}}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \texttt{flat}(\|e\|^j) \rfloor^{\overline{k'}}}$$

In particular, the new rule relaxes the condition on the obligation labels $\overline{k'}$ of the flat contract. The $\overline{k'}$ set is a static conservative approximation of dynamically computed obligation annotations $\overline{k}$. In the original rule the two sets coincide but as computation proceeds and the dynamic information becomes more accurate, the statically predicted responsible parties $\overline{k}$ can be a superset of the dynamically predicted ones, $\overline{k'}$.

Note here that a well-formed program in the sense of the preceding section is a well-formed program in the sense of revised judgment, too. After all, the revised judgment is an extension of the initial one.

Now we are ready to prove our central lemma. A program that is well-formed according to $\Gamma; l \Vdash e$ reduces to well-formed programs. This statement holds for a contract system using the *lax* reduction.

Lemma 3.3.3. *Let e be a program such that* $\varnothing; l_o \Vdash e$. *If* $e \longmapsto_l e_0$, *then* $\varnothing; l_o \Vdash e_0$;

Proof. We proceed by case analysis on the reduction of e:

- $E^l[\|\mathtt{n}_1\|^{\vec{k}} + \|\mathtt{n}_2\|^{\vec{l}}] \longmapsto_l E^l[\mathtt{n}]$. By assumption $\varnothing; l_o \Vdash e$. Consequently from lemma 3.3.4, we get $\varnothing; l \Vdash \|\mathtt{n}_1\|^{\vec{k}} + \|\mathtt{n}_2\|^{\vec{l}}$. We can use the same label to check n via the inference rules, i.e., $\varnothing; l \Vdash \mathtt{n}$. Hence, from lemma 3.3.6, $\varnothing; l_o \Vdash E^l[\mathtt{n}]$.

- The cases for other primitive operations, conditionals, errors and predicate checks are similar to the first.

- $E^l[\mu x.e] \longmapsto_l E^l[\{\|\mu x.e\|^l/x\}e]$: By assumption and lemma 3.3.4, $\varnothing; l \Vdash \mu x.e$. From lemma 3.3.8 we get that $\varnothing; l \Vdash \{\|\mu x.e\|^l/x\}e$ and, in turn, we obtain $\varnothing; l_o \Vdash E^l[\{\|\mu x.e\|^l/x\}e]$.

- $E^l[\|\lambda x.e_0\|^{\vec{k}}\ v] \longmapsto_l E^l[\|\{\|v\|^{l^{\overleftarrow{k}}}/x\}e_0\|^{\vec{k}}]$: Again by using the assumption and lemma 3.3.4, we conclude that $\varnothing; l \Vdash \|\lambda x.e_0\|^{\vec{k}}\ v$ and, therefore, $\varnothing; l \Vdash \|\lambda x.e_0\|^{\vec{k}}$ and $\varnothing; l \Vdash v$.

  Next we distinguish two cases, depending on the length of $\vec{k}$. First assume the vector is empty. In that case, the inference rules imply $\{x : l\}; l \Vdash e_0$. Combining this judgment with $\varnothing; l \Vdash v$, we may conclude that $\varnothing; l \Vdash \{\|v\|^l/x\}e_0$ via lemma 3.3.9. Finally from here it is easy to get $\varnothing; l_o \Vdash E^l[\{\|v\|^l/x\}e_0]$, the desired conclusion from lemma 3.3.6.

  Second, let k_1 be the first element of $\vec{k}$. In that case, the inference rules imply $\{x : k_1\}; k_1 \Vdash e_0$. Since $\varnothing; l \Vdash v$ still holds, we conclude again via lemma 3.3.9

that $\varnothing; k_1 \Vdash \{\|v\|^{l^{\overleftarrow{k}}}/x\}e_0$. Since k_1 is the outermost element of $\vec{k}$, if $k_1 = l$, we use lemma 3.3.6 and we finally get the desired conclusion, $\varnothing; l_o \Vdash E^l[\|\{\|v\|^{l^{\overleftarrow{k}}}/x\}e_0\|^{\vec{k}}]$. Else if $k_1 \neq l$, we employ lemma 3.3.6 to achieve the same result.

- $E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{flat}(e_c)\rfloor^{\overline{l'}}, v)] \longmapsto_l E^l[\mathtt{check}^k_j((e_c\ v), v)]$: The assumptions imply $\varnothing; l \Vdash \mathtt{mon}^{k,l}_j(\lfloor \mathtt{flat}(e_c)\rfloor^{\overline{l'}}, v)$ via lemma 3.3.4 and hence, $e_c = \|e'_c\|^j$ with $\varnothing; j \Vdash e'_c$. The latter implies $\varnothing; j \Vdash e_c$. Furthermore, the same reasoning yields $v = \|v_0\|^k$ and $\varnothing; k \Vdash v_0$. Thus $\varnothing; j \Vdash v$. Since the rules for well-formed expressions imply $\varnothing; l \Vdash \mathtt{check}^l_j((e_c\ v), v)$ is well-formed, the desired conclusion follows immediately.

- $E^l[\mathtt{mon}^{k,l}_j(\kappa_1 \mapsto \kappa_2, v)] \longmapsto_l E^l[\lambda x.\mathtt{mon}^{k,l}_j(\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))]$: With the usual reasoning, we get $\varnothing; l \Vdash \mathtt{mon}^{k,l}_j(\kappa_1 \mapsto \kappa_2, v)$, $v = \|v_0\|^k$, and $\varnothing; k \Vdash v_0$. The contract check yields two pieces of knowledge:

$$\varnothing; \{l\}; \{k\}; j \rhd \kappa_1$$

and

$$\varnothing; \{k\}; \{l\}; j \rhd \kappa_2$$

From an additional application of the inference rules for well-formedness we get $\{x : l\}; k \vDash v\ \mathtt{mon}^{l,k}_j(\kappa_1, x)$ and, with the help of lemma 3.3.10, $\{x : l\}; l \Vdash \mathtt{mon}^{k,l}_j(\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))$. Finally from a last application of the inference rules for well-formedness we get $\varnothing; l \Vdash \lambda x.\mathtt{mon}^{k,l}_j(\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))$.

- Finally, let $e = E^l[\mathtt{mon}^{k,l}_j(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)]$ and observe that

$$e \longmapsto_l E^l[\lambda x.\mathtt{mon}^{k,l}_j(\{x/^c x\}\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))].$$

Via lemma 3.3.4 we derive $\varnothing; l \Vdash \mathtt{mon}^{k,l}_j(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)$. Thus, $v = \|v\|^k$ with $\varnothing; k \Vdash v$, but also $\varnothing; \{l\}; \{k, j\}; j \rhd \kappa_1$ and $\varnothing; \{k\}; \{l\}; j \rhd \kappa_2$. The

rest of this argument uses the same strategy as the preceding case, except that we use lemmas 3.3.11 and 3.3.13 to derive the key result, $\{x : l\}; l \Vdash \mathtt{mon}_j^{k,l}(\{x/^c x\}\kappa_2, v\, \mathtt{mon}_j^{l,k}(\kappa_1, x))$.

The proof of the central lemma depends on a series of auxiliary lemmas about the properties of well-formed terms and contracts, substitution, and contract substitution.

Lemma 3.3.4. *If* $\varnothing; l \Vdash E^k[e]$ *then* $\varnothing; k \Vdash e$.

Proof. By induction on the size of E^k with the help of lemma 3.3.5 for the inductive case. ∎

Lemma 3.3.5. *If* $\varnothing; l \Vdash E^{l_o}[e]$ *then* $\varnothing; l \Vdash e$.

Proof. By induction on the size of E^{l_o}. ∎

Lemma 3.3.6. *If* $\varnothing; l \Vdash E^k[e]$, $\varnothing; k \nvDash e$ *and* $\varnothing; k \Vdash e'$ *then* $\varnothing; l \Vdash E^k[e']$.

Proof. By induction on the size of E^k. ∎

Lemma 3.3.7. *If* $\varnothing; l \Vdash E^k[e]$, $\varnothing; k \vDash e$ *and* $\varnothing; k \Vdash e'$ *then* $\varnothing; l \Vdash E^k[\|e'\|^k]$.

Proof. By induction on the size of E^k. ∎

Lemma 3.3.8. *If* $\Gamma; l \Vdash e$, $k; \Gamma \Vdash e_0$, *and* $x \notin dom(\Gamma)$, $\Gamma; l \Vdash \{\|e_0\|^k/x\}e$.

Proof. By mutual induction on the height of $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 3.3.9. *If* $\Gamma \uplus \{x : k\}; l \Vdash e$, $v = \|v_0\|^k$ *and* $\Gamma; k \Vdash v_0$ *then* $\Gamma; l \Vdash \{v/x\}e$.

Proof. By mutual induction on the height of $\Gamma \uplus \{x : k\}; l \Vdash e$ and $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 3.3.10. *If* $\Gamma; l \Vdash e$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma \uplus \{x : k\}; l \Vdash e$.

Proof. By mutual induction on the height of $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 3.3.11. *If* $\Gamma;\overline{k};\overline{l};j \rhd \kappa$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma \uplus \{x : l'\};\overline{k};\overline{l};j \rhd \{x/^c x\}\kappa$.

Proof. By induction on the height of $\Gamma;\overline{k};\overline{l};l \rhd \kappa$. For the flat contracts case we employ lemma 3.3.12. ∎

Lemma 3.3.12. *If* $\Gamma;l \Vdash e$ *and* $x \notin dom(\Gamma)$, $\Gamma \uplus \{x : k\};l \Vdash \{\|x\|^l/x\}e$.

Proof. First, we generalize the lemma's statement: If $\Gamma;l \Vdash e$ and $x \notin dom(\Gamma)$, $\Gamma \uplus \{x : k\};l \Vdash \{\|x\|^j/x\}e$. Then we proceed by mutual induction on the height of $\Gamma;l \Vdash e$ and $\Gamma;\overline{k};\overline{l};j \rhd \kappa$. ∎

Lemma 3.3.13. *If* $\Gamma;\overline{k};\overline{l};j \rhd \kappa$, $\overline{k'} \subseteq \overline{k}$ *and* $\overline{l'} \subseteq \overline{l}$ *then* $\Gamma;\overline{k'};\overline{l'};j \rhd \kappa$.

Proof. By induction on the height of $\Gamma;\overline{k};\overline{l};j \rhd \kappa$. ∎

3.3.3 Main Theorems

When a program is well-formed, its monitors obviously satisfy the blame correctness criterion.

Theorem 3.3.14. $\longmapsto_l$ *is blame correct.*

Proof. This theorem is a straightforward consequence of lemma 3.3.3. To wit, the subject reduction lemma says that a program satisfies the generalized well-formedness judgment including when its redex is a monitor term containing a flat contract. From the proof of the subject reduction theorem, we know that generalized well-formedness implies

$$\varnothing;l \Vdash \mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c) \rfloor^{\overline{l'}}, \|v\|^k)$$

The label on the context is the same as the client label by lemma 3.3.4 but we also need to know that the server label l is a member of the contract's obligations $\overline{l'}$. This has to be the case given that the monitor term is well-formed. Note how the

proof is independent of the reduction semantics as long as it satisfies the subject reduction property. ∎

In contrast to *lax*, the *picky* contract system fails to satisfy an analogous theorem.

Theorem 3.3.15. *There exists a program e such that* $l_o \vdash e$ *and*

$$e \longmapsto_p^* E^l[\mathtt{mon}_j^{k,l_2}(\lfloor \mathtt{flat}(e) \rfloor^{\bar{l}}, \|v\|^{l_1})]$$

but $k \neq l_1$.

Proof. Here is one such program where $k \neq l_o$:

$$\Pi_l^1 = \mathtt{mon}_l^{k,l_o}(\kappa_l, \|\lambda h_1.h_1\ \lambda x.5\ (\lambda g.g\ 1)\|^k)\ (\lambda f.\lambda h_2.h_2\ \lambda x.6)$$

The restriction on the labels intuitively corresponds to the composition of the two different modules k and l_o through the contract κ_l. Note that Π_l^1 is a family of programs, one per label l. This label is the label of the contract monitor and, consequently, must be the owner of all embedded flat contracts. In principle, l could be the label of the client, the server, or any other non-top-level (l_o) label but in the end, our choice must obey subject reduction.

While Π_l^1 performs no interesting computation, its contract κ_l plays a critical role. To explain the contract though, it is best to start with the type of h_1:

$$[(\mathcal{N} \to \mathcal{N}) \to \{([\mathcal{N} \to \mathcal{N}] \to \mathcal{N}) \to \mathcal{N}\}] \to \mathcal{N}$$

The type tells us that h_1 consumes a complex higher-order function and produces a number. Instead of plain numbers, however, we wish to deal with positive numbers only. We thus know that κ_l must have at least something like the following shape:

$$[(\mathtt{P?}_l \mapsto \mathtt{P?}_l) \mapsto \{([\mathtt{P?}_l \mapsto \mathtt{P?}_l] \mapsto \mathtt{P?}_l) \mapsto \mathtt{P?}_l\}] \mapsto \mathtt{P?}_l$$

Next we add two dependencies:

$$[(\mathtt{P?}_l \mapsto \mathtt{P?}_l) \overset{d}{\mapsto} (\lambda f.\{([\mathtt{P?}_l \mapsto \mathtt{P?}_l] \overset{d}{\mapsto} (\lambda g.\mathtt{P?}_l)) \mapsto \mathtt{P?}_l\})] \mapsto \mathtt{P?}_l$$

The two key points to notice are: (1) in this contract, g is in the scope of f and (2) while f originates in the server (wrapped expression), g originates in the client (context) and both flow into the contract.

Equipped with this informal and approximate understanding, we can now turn to the actual contract:

$$\begin{array}{rcl}
\kappa_l & = & ((\lfloor \mathtt{P?}_l \rfloor^{l_o} \mapsto \lfloor \mathtt{P?}_l \rfloor^{k}) \overset{d}{\mapsto} (\lambda f.\kappa_l^1)) \mapsto \lfloor \mathtt{P?}_l \rfloor^{k} \\
\kappa_l^1 & = & ((\lfloor \mathtt{P?}_l \rfloor^{k} \mapsto \lfloor \mathtt{P?}_l \rfloor^{l_o}) \overset{d}{\mapsto} (\lambda g.\kappa_l^2)) \mapsto \lfloor \mathtt{P?}_l \rfloor^{k} \\
\kappa_l^2 & = & \lfloor \mathtt{flat}(\|\lambda x.\mathtt{zero?}(f\ 1 - g\ 0)\|^l) \rfloor^{k} \\
\mathtt{P?}_l & = & \mathtt{flat}(\|\lambda x.x > 0\|^l)
\end{array}$$

Note how κ_l^2 invokes f on a positive number and g on 0.

Now we show that l must be set to l_o in order to satisfy blame correctness. First, note that for for all $l \in \mathbb{L}$, $l_o \vdash \Pi_l^1$ if $k \neq l_o$. Second, the reduction of Π_l^1 eventually checks that 1 is greater than 0, i.e., that the post-condition contract κ_l^2 is satisfied:

$$\Pi_l^1 \longmapsto_p^* E_0^k[\mathtt{mon}_l^{l_o,k}(\lfloor \mathtt{P?}_l \rfloor^{l_o}, \|\|1\|^l\|^l)]$$

In order for the *picky* system to satisfy the blame correctness condition, l must be equal to l_o, which means the term looks like this:

$$E_0^k[\mathtt{mon}_{l_o}^{l_o,k}(\lfloor \mathtt{P?}_{l_o} \rfloor^{l_o}, \|\|1\|^{l_o}\|^{l_o})]$$

Unfortunately, the next few steps of the reduction process produces a state that is inconsistent with blame correctness. The state involves a monitor with a flat contract where the owner on the checked value differs from the server label on the monitor. Specifically, κ_l^2 also checks that g's pre-condition holds for 0:

$$E_0^k[\mathtt{mon}_{l_o}^{l_o,k}(\lfloor \mathtt{P?}_{l_o} \rfloor^{l_o}, \|\|1\|^{l_o}\|^{l_o})] \longmapsto_p^*$$
$$E_1^{l_o}[\mathtt{mon}_{l_o}^{k,l_o}(\lfloor \mathtt{P?}_{l_o} \rfloor^{l_o}, \|\|0\|^{l_o}\|^{l_o})]$$

This last state, however, is inconsistent with generalized well-formedness because k cannot equal l_o. Indeed, the next few reduction steps result in a failed check. The contract monitor blames k, which isn't the owner l_o of the value. The *picky* system fails to assign blame properly. ∎

In essence the proof of the theorem shows that there is no correct strategy for associating (pieces of) contracts with components in a picky semantics. No matter which labeling strategy we use, a contract violation may blame a component that has no control over the faulty value.

CHAPTER 4

Complete Monitors[11]

4.1 Correct Blame Is Not Enough

The ownership and obligations framework of the previous chapter introduces a criterion for deciding whether a contract system assigns blame correctly. Based on this criterion we show that only the *lax* semantics is blame correct; the *picky* system occasionally blames the wrong party. However, we should acknowledge that the effort of *picky* to restrict the uses of function arguments in the post-condition of dependent contracts is reasonable. Take a second look at the `d/dx` example:

```
;; for some natural number n and real δ
(->d ([f (-> 0<real<1? 0<real<1?)]
       [ε real?])
      (fp (-> 0<real<1? real?))
      #:post-cond
      (for/and ([i (in-range 0 n)])
        (define x (random-number))
        (define slope
          (/ (- (f (- x ε)) (f (+ x ε)))
             (* 2 ε)))
        (<= (abs (- slope (fp x))) δ)))
```

[11] This chapter reports joint work with Tobin-Hochstadt and Felleisen [26].

As we explain in section 3.1, the contract uses `random-number`, which may produce complex numbers in Racket. The *lax* system, though, does not regulate the calls to `f` and `fp` in the **`#:post-cond`** code. The consequences of allowing the application of a complex number to `f` and `fp` are impossible to predict. The functions may not be able to handle a complex number argument. Another possibility is that the two functions return complex number results for complex inputs causing the comparison operator of Racket to raise an error. In both scenarios tracking the error back to the result of `random-number` is challenging and uncertain. Even worse, the use of complex numbers may raise no exceptions and the programmer may assume that the `d/dx` function behaves according to the contract. Naturally, this conclusion is misleading because it does not correspond to a real number argument, the intended domain of use of `d/dx`.

As the example demonstrates, the *lax* system may lead to errors in the face of contracts. We conjecture that a programmer would like the guarantee that the values produced by their components are never used in violation to the interface specifications and, conversely, that their components are not handed values that do not live up to the promises of the specifications.

Blame correctness is thus not strong enough to provide such a universal guarantee. After all, it admits contract systems that entirely discard contract checks. Its power is limited to situations where the contract systems detects a contract violation. As a consequence, it cannot provide any information whether the contract system misses contract violations and contracts fail to protect effectively the components they guard.

In response, we present a generalization of blame correctness, called *complete monitoring*. We take the ownership-and-obligation framework of blame correctness and extend it so that a component may not manipulate values that it does not create or that have not been transferred from other components via a—possibly vacuous—

contract. In short, a complete contract system has the power to monitor *all* value flows across component boundaries.

The next section illustrates *complete monitors* informally. Subsequently, we define them formally in the context of CPCF with ownership and obligations annotations in section 4.3. Finally, the last section presents the main theoretical result of this chapter: a formal definition of complete monitors and theorems that neither neither *lax* nor *picky* are such.

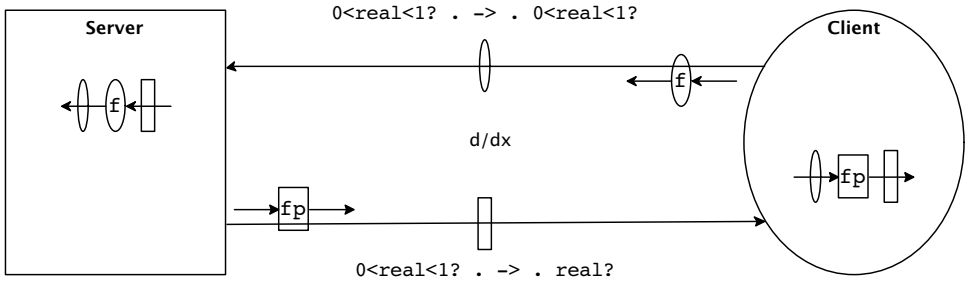
4.2 Complete Monitors With Pictures

While we can use the CPCF model to articulate a formal criterion of monitoring completeness, it can also provide an intuitive understanding of the idea. In this section, we present three diagrams of contract monitoring that employ some of the elements of the CPCF model and introduce complete monitors on this basis.

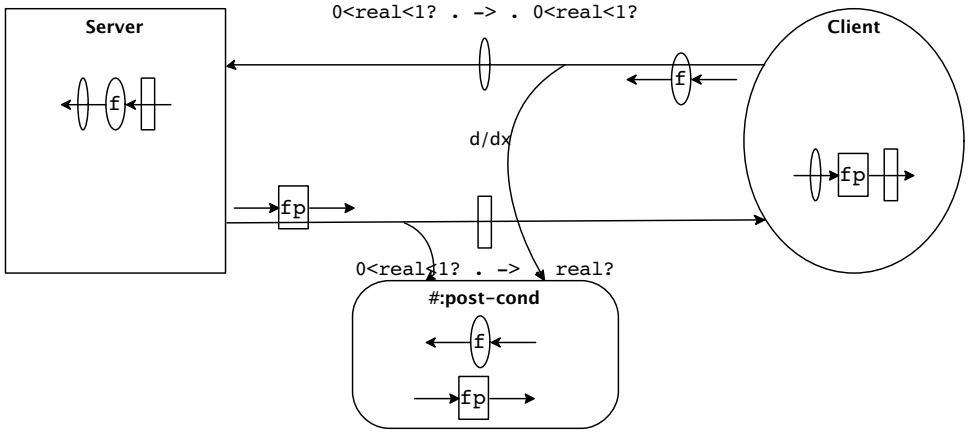
The first diagram in figure 4.1 illustrates how the contract system monitors the contract of `d/dx` without its **`#:post-cond`** clause. The client owns `f` and applies `d/dx` to it. Pictorially, it ships `f` to the server over the `d/dx` channel. The contract system monitors the channel and attaches the appropriate pieces of the contract to `f`. Thus the server component receives a wrapped version of `f`. The wrapper checks that the argument and the result of any application to `f` are real numbers between 0 and 1. The result of the application, `fp`, returns to the client component via a similar channel.

Our diagrams use shapes to express *ownership* of values. Thus `f` comes in an *ellipse* to match the shape of the client and `fp` is in a *rectangle*, like the server component that creates the function. If the client were to pass `fp` back to the server to create the second derivative of `f`, the value would come in an ellipse around the rectangle and the contract wrapper. In other words, wrapping shapes within shapes

Contracts without post-condition:



Lax contracts for post-conditions:



Picky contracts for post-conditions:

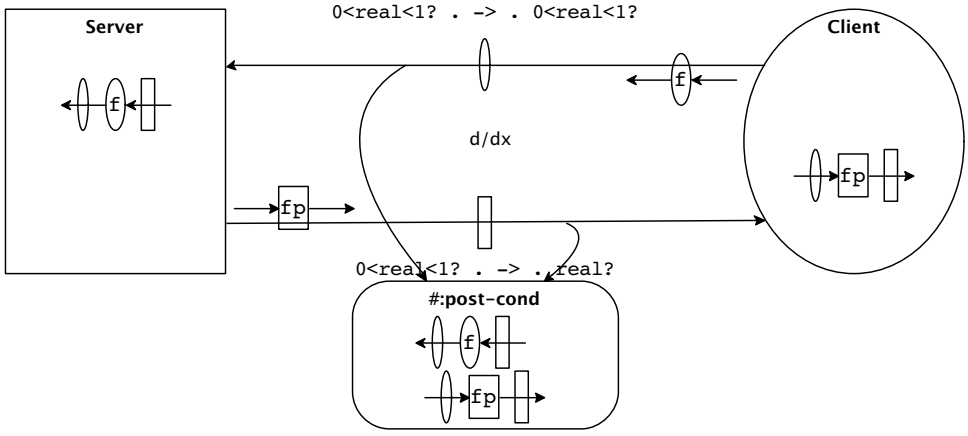


Figure 4.1: Monitoring in diagrams

illustrates how the semantics uses ownership to keep track of a value's provenance.

Similarly, shapes on the input-output arrows mark *obligations*. In particular, the flat contracts that guard channels have the same shape as the component responsible for satisfying them. The characterization holds for both components and higher-order values that flow back and forth and receive wrappers.

Hence, our diagram suggests that if the initial program is *well-formed*, meaning it separates the client and the server component with a properly formed contract boundary, a complete contract system preserves a two-part invariant. The first part dictates that each value has the same shape as its origin and, if the origin differs from the current host component, then the contract system guards the value with contracts. The second part adds that the host component is responsible for meeting the pre-condition for the uses of the foreign value and the origin component is responsible for the post-condition.

Even though the invariant seems easy to maintain, adding the **`#:post-cond`** clause shows that doing so poses subtle challenges. Concretely, a post condition clause consists of a piece of code and thus introduces a new component. In a real-world language such as Racket, this new component could exist within the server module, the client module, or as a third-party component all by itself. No matter where the contract is located, it hosts both `f` and `fp`, and this co-habitation is the source of all subtleties.

The new component connects to the `d/dx` channels with its own branch channels and can thus absorb the values from these channels. As the second diagram of figure 4.1 shows, the *lax* semantics allows `f` and `fp` to enter the new component before they flow through the monitors—meaning no guards are attached to these new channels. Since `f` and `fp` have different owners, at least one of the values must be considered a foreign value and, as such, inhabits the component without the necessary guard.

In contrast to *lax*, *picky* protects these additional channels of communication, too. Figure 4.1 explains this idea with forks in the channels *behind* the monitors that protect the channels. Unfortunately, the obligations for the flat contracts in the `#:post-cond` component do not agree with the second part of the completeness invariant. That is, at least one of the two values inhabits the new component as a foreign value but is protected by misshaped contracts.

The next section presents our formal framework for complete monitoring. Technically, our framework serves as an independent specification of the contract system and excludes scenarios such as the two above by halting computation when a certain single-owner policy condition breaks.

4.3 CPCF With A Single Owner Policy

CPCF with obligations and ownership, see section 2.3, uses annotations to build an independent level for reasoning about component boundaries and value migration. In principle, ownership and obligations could be just observers of the reduction sequence that do not affect evaluation. In fact, in the context of our blame correctness criterion, ownership and obligation do not prevent us from admitting as blame correct contract systems that permit an uncontrolled flow of values between components exactly because ownership and obligation annotations are just decorative. The problem lies with the way the original semantics of CPCF treats ownership. It allows for components to mix freely and for values to acquire multiple owners as they cross boundaries. For instance a term $\|e\|^l$ can show up without any restrictions as a sub-term of a term $\|e'\|^k$. Similarly a value $\|\ldots\|v\|^{l_1}\ldots\|^{l_n}$ comes with multiple owners as the annotations keep track of the whole history of migrations from one module to another. These annotations do not affect evaluation, though, because it ignores them and proceeds as if they are not there.

To prove that the contract system allows values to migrate from one component to another only when they are under the control of a contract, we use ownership to impose restrictions on value flows between components. We enforce a *single owner* policy that disallows mixing terms with different owners. Instead, our reduction relation ensures that foreign values within a component are wrapped in contract checks or that the contract system has completely verified all (flat) specifications during the absorption of a foreign value into a component.

$E^l[\cdots]$	$\longmapsto_m$	$E^l[\cdots]$	
$\|\|\mathtt{n}_1\|\|^l + \|\|\mathtt{n}_2\|\|^l$	.	$\mathtt{n}$	where $n_1 + n_2 = n$
$\|\|\mathtt{n}_1\|\|^l - \|\|\mathtt{n}_2\|\|^l$	.	$\mathtt{n}$	where $n_1 - n_2 = n$
$\mathtt{zero?}(\|\|\mathtt{0}\|\|^l)$	.	$\mathtt{tt}$	
$\mathtt{zero?}(\|\|\mathtt{n}\|\|^l)$	.	$\mathtt{ff}$	if $n \neq 0$
$\|\|v_1\|\|^l \wedge \|\|v_2\|\|^l$	.	v	where $v_1 \wedge v_2 = v$
$\|\|v_1\|\|^l \vee \|\|v_2\|\|^l$	.	v	where $v_1 \vee v_2 = v$
$\mathtt{if}\ \|\|\mathtt{tt}\|\|^l\ e_1\ e_2$	.	e_1	
$\mathtt{if}\ \|\|\mathtt{ff}\|\|^l\ e_1\ e_2$	.	e_2	
$\|\|\lambda x.e\|\|^l\ \|\|v\|\|^l$	.	$\|\{\|\|v\|\|^l/x\}e\|\|^l$	
$\mu x.e$	.	$\{\|\|\mu x.e\|\|^l/x\}e$	
$\mathtt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)$	.	$\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))$	
$\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{l'}, \|\|b\|\|^{l'})$	.	$\mathtt{check}_j^k(e\ b, b)$	
$\mathtt{check}_j^k(\|\|\mathtt{tt}\|\|^j, v)$	.	v	
$\mathtt{check}_j^k(\|\|\mathtt{ff}\|\|^j, v)$	.	$\mathtt{error}_j^k$	
$E^l[\mathtt{error}_j^k]$	$\longmapsto_m$	$\mathtt{error}_j^k$	

Figure 4.2: CPCF semantics enforces the single-owner policy

To implement this policy, we require all terms in a *redex* to have a single owner. Put differently, our semantics does not perform operations on values that have ownership annotations with different owners. We use $||e||^l$ to denote that e may have no ownership annotations but if it has one then the owner label is l for all such

annotations:

$$||e||^l \quad = \quad \|...\|e\|^l...\|^l \qquad \text{where for all labels } k \text{ and terms } e',\ e \neq \|e'\|^k.$$

Proposition 4.3.1. *(The single ownership policy for CPCF)*

If a term e_0 reduces to term $e_1 = E^l[r]$ where r a redex, and the elements of r are not of the form $||e||^l$, then there is no term e_2 such that $e_1 \longmapsto_m e_2$. Also, if a term e_0 reduces to term $e_1 = E^l[r]$ and the elements of r are of the form $||e||^l$, then there is term e_2 such that $e_1 \longmapsto_m e_2$.

Proof. By inspection of the reduction relation.

The *single owner* policy plays a critical role for the definition of the revised reduction semantics for CPCF. A component should be able to perform an operation if and only if it is the owner of all the arguments of the redex. This implies that either the arguments inherit their implicit ownership annotation from the context or that they come with an explicit ownership annotation that matches with the owner of the context. We model implicit ownership with *labeled evaluation contexts* same as in section 3; see figure 3.4.

The reduction relation of figure 4.2 implements the single owner policy by reducing redexes only if the label of the hole matches the owner of the pieces of the redex. For instance the rule for function application is more restrictive than the original rule for CPCF from section 2.3. The latter allows the function and the argument to have different and multiple owners. In contrast, the new rule fires only if l, the owner of the component, is also the only owner of the function and the argument. The argument is substituted in the body of the function, annotated with the common owner so that it keeps its ownership annotation no matter where it lands in the function body. The context absorbs the body of the function, which thus obtains the context's ownership annotation. Since the function and the context have the same owner, however, the body of the function retains its original

owner. This rationale explains all the rules, including the rules for monitors where the client label must be the same as the label of the context. When the reduction rules create new values, as in the case of primitives operators, the context becomes directly responsible for the new value and thus no additional ownership annotation is necessary.

Values retain their owner as long as they move inside the same component. They change owner only when flat contract checking succeeds. When the check succeeds, the contract system gives permission to the surrounding component l to absorb b, and b changes hands between k and l.

The reduction rules concerning monitors for dependent function contracts come in the two flavors $l(ax)$ and $p(icky)$. Their formal definitions remain the same as in section 2.3:

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \longmapsto_l E^l[\lambda x.\mathtt{mon}_j^{k,l}(\{x/^c x\}\kappa_2, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))]$$

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \longmapsto_p E^l[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\{\mathtt{mon}_j^{l,k}(\kappa_1, x)/^c x\}\kappa_2}, v\,\mathtt{mon}_j^{l,k}(\kappa_1, x))]$$

Like in section 2.3, ownership propagation and, by implication, the single-owner policy are only meaningful if ownership and obligations annotations are in specific places in source code. Particularly in this new setting, ownership in valid source code must coincide with monitors and terms have to respect the single-owner policy. The well-formedness judgments from section 2.3, $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$, are sufficient to enforce this restriction and we use them here too to specify valid source programs.

This semantics of CPCF differs from the semantics of the chapter 3. The main deviation is the introduction of the single owner policy. It helps us prove complete monitoring, a notion of correctness for a contract system that subsumes blame correctness.

In the previous chapter, ownership and obligations are used to verify that when-

ever a contract error is raised, its witness value is owned by the party that is blamed and that the party fails to satisfy one of its obligations. Our new semantics turns ownership into a computational device that is exploited to enforce the single owner policy. This change enables us to state when a contract system is a *complete monitor* for all specified properties and show that not only *picky* but also *lax* is incorrect.

4.4 Neither Lax Nor Picky Are Complete Monitors

The CPCF semantics enforces the single owner policy. If a redex does not respect it, the evaluation gets stuck. Since embeddings of foreign terms in a component are wrapped with contract monitors, such stuck states are evidence that a value has leaked from one component to another without the contract system's approval. If a contract system can eliminate all such stuck states and force programs to reduce to a value or to diverge or to raise a contract error, then the contract system insulates the components of the program and regulates all exchanges of values between them. We call such a contract system a *complete monitor*.

Definition 4.4.1 (Complete Monitors for CPCF)**.** *A contract semantics m specifies a* complete monitor *iff for all well typed terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- $e_0 \longmapsto_m^* v$ *or,*
- *for all e_1 such that $e_0 \longmapsto_m^* e_1$ there exists e_2 such that $e_1 \longmapsto_m e_2$ or,*
- $e_0 \longmapsto_m^* e_1 \longmapsto_m^* \texttt{error}_j^k$ *and e_1 is of the form*

$$E^l[\texttt{mon}_j^{k,l}(\lfloor \texttt{flat}(e) \rfloor^{\overline{l}}, v)]$$

and for all such terms e_1, $v = \|v_1\|^k$ and $k \in \overline{l}$.

Complete monitoring takes advantage of the ownership and obligation annotations to verify that well-formed programs do not get stuck. In addition if a contract

error is raised indicating contract j failed, then for all checks of flat contracts from j, the owner k of the guarded value is identical to the server label on the contract monitor and the flat contract is part of the obligations of k. Clearly, this is stronger guarantees than *blame correctness*.

At first glance *complete monitoring* appears too weak to establish the correctness of a contract system; it simply guarantees that when a value crosses a boundary, the contract system attaches *some* contract to it and that if a contract violation is detected, blame is assigned to the party that contributed the witness value. What complete monitoring does *not* require is that the contract system (1) attaches the *proper* contracts to migrating values and (2) checks flat contracts. Point 1 concerns the decomposition of compound contracts, i.e., a contract system must check the pieces of a compound value (for example, functions) with the proper pieces of their contracts (for example, domain and range). In CPCF, this property comes for free with type soundness, which forces the proper distribution of compound contracts over their pieces to retain type safety. As for point 2, it is necessary to ensure that a contract system actually executes the application of the predicate to the witness value. Again, this is obvious in the case of CPCF and it is easy to check in general. In short, complete monitoring is the main ingredient language designers must check if they wish to implement a correct contract system; the remaining properties can be validated by inspection.

We show now that neither *lax* nor *picky* are complete.

As an example where *lax* does not manage to live up to complete monitoring, consider the following program:

$$\Pi_l^1 = \mathtt{mon}_l^{k,l_o}(\kappa_l, \|\lambda h_1.h_1\ \lambda x.5\ (\lambda g.g\ 1)\|^k)\ (\lambda f.\lambda h_2.h_2\ \lambda x.6)$$

where

$$\begin{array}{rcl}
\kappa_l & = & ((\lfloor \mathtt{P?}_l \rfloor^{l_o} \mapsto \lfloor \mathtt{P?}_l \rfloor^{k}) \overset{d}{\mapsto} (\lambda f.\kappa_l^1)) \mapsto \lfloor \mathtt{P?}_l \rfloor^{k} \\
\kappa_l^1 & = & ((\lfloor \mathtt{P?}_l \rfloor^{k} \mapsto \lfloor \mathtt{P?}_l \rfloor^{l_o}) \overset{d}{\mapsto} (\lambda g.\kappa_l^2)) \mapsto \lfloor \mathtt{P?}_l \rfloor^{k} \\
\kappa_l^2 & = & \lfloor \mathtt{flat}(\|\lambda x.\mathtt{zero?}(f\ 1 - g\ 0)\|^{l}) \rfloor^{k} \\
\mathtt{P?}_l & = & \mathtt{flat}(\|\lambda x.x > 0\|^{l}).
\end{array}$$

If $k \neq l_o$, then for all $l \in \mathbb{L}$, $\varnothing; l_o \Vdash \Pi_l^1$. The constraint on k and l_o comes from the rules for well-formed source terms and captures the intuition that contracts are used as the interface between distinct components.

Π_l^1 is the same example as the one that we use to invalidate *picky* in section 3.3. Like before, we start by showing that l must equal l_o in order to satisfy complete monitoring. The reduction of Π_l^1 eventually applies f_v to 1. After the substitution of f_v for f in κ_l^2 we get

$$\kappa_l^{\dagger} = \lfloor \mathtt{flat}(\|\lambda x.\mathtt{zero?}(\|\|f_v\|^{l_o}\|^{l}\ 1 - g\ 0)\|^{l}) \rfloor^{k}.$$

Notice now, the ownership labels on the application $\|\|f_v\|^{l_o}\|^{l}\ 1$. A few steps further down, the reduction sequence evaluates the flat predicate and tries to run the application. If the labels l_o and l do not match, the evaluation of the predicate is going to lead to a stuck state. In order for the *lax* system to avoid this situation and satisfy the complete monitoring condition, $\kappa_l^{\dagger}$ must remain well formed:

$$\{f : l, g : l\}; \{k\}; \{\}; l \rhd \kappa_l^{\dagger}.$$

This judgment, however, demands that $\{f : l, g : l\}; l \Vdash \|\|f_v\|^{l_o}\|^{l}$, which in turn requires that l must be equal to l_o. If so, the contract looks like this:

$$\kappa_{l_o}^{\dagger} = \lfloor \mathtt{flat}(\|\lambda x.\mathtt{zero?}(\|\|f_v\|^{l_o}\|^{l_o}\ 1 - g\ 0)\|^{l_o}) \rfloor^{k}.$$

The next few steps of the reduction process produce a state that is inconsistent with complete monitoring. Specifically, κ_l^2 also applies g_v to 0:

$$\kappa_{l_o}^{\dagger\dagger} = \lfloor \mathtt{flat}(\|\lambda x.\mathtt{zero?}(\|\|f_v\|^{l_o}\|^{l_o}\ 1 - \|\|g_v\|^{k}\|^{l_o}\ 0)\|^{l_o}) \rfloor^{k}.$$

And this last contract leads to a state that *lax* can violate the single owner policy. Indeed $\Pi^1_{l_o} \longmapsto^*_l E^{l_o}[\|\|g_v\|^k\|^{l_o}\ 0]$. This last state is a stuck state as it involves the application of a function that has multiple owner tags, i.e., it has crossed contract-free boundaries between distinct components.

Our example shows that independently of the choice of l, Π^1_l does not respect preservation under the *lax* semantics.

Theorem 4.4.2. *$\longmapsto_l$ is not a complete monitor.*

The same example also shows that *picky* CPCF is not a complete monitor.

Theorem 4.4.3. *$\longmapsto_p$ is not a complete monitor.*

Proof. Unlike for *lax*, the single-owner policy holds through out the computation. The two arguments, f_v and g_v are installed inside κ^2_l wrapped in monitors that mediate appropriately the communication between components l and l_o. However, the reduction sequence does not live up to the correct blame portion of the definition of complete monitoring. The monitors that the reduction installs in the dependent post-condition do not have the right labels. The state that violates the second requirement of complete monitors is the same as the one that violates correct blame for *picky* in section 3.3. ∎

CHAPTER 5

Complete Monitors, Applied[12]

5.1 Fixing Dependent Contracts

Complete monitoring reveals that both *lax* and *picky*, are not correct. The latter blames incorrectly in some case while the former fails to protect components effectively. However, the framework of complete monitors is not only a medium for judging existing contract systems, but also a tool that the programming languages designer can employ to fix existing contract systems and design new ones.

In particular, complete monitoring can guide the construction of a correct semantics for dependent contracts. If we inspect the examples that disprove *lax* and *picky* as complete monitors, we can deduce that in order to avoid stuck states and to assign blame correctly, two changes are necessary:

1. arguments that flow in the post-condition of a dependent contract need to be wrapped with monitors for the pre-condition of the dependent contract,

2. the client party for such monitors needs to be the same independently from whether the server or the client of the dependent contract contribute the argument.

[12] This chapter reports joint work with Tobin-Hochstadt and Felleisen [26].

Starting from these observations and from a close look at the well-formedness rules, we can detect an appropriate party: the contract itself. The violations that render *lax* incomplete and less effective than *picky* are all violations that have to do with the contract misusing values it manipulates. Conversely, the reason *picky* assigns blame incorrectly is because it blames either the client or the server for violations caused by the contract itself. We crystallize these conclusions with a new reduction rule for dependent contracts, dubbed *indy*:

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \longmapsto_s E^l[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\kappa_2'}, v\ \mathtt{mon}_j^{l,k}(\kappa_1, x))] \qquad (\textit{indy})$$
$$\text{where } \kappa_2' \equiv \{\mathtt{mon}_j^{l,j}(\kappa_1, x)/^c x\}\kappa_2$$

The new rule uses the label of the contract j as the client label for the monitors it installs in the post-condition of a dependent contract. This way, we treat the contract as an independent component with its own contractual obligations.

Naturally, the *indy* semantics changes the behavior of programs. Consider this example:

$$\begin{aligned}\Pi^2 \quad &= \quad \mathtt{mon}_j^{k,l_o}(\kappa, \lambda f.f\ 42)\ \lambda x.x \\ &\quad \text{where } \kappa = (\mathtt{P?} \mapsto \mathtt{P?}) \overset{d}{\mapsto} (\lambda f.\mathtt{flat}(\lambda x.f\ 0 > -1))\end{aligned}$$

The evaluation of Π^2 yields the following results for the three different contract monitoring systems:

program	monitoring system	result
Π^2	*lax*	42
Π^2	*picky*	$\mathtt{error}_j^k$
Π^2	*indy*	$\mathtt{error}_j^j$

The table demonstrates several points. First, when a program yields a plain value according to the *lax* system, the *picky* system may still find a fault during contract checking and signal a violation. Second, the *picky* system blames party k, the server

component, for a contract violation. The specific violation is that f is applied to 0 in the dependency assertion, even though the domain contract promises that the function is only applied to positive numbers. Third, the *indy* system blames the contract itself, rather than the server.

Another example shows that while *picky* can blame the client when things go wrong with the contract, *indy* blames the contract:

$$\begin{aligned}\Pi^3 \quad &= \quad \mathtt{mon}_*^{k,l_o}(\kappa, \lambda f.\, f\ \lambda x.x)\ \lambda g.g\ 42 \\ &\text{where } \kappa = ((\mathtt{P?} \mapsto \mathtt{P?}) \overset{d}{\mapsto} (\lambda f.\mathtt{flat}(\lambda x.\, f\ 0 > -1))) \mapsto \mathtt{P?}\end{aligned}$$

Specifically, evaluating Π^3 yields the following results:

program	monitoring system	result
Π^3	*lax*	42
Π^3	*picky*	$\mathtt{error}_j^{l_o}$
Π^3	*indy*	$\mathtt{error}_j^{j}$

Again the *lax* system does not signal contract violations, while the other four report one. Here the *indy* system blames the contract itself rather than the client, which is blamed by the *picky* system.

However, as the examples suggest, the *indy* contract system signals an error when the *picky* system signals an error and vice versa, though the errors are not necessarily labeled with the same party.

Proposition 5.1.1. $e \longmapsto_i^* \mathtt{error}_l^k$ *iff* $e \longmapsto_p^* \mathtt{error}_l^{k'}$

Proof. By a straightforward bi-simulation argument. The bi-simulation used for the proof relates two expressions that are structurally identical except that their labels can differ. ∎

Before proving that *indy* is a correct design theoretically, i.e., it is a complete monitor, note that the *indy* semantics is not just a theoretical artifact. On the pragmatic side, the treatment of contracts as independent parties is compatible with

some practical uses in Racket. First, contracts for Rackets unit system are given as part of the signature. Strickland and Felleisen [62] show that linking such units may necessitate blaming the signature itself. Our framework finally provides a semantic explanation for this phenomenon. Second, in Rackets first-order module system, contracts are specified via **`contract-out`**, i.e., in the export interface of modules. This form combines identifiers with contracts and attaches contracts to these values as they flow across the module boundary. When things go wrong with the dependencies in such contracts, the monitoring system considers the contract a part of the server module and blames the server module. We can express this idea in our framework with the small change of using the module name as the contract label. Finally, Typed Racket [67, 68, 69] protects the interaction of typed and untyped modules with contracts derived from types. Since one of the basic assumptions of Typed Racket is that untyped modules stay unchanged, it implements this protection mechanism with **`require/contract`**. This contract form guarantees that values from an untyped module satisfy the specified contract. Put differently, the form protects the import boundary. If a programmer attached dependencies to these contracts, the code would have to be considered as a part of the client module. We can specify this behavior as instance of the *indy* semantics that uses the import module as the contract label.

The *indy* semantics have influenced important changes in Racket's contract system. The *indy* contract combinator `->i`, that constructs an *indy* dependent function contract, has deprecated the initial `->d` combinator of Racket. Currently, because of their pragmatic significance and their theoretical correctness, the *indy* semantics are the basis of all dependent contracts in Racket.

5.1.1 Complete Monitoring = Progress + Preservation

We now show that *indy* is a *complete monitor*. The proof of complete monitoring follows a subject reduction technique similar to those of type soundness [71]. This sub-section presents the construction of the generalized well-formedness judgment for the subject reduction proof, progress and preservation, and how these results imply completeness for *indy*.

The generalized well-formedness we utilize in section 3.3 (figure 3.6) to show that *lax* is blame correct has almost all the necessary ingredients to also prove *indy* a complete monitor; it makes sure that during computation ownership and monitors specify component boundaries consistently and that obligations are distributed appropriately. The only missing piece is that the original form of the generalized well-formedness does not enforce the single-owner policy that is the core of complete monitoring. We compensate for that by removing from our rules for generalized well-formedness specifically, the rule that allows to embed "foreign" terms that come with an explicit annotation of their owner in a "host" term with a different owner. With this adjustment, generalized well-formedness excludes mixing terms with different owners except under the supervision of contract monitors. Figure 5.1 shows the details of the revised generalized well-formedness which for the proof replaces the judgment of figure 3.6.

Now we are ready to prove *indy* correct.

Theorem 5.1.2. *$\longmapsto_i$ is a complete monitor.*

Proof. As a direct consequence of the preservation and progress lemmas (lemmas 5.1.3 and 5.1.4), we obtain that for all well typed terms e_0, if $\varnothing; l_o \Vdash e_0$,

- $e_0 \longmapsto_i^* v$ or,
- for all e_1 such that $e_0 \longmapsto_i^* e_1$ there exists e_2 such that $e_1 \longmapsto_i e_2$ or,

$\boxed{\Gamma; l \Vdash e}$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1\ e_2} \qquad \frac{\Gamma; l \Vdash e_1}{\Gamma; l \Vdash \mathtt{zero?}(e_1)} \qquad \frac{}{\Gamma; l \Vdash \mathtt{error}^k}$$

$$\frac{\Gamma; j \Vdash e \quad \Gamma; l \Vdash v}{\Gamma; l \Vdash \mathtt{check}^k_j(e, v)} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2 \quad \Gamma; l \Vdash e_3}{\Gamma; l \Vdash \mathtt{if}\ e_1\ e_2\ e_3}$$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 + e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 - e_2}$$

$$\frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \wedge e_2} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash e_1 \vee e_2}$$

$$\frac{}{\Gamma; l \Vdash \mathtt{n}} \qquad \frac{}{\Gamma; l \Vdash \mathtt{tt}} \qquad \frac{}{\Gamma; l \Vdash \mathtt{ff}} \qquad \frac{\Gamma(x) = l}{\Gamma; l \Vdash x}$$

$$\frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \|e\|^l} \qquad \frac{\Gamma; k \vDash e \quad \Gamma; \{k\}; \{l\}; j \rhd \kappa}{\Gamma; l \Vdash \mathtt{mon}^{k,l}_j(\kappa, e)}$$

$\boxed{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa}$

$$\frac{\Gamma; \overline{l}; \overline{k}; j \rhd \kappa_1 \quad \Gamma; \overline{k}; \overline{l}; j \rhd \kappa_2}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \mapsto \kappa_2}$$

$$\frac{\Gamma; \overline{l}; \overline{k} \cup \{j\}; j \rhd \kappa_1 \quad \Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa_2}{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)}$$

$$\frac{\Gamma; j \Vdash e \quad \overline{k} \subseteq \overline{k'}}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{flat}(\|e\|^j) \rfloor^{\overline{k'}}}$$

Figure 5.1: Obligations coincide with labels on monitors (3)

- $e_0 \longmapsto_i^* \mathtt{error}^k_j$.

For the last case, since $\varnothing; l_o \Vdash e_0$ we know that $\mathtt{error}^k_j$ does not occur in e_0. Thus it must be a result of reduction. The only reduction rule that introduces $\mathtt{error}^k_j$ is $E^l_2[\mathtt{check}^k_j(||\mathtt{ff}||^j, v)] \longmapsto_i E^\dagger_2[\mathtt{error}^k_j]$. Also $E^\dagger_1[\mathtt{error}^k_j] \longmapsto_i \mathtt{error}^k_j$. Again $\mathtt{check}^k_j(e, v)$ does not occur in e_0 because $\varnothing; l_o \Vdash e_0$. Hence it must be also the

result of a reduction. The only rule that introduces checks is:

$$E_1^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e)\rfloor^{\overline{l'}}, ||b||^{l'})] \longmapsto_i E_1^l[\mathtt{check}_j^k(e\ b, b)].$$

From the above information, we deduce:

$$\begin{aligned}
&e_0 \longmapsto_i^* E_1^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e)\rfloor^{\overline{l'}}, ||b||^{l'})] \\
&\longmapsto_i E_1^l[\mathtt{check}_j^k(e\ b, b)] \\
&\longmapsto_i^* E_2^l[\mathtt{check}_j^k(||\mathtt{ff}||^j, v)] \\
&\longmapsto_i E_2^l[\mathtt{error}_j^k] \\
&\longmapsto_i \mathtt{error}_j^k.
\end{aligned}$$

By lemma 5.1.4 we get $\varnothing; l_o \Vdash E_1^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e)\rfloor^{\overline{l'}}, ||b||^{l'})]$. Lemma 5.1.6 implies $\varnothing; l \Vdash \mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e)\rfloor^{\overline{l'}}, ||b||^{l'})$. From the rules for well formed terms we obtain $||b||^{l'} = \|v_1\|^k$ and $k \in \overline{l'}$. Finally combining the above results entails that $\longmapsto_i$ is a complete monitor. ∎

The proof is direct consequence of two major lemmas: progress and preservation. The first says that a well-formed typed, term reduces to another term unless is a value or a contract error.

Lemma 5.1.3. *(Progress) For all well typed terms e such that* $\varnothing; l \Vdash e$, $e = v$ *or* $e = \mathtt{error}_j^k$ *or* $e \longmapsto_i e_0$.

Proof. From lemma 5.1.5 we know that there exist e', l and E^l such that $e = E^l[e']$. Via lemma 5.1.6 and by $\varnothing; l_o \Vdash e$, we derive $\varnothing; l \Vdash e'$. We proceed by case analysis on the form of e':

- $e' = v_1 + v_2$: By type soundness for CPCF, we get thet $v_1 = ||\mathtt{n}_1||^k$ and $v_2 = ||\mathtt{n}_2||^j$. Since $\varnothing; l \Vdash e'$ from the inference rules for well-formed terms, we deduce $\varnothing; l \Vdash ||\mathtt{n}_1||^k$ and $\varnothing; l \Vdash ||\mathtt{n}_2||^j$. With additional uses of well-formedness we obtain $k = j = l$. Thus according to the reduction relation $e \longmapsto_i E^l[\mathtt{n}]$ where $n = n_1 + n_2$.

- The rest of the cases are similar to the first. ∎

If a well-formed term reduces according to *indy*, it reduces to a well-formed term.

Lemma 5.1.4. *(Preservation) For all terms e and e_0 such that $\varnothing; l_o \Vdash e$ and $e \longmapsto_i e_0$, $\varnothing; l_o \Vdash e_0$.*

Proof. We proceed by case analysis on the reduction of e:

- $E^l[||\mathtt{n}_1||^l + ||\mathtt{n}_2||^l] \longmapsto_i E^l[\mathtt{n}]$: By assumption $\varnothing; l_o \Vdash e$. For the latter, the redex extraction lemma, lemma 5.1.6, implies that $\varnothing; l \Vdash ||\mathtt{n}_1||^l + ||\mathtt{n}_2||^l$. We can use the same label to check n via the inference rules, i.e., $\varnothing; l \Vdash \mathtt{n}$. Hence, from lemma 5.1.8 $\varnothing; l_o \Vdash E^l[\mathtt{n}]$.

- The cases for other primitive operations, conditionals, checks and errors are similar to the first.

- $E^l[\mu x.e] \longmapsto_i E^l[\{\|\mu x.e\|^l/x\}e]$: By assumption and lemma 5.1.6, we conclude that $\varnothing; l \Vdash \mu x.e$. The inference rules imply $\{x : l\}; l \Vdash e$. Via lemmas 5.1.10, 5.1.12 and 5.1.13, we conclude that $\varnothing; l \Vdash \{\|\mu x.e\|^l/x\}e$. From lemma 5.1.8, we obtain $\varnothing; l_o \Vdash E^l[\{\|\mu x.e\|^l/x\}e]$.

- $E^l[v_f\ ||v||^]] \longmapsto_i E^l[[\|\{\|v\|^l/x\}e_0\|^l]$ where $v_f = ||\lambda x.e_0||^l$: By assumption and lemma 5.1.6, we conclude $\varnothing; l \Vdash v_f\ ||v||^l$. Therefore, $\varnothing; l \Vdash ||\lambda x.e_0||^l$ and $\varnothing; l \Vdash ||v||^l$. The inference rules imply $\{x : l\}; l \Vdash e_0$. Since $\varnothing; l \Vdash \|v\|^l$ still holds, we conclude again via another use of lemmas 5.1.10, 5.1.12 and 5.1.13 that $\varnothing; l \Vdash \{\|v\|^l/x\}e_0$. The desired conclusion $\varnothing; l_o \Vdash E^l[[\|\{\|v\|^l/x\}e_0 l\|^l]$ is derived from the above antecedents and lemma 5.1.8 if $v_f = \lambda x.e_0$ or from the above antecedents and lemma 5.1.9 otherwise.

- $E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c) \rfloor^{\overline{l'}}, ||b||^{l'})] \longmapsto_i E^l[\mathtt{check}_j^k(e_c\ b, b)]$: The assumptions imply $\varnothing; l \Vdash \mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c) \rfloor^{\overline{l'}}, ||b||^{l'})$ via lemma 5.1.6. Hence, $e_c = \|e_c'\|^j$

with $\varnothing; j \Vdash e'_c$. Furthermore, the same reasoning yields $l' = k$ and $\varnothing; k \Vdash b$. We trivially obtain that $\varnothing; l \Vdash b$ and $\varnothing; j \Vdash b$. Thus $\varnothing; j \vDash \|e'_c\|^j\ b$. Since the rules for well-formed expressions imply $\varnothing; l \Vdash \mathtt{check}^k_j((e_c\ b), b)$ is well-formed, the desired conclusion follows immediately.

- Let $e = E^l[\mathtt{mon}^{k,l}_j(\kappa_1 \mapsto \kappa_2, v)]$. According to the reduction relation

$$e \longmapsto_i E^l[\lambda x.\mathtt{mon}^{k,l}_j(\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))].$$

  Applying the usual method, we get $\varnothing; l \Vdash \mathtt{mon}^{k,l}_j(\kappa_1 \mapsto \kappa_2, v)$, $v = \|v_0\|^k$, $\varnothing; k \Vdash v_0$. The contract check yields two pieces of knowledge:

$$\varnothing; \{l\}; \{k\}; j \rhd \kappa_1$$

  and

$$\varnothing; \{k\}; \{l\}; j \rhd \kappa_2.$$

  From an additional application of the inference rules for well-formedness we get $\{x : l\}; k \vDash v\ \mathtt{mon}^{l,k}_j(\kappa_1, x)$ and, using lemmas 5.1.11 and 5.1.14, $\{x : l\}; l \Vdash \mathtt{mon}^{k,l}_j(\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))$. Finally from a last application of the inference rules for well-formedness, we get $\varnothing; l \Vdash \lambda x.\mathtt{mon}^{k,l}_j(\kappa_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))$.

- $E^l[\mathtt{mon}^{k,l}_j(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \longmapsto_i e_0$ where

$$e_0 = E^l[\lambda x.\mathtt{mon}^{k,l}_j(\kappa'_2, v\ \mathtt{mon}^{l,k}_j(\kappa_1, x))]$$

  and

$$\kappa'_2 = \{\mathtt{mon}^{l,j}_j(\kappa_1, x)/^c x\}\kappa_2 :$$

  As usual we derive $\varnothing; l \Vdash \mathtt{mon}^{k,l}_j(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)$ from lemma 5.1.6 and the cases' assumptions. Thus, $v = \|v\|^k$ with $\varnothing; k \Vdash v$, and

$$\varnothing; \{l\}; \{k, j\}; j \rhd \kappa_1$$

plus

$$\{x : j\}; \{k\}; \{l\}; j \rhd \kappa_2 \ .$$

The rest of this argument uses the same strategy as the preceding case, except that we use lemmas 5.1.15 and 5.1.16 to derive the key result, $\{x : l\}; l \Vdash \mathtt{mon}_j^{k,l}(\kappa_2', v\, \mathtt{mon}_j^{l,k}(\kappa_1, x))$. ∎

The proofs of the progress and preservation lemmas depend on a series of easy auxiliary lemmas about the properties of well formed terms and contracts, substitution, contract substitution, and environment extension.

Lemma 5.1.5. *(Unique Decomposition) For all terms e such that $e \neq v$ and $e \neq \mathtt{error}_j^k$, there are unique e', l', and $E^{l'}$ such that $e = E^{l'}[e']$ and $e' = v_0\, op\, v_1$ or $e' = \mathtt{zero?}(v)$ or $e' = \mathtt{if}\ v_0\ e_1\ e_2$ or $e' = \mathtt{check}_j^k(v_0, v_1)$ or $e' = v_1\ v_2$ or $e' = \mu x.e_0$ or $e' = \mathtt{mon}_j^{k,l}(\kappa, v)$ or $e' = \mathtt{error}_j^k$.*

Proof. By induction on the size of e. qed

Lemma 5.1.6. *If $\varnothing; l \Vdash E^k[e]$ then $\varnothing; k \Vdash e$.*

Proof. By induction on the size of E^k with the help of lemma 5.1.7 for the inductive case. ∎

Lemma 5.1.7. *If $\varnothing; l \Vdash E^{l_o}[e]$ then $\varnothing; l \Vdash e$.*

Proof. By induction on the size of E^{l_o}. ∎

Lemma 5.1.8. *If $\varnothing; l \Vdash E^k[e]$, $\varnothing; k \nVdash e$ and $\varnothing; k \Vdash e'$ then $\varnothing; l \Vdash E^k[e']$.*

Proof. By induction on the size of E^k. ∎

Lemma 5.1.9. *If $\varnothing; l \Vdash E^k[e]$, $\varnothing; k \vDash e$ and $\varnothing; k \Vdash e'$ then $\varnothing; l \Vdash E^k[\|e'\|^k]$.*

Proof. By induction on the size of E^k. ∎

Lemma 5.1.10. *If* $\Gamma \uplus \{x : k\}; l \Vdash e$, $\Gamma \uplus \{x : k\}; k \Vdash \|e_0\|^k$, *then*

$$\Gamma \uplus \{x : k\}; l \Vdash \{\|e_0\|^k / x\}e.$$

Proof. By mutual induction on the height of $\Gamma \uplus \{x : k\}; l \Vdash e$ and $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 5.1.11. *If* $\Gamma; l \Vdash e$ *and* $x \notin dom(\Gamma)$, $\Gamma \uplus \{x : k\}; l \Vdash e$.

Proof. By mutual induction on the height of $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 5.1.12. *If* $\Gamma; l \Vdash e$ *and* $x \notin dom(\Gamma)$, *then* $x \notin \mathcal{FV}(e)$.

Proof. By mutual induction on the height of $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 5.1.13. *If* $\Gamma \uplus \{x : k\}; l \Vdash e$ *and* $x \notin \mathcal{FV}(e)$, *then* $\Gamma; l \Vdash e$.

Proof. By mutual induction on the height of $\Gamma \uplus \{x : k\}; l \Vdash e$ and $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 5.1.14. *If* $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \kappa$.

Proof. By induction on the height of $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$ using lemma 5.1.11 for the flat contracts case. ∎

Lemma 5.1.15. *If* $\Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa$ *and* $\Gamma \uplus \{x : k\}; l \Vdash \mathtt{mon}_j^{k,l}(\kappa', x)$, *then* $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \{\mathtt{mon}_j^{k,l}(\kappa', x)/^c x\}\kappa$.

Proof. By induction on the height of $\Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa$. For the flat contracts case we employ lemma 5.1.10. ∎

Lemma 5.1.16. *If* $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$, $\overline{k'} \subseteq \overline{k}$ *and* $\overline{l'} \subseteq \overline{l}$ *then* $\Gamma; \overline{k'}; \overline{l'}; j \rhd \kappa$.

PROOF IDEA. By induction on the height of $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

5.2 Optimizing Dependent Contracts

The *indy* semantics imposes a sufficient amount of contract checking for proving complete monitoring. To our surprise, though some of the checks turn out to be unnecessary.

Let us consider the following example:

$$\Pi^4 = \mathtt{mon}_j^{k,l}(\kappa, \lambda f.0)\ \lambda x.x - 1$$

where $\kappa = (P?_k \mapsto P?_l) \overset{d}{\mapsto} (\lambda f.\lfloor \mathtt{flat}(\|\lambda x.f\ 0 > 0\|^j)\rfloor^k)$ and $P?_l$ stands for the flat contract $\lfloor \mathtt{flat}(\|\lambda x.x > 0\|^j)\rfloor^l$. Π^4 is a perfectly well-formed program. It does not compute anything of interest but the contract has the interesting property that it applies the argument f to 0 and thus violates the precondition $P?_k \mapsto P?_l$. In turn, the *indy* semantics observes that 0 is not a positive number and thus accuses the contract itself for a contract violation.

However if the client owns the contract, i.e., j is equal to l, the monitor injected into the post-condition of κ has the same label l in the server and the client position. One could argue that this is an indication that f has not crossed a boundary but moved from one part of the client to another, and it is therefore not required to protect it with a contract. This observation suggests a variant of the reduction rules for dependent function contracts:

$$E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)] \longmapsto_s E^l[\lambda x.\mathtt{mon}_j^{k,l}(\underline{\kappa_2'}, v\ \mathtt{mon}_j^{l,k}(\kappa_1, x))] \qquad (\textit{semi})$$

where $\kappa_2' \equiv \{\mathtt{mon}_j^{l,j}(\kappa_1, x)/^c x\}\kappa_2$ if $j \neq l$, or $\kappa_2' \equiv \{x/^c x\}\kappa_2$ if $j \neq l$ otherwise

The *semi* semantics avoids the creation of contract monitors when the server and the client labels are the same. Furthermore, it assigns blame differently than *indy*. In fact for Π^4, *semi* does not discover any violation of the pre-condition of κ but it blames k when the post-condition fails. It is easy to show that *semi* may blame different components than *indy* and that it discovers more contract violations than *lax* and fewer than *indy*. Still, *semi* is a complete monitor.

Theorem 5.2.1. $\longmapsto_s$ *is a complete monitor.*

Proof. The proof is similar to the proof that *indy* is a complete monitor. The only difference is the case for dependent contracts in the preservation lemma. For *semi*, we distinguish between two sub-cases: $j \neq l$ and $j = l$. In the first sub-case, the proof is identical to the case for dependent contracts under *indy*. In the second sub-case, we derive the desired conclusion by following the same conclusion but applying lemma 5.1.15, we employ the following lemma 5.2.2. ∎

Lemma 5.2.2. *If* $\Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa$, *then* $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \{x/^c x\}\kappa$.

PROOF IDEA. By induction on the height of $\Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa$. For the flat contracts case we employ lemma 3.3.8. ∎

At this point the alert reader may wonder whether *semi* is preferable to *indy* or vice versa. Our framework suggests a philosophical answer. They behave differently only when *indy* constructs monitors at run-time where the server and the client label is the same. Such monitors enforce invariants described by contracts even in the interior of a component. According to the Findler and Felleisen philosophy of contracts and component-oriented programming, these invariants should be enforced only outside the component. These extra checks should thus be considered redundant and do not make the contract system stronger.

In fact we can also show that *semi* comes with the minimal amount of checking that suffices for complete monitoring. Since well-formed source code forces the server and client labels to be different and the reduction rule for dependent function contracts monitors is the only rule that creates monitors with new labels, the *semi* semantics always produces monitors with different client and server labels. Thus it injects monitors only where they are really necessary, at the boundaries between different components.

5.3 Mutation Needs Complete Monitors

The principle of complete monitoring provides guidance for the addition of linguistic features to CPCF. Concretely, consider the addition of reference cells, i.e., sharable, mutable data. Doing so requires both a notation for contracts on cells and also a mechanism that monitors all channels of communication between components that exchange cells.

We investigate this setting via CPCF!, an imperative variant of CPCF. The source syntax of CPCF!, specified in figure 5.2, extends the source syntax of CPCF with the standard operators of a language with mutable cells. CPCF! also comes with contracts for mutable cells, $\texttt{ref/c}(\kappa)$. Intuitively the contract specifies that the protected cell should conform at any point with κ. CPCF!, just like CPCF, is typed. The type system and its soundness pose no challenges and are omitted.

Types	τ	$=$	$\ldots \mid \mathsf{ref}(\tau)$	**Values**	v	$=$	$\ldots \mid \mathit{loc} \mid \gamma$
Contracts	κ	$=$	$\ldots \mid \texttt{ref/c}(\kappa)$	**Guards**	γ	$=$	$\texttt{G}\{v\ (\kappa\ l\ l\ l)\}$
Terms	e	$=$	$\ldots \mid \texttt{ref}(e) \mid \texttt{get}(e) \mid \texttt{set}(e,e)$				

Figure 5.2: CPCF!: source (left) and evaluation syntax (right)

The additions to the source syntax demand additions to the definitions of well-formed terms and contracts. The first are straightforward requiring that the arguments of the operators related to store are well-formed under the same owner as the operator:

$$\frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \texttt{ref}(e)} \qquad \frac{\Gamma; l \Vdash e}{\Gamma; l \Vdash \texttt{get}(e)} \qquad \frac{\Gamma; l \Vdash e_1 \quad \Gamma; l \Vdash e_2}{\Gamma; l \Vdash \texttt{set}(e_1, e_2)}$$

The second addition poses a small challenge. The same component can read from, and write to, a mutable cell. Thus the distinction between clients and servers of the contents of the cells collapses. To reflect this insight, the rule for well-formed contracts on cells merges the parties responsible for the negative and positive pieces

of the contract when assigning obligations for the contract that protects the contents of a cell. All parties $\overline{l}$ and $\overline{k}$ have the obligation to treat the contents according to the contract both as clients and servers:

$$\frac{\Gamma;\overline{lk};\overline{lk};\, j \rhd \kappa}{\Gamma;\overline{k};\overline{l};\, j \rhd \mathtt{ref/c}(\kappa)}$$

Mutable cells are represented at run-time as memory locations *loc*. To enforce contract checks on the contents of memory locations we have to delay checking until a component tries to read the location. For that reason we introduce guards $\mathtt{G}\{v\ (\kappa\ k\ l\ j)\}$ as intermediate terms. They are contract monitors similar to $\mathtt{mon}_j^{k,l}(\kappa,e)$. The difference is that in contrast with monitors, guards are values, and thus they attach themselves permanently around locations when the locations cross component boundaries. Figure 5.2 shows the intermediate syntax for CPCF! that extends the intermediate syntax of CPCF.

$$\begin{array}{lll}
E^l[\mathtt{ref}(v)],\sigma & \longmapsto_m & E^l[loc],\sigma' \\
\quad \text{where } loc \notin dom(\sigma) \text{ and } \sigma' = \sigma \uplus \{loc \mapsto v\} & & \\
E^l[\mathtt{get}(||loc||^l)],\sigma & \longmapsto_m & E^l[\|v\|^l],\sigma \\
\quad \text{where } \sigma = \sigma' \uplus \{loc \mapsto v\} & & \\
E^l[\mathtt{set}(||loc||^l,v')],\sigma & \longmapsto_m & E^l[\|loc\|^l],\sigma' \\
\quad \text{where } \sigma = \sigma'' \uplus \{loc \mapsto v\} \text{ and } \sigma' = \sigma'' \uplus \{loc \mapsto v'\} & & \\
E^l[\mathtt{mon}_j^{k,l}(\mathtt{ref/c}(\kappa),v)],\sigma & \longmapsto_m & E^l[\mathtt{G}\{v\ (\kappa\ k\ l\ j)\}],\sigma \\
E^l[\mathtt{get}(||\gamma||^l)],\sigma & \longmapsto_m & E^l[\|\mathtt{mon}_j^{k,l}(\kappa,\mathtt{get}(v))\|^l],\sigma \\
\quad \text{where } \gamma = \mathtt{G}\{v\ (\kappa\ k\ l\ j)\} & & \\
E^l[\mathtt{set}(||\gamma||^l,||v'||^l)],\sigma & \longmapsto_m & \\
\qquad E^l[\|(\lambda x.\mathtt{mon}_j^{k,l}(\mathtt{ref/c}(\kappa),\mathtt{set}(v,\mathtt{mon}_j^{l,k}(\kappa,x))))\ v''\|^l],\sigma & & \\
\quad \text{where } \gamma = \mathtt{G}\{v\ (\kappa\ k\ l\ j)\} \text{ and } v'' = \|v'\|^l & &
\end{array}$$

Figure 5.3: Operations on mutable data

The definition of the reduction relation for CPCF! demands some preparation. Locations require the addition of a store, which changes the shape of states. They now have two parts: e and σ. The reduction relation now describes transitions

between such states: $E^l[e], \sigma \longmapsto_m E^l[e'], \sigma'$. Moreover we derive additional evaluation contexts for the new operators just like for the primitive operators in CPCF.

The reduction rules for CPCF become reduction rules for CPCF! after adding the same store on both sides of each rule. The additional operations on mutable data, see figure 5.3, are straightforward when they are performed directly on store locations. They only fire when the context owns the location in order to guarantee that a component can read or write to properly acquired cells. Things become interesting when a component other than the creator and owner of the location tries to access or modify the location's contents. Doing so requires a guard $\mathtt{G}\{v\ (\kappa\ k\ l\ j)\}$. Guards are the result of a monitor of a contract $\mathtt{ref/c}(\kappa)$ on a value v. They contain the guarded value, the contract κ and the labels that decorate the monitor. A $\mathtt{get}(||\gamma||^l)$ opens the guard γ and delegates the `get` operation to the value that resides in γ. Moreover it wraps the result with a monitor built out of the contract and the labels from γ. This ensures that the contract is checked when the host component tries to use the value obtained from the location.

Writing a value v' to a mutable cell via a guard γ is also delegated to the value that resides in γ; v' is wrapped with the appropriate contract monitor. However, in this case the semantics must take into account two other factors. First, a `set` operation creates a flow of values in the opposite direction than a `get` operation. Thus the server for the new content of the location should be the client for the old one and vice versa. Second, the result of `set` should be the same guard γ as the one applied on the operation. This ensures that the location remains protected. To achieve this, the reduction rule reverses the labels on the monitor of the term that is written in the location and wraps the whole operation with a monitor that is going to reproduce γ. Finally the rule expands the operation into a function application so that v' becomes explicitly decorated with the label of the host component before written to the location.

Proving that *indy* CPCF! is a complete monitor follows the same pattern as for CPCF. We first adapt the definition of complete monitoring to a store semantics.

Definition 5.3.1 (Complete Monitors for CPCF!)**.** *A contract semantics m specifies a* complete monitor *iff for all well typed terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- $e_0, \varnothing \longmapsto_m^* v, \sigma_1$ *or,*
- *for all terms e_1 and stores σ_1 such that $e_0, \varnothing \longmapsto_m^* e_1, \sigma_1$ there exists term e_2 and store σ_2 such that $e_1, \sigma_1 \longmapsto_m e_2, \sigma_2$ or,*
- $e_0, \varnothing \longmapsto_m^* e_1, \sigma_1 \longmapsto_m^* \mathtt{error}_j^k, \sigma_2$*, e_1 is of the form $E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l}}, v)]$ and for all such terms e_1, $v = \|v_1\|^k$ and $k \in \overline{l}$.*

Then we generalize well-formedness for source code and contracts to intermediate terms and prove preservation and progress lemmas. Generalized well-formedness consists of two new judgments, $\Sigma; \Gamma; l \Vdash e$ and $\Sigma; \Gamma; \overline{k}; \overline{l}; j \rhd \kappa$.

The most important modification to the corresponding generalized judgment for well-formed CPCF terms and contracts is the introduction of store ownership, which establishes that the store is well-formed. The store ownership relates locations and owners. A store is well-formed if its content is well-formed under the owner store ownership points to:

$$\frac{\text{for all } loc \in dom(\Sigma),\ \Sigma; \varnothing; \Sigma(loc) \Vdash \sigma(loc)}{\Sigma \sim \sigma}$$

This is necessary for the same reason that store typing is necessary to prove type soundness for languages with mutable data; it admits circularity in the store.

The generalized judgment for well-formed terms and contracts is almost the same as the corresponding well-formed judgments in CPCF. The differences are the additional rules for store operations and that together with the environment, it propagates the ownership typing. There are also rules for guards and locations, the intermediate terms of CPCF!:

$$\frac{\Sigma(loc) = l}{\Sigma;\Gamma;l \Vdash loc} \qquad \frac{\Sigma;\Gamma;k \Vdash v \qquad \Sigma;\Gamma;\{k,l\};\{k,l\};j \rhd \kappa}{\Sigma;\Gamma;l \Vdash \mathtt{G}\{\|v\|^k\ (\kappa\ k\ l\ j)\}}$$

A location is well-formed only under the owner according to the store ownership. Well-formed guards are only those where the guarded value is explicitly annotated as owned by the component with the positive label (k) in the guard. Furthermore, the contract κ must be also well-formed. The last label (j) serves as the owner of the contract's code. Since guards are used to protect locations and locations can be used by components both for writing and reading, both the negative label l and positive label k must be responsible for the positive and negative pieces of κ.

Furthermore we need to extend the CPCF rules for loosely well-formed terms with store ownership. We also add two rules due to store related operations:

$$\frac{\Sigma;\Gamma;l \Vdash e}{\Sigma;\Gamma;l \vDash \mathtt{get}(\|e\|^l)} \qquad \frac{\Sigma;\Gamma;l \Vdash e_1 \qquad \Sigma;\Gamma;l \Vdash e_2}{\Sigma;\Gamma;l \vDash \mathtt{set}(\|e_1\|^l, e_2)}$$

The `get` and `set` operators are loosely well-formed if the term in the position of the mutable cell is tagged with the owner of the operation. The reduction semantics guarantees that reducing the operation produces a term explicitly owned by l.

We can now show that the *indy* semantics is a complete monitor for CPCF! using the technique of the preceding section.

Theorem 5.3.2. *$\longmapsto_i$ is a complete monitor.*

Proof. As a direct consequence of lemmas 5.3.3 and 5.3.4 we obtain that for all well typed terms e_0, if $\varnothing; l_o \Vdash e_0$,

- $e_0, \varnothing \longmapsto_i^* v, \sigma$ or,
- $e_0, \varnothing \longmapsto_i^* e_1, \sigma_1$ and for all e_1, $e_1 \longmapsto_i e_2, \sigma_2$ or,
- $e_0, \varnothing \longmapsto_i^* \mathtt{error}_j^k, \sigma_2$.

For the last case, since $\varnothing; l_o \Vdash e_0$ we know that $\mathtt{error}_j^k$ does not occur in e_0. Thus it must be a result of reduction. The only reduction rule that introduces $\mathtt{error}_j^k$ is:

$E_2^l[\mathtt{check}_j^k(||\mathtt{ff}||^j, v)], \sigma \longmapsto_m E_2^\dagger[\mathtt{error}_j^k], \sigma$. Also $E_1^\dagger[\mathtt{error}_j^k], \sigma \longmapsto_i \mathtt{error}_j^k, \sigma$. Again $\mathtt{check}_j^k(e, v)$ does not occur in e_0 because $\varnothing; l_o \Vdash e_0$. Hence it must be also a result of reduction. The only rule that introduces checks is:

$$E_1^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})], \sigma \longmapsto_i E_1^l[\mathtt{check}_j^k(e\ b, b)], \sigma.$$

From the above information we deduce:

$$\begin{aligned}
& e_0, \varnothing \longmapsto_i^* E_1^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})], \sigma_1 \\
& \longmapsto_i E_1^l[\mathtt{check}_j^k(e\ b, b)], \sigma_1 \\
& \longmapsto_i^* E_2^l[\mathtt{check}_j^k(||\mathtt{ff}||^j, v)], \sigma_2 \\
& \longmapsto_i E_2^l[\mathtt{error}_j^k], \sigma_2 \\
& \longmapsto_i \mathtt{error}_j^k, \sigma_2.
\end{aligned}$$

From lemma 5.3.4, we derive that for Σ_1 such that $\Sigma_1 \sim \sigma_1$,

$$\Sigma_1; \varnothing; l_o \Vdash E_1^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})].$$

Subsequently by lemma 5.3.6 we obtain $\Sigma_1; \varnothing; l \Vdash \mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})$. From the rules for well formed terms we get $||b||^{l'} = \|v_1\|^k$ and $k \in \overline{l'}$. Finally combining the above results entail that $\longmapsto_i$ is a complete monitor. ∎

Lemma 5.3.3. *(Progress) For all well typed terms e, well typed stores σ and store ownerships Σ such that $\Sigma; \varnothing; l \Vdash e$ and $\Sigma \sim \sigma$, $e = v$ or $e = \mathtt{error}_j^k$ or $e, \sigma \longmapsto_i e_0, \sigma'$.*

Proof. From lemma 5.3.5 we know that there exist e', l and E^l such that $e = E^l[e']$. Via lemma 5.3.6 and by $\Sigma; \varnothing; l_o \Vdash e$ we derive $\Sigma; \varnothing; l \Vdash e'$. We proceed by case analysis on the form of e':

- $e' = v_1 + v_2$: By type soundness for CPCF! we get that $v_1 = ||\mathtt{n}_1||^k$ and $v_2 = ||\mathtt{n}_2||^j$. Since $\Sigma; \varnothing; l \Vdash e'$, from the inference rules for well-formed terms we deduce $\Sigma; \varnothing; l \Vdash ||\mathtt{n}_1||^k$ and $\Sigma; \varnothing; l \Vdash ||\mathtt{n}_2||^j$. With additional uses of well-formedness we obtain $k = j$. Thus according to the reduction relation $e \longmapsto_i E^l[\mathtt{n}]$ where $n = n_1 + n_2$.

- The rest of the cases are similar to the first. ∎

Lemma 5.3.4. *(Preservation) For all terms e, stores σ and store ownerships Σ such that $\Sigma;\varnothing;l_o \Vdash e$ and $\Sigma \sim \sigma$ if $e,\sigma \longmapsto_i e_0,\sigma'$, there exists Σ' such that $\Sigma \subseteq \Sigma'$, $\Sigma' \sim \sigma'$ and $\Sigma';l_o;\varnothing \Vdash e_0$.*

Proof. We proceed by case analysis on the reduction of e:

- $E^l[||\mathtt{n}_1||^l + ||\mathtt{n}_2||^l],\sigma \longmapsto_i E^l[\mathtt{n}],\sigma$: By assumption $\Sigma \sim \sigma$ and $\Sigma;\varnothing;l_o \Vdash e$. For the latter lemma 5.3.6 implies that $\Sigma;\varnothing;l \Vdash ||\mathtt{n}_1||^l + ||\mathtt{n}_2||^l$. We can use the same label to check $\mathtt{n}$ via the inference rules, i.e., $\Sigma;\varnothing;l \Vdash \mathtt{n}$. Hence, from lemma 5.3.8 $\Sigma;\varnothing;l_o \Vdash E^l[\mathtt{n}]$.

- The cases for other primitive operations, conditionals, checks and errors are similar to the first.

- $E^l[\mu x.e],\sigma \longmapsto_i E^l[\{\|\mu x.e\|^l/x\}e],\sigma$: By assumption and lemma 5.3.6, we conclude that $\Sigma;\varnothing;l \Vdash \mu x.e$ and $\Sigma \sim \sigma$. The inference rules imply $\Sigma;\{x : l\};l \Vdash e$. Via lemmas 5.3.10, 5.3.12 and 5.3.13 we conclude that $\Sigma;\varnothing;l \Vdash \{\|\mu x.e\|^l/x\}e$. From lemma 5.3.8 we obtain $\Sigma;\varnothing;l_o \Vdash E^l[\{\|\mu x.e\|^l/x\}e]$.

- $E^l[v_f\ ||v||^l],\sigma \longmapsto_i E^l[\|\{\|v\|^l/x\}e_0\|^l],\sigma$ where $v_f = ||\lambda x.e_0||^l$: Again by assumption and with the help of lemma 5.3.6, we conclude that $\Sigma;\varnothing;l \Vdash v_f\ ||v||^l$ and $\Sigma \sim \sigma$. Therefore, $\Sigma;\varnothing;l \Vdash ||\lambda x.e_0||^l$ and $\Sigma;\varnothing;l \Vdash ||v||^l$. The inference rules imply $\Sigma;\{x : l\};l \Vdash e_0$. Since $\Sigma;\varnothing;l \Vdash v$ still holds, we conclude again via another use of lemmas 5.3.10, 5.3.12 and 5.3.13 that $\Sigma;\varnothing;l \Vdash \{\|v\|^l/x\}e_0$. Finally the desired conclusion $\Sigma;\varnothing;l_o \Vdash E^l[\|\{\|v\|^l/x\}e_0\|^l]$ is derived from the above antecedents and lemma 5.3.8 if $v_f = \lambda x.e_0$ or from the above antecedents and lemma 5.3.9 otherwise.

- Let $e = E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{flat}(e_c)\rfloor^{\overline{l'}}, ||b||^{l'})]$. The reduction rules dictate that

$$e,\sigma \longmapsto_i E^l[\mathtt{check}^k_j(e_c\ b,b)],\sigma.$$

The assumptions imply $\Sigma;\varnothing;l \Vdash \mathtt{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c)\rfloor^{\overline{l'}}, ||b||^{l'})$ via lemma 5.3.6 and $\Sigma \sim \sigma$. Hence, $e_c = \|e_c'\|^j$ with $\Sigma;\varnothing;j \Vdash e_c'$. Furthermore, the same reasoning yields $l' = k$ and $\Sigma;\varnothing;k \Vdash b$. We trivially obtain that $\Sigma;\varnothing;l \Vdash b$ and $\Sigma;\varnothing;j \Vdash b$. Thus $\Sigma;\varnothing;j \vDash \|e_c'\|^j\ b$. Since the rules for well-formed expressions imply $\Sigma;\varnothing;l \Vdash \mathtt{check}_j^k((e_c\ b),b)$ is well-formed, the desired conclusion follows immediately.

- Let $e = E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)]$. According to the reduction relation

$$e,\sigma \longmapsto_i E^l[\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\ \mathtt{mon}_j^{l,k}(\kappa_1,x))],\sigma.$$

  Applying the usual method, we get $\Sigma;\varnothing;l \Vdash \mathtt{mon}_j^{k,l}(\kappa_1 \mapsto \kappa_2, v)$, $v = \|v_0\|^k$, $\Sigma;\varnothing;k \Vdash v_0$, and $\Sigma \sim \sigma$. The contract check yields two pieces of knowledge:

$$\Sigma;\varnothing;\{l\};\{k\};j \rhd \kappa_1$$

  and

$$\Sigma;\varnothing;\{k\};\{l\};j \rhd \kappa_2.$$

  From an additional application of the inference rules for well-formedness we get $\Sigma;\{x : l\};k \vDash v\ \mathtt{mon}_j^{l,k}(\kappa_1,x)$ and, using lemmas 5.3.11 and 5.3.14, $\Sigma;\{x : l\};l \Vdash \mathtt{mon}_j^{k,l}(\kappa_2, v\ \mathtt{mon}_j^{l,k}(\kappa_1,x))$. Finally from a last application of the inference rules for well-formedness we get $\Sigma;\varnothing;l \Vdash \lambda x.\mathtt{mon}_j^{k,l}(\kappa_2, v\ \mathtt{mon}_j^{l,k}(\kappa_1,x))$.

- $E^l[\mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)],\sigma \longmapsto_i e_0,\sigma$ where

$$e_0 = E^l[\lambda x.\mathtt{mon}_j^{k,l}(\kappa_2', v\ \mathtt{mon}_j^{l,k}(\kappa_1,x))]$$

  and

$$\kappa_2' = \{\mathtt{mon}_j^{l,j}(\kappa_1,x)/^c x\}\kappa_2 :$$

  From assumptions we know that $\Sigma \sim \sigma$ and via lemma 5.3.6 we derive $\Sigma;\varnothing;l \Vdash \mathtt{mon}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v)$. Thus, $v = \|v\|^k$ with $\Sigma;\varnothing;k \Vdash v$, but also

$$\Sigma;\varnothing;\{l\};\{k,j\};j \rhd \kappa_1$$

and

$$\Sigma; \{x : j\}; \{k\}; \{l\}; j \rhd \kappa_2 \ .$$

The rest of this argument uses the same strategy as the preceding case, except that we use lemmas 5.3.15 and 5.3.16 to derive the key result, $\Sigma; \{x : l\}; l \Vdash \mathtt{mon}_j^{k,l}(\kappa_2', v\, \mathtt{mon}_j^{l,k}(\kappa_1, x))$.

- $E^l[\mathtt{mon}_j^{k,l}(\mathtt{ref/c}(\kappa), v)], \sigma \longmapsto_i E^l[\mathtt{G}\{v\ (\kappa\ k\ l\ j)\}], \sigma$: By assumptions and lemma 5.3.6 we obtain $\Sigma \sim \sigma$ and $\Sigma; \varnothing; l \Vdash \mathtt{mon}_j^{k,l}(\mathtt{ref/c}(\kappa), v)$. Using the inference rules for well formed terms we extract from the latter $v = \|v_0\|^k$, $\Sigma; \Gamma; k \Vdash v_0$ and $\Sigma; \Gamma; \{k,l\}; \{k,l\}; j \rhd \kappa$. Thus with another application of the inference rules for well formed terms we get the essential result $\Sigma; \varnothing; l \Vdash \mathtt{G}\{v\ (\kappa\ k\ l\ j)\}$.

- $E^l[\mathtt{ref}(v)], \sigma \longmapsto_i E^l[loc], \sigma \uplus \{loc \mapsto v\}$: Since $loc \notin dom(\sigma)$ and $\Sigma \sim \sigma$, from lemma 5.3.17 we get $\Sigma \uplus \{loc : l\}; \varnothing; l_o \Vdash E^l[\mathtt{ref}(v)]$. As in the previous cases from lemma 5.3.6, we conclude $\Sigma \uplus \{loc : l\}; \varnothing; l \Vdash v$. Via the definition of well formed store the above judgment implies $\Sigma \uplus \{loc : l\} \sim \sigma \uplus \{loc \mapsto v\}$. Finally from the inference rules for well formed terms we obtain $\Sigma \uplus \{loc : l\}; \varnothing; l \Vdash loc$.

- $E^l[\mathtt{get}(v_l)], \sigma \longmapsto_i E^l[\|v\|^l], \sigma$ where $\sigma = \sigma' \uplus \{loc \mapsto v\}$ and $v_l = ||loc||^l$: As in the previous cases, we derive $\Sigma \sim \sigma' \uplus \{loc \mapsto v\}$. Thus $\Sigma; \varnothing; l \Vdash v$. If $v_l = loc$ from lemma 5.3.8 we get the desired result $\Sigma; \varnothing; l_o \Vdash E^l[\|v\|^l]$ else we reach the same conclusion via lemma 5.3.9.

- $E^l[\mathtt{set}(||loc||^l, v')], \sigma \longmapsto_i E^l[\|loc\|^l], \sigma'$ where $\sigma = \sigma'' \uplus \{loc \mapsto v\}$ and $\sigma' = \sigma'' \uplus \{loc \mapsto v'\}$: Following the usual path, we derive $\Sigma \sim \sigma'' \uplus \{loc \mapsto v\}$ and $\Sigma; \varnothing; l \Vdash v'$. By the definition of well formed store we get $\Sigma \sim \sigma'' \uplus \{loc \mapsto v'\}$ and by the inference rules for well formed terms we conclude $\Sigma; \varnothing; l \Vdash ||loc||^l$. Finally we show that $\Sigma; \varnothing; l_o \Vdash E^l[\|loc\|^l]$ like in the previous case.

- $E^l[\mathtt{get}(||\gamma||^l)], \sigma \longmapsto_i E^l[\|\mathtt{mon}^{k,l}_j(\kappa, \mathtt{get}(v))\|^l], \sigma$ where $\gamma = \mathtt{G}\{v\ (\kappa\ k\ l\ j)\}$: By assumptions and lemma 5.3.6 we get $\Sigma \sim \sigma$ and $\Sigma; \varnothing; l \Vdash \mathtt{get}(||\gamma||^l)$. From the inference rules for well formed terms we derive $\Sigma; \varnothing; l \Vdash \gamma$, $v = \|v_0\|^k$, $\Sigma; \varnothing; k \Vdash v_0$ and $\Sigma; \Gamma; \{k,l\}; \{k,l\}; j \rhd \kappa$. Hence, we infer $\Sigma; \varnothing; k \vDash v$ and, via lemma 5.3.16, $\Sigma; \Gamma; \{k\}; \{l\}; j \rhd \kappa$. The above judgments imply $\Sigma; \Gamma; l \Vdash \|\mathtt{mon}^{k,l}_j(\kappa, \mathtt{get}(v))\|^l$ which entails our goal via lemma 5.3.8 or lemma 5.3.9 like in the previous case.

- $E^l[\mathtt{set}(||\gamma||^l, v')], \sigma \longmapsto_i e_0, \sigma$ where $\gamma = \mathtt{G}\{v\ (\kappa\ k\ l\ j)\}$, $e_0 = E^l[e_1]$ and

$$e_1 = \|(\lambda x.\mathtt{mon}^{k,l}_j(\mathtt{ref/c}(\kappa), \mathtt{set}(v, \mathtt{mon}^{l,k}_j(\kappa, x))))\ v'\|^l :$$

  With the same method as in the previous cases we obtain $\Sigma \sim \sigma$ and $\Sigma; \varnothing; l \Vdash \mathtt{set}(||\gamma||^l, v')$. From the inference rules for well formed terms we get $\Sigma; \varnothing; l \Vdash \gamma$, $v = \|v_0\|^k$, $\Sigma; \varnothing; k \Vdash v_0$, $\Sigma; \varnothing; l \Vdash v'$ and $\Sigma; \Gamma; \{k,l\}; \{k,l\}; j \rhd \kappa$. We put all the pieces together and we obtain

$$\Sigma; \Gamma \uplus \{x : l\}; l \Vdash \mathtt{mon}^{k,l'}_j(\mathtt{ref/c}(\kappa), \mathtt{set}(v, \mathtt{mon}^{l',k}_j(\kappa, x))).$$

  With another use of the inference rules for well formed terms we conclude $\Sigma; \Gamma; l \Vdash \lambda x.\mathtt{mon}^{k,l}_j(\mathtt{ref/c}(\kappa), \mathtt{set}(v, \mathtt{mon}^{l',k}_j(\kappa, x)))$. Thus, also, $\Sigma; \Gamma; l \Vdash e_1$. Finally the desired result $\Sigma; \Gamma; l_o \Vdash e_0$ is produced with the help of lemma 5.3.8 or lemma 5.3.9 like before. ∎

Lemma 5.3.5. *(Unique Decomposition) Let e and e' terms such that $e \neq v$ and $e \neq \mathtt{error}^k_j$, and $e' = v_0\, op\, v_1$ or $e' = \mathtt{zero?}(v)$ or $e' = \mathtt{if}\ v_0\ e_1\ e_2$ or $e' = \mathtt{check}^k_j(v_0, v_1)$ or $e' = v_1\ v_2$ or $e' = \mu x.e_0$ or $e' = \mathtt{ref}(v)$ or $e' = \mathtt{get}(v)$ or $e' = \mathtt{set}(v_1, v_2)$ or $e' = \mathtt{mon}^{k,l}_j(\kappa, v)$ or $e' = \mathtt{error}^k_j$, there are unique l' and $E^{l'}$ such that $e = E^{l'}[e']$.*

Proof. By induction on the size of e. ∎

Lemma 5.3.6. *If $\Sigma; \varnothing; l \Vdash E^k[e]$ then $\Sigma; \varnothing; k \Vdash e$.*

Proof. By induction on the size of E^k with the help of lemma 5.3.7 for the inductive case. ∎

Lemma 5.3.7. *If* $\Sigma;\varnothing;l \Vdash E^{l_o}[e]$ *then* $\Sigma;\varnothing;l \Vdash e$.

Proof. By induction on the size of E^{l_o}. ∎

Lemma 5.3.8. *If* $\Sigma;\varnothing;l \Vdash E^k[e]$, $\Sigma;\varnothing;k \nVdash e$ *and* $\Sigma;\varnothing;k \Vdash e'$ *then* $\Sigma;\varnothing;l \Vdash E^k[e']$.

Proof. By induction on the size of E^k. ∎

Lemma 5.3.9. *If* $\Sigma;\varnothing;l \Vdash E^k[e]$, $\Sigma;\varnothing;k \vDash e$ *and* $\Sigma;\varnothing;k \Vdash e'$ *then* $\Sigma;\varnothing;l \Vdash E^k[\|e'\|^k]$.

Proof. By induction on the size of E^k. ∎

Lemma 5.3.10. *If* $\Sigma;\Gamma \uplus \{x:k\};l \Vdash e$, $\Sigma;\Gamma \uplus \{x:k\};j \Vdash \|e_0\|^k$, *then*

$$\Sigma;\Gamma \uplus \{x:k\};l \Vdash \{\|e_0\|^k/x\}e.$$

Proof. By mutual induction on the height of $\Sigma;\Gamma \uplus \{x:k\};l \Vdash e$ and $\Sigma;\Gamma \uplus \{x:k\};\overline{k};\overline{l};j \rhd \kappa$. ∎

Lemma 5.3.11. *If* $\Sigma;\Gamma;l \Vdash e$ *and* $x \notin dom(\Gamma)$, $\Sigma;\Gamma \uplus \{x:k\};l \Vdash e$.

Proof. By mutual induction on the height of $\Sigma;\Gamma;l \Vdash e$ and and $\Sigma;\Gamma;\overline{k};\overline{l};j \rhd \kappa$. ∎

Lemma 5.3.12. *If* $\Sigma;\Gamma;l \Vdash e$ *and* $x \notin dom(\Gamma)$, *then* $x \notin \mathcal{FV}(e)$.

Proof. By mutual induction on the height of $\Sigma;\Gamma;l \Vdash e$ and $\Sigma;\Gamma;\overline{k};\overline{l};j \rhd \kappa$. ∎

Lemma 5.3.13. *If* $\Sigma;\Gamma \uplus \{x:k\};l \Vdash e$ *and* $x \notin \mathcal{FV}(e)$, *then* $\Sigma;\Gamma;l \Vdash e$.

Proof. By mutual induction on the height of $\Sigma;\Gamma \uplus \{x:k\};l \Vdash e$ and $\Sigma;\Gamma \uplus \{x:k\};\overline{k};\overline{l};j \rhd \kappa$. ∎

Lemma 5.3.14. *If* $\Sigma;\Gamma;\overline{k};\overline{l};j \rhd \kappa$ *and* $x \notin dom(\Gamma)$, *then* $\Sigma;\Gamma \uplus \{x:k\};\overline{k};\overline{l};j \rhd \kappa$.

Proof. By induction on the height of $\Sigma;\Gamma;\overline{k};\overline{l};j \rhd \kappa$ using lemma 5.3.11 for the flat contracts case. ∎

Lemma 5.3.15. *If* $\Sigma;\Gamma \uplus \{x : j\};\overline{k};\overline{l};j \rhd \kappa$ *and* $\Sigma;\Gamma \uplus \{x : k\};l \Vdash \mathtt{mon}_j^{k,l}(\kappa', x)$, *then* $\Sigma;\Gamma \uplus \{x : k\};\overline{k};\overline{l};j \rhd \{\mathtt{mon}_j^{k,l}(\kappa', x)/^c x\}\kappa$.

Proof. By induction on the height of $\Sigma;\Gamma \uplus \{x : j\};\overline{k};\overline{l};j \rhd \kappa$. For the flat contracts case we employ lemma 5.3.10. ∎

Lemma 5.3.16. *If* $\Sigma;\Gamma;\overline{k};\overline{l};j \rhd \kappa$, $\overline{k'} \subseteq \overline{k}$ *and* $\overline{l'} \subseteq \overline{l}$ *then* $\Sigma;\Gamma;\overline{k'};\overline{l'};j \rhd \kappa$.

Proof. By induction on the height of $\Sigma;\Gamma;\overline{k};\overline{l};j \rhd \kappa$. ∎

Lemma 5.3.17. *Let store* σ *and store ownership* Σ *such that* $\Sigma \sim \sigma$. *If* $\Sigma;\Gamma;l \Vdash e$ *and* $loc \notin dom(\sigma)$, $\Sigma \uplus \{loc : k\};\Gamma;l \Vdash e$.

Proof. By induction on the height of $\Sigma \uplus \{loc : k\};\Gamma;l \Vdash e$. ∎

5.4 Complete Monitors and the Blame Theorem

Typed Racket [68] enables mixing typed modules with untyped Racket modules. Type-like contracts prevent untyped code from violating the type discipline when interacting with typed code. Tobin-Hochstadt and Felleisen [67] define and prove the soundness of this approach in a multi-lingual setting via a so-called Blame Theorem, a name due to Wadler and Findler [70], which establishes that a program execution can only raise contract violations in the untyped part.

To prove type soundness for an imperative version of this system we create an untyped sister language of CPCF!, UCPCF!, with a shared term syntax, and prove the corresponding blame theorem exploiting complete monitoring for CPCF!. As CPCF! has only base types, function types, and reference types, it suffices to

consider only the corresponding contracts:

$$\kappa = \lfloor \mathtt{I} \rfloor^{\bar{l}} \mid \lfloor \mathtt{B} \rfloor^{\bar{l}} \mid \kappa \mapsto \kappa \mid \mathtt{ref/c}(\kappa)$$

This restriction enables a series of additional simplifications in our framework. First, flat contracts contain only built-in predicates and not arbitrary code. Thus their code is not the property of any specific party. This decision is reflected in simplified rules for well-formed flat contracts:

$$\frac{}{\Gamma;\bar{k};\bar{l};j \rhd \lfloor \mathtt{I} \rfloor^{\bar{k}}} \qquad \frac{}{\Gamma;\bar{k};\bar{l};j \rhd \lfloor \mathtt{B} \rfloor^{\bar{k}}}$$

Second, the omission of dependent function contracts makes the distinction between *lax*, *picky* and *indy* irrelevant. We use $\longmapsto$ without any subscript to denote the reduction relation for UCPCF!.

Third, checking of flat contracts does not require the special check construct:

$$\begin{array}{ll}
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{I} \rfloor^{\bar{l'}}, ||\mathtt{n}||^{l'})], \sigma \longmapsto E^l[\mathtt{n}], \sigma & \\
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{I} \rfloor^{\bar{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[\mathtt{error}_j^k], \sigma & \text{if } b \neq \mathtt{n} \\
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{B} \rfloor^{\bar{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[b], \sigma & \text{if } b \in \{\mathtt{tt}, \mathtt{ff}\} \\
E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{B} \rfloor^{\bar{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[\mathtt{error}_j^k], \sigma & \text{if } b \notin \{\mathtt{tt}, \mathtt{ff}\}
\end{array}$$

The untyped nature of UCPCF! obliges us to extend the reduction relation of the language. Type soundness for CPCF! allowed us to ignore redexes such as $||v_1||^l \; ||v_2||^l$ where v_1 is not a function. In UCPCF! such states *can* occur. We deal with them by introducing dynamic type errors $\mathtt{error}^l_{\mathcal{T}}$ where l is the owner of the hole in which the ill-formed redex occurs.

This change must be propagated to our definition of complete monitoring. The definition of the property includes an extra case for run-time type errors.

Definition 5.4.1 (Complete Monitors for UCPCF!)**.** *A contract semantics m specifies a* complete monitor *iff for all terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- $e_0, \varnothing \longmapsto_m^* v, \sigma$ *or,*
- $\underline{e_0, \varnothing \longmapsto_m^* \mathtt{error}^l_{\mathcal{T}}, \sigma}$ *or,*
- *for all terms* e_1 *and stores* σ_1 *such that* $e_0, \varnothing \longmapsto_m^* e_1, \sigma_1$ *there exists term* e_2 *and store* σ_2 *such that* $e_1, \sigma_1 \longmapsto_m e_2, \sigma_2$ *or,*
- $e_0, \varnothing \longmapsto_m^* e_1, \sigma_1 \longmapsto_m^* \mathtt{error}^k_j, \sigma_2$, $\underline{j \neq \mathcal{T}}$, e_1 *is of the form* $E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{I} \rfloor^{\overline{l'}}, v)]$ *or* e_1 *is of the form* $E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{B} \rfloor^{\overline{l'}}, v)]$ *and for all such* e_1, $v = \|v_1\|^k$ *and* $k \in \overline{l}$.

The addition of run-time type errors does not eliminate all stuck states. The single owner policy still must hold for a redex to reduce. We can show, though, that these stuck states are not reachable and establish that $\longmapsto$ *is* a complete monitor for UCPCF!.

Theorem 5.4.2. $\longmapsto$ *is a complete monitor.*

Proof. As a direct consequence of lemmas 5.4.3 and 5.4.4 we obtain that for all terms e_0, if $\varnothing; l_o \Vdash e_0$,

- $e_0, \varnothing \longmapsto^* v, \sigma$ or,
- $\underline{e_0, \varnothing \longmapsto^* \mathtt{error}^l_{\mathcal{T}}, \sigma}$ or,
- for all terms e_1 and stores σ_1 such that $e_0, \varnothing \longmapsto^* e_1, \sigma_1$ there exists term e_2 and store σ_2 such that $e_1, \sigma_1 \longmapsto e_2, \sigma_2$ or,
- $e_0, \varnothing \longmapsto^* e_1, \sigma_1 \longmapsto^* \mathtt{error}^k_j, \sigma_2$ where $\underline{j \neq \mathcal{T}}$.

For the last case, since $\varnothing; \varnothing; l_o \Vdash e_0$ we know that $\mathtt{error}^k_j$ does not occur in e_0. Thus it must be a result of reduction. The only reduction rules that introduce $\mathtt{error}^k_j$ are:

$$E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{I} \rfloor^{\overline{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[\mathtt{error}^k_j], \sigma \quad \text{if } b \neq \mathtt{n}$$
$$E^l[\mathtt{mon}^{k,l}_j(\lfloor \mathtt{I} \rfloor^{\overline{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[\mathtt{error}^k_j], \sigma \quad \text{if } b \notin \{\mathtt{tt}, \mathtt{ff}\}.$$

From the above information we deduce:

$$e_0 \longmapsto E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{I} \rfloor^{\overline{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[\mathtt{error}_j^k], \sigma \longmapsto \mathtt{error}_j^k, \sigma$$
$$e_0 \longmapsto E^l[\mathtt{mon}_j^{k,l}(\lfloor \mathtt{I} \rfloor^{\overline{l'}}, ||b||^{l'})], \sigma \longmapsto E^l[\mathtt{error}_j^k], \sigma \longmapsto \mathtt{error}_j^k, \sigma.$$

From the rules for well formed terms we get $||b||^{l'} = \|v_1\|^k$ and $k \in \overline{l'}$ and the extra conditions of the last case are satisfied trivially. Finally combining the above results entail that $\longmapsto$ is a complete monitor. ∎

Lemma 5.4.3. *(Progress) For all terms e, stores σ and store ownerships Σ such that* $\Sigma; \varnothing; l \Vdash e$ *and* $\Sigma \sim \sigma$, $e = v$ *or* $e = \mathtt{error}_j^k$ *or* $e, \sigma \longmapsto e_0, \sigma'$.

Proof. The proof is similar to that of lemma 5.3.3. The differences are the following:

1. Additional cases for not well typed terms that reduce to a term with a dynamic type error in the hole of the evaluation context.

2. Modified cases for well typed terms where the redex is a monitor of a flat contract. These redexes reduce to a term with a value or a contract error in the hole of the evaluation context. ∎

Lemma 5.4.4. *(Preservation) For all terms e, stores σ and store ownerships Σ such that* $\Sigma; \varnothing; l_o \Vdash e$ *and* $\Sigma \sim \sigma$ *if* $e, \sigma \longmapsto e_0, \sigma'$, *there exists* Σ' *such that* $\Sigma \subseteq \Sigma'$, $\Sigma' \sim \sigma'$ *and* $\Sigma'; l_o; \varnothing \Vdash e_0$.

Proof. The proof is similar to that of lemma 5.3.4. The differences are the following:

1. Additional cases for reductions that produce dynamic type errors in the hole of the evaluation context. These reductions trivially produce well-formed terms.

2. Additional cases for reductions for flat contracts that do not produce type errors. These reductions trivially produce well-formed terms. ∎

Since CPCF! and UCPCF! share the same source code syntax, there is a subset of UCPCF! programs that are well-typed under CPCF!'s sound type system. We can extend CPCF!'s type system to allow for embedding of untyped UCPCF! code. For simplicity we assume that there are only two component labels: u for untyped code and t for typed code. We use $\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash e : \tau$ to express that a term e has type τ given type environment $\mathcal{G}$, store typing $\mathcal{S}$ and untyped locations $\mathcal{S}^u$:

$$\frac{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash e}{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash \mathtt{mon}_j^{u,t}(\kappa, e) : \mathcal{T}[\![\kappa]\!]} \qquad \frac{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash v}{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash \mathtt{G}\{v\ (\kappa\ u\ t\ j)\} : \mathcal{T}[\![\kappa]\!]}$$

The meta-function $\mathcal{T}$ maps a contract to the corresponding type. For flat contracts, $\mathcal{T}[\![\lfloor \mathtt{I} \rfloor^{\bar{k}}]\!] = \mathcal{N}$ and $\mathcal{T}[\![\lfloor \mathtt{B} \rfloor^{\bar{k}}]\!] = \mathcal{B}$.

The judgment $\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash e$ denotes that any typed code embedded in untyped code is well-typed. The judgment structurally decomposes e. Things become more interesting when a sub-term is typed:

$$\frac{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash e : \mathcal{T}[\![\kappa]\!]}{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash \mathtt{mon}_j^{t,u}(\kappa, e)} \qquad \frac{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash v : \mathcal{T}[\![\mathtt{ref/c}(\kappa)]\!]}{\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash \mathtt{G}\{v\ (\kappa\ t\ u\ j)\}}$$

Free variables and locations in typed code can only originate from typed code. $\mathcal{G}$ and $\mathcal{S}$ record this information. This goes hand in hand with the idea that a well-formed term can refer only to variables and locations of the same owner as the term and writing and reading foreign mutable cells can be done only through guards. In turn, $\mathcal{S}^u$ tracks all the untyped locations. We use it to make sure that typed and untyped locations are disjoint.

We can now state and prove the Blame Theorem.

Theorem 5.4.5. *(Blame Theorem) For all UCPCF! terms e_0 such that $\varnothing, \varnothing, \varnothing \vdash e_0$ and $\varnothing; u \Vdash e_0$, $e_0, \varnothing \not\longrightarrow^* \mathtt{error}_j^t, \sigma_0$.*

Proof. The proof is a direct consequence of a standard reduction technique based on the typed/untyped preservation lemma 5.4.6 and of complete monitoring for UCPCF!. For the preservation lemma we need to define an extension of store typing and store correspondence:

$$\frac{dom(\mathcal{S}) \cap \mathcal{S}^u = \varnothing \qquad dom(\mathcal{S}) \cup \mathcal{S}^u = dom(\sigma) \quad \text{for all } loc \in dom(\mathcal{S}),\ \mathcal{S}, \mathcal{S}^u, \varnothing \vdash \sigma(loc) : \mathcal{S}(loc)}{\mathcal{S}, \mathcal{S}^u \sim \sigma}$$

Lemma 5.4.6. *(Typed/Untyped Preservation) If* $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e$, $\Sigma; \varnothing; u \Vdash e$, $\mathcal{S}, \mathcal{S}^u \sim \sigma$, *and* $e, \sigma \longmapsto e', \sigma'$ *then* $\mathcal{S}', \mathcal{S}^{u\prime} \sim \sigma'$, $\mathcal{S} \subseteq \mathcal{S}'$, $\mathcal{S}^u \subseteq \mathcal{S}^{u\prime}$, *and* $\mathcal{S}', \mathcal{S}^{u\prime}, \varnothing \vdash e'$.

Proof. We proceed by induction on the length of the evaluation trace. Type soundness for CPCF! and complete monitoring for UCPCF! help us reduce the case analysis to three disjoint groups: redexes that involve only untyped terms are not monitors or guards, redexes that consist only of typed terms that are not monitors or guards, and redexes that manipulate monitors or guards. The first category can only produce type or contract errors blaming the untyped code. The second category can produce no errors. The third category deserves a detailed analysis. Lets look at these three cases one by one:

1. The redex e involves only untyped code terms that are not monitors or guards. We know that $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e$. By complete monitoring and the single owner policy it enforces we deduce that such a reduction can either raise $\texttt{error}^u_{\mathcal{T}}$ if e is not well typed or step to an untyped term e'. By complete monitoring, $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e'$ and hence, we conclude that the result of the reduction preserves the subject.

2. The redex e involves only typed code terms that are not monitors or guards. We know that $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e : \tau$ for some τ. By complete monitoring and type soundness for CPCF! we deduce that such a redex can only step to a typed

term e' such that $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e' : \tau$. Like in the previous case we conclude that the result of the reduction preserves the subject.

3. The redex involves monitors or guards. Since typed and untyped code have different owners we know, from complete monitoring, that these reductions concern the manipulation of contract monitors between typed and untyped code. For each such reduction we need to consider both the sub-case where the positive label on the monitor or guard is u and the negative label is t and the reverse case. Because of blame correctness, the first sub-case arises when typed code is executed and the second when untyped code is executed. Note that for all ill-formed redexes that can show up in the untyped sub-case, a type error $\texttt{error}^u_{\mathcal{T}}$ is raised and the subject is preserved trivially. Ill-formed redexes cannot show up in typed subcase by the definition of $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e_0$. Thus an $\texttt{error}^t_{\mathcal{T}}$ cannot be raised in this case either. So in any case we do not need to further consider reductions that raise type errors. Let us see here both sub-cases for the most interesting reduction:

   - Evaluation of untyped code:

     $$E^u[\texttt{set}(||\gamma||^u, v')], \sigma \longmapsto E^u[e'], \sigma$$

     where $v' = ||v_0||^l$, $\gamma = \texttt{G}\{v\ (\kappa\ t\ u\ j)\}$, $v'' = \|v_0\|^l$ and

     $$e' = \|(\lambda x.\texttt{mon}^{t,u}_j(\texttt{ref/c}(\kappa), \texttt{set}(v, \texttt{mon}^{u,t}_j(\kappa, x))))\ v''\|^u.$$

     The choice of the labels in the resulting term is due to complete monitoring. All we need to show in this case is that

     $$\mathcal{S}, \mathcal{S}^u, \varnothing \vdash \texttt{mon}^{t,u}_j(\texttt{ref/c}(\kappa), \texttt{set}(v, \texttt{mon}^{u,t}_j(\kappa, x))) : \mathcal{T}[\![\texttt{ref/c}(\kappa)]\!].$$

     By inversion, this boils down to proving that $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash v : \mathcal{T}[\![\texttt{ref/c}(\kappa)]\!]$ and $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash \texttt{mon}^{u,t}_j(\kappa, x) : \mathcal{T}[\![\kappa]\!]$. The first is a consequence of the

fact that the left hand side of the reduction meets the subject and the second falls out for the definition of the subject. Thus in this sub-case the subject is preserved.

- Evaluation of typed code:

$$E^t[\mathtt{set}(||\gamma||^t, v')], \sigma \longmapsto E^t[e'], \sigma$$

where $e' = \|(\lambda x.\mathtt{mon}_j^{u,t}(\mathtt{ref/c}(\kappa), \mathtt{set}(v, \mathtt{mon}_j^{t,u}(\kappa, x))))\ v''\|^t$, $v' = ||v_0||^l$, $v'' = \|v_0\|^l$ and $\gamma = \mathtt{G}\{v\ \ (\kappa\ \ u\ \ t\ \ j)\}$. Again the label unification is dictated by complete monitoring. We must establish $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash e' : \mathcal{T}[\![\mathtt{ref/c}(\kappa)]\!]$. For this it suffices to show that $\mathcal{S}, \mathcal{S}^u, \{x : \tau\} \vdash x : \mathcal{T}[\![\kappa]\!]$. In order for the last judgment to hold τ must be equal to $\mathcal{T}[\![\kappa]\!]$. We obtain this from the construction of the λ-term and the assumption that $\mathcal{S}, \mathcal{S}^u, \varnothing \vdash v' : \mathcal{T}[\![\kappa]\!]$. We conclude that in this last sub-case the subject is preserved. ∎

The proof of the theorem benefits greatly from complete monitoring as it allows us to reduce the space of the proof cases. For instance when typed code retrieves values from the store, complete monitoring guarantees that those are either the property of typed code and thus, from type soundness for CPCF!, they are well-typed, or they come from the untyped code and thus they are wrapped in a contract monitor. This observation reduces the proof cases essentially to only those that create new contract monitors. There we utilize the subject introduced in this section to make sure that the new monitors that contain terms from the typed party are protecting the code with contracts that correspond to their type.

In essence the proof of the Blame Theorem says that typed terms e can only show up inside monitors of the form $\mathtt{mon}_j^{t,u}(\kappa, e)$ and that for some $\mathcal{S}$, $\mathcal{S}^u$ and $\mathcal{G}$, $\mathcal{S}, \mathcal{S}^u, \mathcal{G} \vdash e : \mathcal{T}[\![\kappa]\!]$. Since type safety guarantees that type errors $\mathtt{error}_{\mathcal{T}}^t$ do not emerge in any case, we must simply rule out contract errors blaming the typed

code. From complete monitoring, this requires a failure of a contract check of the form $\mathtt{mon}_j^{t,u}(\kappa, ||b||^t)$ where κ is a flat contract. However, this is impossible since $\varnothing, \varnothing, \varnothing \vdash b : \mathcal{T}[\![\kappa]\!]$ and by the semantics for flat contract monitors and the translation of contracts to types no such check can fail. Thus no error blaming the typed code ever occurs.

CHAPTER 6

Option Contracts with Complete Monitoring[13]

6.1 Contracts: Correctness or Efficiency?

When programming languages provide expressive power to specify and enforce invariants, good programmers try to exploit every bit of it. Thus, hackers like Haskell [52] because its type system is a novel kind of tool for static checking. Programmers compelled to work with mainstream languages exploit assertions to dynamically check the state of executions at important steps. In languages with contracts—assertions on interfaces between components—programmers often start with simple checks that ensure basic properties and work their way toward complete correctness.

Since checking complete correctness tends to impose a large computational cost, algorithmicists have developed so-called *spot checkers*, algorithms that probabilistically check the correctness of the results of functions and methods and the contents of data structures [2, 5, 14, 17, 28, 29]. In the world of higher-order contracts [32], we have observed a similar phenomenon with the use of random testing of objects or closures passed to or returned from functions and methods, such as the example of the contract for `d/dx` from sections 3.1 and 4.1.

[13] This chapter reports work in collaboration with Ahmed, Sabry and Felleisen [25].

Consider, another example: the contract of a search function on a large vector v. Binary search works efficiently in this case but requires that v is sorted. If we add a sorted? contract that checks all of v, we radically alter its complexity. An alternative specify that v is locally sorted. See figure 6.1 for the essence of this contract, written in Racket's contract notation [35] where we have represented v with an access function.

```
;; searches a sorted vector v in [0,N) for element e
;; v is represented as a mapping from [0,N) to reals
[search
 (->i ([e number?]
       [N index?]
       [v (N) ((->sorted@i N) index? number?)])
      #:pre (v N) (for/and ((count K)) [v (random N)])
      [index-of-e (N) (or/c false/c (</c N))])]

...

;; contract: ensures for each access v[i] that
;;      v[i-1] <= v[i] <= v[i+1]
;; (if v[i-1] and v[i+1] exist)
(define ((->sorted@i s) dom rng) ...)
```

Figure 6.1: Binary search, with randomly checked contracts

Since contracts are coded in the full-fledged language, we can supplement it with an additional #:pre-condition (underlined) that looks at K randomly chosen points in v. If this Boolean loop produces true, we know that v is sorted around these random points because the application [v (random N)] checks the ->sorted@i property. While this contract is less precise than an assertion that v is completely sorted, it does increase our confidence and it does not affect the asymptotic complexity of search.

Unfortunately the injection of random testing into conventional interface contracts adds uncertainty to the software development and maintenance process. In

particular, if a contract fails, the contract system should not blame just a single component—after all, other components consume randomly tested values as if they are checked for the advertised property. Hence the error could have originated somewhere else, meaning the blame assignment of the contract system can no longer serve as a starting point for the debugging process. Worse, if the creators of components do not trust the random testing in the contract of some server component, they cannot opt out. The contract system lacks the expressiveness to just say no.

This chapter presents a contract system that overcomes these problems and naturally accommodates the marriage of contracts and probabilistic (or random) checking. The key innovation is the contract combinator `option/c`, which combines a contract `c` and a testing procedure `t` into an *option* contract. The contract system checks whether a value `v` satisfies `c` by applying `t` to `v` guarded with `c`. Once `t` succeeds, `c` is disabled until a consumer of the value exercises the contract option and reinstates `c`.

Option contracts correspond to options in the business world in the same way as Eiffel's contracts are analogous to business contracts. An option contract is not in force until the option is exercised but the option can be traded on its own. In the world of software components, a component may import a value via an option contract by **`accept`**ing it as is or by exercising the option. An **`accept`** means that the importing component agrees to shoulder joint responsibility for this value. Exercising an option means that the contract system extracts contract `c` and uses it to monitor the value's behavior. If anything goes wrong after an option is exercised, the contract system will blame *all components* that **`accept`**ed to share responsibility for an obligation.

With an option contract system, programmers can freely use probabilistic spot checkers and random testing in contracts to lower the cost of contract checking. The

```
#lang racket+options ;; revised-binary

(provide
  (contract-out
   ;; searches a sorted vector v in [0,N) for element e
   ;; v is represented as a mapping from [0,N) to reals
   [search
    (->i ([e number?]
                [N index?]
                [v (N) (option/c ((->sorted@i N) index? number?)
                                 (spot-check-v N))]
         [index-of-e (N) (or/c false/c (</c N))]))]))

;; contract: ensures for each access v[i] that
;;      v[i-1] <= v[i] <= v[i+1]
;; (if v[i-1] and v[i+1] exist)
(define ((->sorted@i s) dom rng) ...)

;; exercises the contract on v for 10 indexes in [0,N)
(define ((spot-check-v N) v)
  (for/and ((count (log N)))
    (define index (random N))
    (= (search [v index]) index)))
...
```

Figure 6.2: Binary search, with option contracts

use of `option/c` signals to both the client of a component and the contract system that properties are partially checked. A client can protect itself by exercising the option contract. The contract system knows to blame all components that propagate some randomly checked value.

The rest of the chapter starts with a section that illustrates the design of option contracts with some example. Sections 6.3 and 6.4 formalize the semantics of the new contract system and prove that it satisfies the "complete monitor" property from chapter 4. Section 6.5 presents some evidence that option contracts are practically feasible and relevant.

6.2 Option Contracts by Example

To understand the pragmatics of option contracts, we have implemented an extension of Racket—called `racket+options`—with two additions to the contract system: `option/c` and **`accept`**. The first combines a contract `c` with a testing predicate `t`. When a module exports or imports a function (or vector or object) `f` via an option contract, the contract system constructs a wrapper function from `c` and `f` and supplies the wrapper to the tester. If the wrapper signals a mistake or if the tester produces `false`, the contract system signals a contract violation. If it succeeds, the contract system returns the wrapper with the *contract disabled*. The second construct is **`accept`**, which imports an option contract *without* exercising the option. In contrast, the regular Racket import specification, called **`require`**, exercises the contract option, meaning it *enables* `c` and uses it to monitor the behavior of `f`. This section uses illustrative examples to demonstrate the workings of `option/c` and **`accept`**; it does not aim to explain their pragmatics.

Figure 6.2 sketches the `revised-binary` module, which is a realistic reformulation of the interface fragment in figure 6.1 based on a spot-checker [28]. The tester picks $\log N$ random vector indexes where N is the size of the vector, extracts the respective elements and `search`es for these elements. The test fails if any of the binary searches return an index different from the chosen one. More importantly, for each access into the vector, the procedure ensures that the vector is locally sorted. The spot-checker fails with probability $3/4$ for vectors without a sorted segment of length $(1 - 1/\log N) \cdot N$; its complexity is $O((\log n)^2)$, i.e., its effect on the complexity of binary search is tolerable.

Recall that once the option contract has tested the incoming vector with the predicate `spot-check-v`, the contract system hands over the vector to `search` with a *disabled contract*. Hence, `search` can use `v` without running any checks concerning local sorting. In short, option contracts allow programmers to run functions

without further checking because the tester checks the specification to (whatever is considered) a reasonable degree.

```
#lang racket+options ;; math

;; check monotonicity on real?
(define (monotone/c)
  (let ([seen (sorted-table)])
    (->i ([x real?])
         (r (x) (and real? (insert-and-check seen x))))))

(provide
  (contract-out
   [exp (option/c (monotone/c) apply-100-random-inputs)]
   ...
   [adaptive
    (->i ([f (monotone/c)])
         (intgr (f)
                (->i ([L real?] [R real?]) (A real?)
                     #:post (intgr L R A)
                            (approx? intgr f L R A))))]))

;; an exponentiation function
(define (exp x) ...)
...
;; an adaptive integration algorithm
(define (adaptive f)
  (λ (L R)
    ...))
```

Figure 6.3: A mathematics module

While the first option contract example works for exports from the client to the service provider, **`accept`** specifications concern imports from the server to the client. To demonstrate the workings of **`accept`**, we use a second example motivated by high-school calculus. The example begins with the `math` module in figure 6.3.

The `math` module exports the usual mathematical functions, e.g., `exp`, as well as an adaptive integration operator for monotonic functions on the reals. The con-

tract `monotone/c` uses a (local) table to keep track of all arguments and results encountered so far and ascertains that this finite snapshot of the function is appropriately sorted. The `exp` function is exported via an option contract that combines `monotone/c` and a tester that generates 100 random inputs and applies `exp` to them. The `adaptive` operator consumes a `monotone/c` function `f` and produces a function `intgr`, which consumes two reals and produces a real that approximates the area under the graph of `f` between the two boundaries.

Now image a module that **`require`**s `math` and computes the indeterminate integral of `exp` using `adaptive` for export:

```
#lang racket+options ;; integr

(accept math)
(provide exp exp-area)
(define exp-area (adaptive exp))
```

In the absence of option contracts, the `integr` module would have to pay for `monotone/c` twice: once because of `exp`'s contract and a second time because of `adaptive`'s argument contract. Option contracts offer an alternative. Instead of **`require`**, `integr` uses **`accept`** to import `exp` and relies on the tester of its contract; the price is that it accepts responsibility for `exp`'s behavior.

Even clients of `integr` may **`accept`** `exp` as is:

```
#lang racket+options ;; area

(accept integr)
(provide exp)
...
```

A `client` module may import `area` and use `exp`:

```
#lang racket+options ;; geometry

(require area)
... (exp x) ...
```

Since `client` imports `exp` via **`require`**, the contract system activates checking for `exp`. The question is who gets blamed if anything goes wrong with an application of `exp`.

In order to understand the answer, we need to look at the management of responsibility information. When a module **`accept`**s a value with an option contract, the contract system adds the module to the responsible parties for uses of the imported value. Similarly, when a module re-**`provide`**s a value, it is added as a responsible party for the value's services. Based on this explanation, we can see that if `client` abuses `exp` with respect to its contract, the monitoring system blames both `area` and `integr` for these bad uses. In the same vein, if the contract system discovers that `exp` is not monotonic, it blames both `math` and `integr` for providing bad services. In short, **`accept`**ing option contracts comes with the price of shared responsibility.

Figure 6.4 depicts the interaction between the various modules in a graphical. All four modules contain a possibly wrapped version of `exp`, depicted as a box. The boundaries between modules are single vertical lines if the import takes place via **`accept`**. A double-line boundary represents an import via **`require`**. The diagram shows how the contract system wraps `exp` as it passes from `math` to `integr`. The wrapper does not enforce contract checking but keeps track of the service provider `math` and the client `integr`. When `exp` migrates from `integr` to `area`, the contract system still does not activate contract checking but adds `integr` and `area` to the list of server and client parties, respectively. Finally the wrapped `exp` reaches `geometry` where the contract system turns contract checking on—shown as a double margin around `exp`—and keeps the existing list of contract parties. In short, the contract system manages to keep track of the migration history of `exp` even while it does not enforce the corresponding contract. When the option is exercised, the names of all responsible parties are available and can be used to

launch a bug search if anything goes wrong.

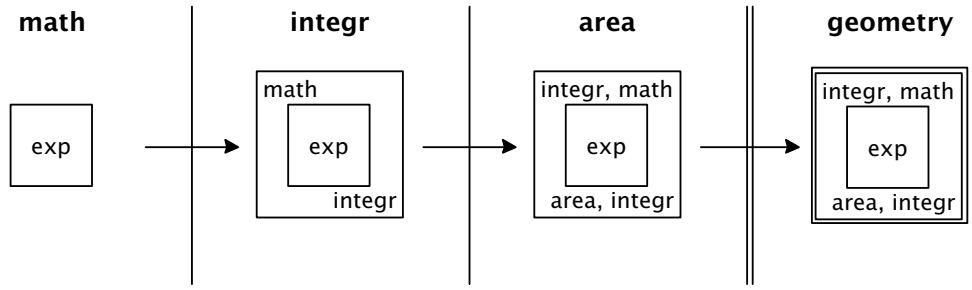


Figure 6.4: Module composition via **accept** and **require**

In summary, option contracts implement a natural mechanism for probabilistically or randomly checked program properties. The contract system allows components to place their trust in these checks in return for making them parties to the contract. Once the option is exercised, the contract system has enough blame information to pinpoint guilty parties if something goes wrong.

6.3 OCPCF:A Model for Option Contracts

We specify the semantics of option contracts as an extension to *CPCF* (see section 2.3), called *OCPCF*.

Types	τ	$=$	$\mathcal{N} \mid \mathcal{B} \mid \tau \to \tau \mid con(\tau)$
Contracts	κ	$=$	$\texttt{flat}(e) \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa) \mid \texttt{option}(\kappa, e)$
Terms	e	$=$	$v \mid x \mid e\,e \mid \mu x{:}\tau.e \mid e+e \mid e-e$
		$\mid$	$e \wedge e \mid e \vee e \mid \texttt{zero?}(e) \mid \texttt{if}\ e\ e\ e$
		$\mid$	$\texttt{mon}_l^{l,l}(\kappa, e) \mid \texttt{accept}^{l,l}(e)$
C. Values	v^*	$=$	$\lambda x{:}\tau.e$
Values	v	$=$	$b \mid v^*$
Base Values	b	$=$	$0 \mid 1 \mid -1 \mid \ldots \mid \texttt{tt} \mid \texttt{ff}$

Figure 6.5: OCPCF: source syntax

Figure 6.5 shows the source syntax for *OCPCF*. Like *CPCF*, *OCPCF* is a typed language that comes with predicate contracts for primitive values, $\texttt{flat}(e)$, and dependent contracts for higher-order functions, $\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2)$.[14] In addition, *OCPCF* offers option contracts $\texttt{option}(\kappa, e)$, which pair a function contract κ with a tester e that exercises the function. Note that option contracts are applied only to functions.[15]

Similarly to CPCF, the programmer can apply a contract κ to a component e using the monitor $\texttt{mon}_j^{k,l}(\kappa, e)$. A monitor separates two components: the service provider e and its context, the client. It combines our prototype's **`require`** and **`contract-out`**. The labels k, l and j serve as identifiers for the exporting component, dubbed the server, the importing component, dubbed the client, and the contract, respectively. In source code monitors have only one server and client label, but due to option contracts monitors may accumulate multiple server and client labels during evaluation. As we show, these sets of labels correspond to components that have accepted a randomly checked value either as servers or clients.

The form $\texttt{accept}^{k,l}(e)$ is an alternative component boundary, which imports and exports values without applying a contract, thus combining Racket's **`provide`** and **`accept`**. It declares that an importing component trusts the exporting component, even if it comes with an option contract. In particular, it does not activate the option that comes with the contract. Again the labels k and l name the server and the client side, respectively, of the expression.

[14] We omit simple function contracts as dependent function contracts subsume them

[15] The type system enforces this restriction. It also guarantees that function contracts are applied only to functions while flat contracts are applied to basic values. The type system rules are straightforward and therefore omitted.

Here is an example contract for a derivative operator:

$$\begin{aligned}\kappa &= \kappa_1 \overset{d}{\mapsto} (\lambda f.\texttt{option}(\kappa_2, \text{tester}))\\ \kappa_1 &= \text{positive} \overset{d}{\mapsto} (\lambda x.\,\text{positive})\\ \kappa_2 &= \text{positive} \overset{d}{\mapsto} (\lambda x.\,\text{close-to-slope-of-f@x})\end{aligned}$$

The post-condition of κ_2 asserts that the slope of f around x is close to the result of deriv applied to f at x. We use an option contract for κ_2 that performs some random testing. For conventional contracts, the monitor for deriv checks that its arguments respect κ_1 and its results κ_2. However κ_2 comes with an option contract. This replaces the checks for κ_2 with the random testing that tester performs when deriv returns its result. After that point, the contract system does not monitor the execution for κ_2 unless a component activates the option.

Component k_1 imports deriv with contract κ and applies it to a function f:

$$e_1 = \texttt{mon}_j^{s,k_1}(\kappa, \text{deriv})\ f$$

To avoid exercising the option, a client k_2 imports component k_1 via `accept`:

$$e_2 = \texttt{accept}^{k_1,k_2}(e_1)$$

In turn, another client k_3 employs `accept` to import k_2 without contracts:

$$e_3 = \texttt{accept}^{k_2,k_3}(e_2)\ 0$$

Components k_2 and k_3 both accept the result of e_1, they trust the random testing and opt out of any further checks related to κ_2. Thus k_3 can use the result of e_1 without any contract checking. By using `accept`, clients k_1, k_2 and k_3 acknowledge responsibility as clients of the randomly checked result of e_1. Moreover, since k_1 and k_2 export the value that k_2 and k_3 accept, respectively, they also become servers for the value of e_1.

A different client k_4 can activate the option by importing e_2 through a monitor with any function contract; here we use the contract κ_3 that always succeeds:

$$\begin{array}{lcl} \kappa_3 & = & \text{always} \overset{d}{\mapsto} (\lambda x.\,\text{always}) \\ e_4 & = & \mathtt{mon}_{j'}^{k_2,k_4}(\kappa_3, e_2)\ 0 \end{array}$$

Let f' denote the derivative function of f and κ_2' and tester$'$ the result of substituting f for x in κ_2 and tester respectively (necessary due to the dependent nature of κ). Client k_4 considers κ_2', the contract of f', to be critical and thus decides not to trust the random testing. The monitor with the contract that always succeeds achieves this goal without imposing any more obligations on k_2 and k_4. More specifically, the monitor activates the option contract on κ_2' and alerts the contract system to inspect any further uses of f'. If the contract system detects a violation of the pre-condition of κ_2', it blames all the components that accepted responsibility for uses of f', namely k_1 and k_2. If it detects a violation of the post-condition of κ_2', it blames all the components that accepted to become servers for f', in this case s and k_1. Notice that the sets of clients and servers do not include k_4 and k_2 respectively. Since k_4 activates the option it does not bear any responsibility for uses of f' and also it does not hold k_2 responsible for its results.

$$\begin{array}{llcl} \textbf{Terms} & e & = & \ldots \mid \mathtt{check}_{\bar{l}}(e,v) \mid \mathtt{try}_{\bar{l}}(e,e) \\ & & \mid & \underline{mg} \mid og \\ \textbf{Guards} & mg & = & \mathtt{M}_{\underline{\bar{l}}}^{l,l}(\kappa,e) \\ & og & = & \mathtt{O}_{l}^{l,l}(\kappa,e) \\ \textbf{Values} & & & \\ MGV: & mgv & = & \ldots \mid \mathtt{M}_{\underline{\bar{l}}}^{\bar{l},l}(\kappa,v^*) \\ OGV: & ogv & = & \ldots \mid \mathtt{O}_{l}^{l,l}(\kappa,v^*) \\ & v^* & = & \ldots \mid mgv \mid ogv \end{array}$$

Figure 6.6: OCPCF: evaluation syntax

$$
\begin{array}{rcl}
E & = & [\,] \mid E\ e \mid v\ E \mid E + e \mid v + E \mid E - e \mid v - E \\
& \mid & E \wedge e \mid v \wedge E \mid E \vee e \mid v \vee E \mid \texttt{zero?}(E) \\
& \mid & \texttt{if}\ E\ e\ e \mid \texttt{mon}_j^{\bar{k},\bar{l}}(\kappa, E) \mid \texttt{check}_l^{\bar{k}}(E, v) \\
& \mid & \texttt{try}_l^{\bar{l}}(E, v) \mid \texttt{accept}^{k,l}(E) \mid \texttt{O}_j^{\bar{k},\bar{l}}(\kappa, E)
\end{array}
$$

Figure 6.7: OCPCF: evaluation contexts

We define the behavior of *OCPCF* with a reduction semantics. Figures 6.6, 6.7 and 6.8 present the evaluation syntax, the evaluation contexts, and the reduction rules for *OCPCF*. The rules that do not involve partial contracts are similar to *indy* CPCF.[16]

Monitors guards for option contracts, $\texttt{M}_j^{k,l}(\texttt{option}(\kappa, e), v)$, employ `try` to check that a guard for v with κ satisfies the tester e. Since e is part of the contract itself, the monitor for the test carries j as its client label. If the test does not lead to a contract error, `try` returns an option guard $\texttt{O}_j^{k,l}(\kappa, v)$. The option guard is a value that stores v together with κ and the labels of the original monitor.

Option guards represent partially checked functions. The guards keep track of the contract so that they can restore it upon activation of the option. In addition they accumulate the labels of the components that accepted the option guard in order to provide accurate blame if, after activation, contract checking detects a contract breach. A component can use an option guard like any other function. No contract checking takes place. In particular, $\texttt{O}_j^{\bar{k},\bar{l}}(\kappa, v_f)\ v$ frees v_f from the guard and reduces to an ordinary application $v_f\ v$.

An $\texttt{accept}^{k,l}(\texttt{O}_j^{\bar{h},\bar{q}}(\kappa, v))$ statement updates the option guard's labels; it adds the server and client label of the `accept` statement to the server and client labels of the guard respectively, which yields $\texttt{O}_j^{k\bar{h},l\bar{q}}(\kappa, v)$. This labeling shows how the importing component accepts responsibility as a client of the option guard and the

[16] Since we only consider *indy* semantics we drop the i subscript from the reduction relations symbols

$E[\cdots]$	$\longmapsto$	$E[\cdots]$	
$\mathtt{n}_1 + \mathtt{n}_2$	.	$\mathtt{n}$	where $n_1 + n_2 = n$
$\mathtt{n}_1 - \mathtt{n}_2$	.	$\mathtt{n}$	where $n_1 - n_2 = n$
$\mathtt{zero?}(\mathtt{0})$	.	$\mathtt{tt}$	
$\mathtt{zero?}(\mathtt{n})$	.	$\mathtt{ff}$	if $n \neq 0$
$v_1 \wedge v_2$	.	v	where $v_1 \wedge v_2 = v$
$v_1 \vee v_2$	.	v	where $v_1 \vee v_2 = v$
$\mathtt{if}\ \mathtt{tt}\ e_1\ e_2$	.	e_1	
$\mathtt{if}\ \mathtt{ff}\ e_1\ e_2$	.	e_2	
$\lambda x.e\ v$	.	$\{v/x\}e$	
$\mu x.e$	.	$\{\mu x.e/x\}e$	
$\mathtt{mon}_j^{k,l}(\kappa, v)$	.	$\mathtt{M}_j^{k,l}(\kappa, v)$	
$\mathtt{M}_j^{\bar{k},\bar{l}}(\mathtt{flat}(e), b)$	.	$\mathtt{check}_j^{\bar{k}}(e\ b, b)$	
$\mathtt{M}_j^{k,l\bar{l}}(\mathtt{option}(\kappa, e), v)$	.	$\mathtt{try}_j^{\bar{k}}(e\ \mathtt{M}_j^{\bar{k},j\bar{l}}(\kappa, v), \mathtt{O}_j^{\bar{k},l\bar{l}}(\kappa, v))$	
$\mathtt{M}_j^{k\bar{k},\bar{l}}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v_f)\ v$	.	$\mathtt{M}_j^{k\bar{k},\bar{l}}(\{\mathtt{M}_j^{\bar{l},j\bar{k}}(\kappa_1, v)/x\}\kappa_2, v_f\ \mathtt{M}_j^{\bar{l},k\bar{k}}(\kappa_1, v))$ if $v_f \notin OGV$	
$\mathtt{M}_j^{\bar{k},\bar{l}}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), ogv)\ v$	.	$\mathtt{M}_j^{\bar{k},\bar{l}}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), mgv)\ v$ where $ogv = \mathtt{O}_j^{\bar{h},\bar{q}}(\kappa, v)$ and $mgv = \mathtt{M}_j^{\bar{h},\bar{q}}(\kappa, v)$	
$\mathtt{check}_j^{\bar{k}}(\mathtt{tt}, v)$	.	v	
$\mathtt{check}_j^{\bar{k}}(\mathtt{ff}, v)$	.	$\mathtt{error}_j^{\bar{k}}$	
$\mathtt{try}_j^{\bar{k}}(\mathtt{tt}, v)$	.	v	
$\mathtt{try}_j^{\bar{k}}(\mathtt{ff}, v)$	.	$\mathtt{error}_j^{\bar{k}}$	
$\mathtt{accept}^{k,l}(v)$	.	v	if $v \notin OGV$
$\mathtt{accept}^{k,l}(\mathtt{O}_j^{\bar{h},\bar{q}}(\kappa, v))$	.	$\mathtt{O}_j^{k\bar{h},l\bar{q}}(\kappa, v)$	
$\mathtt{O}_j^{\bar{k},\bar{l}}(\kappa, v_f)\ v$	.	$v_f\ v$	
$E[\mathtt{error}_j^{\bar{k}}]$	$\longmapsto$	$\mathtt{error}_j^{\bar{k}}$	

Figure 6.8: OCPCF: reduction semantics

exporting component as a server. The `accept` expression does not affect values other than option guards.

Notice also that due to `accept` and activation, monitor guards carry a pair of sets of labels instead of one server and one client label. This implies that in case of a contract failure the `check` blames all the labels at the server position on the guard, i.e., it blames all the components that accepted responsibility for the guarded value. Also, it requires the addition of sets of labels to all the rules that involve monitors.

Activation of option guards occurs when a component passes an option guard to another component without using an `accept` statement. In our model this happens only via a contract monitor guard $\mathtt{M}_j^{k,l}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), ogv)$. In this case the option guard $\mathtt{O}_j^{\overline{k},\overline{l}}(\kappa, v)$ becomes a monitored guard $\mathtt{M}_j^{\overline{k},\overline{l}}(\kappa, v)$. Monitored guards cannot be deactivated again.

To demonstrate the workings of our model, we revisit our example from above. Under the semantics of the model, e_3 reduces as follows:

$$\begin{aligned}
e_3 &\longmapsto^* \mathtt{accept}^{k_2,k_3}(\mathtt{accept}^{k_1,k_2}(\mathtt{mon}_j^{s,k_1}(\mathtt{option}(\kappa_2', \mathrm{tester}'), f'))) \\
&\longmapsto^* \mathtt{accept}^{k_2,k_3}(\mathtt{accept}^{k_1,k_2}(\mathtt{O}_j^{s,k_1}(\kappa_2', f'))) \\
&\longmapsto^* \mathtt{O}_j^{\{k_2,k_1,s\},\{k_3,k_2,k_1\}}(\kappa_2', f')\ 0 \\
&\longmapsto f'\ 0
\end{aligned}$$

Since the option contract is not activated the result of e_1, f', does not come with a guard around it and the application proceeds without any contract checking. Thus the use of 0—not a positive number—does not result in a violation of κ_2'.

In the case of e_4, the reduction is different:

$$e_4 \longmapsto^* \mathtt{mon}^{k_2,k_4}(\kappa_3, \mathtt{M}_j^{\{k_1,s\},\{k_2,k_1\}}(\kappa_2', f'))\,0$$

Since e_4 chooses to activate the option contract, the application involves checking κ_2' and thus results in a contract error blaming k_1 and k_2, the components that

accepted responsibility for using f':

$$e_4 \longmapsto^* \texttt{error}_j^{\{k_2,k_1\}}$$

As the example points out, our model introduces a policy for assigning blame that deviates from the blame behavior of Findler-Felleisen contracts. Instead of blaming one component for violating its contractual obligations, our new model assigns blame to many components. Considering the difficulties of getting blame assignment correct for the Findler-Felleisen version of contracts this change calls for a formal investigation of the correctness of our contract system.

6.4 Complete Monitoring for Option Contracts

In this section we prove a variant of complete monitoring [26] for *OCPCF*. Complete monitoring guarantees that the contract system assigns blame accurately and protects components effectively. A complete monitoring theorem is thus the minimal standard that a contract system should satisfy.

Unfortunately, the *OCPCF* model does not come with the necessary infrastructure to prove complete monitoring. We deal with this issue via an indirect approach. We first define another model **OCPCF*, which has the required hooks for establishing complete monitoring. After that we show that the two models are observably equivalent. The rest of this section introduces **OCPFC*, proves complete monitoring for the latter, and finally shows that *OCPCF* and **OCPCF* specify the same behavior for programs.

6.4.1 *OCPCF: a Theory-Friendly Model for Partial Contracts

Adding ownership and obligations annotations to *OCPCF* poses a serious challenge. The `accept` expression allows values to circumvent the contract system

as they migrate from one component to another. Thus a naive annotation of the *OCPCF* semantics with ownership labels obviously violates the single-owner policy and thus breaks complete monitoring. Fortunately, we can construct an extension of *CPCF* that is equivalent to *OCPCF* and tracks uninspected values. We call the new language **OCPCF*.

The source syntax of **OCPCF*, specified in figure 6.9, borrows many elements from the source syntax of the annotated *CPCF*.

Types	τ	$=$	$\mathcal{N} \mid \mathcal{B} \mid \tau \to \tau \mid con(\tau)$
Contracts	κ	$=$	$\lfloor \texttt{flat}(\lVert e \rVert^l) \rfloor^{\bar{l}} \mid \kappa \mapsto \kappa \mid \kappa \overset{d}{\mapsto} (\lambda x.\kappa) \mid \lfloor \texttt{option}(\kappa, e) \rfloor^{\bar{l}}$
Terms	e	$=$	$x \mid e+e \mid e-e \mid e \wedge e \mid e \vee e \mid \texttt{zero?}(e)$
		$\mid$	$\texttt{if}\ e\ e\ e \mid \mu x\!:\!\tau.e \mid e\ e \mid \texttt{accept}^{l,l}(e)$
C.Values	v^*	$=$	$\lambda x\!:\!\tau.e$
Values	v	$=$	$b \mid v^*$
BaseValues	b	$=$	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid \texttt{tt} \mid \texttt{ff} \mid \lambda x\!:\!\tau.e$

Figure 6.9: *OCPCF: source syntax

Notice, that the source syntax of **OCPCF* is the same as that of *OCPCF* except the ownership and obligations annotations.

Figure 6.10 shows the necessary extra rules for well-formed source programs and contracts. An `accept` expression defines a component boundary and thus the client label should match the owner of the context while the server label has to match the label on its sub-expression. In turn, the argument term should be a well-formed server expression. Option contracts may raise a contract violation due to a failed test and thus they impose obligations like flat contracts. In addition, an option contract must own its tester term.

In addition to `try` and `check`, the evaluation syntax of **OCPCF* (figure 6.11) introduces a small ecosystem of guards. We group guards in three categories: monitor guards, option guards and *trusted guards*. Trusted guards mark "foreign"

$\boxed{\Gamma; l \Vdash e}$

$$\frac{\Gamma; k \Vdash e \qquad k \neq l}{\Gamma; l \Vdash \mathtt{accept}^{k,l}(\|e\|^k)}$$

$\boxed{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa}$

$$\frac{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa \qquad \Gamma; j \Vdash e}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{option}(\kappa, \|e\|^j) \rfloor^{\overline{k}}}$$

Figure 6.10: *OCPCF: well-formed source programs and contracts

unchecked values as welcomed values in a component while wrapping them with appropriate ownership annotations. Unlike *OCPCF*, here both monitor and option guards carry only two labels rather than two sets of labels. However, the star versions of the guards, $^*M$ and $^*O$, allow us to record multiple components that have accepted an option guard. For instance a stack of star-option guards with an option guard at the bottom in **OCPCF*, $^*\mathtt{O}^{k_1,l_1}(\ldots{}^*\mathtt{O}^{k_n,l_n}(\mathtt{O}_j^{h,q}(\kappa,v))\ldots)$, corresponds to a single option guard $\mathtt{O}_j^{k_1\ldots k_n h, l_1\ldots l_n q}(\kappa,v)$ in *OCPCF*. Stacks of guards have the advantage of introducing separate boundaries between components that have accepted an option guard. This allows the guards to accommodate ownership annotations without breaking the single-owner policy.

The guards, together with the `check` and `try` constructs introduce points where ownership changes. Hence, the owner of the hole of evaluation contexts changes at corresponding positions. Figure 6.12 defines the evaluation contexts of **OCPCF* as an extension of those of *CPCF* with annotations (figure 3.4).

Before we present the semantics of **OCPCF*, we need to add another case to the special contract substitution function so that it handles option contracts:

$$\{e/^c x\}\lfloor \mathtt{option}(\kappa, \|e'\|^k) \rfloor^{\overline{l}} = \lfloor \mathtt{option}(\{e/^c x\}\kappa, \{\|e\|^k/x\}\|e'\|^k) \rfloor^{\overline{l}}$$

$$
\begin{array}{llcl}
\textbf{Terms} & e & = & \ldots \mid \mathtt{check}_l^{\bar{l}}(e,v) \mid \mathtt{try}_l^{\bar{l}}(e,e) \\
 & & \mid & mg \mid og \mid tg \\
\textbf{Guards} & mg & = & \mathtt{M}_l^{l,l}(\kappa,e) \mid {}^*\mathtt{M}^{l,l}(e) \mid \|mg\|^l \\
 & og & = & \mathtt{O}_l^{l,l}(\kappa,e) \mid {}^*\mathtt{O}^{l,l}(e) \mid \|og\|^l \\
 & tg & = & \mathtt{T}^{l,l}(e) \mid \|tg\|^l \\
\textbf{Values} & & & \\
MGV: & mgv & = & \ldots \mid \mathtt{M}_l^{l,l}(\kappa,v^*) \mid {}^*\mathtt{M}^{l,l}(v^*) \mid \|mgv\|^l \\
OGV: & ogv & = & \ldots \mid \mathtt{O}_l^{l,l}(\kappa,v^*) \mid {}^*\mathtt{O}^{l,l}(v^*) \mid \|ogv\|^l \\
TGV: & tgv & = & \ldots \mid \mathtt{T}^{l,l}(v^*) \mid \|tgv\|^l \\
 & v^* & = & \ldots \mid mgv \mid ogv \mid tgv
\end{array}
$$

Figure 6.11: *OCPCF: evaluation syntax

$$
\begin{array}{rcl}
E^l & = & \ldots \mid \_\mathtt{check}_l^{\bar{k}}(E^{l_o},v) \mid \mathtt{check}_h^{\bar{k}}(E^l,v) \mid \mathtt{try}_l^{\bar{k}}(E^{l_o},v) \\
 & \mid & \mathtt{try}_h^{k}(E^l,v) \mid \mathtt{accept}^{l,k}(E^{l_o}) \mid \mathtt{accept}^{h,k}(E^l) \\
 & \mid & \mathtt{M}_j^{l,k}(\kappa,E^{l_o}) \mid \mathtt{M}_j^{h,k}(\kappa,E^l) \mid {}^*\mathtt{M}^{l,k}(E^{l_o}) \mid {}^*\mathtt{M}^{h,k}(E^l) \\
 & \mid & \mathtt{O}_j^{l,k}(\kappa,E^{l_o}) \mid \mathtt{O}_j^{h,k}(\kappa,E^l) \mid {}^*\mathtt{O}^{l,k}(E^{l_o}) \mid {}^*\mathtt{O}^{h,k}(E^l) \\
 & \mid & \mathtt{T}^{l,k}(E^{l_o}) \mid \mathtt{T}^{h,k}(E^l)
\end{array}
$$

Figure 6.12: *OCPCF: evaluation contexts

The reduction semantics for *$OCPCF$* adopts some rules from the semantics of *$CPCF$* and adds a few, see figures 6.13 and 6.14.

Rules for `check` and `try` are similar to those of *$OCPCF$* except that they come with ownership annotations.

The rules for `accept` are slightly modified. Accepting an option guard *ogv* results in a star-option guard around *ogv* that uses the same labels as the `accept` expression. Accepting any other value returns a trusted guard around the value. We use trusted guards to mark a function as foreign inside a component when the function crosses the component's boundary without the contract system's protection. This should only happen via an `accept`.

When a stack of trusted guards holds a basic value, the reduction eliminates the

$E^l[\cdots]$	$\longmapsto$	$E^l[\cdots]$	
$\texttt{check}^{\bar{k}}_j(\|\|\texttt{tt}\|\|^j, v)$	.	v	
$\texttt{check}^{\bar{k}}_j(\|\|\texttt{ff}\|\|^j, v)$	.	$\texttt{error}^{\bar{k}}_j$	
$\texttt{try}^{\bar{k}}_j(\|\|\texttt{tt}\|\|^j, v)$	.	v	
$\texttt{try}^{\bar{k}}_j(\|\|\texttt{ff}\|\|^j, v)$	.	$\texttt{error}^{\bar{k}}_j$	
$\texttt{accept}^{k,l}(v)$	.	$\texttt{T}^{k,l}(v)$	if $v \notin OGV$
$\texttt{accept}^{k,l}(ogv)$	.	${}^*\texttt{O}^{k,l}(ogv)$	
tg	.	b	
if $tg = \texttt{T}^{k_1,l_1}(\ldots\|\|\texttt{T}^{k_n,l_n}(\|\|\texttt{T}^{m,p}(\|\|b\|\|^r)\|\|^h)\|\|^{q_n}\ldots)$			
$\|\|tgv\|\|^l \ \|\|v\|\|^l$	.	$\|\|tg\|\|^l$	
if $\|\|tgv\|\|^l = \|\|\texttt{T}^{k_1,l_1}(\ldots\|\|\texttt{T}^{k_n,l_n}(\|\|\texttt{T}^{m,p}(v')\|\|^h)\|\|^{q_n}\ldots)\|\|^{q_1}$			
where $\|\|tg\|\|^l = \|\|\texttt{T}^{k_1,l_1}(\ldots\|\|\texttt{T}^{k_n,l_n}(\|\|\texttt{T}^{m,p}(v'\ tgv')\|\|^h)\|\|^{q_n}\ldots)\|\|^{q_1}$			
and $tgv' = \texttt{T}^{p,m}(\|\|\texttt{T}^{l_n,k_n}(\|\|\ldots\texttt{T}^{l_1,k_1}(\|\|\|\|v\|\|^l\|\|^{q_1})\ldots\|\|^{q_n})\|\|^h)$			
$\|\|ogv\|\|^l \ \|\|v\|\|^l$	.	$\|\|tgv\|\|^l \ \|\|v\|\|^l$	
if $\|\|ogv\|\|^l = \|\|{}^*\texttt{O}^{k_1,l_1}(\ldots\|\|{}^*\texttt{O}^{k_n,l_n}(\|\|\texttt{O}^{m,p}_r(\kappa', v)\|\|^s)\|\|^{q_n}\ldots)\|\|^{q_1}$			
and where $\|\|tgv\|\|^l = \|\|\texttt{T}^{k_1,l_1}(\ldots\|\|\texttt{T}^{k_n,l_n}(\|\|\texttt{T}^{m,p}(v')\|\|^s)\|\|^{q_n}\ldots)\|\|^{q_1}$			

Figure 6.13: *OCPCF: annotated semantics, part 1

guards and delivers the value after removing all ownership annotations. After all, basic values are safe for the context to absorb.

Applying a function wrapped in a stack of trusted guards reduces to an application wrapped with the same guards. The argument is wrapped with a reversed stack of guards plus ownership annotations. The labels on the guards for the argument are swapped just as for monitors for function contracts. An application of a stack of option guards reduces to an application of a homomorphic trusted guard. The trusted guard marks the function as a foreign value but does not add any contract-related constraints on its use.

The four rules that involve stacks of monitor guards are the most complex rules.

$E^l[\cdots] \longmapsto E^l[\cdots]$

$\mathtt{mon}_j^{k,l}(\kappa, e) \quad . \quad \mathtt{M}_j^{k,l}(\kappa, e)$
$mg \quad . \quad \mathtt{check}_j^{k_1 \ldots k_n m}(e\ b, b)$
if $mg = {}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\lfloor \mathtt{flat}(e) \rfloor^{\bar{l}}, ||b||^r)||^k)||^{q_n}\ldots)$

$mg \quad . \quad \mathtt{try}_j^{k_1 \ldots k_n m}(e\ mg', og)$
if $mg = {}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\lfloor \mathtt{option}(\kappa, e) \rfloor^{\bar{l}}, v^*)||^k)||^{q_n}\ldots)$
where $mg' = {}^*\mathtt{M}^{k_1,j}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\kappa, v^*)||^k)||^{q_n}\ldots)$
and $og = {}^*\mathtt{O}^{k_1,l_1}(\ldots||{}^*\mathtt{O}^{k_n,l_n}(||\mathtt{O}_j^{m,p}(\kappa, v^*)||^k)||^{q_n}\ldots)$

$||mgv||^l\ ||v||^l \quad . \quad ||mgv'||^l\ ||v||^l$
if $||mgv||^l = ||{}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), ogv)||^h)||^{q_n}\ldots)||^{q_1}$
where $ogv = ||{}^*\mathtt{O}^{s_1,t_1}(\ldots||{}^*\mathtt{O}^{s_n,t_n}(||\mathtt{O}_j^{s,t}(\kappa, v')||^r)||^{u_n}\ldots)||^{u_1}$
and $mgv' = ||{}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), mgv'')||^h)||^{q_n}\ldots)||^{q_1}$
and $mgv'' = ||{}^*\mathtt{M}^{s_1,t_1}(\ldots||{}^*\mathtt{M}^{s_n,t_n}(||\mathtt{M}_j^{s,t}(\kappa, v')||^r)||^{u_n}\ldots)||^{u_1}$

$||mgv||^l\ ||v||^l \quad . \quad ||mg||^l$
if $||mgv||^l = ||{}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\kappa_1 \overset{d}{\mapsto} (\lambda x.\kappa_2), v')||^h)||^{q_n}\ldots)||^{q_1}$
and $v' \notin OGV$
where $||mg||^l = ||{}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{m,p}(\{mg''/^c x\}\kappa_2, v'\ mg')||^h)||^{q_n}\ldots)||^{q_1}$
and $mg' = {}^*\mathtt{M}^{p,m}(||{}^*\mathtt{M}^{l_n,k_n}(||\ldots\mathtt{M}_j^{l_1,k_1}(\kappa_1, ||||v||^l||^{q_1})\ldots||^{q_n})||^h)$
and $mg'' = {}^*\mathtt{M}^{p,j}(||{}^*\mathtt{M}^{l_n,k_n}(||\ldots\mathtt{M}_j^{l_1,k_1}(\kappa_1, ||||v||^l||^{q_1})\ldots||^{q_n})||^h)$

Figure 6.14: *OCPCF: annotated semantics, part 2

Figure 6.14 shows the corresponding reductions. Note that these rules also cover the cases where there is just a single monitor around a value instead of a stack of star-monitors. Thus they subsume the corresponding rules of *CPCF*.

If the bottom of the stack is a guard for a flat contract on a basic value, the stack reduces to a `check` where the blame labels are all the labels in server position on the stack of the guards. If the check fails, the contract system blames all the components that accepted responsibility for that value.

If the bottom of the stack is a guard for an option contract, the stack reduces to a `try` where the test term applies the tester of the contract to the stack of monitors. The rule replaces, though, the client label of the top of the stack with j to get the *indy* effect. If the test succeeds `try` returns an option guard with the same structure as the original monitor guard but without the option around the contract; else it raises a contract error blaming all the labels in server position on the stack of the guards.

An application of a stack of monitor guards depends on the value that resides in the guard at the bottom of the stack. If it is a stack of option guards, the reduction activates the option by turning the option guards into corresponding monitors and then performs the application. If not, the application is similar to that of trusted guards. The difference is that the rule also decomposes the function contract at the bottom of the stack and uses the pre-condition as the contract at the bottom of the argument stack and the post-condition as the contract at the bottom of the application stack. Furthermore, the stack of guards substituted in the post-condition uses the contract label, namely j, as the client label on the top star-guard.

Before delving into proving complete monitoring for **OCPCF*, we go back to the example of section 6.3 and examine how its behavior changes under the semantics of our new model. We omit obligation annotations, which after all do not affect computation, and we focus on ownership and guards. In terms of the

syntax of $^*OCPCF$, the example's terms become:

$$\begin{aligned} e_1 &= \|\texttt{accept}^{k_1,k_2}(\|\texttt{mon}_j^{s,k_1}(\kappa, \|\,\text{deriv}\,\|^s)\ f\|^{k_1})\|^{k_2} \\ e_2 &= \|\texttt{accept}^{k_1,k_2}(e_1)\|^{k_2} \\ e_3 &= \|\texttt{accept}^{k_3,k_2}(e_2)\ 0\|^{k_3} \\ e_4 &= \|\texttt{mon}_{j'}^{k_3,k_4}(\kappa_3, e_2)\ 0\|^{k_3} \end{aligned}$$

The execution of e_3 leads to an application of a stack of trusted guards:

$$\begin{aligned} e_3 &\longmapsto^* \|^*\texttt{O}^{k_2,k_3}(\|^*\texttt{O}^{k_1,k_2}(\|\texttt{O}_j^{s,k_1}(\kappa_2', \|\,f'\|^s)\|^{k_1})\|^{k_2})\ 0\|^{k_3} \\ &\longmapsto^* \|\texttt{T}^{c_2,k_3}(\|\texttt{T}^{k_1,k_2}(\|\texttt{T}^{s,k_1}(\|\,f'\|^s)\|^{k_1})\|^{k_2})\ 0\|^{k_3} \end{aligned}$$

Like in section 6.3, e_3 reduces to the result of applying 0 to f'.

Similarly, e_4 reduces to a stack of monitors and produces the same error as in section 6.3:

$$\begin{aligned} e_4 &\longmapsto^* \|\texttt{mon}_{j'}^{k_2,k_4}(\kappa_3, \|^*\texttt{O}^{k_1,k_2}(\|\texttt{O}_j^{s,k_1}(\kappa_2', \|\,f'\|^s)\|^{k_1})\|^{k_2})\ 0\|^{k_4} \\ &\longmapsto^* \|\texttt{mon}_{j'}^{k_2,k_4}(\kappa_3, \|^*\texttt{M}^{k_1,k_2}(\|\texttt{M}_j^{s,k_1}(\kappa_2', \|\,f'\|^s)\|^{k_1})\|^{k_2})\ 0\|^{k_4} \\ &\longmapsto^* \texttt{error}_j^{k_2,k_1} \end{aligned}$$

6.4.2 Complete Monitoring for *OCPCF

The semantics of $^*OCPCF$ introduce a novel element for a contract system; component boundaries that do not behave the same. More specifically, in CPCF all boundaries are contract monitors that decompose and enforce contracts, while in $^*OCPCF$ there are various forms of guards. Except monitor guards, these guards do not enforce contracts and thus they allow values to cross component boundaries without the contract system's inspection.

In this setting, complete monitoring is meaningful only if it implies that the contract system gives to the programmer the power to control how boundaries behave and thus relax or strenghten contract monitoring at will. Concretely, a contract sytem with option contracts is complete if guards other than monitor guards occur

during evaluation only between componennts that the programmer chose to allow to communicate under a more relaxed protocol, i.e., only between components that exchange values via "accepted" option contracts.

The set $\mathbb{T}(e)$ of label pair sets records the components of e that the programmer indicates to be in such a trust relation:

$$\mathbb{T} = \begin{array}{l}\{\{k,l\} \mid e \text{ contains } \mathtt{accept}^{k,l}(e') \text{ or } \mathtt{mon}^{k,l}_j(\kappa, e') \\ \text{where } \kappa \text{ consists of at least one option contract}\}\end{array}$$

Intuitively, complete monitoring poses an extra requirement to contract systems with option contracts; non-monitor guards should appear only between pairs of components in the $\mathbb{T}$ set of the source program.

The definition of complete monitoring for **OCPCF* guarantees the above requirement by demanding that intermediate states of evaluation do not have a $\mathbb{T}$ set larger than that of the source program. As for the rest of the definition, it is similar to that for *CPCF* except that the case for failures of flat contracts is slightly different, an additional case for failed option contract tests.

Definition 6.4.1 (Complete Monitoring for *OCPCF)**.** *A reduction relation $\hookrightarrow$ is a* complete monitor *for *OCPCF iff for all terms e_0 such that $\varnothing; l_o \Vdash e_0$,*

- *$e_0 \hookrightarrow^* v$ or,*
- *for all e_1 such that $e_0 \hookrightarrow^* e_1$ there exists e_2 such that $e_1 \hookrightarrow e_2$ or,*
- *$e_0 \hookrightarrow^* e_1 \hookrightarrow^* \mathtt{error}^{\overline{h}}_j$ and e_1 is of the form $E^l[e_2]$ where*
  - $e_2 = {}^*\mathtt{M}^{l_2,l_1}(\ldots||{}^*\mathtt{M}^{l_{n+1},l_n}(||\mathtt{M}^{k,l_{n+1}}_j(\lfloor \mathtt{flat}(e) \rfloor^{\overline{l}}, v)||^{l_{n+1}}_+)||^{l_n}_+\ldots)$ *or,*
  - $e_2 = {}^*\mathtt{M}^{l_2,l_1}(\ldots||{}^*\mathtt{M}^{l_{n+1},l_n}(||\mathtt{M}^{k,l_{n+1}}_j(\lfloor \mathtt{option}(\kappa, e) \rfloor^{\overline{l}}, v)||^{l_{n+1}})||^{l_n}\ldots)$

  and for all such terms e_1, $v = \|v_1\|^k$, $\overline{h} = l_2 \ldots l_{n+1} k$ and $k \in \overline{l}$,

and for all intermediate states e' of the above computations, $\mathbb{T}(e') \subseteq \mathbb{T}(e)$.

The cases for contract failures guarantee that if the contract system raises a blame error then the blamed components are all the components that accepted responsibility for the monitor guards. In addition, the definition requires that the flat or option contract at the bottom of the stack is amongst the obligations of one of the blamed components, specifically of the first component that took responsibility for the value.

$\boxed{\Gamma; l \Vdash e}$

$$\frac{}{\Gamma; l \Vdash \mathtt{error}_j^{\overline{k}}} \qquad \frac{\Gamma; j \vDash e \qquad \Gamma; l \Vdash v}{\Gamma; l \Vdash \mathtt{check}_j^{\overline{k}}(e, v)}$$

$$\frac{\Gamma; l \Vdash e_1 \qquad \Gamma; j \Vdash e_2}{\Gamma; l \Vdash \mathtt{try}_{l_j}^{\overline{k}}(e_1, e_2)} \qquad \frac{\Gamma; k \vDash e}{\Gamma; l \Vdash \mathtt{T}^{k,l}(e)}$$

$$\frac{\Gamma; k \vDash e \qquad \Gamma; l_2 ... l_{n+1} k; l_1 ... l_n; j \rhd \kappa}{\Gamma; l_1 \Vdash {}^*\mathtt{M}^{l_2,l_1}(...||{}^*\mathtt{M}^{l_{n+1},l_n}(||\mathtt{M}_j^{k,l_{n+1}}(\kappa, e)||_+^{l_{n+1}})||_+^{l_n}...)}$$

$$\frac{\Gamma; k \vDash e \qquad \Gamma; l_2 ... l_{n+1} k; l_1 ... l_n; j \rhd \kappa}{\Gamma; l_1 \Vdash {}^*\mathtt{O}^{l_2,l_1}(...||{}^*\mathtt{O}^{l_{n+1},l_n}(||\mathtt{O}_j^{k,l_{n+1}}(\kappa, e)||_+^{l_{n+1}})||_+^{l_n}...)}$$

Figure 6.15: *OCPCF: well-formed evaluation terms

$\boxed{\Gamma; l \vDash e}$

$$\frac{\Gamma; l \Vdash e}{\Gamma; l \vDash \|e\|^l} \qquad \frac{\Gamma(x) = l}{\Gamma; l \vDash x} \qquad \frac{\Gamma; l \Vdash e_1 \qquad \Gamma; l \Vdash e_2}{\Gamma; l \vDash \|e_1\|^l \; e_2}$$

Figure 6.16: *OCPCF: loosely well-formed terms

The proof of complete monitoring for *OCPCF follows that of chapter 4. We develop a subject that generalizes well-formedness and use a progress and preservation subject reduction technique. Figure 6.15 shows the extended well-formedness relation. The rules cover the terms of the evaluation syntax of *OCPCF. Es-

$\boxed{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa}$

$$\frac{\Gamma; j \Vdash e \qquad last(\overline{k}) \in \overline{k'}}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{flat}(\|e\|^j) \rfloor^{\overline{k'}}}$$

$$\frac{\Gamma; \overline{k}; \overline{l}; j \rhd \kappa \qquad \Gamma; j \Vdash e \qquad last(\overline{k}) \in \overline{k'}}{\Gamma; \overline{k}; \overline{l}; j \rhd \lfloor \mathtt{option}(\kappa, \|e\|^j) \rfloor^{\overline{k'}}}$$

Figure 6.17: *OCPCF: well-formed contracts

pecially for stacks of guards they make sure that appropriate ownership annotations are present in between each guard layer. Moreover the rules for monitors and stacks of guards use a generalized notion of well-formedness (figure 6.16) to inspect terms inside the bottom guard of the stack of guards. This extention is needed because beta-reduction introduces terms that temporarily deviate from well-formedness [24]. The generalized well-formedness permits us to handle these temporarily out-of-order terms without disturbing the single-ownership and obligations principles. The rules in figure 6.17 replace the corresponding rules for well-formed flat and option contracts. Since `accept` expressions dynamically change the components that are responsible for meeting a contract, statically determined obligations cannot provide an accurate prediction of the blamable parties. Nevertheless, obligations can still tell us that the initially responsible component for a contract is amongst the components that get blamed if the contract fails.

With the extended definition for well-formedness in hand, we proceed to prove that $\longmapsto$ defines a complete monitor.

Theorem 6.4.2. *$\longmapsto$ is a complete monitor for *OCPCF.*

Proof. As a direct consequence of lemmas 6.4.3 and 6.4.4, we obtain that if $\varnothing; l_o \Vdash e_0$,

- $e_0 \longmapsto^* v$ or,

- for all e_1 such that $e_0 \longmapsto^* e_1$ there exists e_2 such that $e_1 \longmapsto e_2$ or,
- $e_0 \longmapsto^* \texttt{error}_j^k$,

and for all intermediate states e' of the above computations, $\mathbb{T}(e') \subseteq \mathbb{T}(e)$. In addition, for the contract error case, since $\varnothing; l_o \Vdash e_0$ we know that $\texttt{error}_j^k$ does not occur in e_0. Thus it must be a result of reduction. The only reduction rule that introduces $\texttt{error}_j^k$ is $E_2^l[\texttt{check}_j^k(||\texttt{ff}||^j, v)] \longmapsto E_2^\dagger[\texttt{error}_j^k]$. Also from the reduction rules, we get $E_2^\dagger[\texttt{error}_j^k] \longmapsto \texttt{error}_j^k$. Again $\texttt{check}_j^k(e, v)$ does not occur in e_0 because $\varnothing; l_o \Vdash e_0$. Hence it must be also the result of a reduction. There are only two rules that introduce checks is:

- $E_1^l[\texttt{mon}_j^{k,l}(\lfloor \texttt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})] \longmapsto E_1^l[\texttt{check}_j^k(e\ b, b)]$. From the above information we deduce:

$$\begin{aligned}
& e_0 \longmapsto^* E_1^l[\texttt{mon}_j^{k,l}(\lfloor \texttt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})] \\
& \longmapsto E_1^l[\texttt{check}_j^k(e\ b, b)] \\
& \longmapsto^* E_2^l[\texttt{check}_j^k(||\texttt{ff}||^j, v)] \\
& \longmapsto E_2^l[\texttt{error}_j^k] \\
& \longmapsto \texttt{error}_j^k
\end{aligned}$$

  By lemma 6.4.4 we get $\varnothing; l_o \Vdash E_1^l[\texttt{mon}_j^{k,l}(\lfloor \texttt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})]$. Lemma 6.4.6 implies $\varnothing; l \Vdash \texttt{mon}_j^{k,l}(\lfloor \texttt{flat}(e) \rfloor^{\overline{l'}}, ||b||^{l'})$. From the rules for well formed terms we obtain $||b||^{l'} = \|v_1\|^k$ and $k \in \overline{l'}$.

- $E_1^l[^*\texttt{A}^{k_1,l_1}(\ldots||^*\texttt{A}^{k_n,l_n}(||\texttt{A}_j^{k,l}(\lfloor \texttt{flat}(e) \rfloor^{\overline{l}}, ||b||^r)||^h)||^{q_n}\ldots)] \longmapsto E_1^l[\texttt{check}_j^{k_1\ldots k_n h}(e\ b, b)]$

  With an analysis of the reduction sequence similar to the first case and using the same lemmas, we derive the desired conclusion.

Finally combining the above results entails that $\longmapsto$ is a complete monitor. ∎

The proof is a direct consequence of the progress and preservation lemmas. The progress lemma shows that well-formed terms always take a step instead of getting stuck.

Lemma 6.4.3. *(Progress) For all terms e such that* $\varnothing; l \Vdash e$, $e = v$ *or* $e = \texttt{error}^k_j$ *or* $e \longmapsto e_0$.

Proof. From lemma 6.4.5 we know that there exist e', l and E^l such that $e = E^l[e']$. Via lemma 6.4.6 and by $\varnothing; l_o \Vdash e$, we derive $\varnothing; l \Vdash e'$. We proceed by case analysis on the form of e' and show that in each case, under the reduction relation, e' takes a step. ∎

The preservation lemma says that well-formed terms step to well-formed terms.

Lemma 6.4.4. *(Preservation) For all e and* e_0 *such that* $\varnothing; l_o \Vdash e$ *and* $e \longmapsto e_0$, $\varnothing; l_o \Vdash e_0$ *and* $\mathbb{T}(e_0) \subseteq \mathbb{T}(e)$.

Proof. We proceed by case analysis on the reduction of e . We employ lemma 6.4.6 to establish that the redex is well-formed. Then we apply the appropriate reduction to the redex and using the rules for well-formed terms and contracts and lemmas 6.4.10 to 6.4.16 we show that the reduction produces a well-formed term. Finally via lemmas 6.4.8 and 6.4.9 we plug in the result in the evaluation context and establish that the resulting term is well-formed which gives us the desired conclusion. The conclusion reguarding the $\mathbb{T}$ tests follows from the fact that no reduction creates non-monitor boundaries for a pair of components unlesss if the left-hand side of the reduction involves an `accept` statement or an option contract. ∎

In turn, the progress and preservation lemmas depend on a series of lemmas involving unique decomposition, preservation of well-formedness under substitution in contracts, terms and evaluation context holes, extraction of the redex of a term, and environment weakening.

Lemma 6.4.5. *(Unique Decomposition) For all terms e such that* $e \neq v$ *and* $e \neq \mathtt{error}^k_j$, *there are unique* e', l', *and* $E^{l'}$ *such that* $e = E^{l'}[e']$ *and* $e' = v_0\, op\, v_1$ *or* $e' = \mathtt{T}^{k_1,l_1}(...||\mathtt{T}^{k_n,l_n}(||\mathtt{T}^{m,p}(||b||^r)||^h)||^{q_n}...)$ *or* $e' = \mathtt{check}^k_j(v_0, v_1)$ *or* $e' = v_1\ v_2$ *or* $e' = \mu x.e_0$ *or* $e' = \mathtt{mon}^{k,l}_j(\kappa, v)$ *or* $e' = {}^*\mathtt{M}^{k_1,l_1}(...||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}^{k,l}_j(\lfloor \mathtt{flat}(e)\rfloor^{\bar{l}}, ||b||^r)||^h)||^{q_n}...)$ *or* $e' = {}^*\mathtt{M}^{k_1,l_1}(...||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}^{k,l}_j(\mathtt{option}(\kappa, e), v)||^h)||^{q_n}...)$ *or* $e' = \mathtt{zero?}(v)$ *or* $e' = \mathtt{if}\ v_0\ e_1\ e_2$ *or* $e' = \mathtt{error}^{\bar{k}}_j$.

Proof. By induction on the size of e. qed

Lemma 6.4.6. *If* $\varnothing; l \Vdash E^k[e]$ *then* $\varnothing; k \Vdash e$.

Proof. By induction on the size of E^k with the help of lemma 5.1.7 for the inductive case. ∎

Lemma 6.4.7. *If* $\varnothing; l \Vdash E^{l_o}[e]$ *then* $\varnothing; l \Vdash e$.

Proof. By induction on the size of E^{l_o}. ∎

Lemma 6.4.8. *If* $\varnothing; l \Vdash E^k[e]$, $\varnothing; k \nVdash e$ *and* $\varnothing; k \Vdash e'$ *then* $\varnothing; l \Vdash E^k[e']$.

Proof. By induction on the size of E^k. ∎

Lemma 6.4.9. *If* $\varnothing; l \Vdash E^k[e]$, $\varnothing; k \vDash e$ *and* $\varnothing; k \Vdash e'$ *then* $\varnothing; l \Vdash E^k[\|e'\|^k]$.

Proof. By induction on the size of E^k. ∎

Lemma 6.4.10. *If* $\Gamma \uplus \{x : k\}; l \Vdash e$, $\Gamma \uplus \{x : k\}; k \Vdash \|e_0\|^k$ *then*

$$\Gamma \uplus \{x : k\}; l \Vdash \{\|e_0\|^k / x\}e.$$

Proof. By mutual induction on the height of the derivation trees for $\Gamma \uplus \{x : k\}; l \Vdash e$ and $\Gamma \uplus \{x : k\}; \bar{k}; \bar{l}; j \rhd \kappa$. ∎

Lemma 6.4.11. *If* $\Gamma; l \Vdash e$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma \uplus \{x : k\}; l \Vdash e$.

Proof. By mutual induction on the height of the derivation trees for $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 6.4.12. *If* $\Gamma; l \Vdash e$ *and* $x \notin dom(\Gamma)$, *then* $x \notin FV(e)$.

Proof. By mutual induction on the height of the derivation trees for $\Gamma; l \Vdash e$ and $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 6.4.13. *If* $\Gamma \uplus \{x : k\}; l \Vdash e$ *and* $x \notin FV(e)$, *then* $\Gamma; l \Vdash e$.

Proof. By mutual induction on the height of the derivation trees for $\Gamma \uplus \{x : k\}; l \Vdash e$ and $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

Lemma 6.4.14. *If* $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma \uplus \{x : k\}; \overline{k}; \overline{l}; j \rhd \kappa$.

Proof. By induction on the height of $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$ using lemma 6.4.11 for the flat contracts case. ∎

Lemma 6.4.15. *If* $\Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa$ *and* $\Gamma; j \Vdash mg$, *then* $\Gamma; \overline{k}; \overline{l}; j \rhd \{mg/^c x\}\kappa$.

Proof. By induction on the height of $\Gamma \uplus \{x : j\}; \overline{k}; \overline{l}; j \rhd \kappa$. For the flat contracts case we employ lemma 6.4.10. ∎

Lemma 6.4.16. *If* $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$, $last(\overline{k'}) \in \overline{k}$ *and* $last(\overline{l'}) \in \overline{l}$ *then* $\Gamma; \overline{k'}; \overline{l'}; j \rhd \kappa$.

By induction on the height of $\Gamma; \overline{k}; \overline{l}; j \rhd \kappa$. ∎

6.4.3 Complete Monitoring for OCPCF

The introduction of $^*OCPCF$ is a detour to prove complete monitoring for *OCPCF*. To transfer the result to *OCPCF*, we prove a bisimulation theorem between the two languages. Figure 6.18 specifies the relation between *OCPCF* and $^*OCPCF$ terms.

The relation associates monitor and option guards in *OCPCF* to corresponding stacks of monitor and option guards in $^*OCPCF$. Also it relates stacks of trusted

$\boxed{e \sim e'}$

$$\frac{e \sim^{ctx} e'}{e \sim \|e'\|^{l}} \qquad \frac{e \sim^{ctx} e'}{e \sim \mathtt{T}^{k,l}(e')}$$

$$\frac{e \sim^{ctx} e' \qquad \kappa \sim^{ctx} \kappa'}{\mathtt{M}_j^{k_1 \ldots k_n k, l_1 \ldots l_n l}(\kappa, e) \sim {}^*\mathtt{M}^{k_1,l_1}(\ldots||{}^*\mathtt{M}^{k_n,l_n}(||\mathtt{M}_j^{k,l}(\kappa', e')||^{l})||^{l_n}\ldots)}$$

$$\frac{e \sim^{ctx} e' \qquad \kappa \sim^{ctx} \kappa'}{\mathtt{O}_j^{k_1 \ldots k_n k, l_1 \ldots l_n l}(\kappa, e) \sim {}^*\mathtt{O}^{k_1,l_1}(\ldots||{}^*\mathtt{O}^{k_n,l_n}(||\mathtt{O}_j^{k,l}(\kappa', e')||^{l})||^{l_n}\ldots)}$$

$\boxed{\kappa \sim \kappa'}$

$$\frac{e \sim^{ctx} e'}{\mathtt{flat}(e) \sim \lfloor \mathtt{flat}(e') \rfloor^{\bar{l}}}$$

Figure 6.18: OCPCF-*OCPCF bisimulation

guards and ownership annotations around a term in ${}^*OCPCF$ to the related term in $OCPCF$.

We establish that $OCPCF$ and ${}^*OCPCF$ define the same behavior by showing that the compatible closure $\sim^{ctx}$ of the relation $\sim$ defines a lockstep bisimulation between programs in the two languages.

Theorem 6.4.17. *Let e a term of PCPCF and e^* a term of ${}^*PCPCF$. If $\varnothing; l_o \Vdash e^*$ and $e \sim^{ctx} e^*$ then*

- *$e \longmapsto^* v$ iff $e^* \longmapsto^* v^*$ and $v \sim^{ctx} v^*$ and,*
- *$e \longmapsto^* \mathtt{error}_j^{\bar{k}}$ iff $e^* \longmapsto^* \mathtt{error}_j^{\bar{k}}$.*

Proof. By induction on the evaluation trace. For the inductive case, we proceed by case analysis on the form of e and we show that $\sim^{ctx}$ is a lockstep bisimulation for e and e'. Complete monitoring for ${}^*PCPCF$ guarantees that the reduction under $\longmapsto$ never reaches a stuck state. ∎

Since programs in the two languages behave the same, we reach the conclusion that *OCPCF* assigns blame correctly and protects components completely.

6.5 Making Option Contracts Feasible

In this section, we present some evidence in favor of option contracts via performance measurements. The conjectures we confirm are (1) that the option contract system satisfies a pay-as-you-go property—i.e., only components that use `option/c` experience a cost—and (2) that using option contracts with **`accept`** helps to lower the cost of contract checking.

Our benchmark suite includes six programs:

`mono` The `mono` program is based on the last example of section 6.2. It consists of one server and ten clients. The server exports a function `f` with a contract that checks whether the function is monotonic. It also supplies an integration operator `integrate` with a contract that checks whether its input is a monotonic function and whose result is a function that computes a good approximation of the area bounded by the graph of the input function. Each client imports `f` and `integrate` and exports them to the next client. The last client computes the integral of `f` and then calls the result of integration 5,000 times. We attach an option contract to `f` and an option contract to the result of `integrate`. The first option contract comes with a tester that checks `f` on 2,500 random inputs. The second option contract tests whether the result of `integrate` computes a good approximation of the area bound by the graph of `f` for 2,500 random inputs.

`deriv` The `d/dx` program has the same structure as the `mono` program. The server module exports a derivative operator. The contract of the operator asserts that the inputs and outputs are functions from reals to reals. In addition the contract

of the output function checks whether its result approximates the slope of the input function. The first client imports the derivative operator and uses it to compute the derivative of a polynomial function of high degree. Then it exports this result. The rest of the clients import and export the derivative of the polynomial, dubbed `fp`. The last client applies `fp` on different arguments 2,000,000 times. We attach an option contract to the result of the derivative function. The tester of the option contract creates 100,000 random inputs for the output of the derivative function.

`integr` This program is similar to the `d/dx` program except that the server module provides an adaptive integration operator. The contract for its output checks whether the result of the output function is a good approximation of the area defined by `integr`'s result. Again, the first client imports the integral operator and uses it to compute the integral of a polynomial function of high degree. Then it exports this result. The rest of the clients import and export the integral of the polynomial, dubbed `intgr`. The last client applies `intgr` on different arguments 1,250 times. The tester of the option contract of `intgr` creates 100 random inputs.

`sup` The server module of the `sup` program provides a `supremum` function that consumes a list `l`, a numerical comparison operator `R` and a function `f` from elements of the list to numbers. It returns a function that given a number `b`, returns the element of `l` that maximizes `f` according to `R` for the first `b` elements of `l`. The contract on the output of `supremum` checks whether the result of the output function is the maximum as specified. The first client imports `supremum` and applies it on a list of 100 numbers with the identity function as `f` and `<=` as `R`. Then it exports the result of the application. The rest of the clients import and export this result. The last client applies the result on 100,000 randomly generated limits. The tester of the option contract provides to the result of `supremum` as many random inputs as half the size of the input list of `supremum`.

vec This program is based on the first example of section 6.2. It consists of a server and a client module. The server module provides a binary search function on vectors for numbers. The contract of the function checks whether the vector is sorted around the area of the index. The client calls binary search 100,000 times on a vector of 10,000,000 elements. This time we attach an option contract on the vector argument of binary search. The tester of the option contract implements a spot checker [28]; it picks 15 random indices of the vector, retrieves the element each index points to and then checks whether binary search on the vector for each element returns the corresponding index.

tree The `tree` program resembles the `vec` program except that it searches binary search trees. The contract on the input tree of binary search checks whether the tree is a binary search tree around the nodes the search accesses. The client passes constructs a tree with 100000 nodes and then calls binary search 100000 times. The tester of the option contract of the tree implements a spot checker again [29]. It picks 25 nodes of the tree and checks whether binary search successfully returns each node when asked to search for the node's datum.

Figure 6.19 shows the results of our experiments. We measured each program in three different ways. The first one, called racket, runs each programs in plain Racket 5.3.0.11 with plain contracts. The second execution, called racket+, runs each program just like the first one but in `racket+options`. Its purpose is to check the pay-as-you-go conjecture. The last execution, called racket+o, runs each program in our implementation with accepted option contracts. Its purpose is to validate the reduced-cost conjecture concerning `accepted` contracts. For each program we measure execution time and memory consumption on a Mac Book Pro with an Intel Core 2 Duo at 2.53 GHz and 4GB of RAM. For each execution we report normalized values over the racket execution of the average of a few repeti-

time	`mono`	`d/dx`	`integr`	`sup`	`vec`	`tree`
racket	1	1	1	1	1	1
racket+	0.98	1.01	0.99	1	0.98	1
racket + *o*	0.33	0.52	0.60	0.52	0.98	1.01

memory	`mono`	`d/dx`	`integr`	`sup`	`vec`	`tree`
racket	1	1	1	1	1	1
racket+	1.02	1.03	1.03	1.03	1.01	1.03
racket + *o*	0.99	1.05	1.04	1.04	1.01	1.03

Figure 6.19: Experimental results

tions.

The results of our experiments provide evidence that our implementation of `option/c` and **`accept`** imposes no time overhead and a negligible 3% space overhead on programs that do not use them. More importantly, the first four programs demonstrate significant improvement in execution time while performing an adequate amount of checking to detect bugs. In the last two programs, the performance of option contracts is comparable to that of plain contracts. However, the option contracts in the last two programs use spot checkers that give more than 90% confidence that the checked property holds. Overall, our data suggests that an efficient implementation of option contracts is plausible and that option contracts can help reduce the overhead of contract checking.

6.6 Related Work

In this section we discuss results from the literature that are specifically related to option contracts. One such line of research tackles the computational overhead from checking precise contracts. Findler et al. [34] observe that traditional contract systems check properties of a data structure as soon as computation tries to access the data structure. As a result, contract checking may severely affect the complexity

of a function that traverses a list; a contract on the list is checked every time the traversal reaches a sub-list of the list. In response, Findler et al. suggest delaying contract checking and enforcing contracts per individual elements of the list as computation tries to access them. On a different direction but with a similar general goal, Dimoulas et al. [23] propose the use of concurrent contract checking to reduce the overhead of contracts in general. They put at work idle processors in multi-core machines to check contracts while the main thread evaluates speculatively the rest of the program. They prove that their system gives the same results as sequential contract checking and show through benchmarks that programs with heavy computationally contracts could benefit from their approach.

Another source of inspiration for option contracts is random testing [18]. There is a vast literature on the effectiveness of random checking and especially on how random tests generated by and checked against complete or partial specifications, like contracts, can boost the bug finding abilities of random testing [8, 11, 50, 51, 57]. Quickcheck [15, 16] is a popular tool that uses Haskell's type system to generate random inputs for functions and checks whether the functions meet user-specified properties. Quickcheck has inspired similar tools in many other languages. Eiffel's autotest [48] also creates random inputs based on the class hierarchy of programs but uses contracts as test oracles. Our result shows that it is feasible to make random testing part of a contract system without giving up complete monitoring.

Finally, the algorithms community has contributed sophisticated ideas for probabilistically checking expensive contracts with high assurance. For instance, spot-checkers [2, 5, 14, 17, 28, 29] are algorithms that can exercise pre-conditions and post-conditions of a function without affecting the asymptotic complexity of the function. Option contracts provide the linguistic support for turning spot-checkers into contracts.

CHAPTER 7

Related Work

7.1 Contracts

In general, this work extends the decades-old linguistic investigation of behavioral software contracts. While Parnas [49] introduced contracts as early as 1972, they became popular only via the design of the Eiffel programming language and the creation of a contract-oriented software engineering curriculum [46, 47]. Since then contracts have been used both for extended static checking [4, 21] and run-time monitoring of higher-order programs [32]. Currently, in one form or another, contracts are part of many platforms [27, 36, 40, 41, 43, 44, 45, 53, 58, 59]. The system of Findler and Felleisen [32], which has brought contracts into the world of higher-order functional programming languages, is the one most closely related to our framework.

The work of Findler and Felleisen initiated a continuing and intense search for the meaning of higher-order contracts. In turn research in this direction inspired novel contract systems. Blume and McAllester [6] introduce *picky* contract monitoring and explore a quotient model for higher-order contracts and use it to prove properties of Findler and Felleisen's higher-order contracts. Findler et al. (2004, 2006) propose an alternative view, namely, "contracts as projections,"

which relates contracts to Scott [60]'s denotation model of types. Gronski and Flanagan [38] relate Findler and Felleisen's higher-order contracts to type casts. Their result motivates a type-oriented form of extended static checking [42], which Greenberg et al. [37] consider a manifest form of contract. Xu et al. [72] use Blume and McAllester's ideas to develop static contract checking for Haskell using symbolic evaluation. Hinze et al. [39] and Chitil et al. [12, 13] introduce contracts to Haskell but end up with two different contract systems. The first performs eager contract checking while the second is lazy. Both research efforts also construct an algebra for contracts based on properties such as semantics-preserving contracts erasure and idempotence. Degen et al. [19] compare eager and lazy contract checking for lazy languages through a series of formal properties but do not reach a definitive conclusion.

7.2 Provenance and Syntactic Type Abstraction

Provenance is the information about the origin, context, and history of data. It plays an important role for the correct and secure behavior of large software and hardware systems [7, 61]. The study of provenance from a formal linguistic perspective is still at an initial stage [1, 9, 10]. Our notion of ownership is a means of keeping around some origin and context information about program values. Hence, it can and should be viewed as a form of provenance. Also ownership can be used as a basis for studying formal properties of this kind of provenance. However our technique and its use to prove properties of contract systems is not related to any provenance tracking technology.

Tracking information flow in a computer system is a specialized kind of provenance. Secure information flow as pioneered by Denning [20] suggests the restriction of flow of data in a computer system between agents that have the appropriate

level of clearance. There are both software and hardware techniques for imposing secure information flow. Our instrumentation of the dynamic semantics resembles techniques used for proving sound type systems that enforce secure information flow.

Zdancewic et al. [73] introduce (program) principals as a means to prove type abstraction properties related to information flow with a syntactic proof technique. In such a semantics, different principals "own" different components and exported values carry the principal of their component of origin. Since the principals semantics prevents reductions that involve values with different principals, a client component is obliged to use a server's functions on the server's values. In short, the semantics dynamically enforces a form of information hiding. It is now easy to see how a principals semantics supports a syntactic soundness proof of type abstraction. If the interface between the server component and the client employs abstract types and if the type system soundly enforces type abstraction, stuck states become unreachable during computation.

While Zdancewic et al. directly inspire our single owner policy, our semantics is unrelated to theirs and we apply the idea to define and prove a novel property of contract systems instead of type systems.

CHAPTER 8

Conclusion

The design of monitors for higher-order contracts raises theoretical problems of practical importance. The biggest obstacle to tackle these problems was the lack of a formal framework for validating the design of contract systems.

This dissertation presents complete monitoring, the first criterion for the correct design of contract systems. In short, if a contract system is a complete monitor, it manages to mediate all exchange of values between the components that contracts protect and it provably assigns blame to the faulty component in case of contract violations. Thus a complete monitor assures the programmer of two pragmatically key properties for any contract system.

To exemplify the applicability and scalability of complete monitoring, we use our framework in a series of scenarios. First, we employ complete monitors to show that existing semantics for dependent contracts are flawed and to design a new correct semantics. Second, we apply successfully our framework on the problem of safe inter-language component composition and the design of new contract systems. Finally, as an extended case study for the usefulness of complete monitors as a language design tool, we develop option contracts. These novel contracts address the pressing problem of the significant overhead that contract checking of precise properties adds to program execution. Specifically, option contracts allow

the programmer to replace expensive full-fledged contract checking with cheaper probabilistic and random tests in a controlled manner. In this setting, complete monitoring helps balance the uncertainty of relaxing contract checking. We show that our design of option contract is in fact a complete monitor and hence provides strong guarantees to the programmer about blame assignment and components protection.

As a final remark on the impact of this dissertation, its results have fundamentally altered Racket and helped the evolution of its contract system [63, 65, 66].

Bibliography

[1] U. A. Acar, A. Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. In *Proceedings of the 1st Conference on Principles of Security and Trust (POST)*, pages 410–429, 2012.

[2] S. Ar, M. Blum, B. Codenotti, and P. Gemmell. Checking approximate computations over the reals. In *Proceedings of the 35th ACM Symposium on the Theory of Computing (STOC)*, pages 786–795, 1993.

[3] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] M. Barnett, K. Rustan M. Leino, and W. Schulte. The Spec# programming system: an overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, pages 49–69, 2004.

[5] M. Blum and S. Kanna. Designing programs that check their work. In *Proceedings of the 31th ACM Symposium on the Theory of Computing (STOC)*, pages 86–97, 1989.

[6] M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

[7] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Survey*, 37(1):1–28, 2005.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[9] J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis. In *Proceedings of the 11th International Conference on Database Programming Languages (DBPL)*, pages 138–152, 2007.

[10] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: a future history. In *Proceedings of the 3rd Onward! Conference*, pages 957–964, 2009.

[11] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, pages 231–255, 2002.

[12] O. Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 67–76, 2012.

[13] O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *Proceedings of the 15th International Conference on Implementation and Application of Functional Languages (IFL)*, pages 1–19, 2003.

[14] M. Chu, S. Kannan, and A. Mcgregor. Checking and spot-checking the correctness of priority queues. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 728–739, 2007.

[15] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

[16] K. Claessen and J. Hughes. Testing monadic code with quickcheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

[17] A. Czumaj and C. Sohler. Soft kinetic data structures. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 865–872, 2001.

[18] Joe W. D. and S. Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 179–183, 1981.

[19] M. Degen, P. Thiemann, and St. Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *Journal of Logic and Algebraic Programming*, pages 515–549, 2010.

[20] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[21] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 158, Compaq SRC Research Report, 1998.

[22] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, 33(5): 16:1 – 16:29, 2011.

[23] C. Dimoulas, R. Pucella, and M. Felleisen. Future contracts. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 195 – 206, 2009.

[24] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*, pages 215 – 226, 2011.

[25] C. Dimoulas, A. Ahmed, A. Sabry, and M. Felleisen. Option contracts: Coupling contract checking with random and probabilistic testing. unpublished manuscript, 2012.

[26] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *Proceedings of the 21st European Symposium on Programming (ESOP)*, pages 211 – 230, 2012.

[27] A. Duncan and U. Hoelzle. Adding Contracts to Java with Handshake. Technical report, Santa Barbara, CA, USA, 1998.

[28] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. *Journal of Computer and System Sciences*, 60(3):717–751, 2000.

[29] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. Testing and spot-checking of data streams. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 165–174, 2000.

[30] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[31] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the 8th International Conference on Functional and Logic Programming (FLOPS)*, pages 226–241, 2006.

[32] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 48–59, 2002.

[33] R. B. Findler, M. Felleisen, and M. Blume. An investigation of contracts as projections. Technical Report TR-2004-02, University of Chicago, Computer Science Department, 2004.

[34] R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Proceedings of the 19th International Conference on Implementation and Application of Functional Languages (IFL)*, pages 111–128, 2007.

[35] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[36] B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. A Language Manual for Sather 1.1, 1996.

[37] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*, pages 353–364, 2010.

[38] J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *In Proceedings of the 8th Symposium on Trends in Functional Programming (TFP)*, pages 54–69, 2007.

[39] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Proceedings of the 8th International Conference on Functional and Logic Programming (FLOPS)*, pages 208–235, 2006.

[40] R. Holt, P. Matthews, J. Rosselet, and J. Cordy. *The Turing programming language: design and definition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[41] M. Karaorman, U. Holzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design by Contract. Technical report, Santa Barbara, CA, USA, 1999.

[42] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic, 2006. URL `http://sage.soe.ucsc.edu/`.

[43] M. Kölling and J. Rosenberg. Blue:Language Specification, version 0.94, 1997.

[44] R. Kramer. iContract - The Java(tm) design by Contract(tm) Tool. In *Technology of Object-Oriented Languages and Systems*, page 295, 1998.

[45] D. Luckham and F. Von Henke. An Overview of Anna, a Specification Language for Ada. *IEEE Software*, 2(2):9–22, 1985.

[46] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[47] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[48] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.

[49] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[50] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[51] Alexandre Petrenko. Specification based testing: Towards practice. In *Proceedings of 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI)*, pages 287–300, 2001.

[52] S. Peyton Jones. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press, 2003.

[53] R. Ploesch and J. Pichler. Contracts: From Analysis to C++ Implementation. In *Technology of Object-Oriented Languages and Systems*, page 248, 1999.

[54] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[55] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[56] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953.

[57] D. Richardson, O. O'Malley, and C. Tittle. Proceedings of the acm sigsoft 3rd symposium on software testing, analysis, and verification approaches to specification-based testing (tav). pages 86–96, 1989.

[58] D. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[59] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In *Proceedings of the 15th International Symposium on Formal Methods (FM)*, pages 68–83, 2008.

[60] Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3): 522–587, 1976.

[61] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, 2005.

[62] T. S. Strickland and M. Felleisen. Contracts for first-class modules. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS)*, pages 27–38, 2009.

[63] T. S. Strickland, C. Dimoulas, A. Takikawa, and M. Felleisen. Gradual typing for first-class classes. *unpublished manuscript*, 2012.

[64] C. Szyperski. *Component Software*. Addison Wesley, 1997.

[65] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. *unpublished manuscript*, 2012.

[66] A. Takikawa, T. S. Strickland, and S. Tobin-Hochstadt. Constraining delimited control with contracts. unpublished manuscript, 2012.

[67] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *Proceedings of the 2nd Symposium on Dynamic Languages (DLS)*, pages 964–974, 2006.

[68] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*, pages 395–407, 2008.

[69] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 117–128, 2010.

[70] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP)*, pages 1–16, 2009.

[71] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994.

[72] D. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*, pages 41–52, 2009.

[73] S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: a syntactic proof technique. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 197–207, 1999.