CONTRACTS AND EFFECTS

CAMERON MOY

Submitted to the faculty of Northeastern University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

December 2025

Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Thesis Title: Contracts and Effects

Author: Cameron Moy

PhD Program: X Computer Science ____ Cybersecurity ____ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

M. Felleisen	21 Dec 2025
Thesis Advisor	*Date*
Manuel Serrano	19 dec 2025
Thesis Reader	*Date*
	22 December 2025
Thesis Reader	*Date*
Robby Findler	December 22, 2025
Thesis Reader	*Date*
	December 22, 2025
Thesis Reader	*Date*

KHOURY COLLEGE APPROVAL:

	December 22, 2025
Associate Dean for Graduate Programs	*Date*

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

	13 January 2026
Recipient's Signature	*Date*

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

Cameron Moy
Contracts and Effects

Funding was provided by the American people through the National Science Foundation (SHF 2116372, SHF 2315884). Some material from this dissertation appears in the following publications: Moy and Felleisen [108], Moy et al. [107], Moy et al. [109], Moy and Patterson [110], and Andersen et al. [5].

ABSTRACT

Contracts empower programmers to articulate specifications concerning the behavior of components. In contrast to many other forms of specification, contracts are essentially written using ordinary code. Consequently, they can perform effects. To the sympathetic, this capability is viewed as neither useful nor harmful in practice. To the skeptic, it is questionable and potentially dangerous. Rare are those who see a *need* for effectful contracts. Over the past fifteen years there has been a slow but steady stream of publications that present the design of various effectful contracts, not out of whimsical desire, but out of *necessity*. Ignoring effectful contracts entirely, or treating them as an exotic curiosity, is no longer justified.

This dissertation is the first systematic investigation of effectful contracts, showing that they can be used as a principled foundation to build expressive high-level specification languages. In support of this thesis, my dissertation contributes both theoretical and practical results. On the theory side, I formalize the semantics of effectful higher-order contracts that compose reliably and extend it to a featureful contract system that provably does not interfere with host programs. On the practice side, I develop trace contracts that enforce multi-call constraints and show how they can serve as a compilation target for high-level specification languages. These contributions are buttressed by a variety of evidence: formal models, proofs, prototypes, shipped software, benchmarks, case studies, and interviews. Drawing from this breadth of evidence, my dissertation seeks to establish that, rather than being an oddity or a nuisance, the ability to perform effects is a unique *advantage* of contracts among the many forms of specification.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Matthias Felleisen. More so than anyone else, Matthias has shaped my intellectual development over the past six years. His commitment to research, teaching, and mentorship is second to none in the programming-languages community. It has been an absolute privilege to learn from his example.

I owe thanks to my committee members: Amal Ahmed, Robby Findler, Manuel Serrano, and Jan Vitek. Each committee member has not only supplied helpful feedback on the dissertation itself, but has been incredibly supportive outside their role on the committee.

This dissertation would not be possible without the contributions of my collaborators. Thanks to Leif Andersen, Michael Ballantyne, Stephen Chang, Christos Dimoulas, Ryan Jung, Daniel Patterson, and Andrew Wagner. Their expertise and enthusiasm has pushed this work far beyond what I could have accomplished alone.

Over the years I have been fortunate enough to work with many folks on other research projects. While their contributions do not appear directly in the dissertation, they have greatly enriched my time as a graduate student. Thanks to Jack Czenszak, Steven Holtzen, Luke Jianu, John Li, Brianna Marshall, Phil Nguyễn, Lisa Oakley, Temur Saidkhodjaev, Sam Stites, Smaran Teja, Sam Tobin-Hochstadt, and David Van Horn. Special thanks to David, who served as my undergraduate advisor and got me hooked on programming-languages research.

At Northeastern, I have gotten to know many people in PLT and the PRL. Being part of these labs has been an absolute joy. I will always carry cherished memories of Wollaston's lunches and coffee discussions. I look forward to many more years of friendship with you all.

Thanks to Jenny, Hasan, Suteerth, and Rachael for keeping me sane, especially during the pandemic. Finally, I owe the greatest amount of gratitude to my parents, who have always been there for me.

CONTENTS

1 WHY EFFECTFUL CONTRACTS?

Behavioral contracts bring many benefits to software developers. A contract describes the promises a library makes about exported values and the obligations it imposes on uses. In other words, a contract is an agreement between modules about values that flow from one to the other. These agreements are specified using boolean-valued assertions from the host language that can be enforced at run time. Since they are executable, contracts provide a form of evergreen documentation for a library's interface.

As a specification medium, contracts confer numerous advantages. Because they are mostly written using ordinary code, the learning curve for contracts is a gentle slope. Once authored, contracts help programmers locate bugs by raising an exception when the system exhibits a violation of its specification. The error message in these exceptions includes both a witness to the violation and blame information pointing to the component that broke its side of the contract. But expressing specifications as ordinary code invites a question. Should contracts be free of side effects?

In practice, nearly all contracts programmers write are pure, even though API documentation often contains informal constraints that could be enforced with effectful contracts. Programmers may be reluctant to turn these informal constraints into contracts because, while most contract systems permit effectful contracts, none provide dedicated linguistic support for them. My dissertation aims to narrow this gap by making effectful contracts a standard part of the contract toolkit.

1.1 THESIS

If the philosophy of contracts is that specifications are code, I propose taking this position to its logical conclusion. Specifications are code that can perform effects. In particular, my thesis is:

> *Effectful contracts can be used as a principled foundation to build expressive high-level specification languages.*

Justifying this thesis requires being precise about what it means. A *principled* contract formally guarantees reasonable behavior; contracts should not be able to arbitrarily affect a program's evaluation. A *foundational* contract system is suitably generic, i.e., it supports a diverse array of mechanisms. An *expressive* specification language copes with complex properties that actually call for expressive power beyond pure boolean expressions. A *high-level* specification language allows

programmers to state properties concisely, without leaking low-level enforcement details involving effectful operations.

An easy way to understand the thesis is by example. Consider the HasNext property that Java programmers must respect when they use an iterator: each call to `next` must follow a call to `hasNext` [116]. Contracts enable the developer of an interface to state such specifications and attach them to all implementations.

```
(provide
  (contract-out [an-iterator-object has-next/c]))

(define has-next/c
  (object-trace/c
   #:satisfies (re (star (seq 'has-next (opt 'next))))
   [has-next (->m boolean?) 'has-next]
   [next     (->m any/c)    'next]))

(define an-iterator-object
  ⟨⟨ definition of the iterator ⟩⟩)
```

Figure 1.1: Checking the HasNext Property

Figure 1.1 shows a module that exports an iterator object and, in doing so, attaches a contract that concisely states the HasNext property with the regular expression $(\texttt{hasNext}, \texttt{next?})^*$ over an alphabet of method names. Using a concise notation, here regular expressions, clarifies the developer's intent, while permitting the automated construction of reasonably efficient dynamic checks. These checks rely on mutation to track the current state of the automaton underlying the regular expression.

1.2 CONTRIBUTIONS

In the sciences, defense of an idea demands evidence. In programming-languages research, legitimate evidence comes in many forms. This dissertation is structured around four kinds of evidence to support the thesis: theoretical, constructive, practical, and empirical.

THEORETICAL Part I considers whether effectful contracts can be principled. It turns out that the traditional and widely used model of higher-order contracts is incompatible with effects. Chapter 3 provides a new semantic substrate for higher-order contracts that perform effects consistently. Building on that model, Chapter 4 shows the design of an effectful-contract mechanism, based on effect handlers, that satisfies an erasure theorem. Roughly speaking, erasure says that the only possible impact of contracts on program execution is raising an

exception. Chapter 5 describes the design and formal semantics of a particularly useful kind of effectful contract, called a trace contract.

CONSTRUCTIVE Part II considers whether a principled design of effectful contracts can be realized. Chapter 6 discusses an implementation of effect-handler contracts in an idealized language called Effect Racket. While helpful to validate the feasibility of a design, the implementation is not yet suitable for real-world use. Thus, Chapters 7 to 9 present a trio of effectful contracts that are extensions to, or modifications of, Racket's existing contract system. Coming full circle, these three effectful contracts can be expressed as macros in Effect Racket.

PRACTICAL Part III demonstrates some affordances needed to make effectful contracts practical for software developers. One challenge is that, in their raw form, trace contracts are still too low level. Chapter 10 examines several case studies that use declarative languages, such as temporal logic, to express specifications at a higher level of abstraction. These specifications can then be compiled into trace contracts. One requirement of any new contract combinator is that it composes with existing contracts in the system. Chapter 11 illustrates the composability of trace contracts by combining them with channel contracts to enforce protocols in concurrent systems.

EMPIRICAL Part IV aims to answer, with data, whether developers might actually be able to program with effectful contracts. Chapter 12 describes the performance overhead of some effectful contracts and shows that the lightly optimized Racket implementations exhibit modest overhead. Chapter 13 reports on an effort to bring effectful contracts, and contracts generally, into the classroom. First-year students with a single semester of prior programming experience were able to use trace contracts to express fairly sophisticated temporal invariants.

In sum, this dissertation furnishes a broad spectrum of evidence seeking to establish that effectful contracts *can* be used as a principled foundation to build expressive high-level specification languages.

2 BACKGROUND

In the late 1960s, software engineering was in a crisis. With computational power accelerating exponentially [103] one might expect that programmers' ability to deliver complex software would improve at the same pace. It did not. Researchers began to ponder why. At the first NATO software engineering conference, convened specifically to understand this question, McIlroy [99] proposed a component marketplace that could supply high-quality, interchangeable, and configurable modules to deploy across different systems. A few years later, Parnas [120] recognized that, to be practical, such components would need to be accompanied by formal interface specifications.

Eiffel [100] was the first programming language to put enforceable interface specifications front and center. Contracts in Eiffel allow programmers to express preconditions that a caller must satisfy and postconditions that the result is guaranteed to satisfy. If a method's precondition fails, it is the caller's fault; if the postcondition fails, it is the callee's fault. For Meyer, the contract system undergirds an entire software design philosophy that he calls *design by contract* [101].

```
set_second (s: INTEGER)
  require
    valid_argument_for_second: 0 <= s and s <= 59
  do
    second := s
  end
```

Figure 2.1: Contracts in Eiffel

Figure 2.1 shows an example specification from Eiffel's documentation of a method that sets the `second` field of a clock object. By monitoring the precondition, Eiffel's contract system can signal an error when control enters the method—not after `second` has been set and retrieved elsewhere. Notice how Eiffel's `require` keyword cleanly separates the specification from the method body.

Inspired by Eiffel, Findler and Felleisen [56] generalized software contracts to modern higher-order languages. They implemented a higher-order contract system for Racket [52, 59], supplying programmers with notation similar to that of type systems. The contract system automatically manages blame information, ensuring that the appropriate module is named if a contract exception is raised.

```
(provide
  (contract-out
   [find-fixpoint (-> (-> real? real?) real? real?)]))

(define (find-fixpoint f x)
  (define y (f x))
  (if (= x y) x (find-fixpoint f y)))
```

Figure 2.2: Contracts in Racket

Figure 2.2 shows a higher-order contract for the `find-fixpoint` function. With the `->` contract, the function argument `f` of `find-fixpoint` is wrapped in a proxy that checks its precondition (`real?`) and postcondition (`real?`). Blame management is complex for higher-order functions because `->` is contravariant in its domain, reversing responsibility for a potentially faulty value. Without contracts, a programmer who wants to enforce this specification would be forced to pollute `find-fixpoint` with defensive checks—obscuring its implementation.

```
(define (find-fixpoint unsafe-f unsafe-x)
  (define (check-real->real g)
    (λ (x)
      (check-real (g (check-real x)))))
  (define f (check-real->real unsafe-f))
  (define x (check-real unsafe-x))
  (define y (f x))
  (check-real (if (= x y) x (find-fixpoint f y))))
```

Figure 2.3: Contracts as Ordinary Code

Figure 2.3 shows (approximately) how the `find-fixpoint` contract may be enforced. Notice how the `->` contract on the higher-order input `f` is replaced with a proxy that checks the precondition and postcondition. The underlying contract system generates all these checks automatically. Programmers should never read, let alone write, such complex code. Contract systems eliminate these awkward and repetitive patterns, allowing programmers to focus on the actual code.

The remainder of this chapter sets the stage for the theoretical part of the dissertation, establishing conventions that will be used throughout (Section 2.1) and reviewing the standard semantics [38, 41] of dependent higher-order contracts (Section 2.2).

2.1 A FUNCTIONAL BASE

This section describes a programming language, dubbed FUN, based on the untyped call-by-value λ-calculus [122]. All the models developed in this dissertation will be built atop FUN.

FUN (CORE)

$$
\begin{aligned}
\mathsf{Expr} \ni e &::= x \mid \mathtt{f} \mid e\ e \mid \mathtt{b} \mid \mathtt{if}\ e\ e\ e \\
\mathsf{Lam} \ni \mathtt{f}, \mathtt{g} &::= \lambda x.e \\
\mathsf{Bool} \ni \mathtt{b} &::= \mathtt{tt} \mid \mathtt{ff} \\
\mathsf{Var} \ni x, y, z
\end{aligned}
$$

Figure 2.4: Core Syntax of FUN

Figure 2.4 defines the core syntax of FUN. The core syntax defines a set of expressions that includes variables, functions, applications, booleans, and conditionals.

FUN (DYNAMIC)

$$
\begin{aligned}
\mathsf{Expr} \ni e &::= \ldots \mid \mathtt{err}^{k}_{j} \\
\mathsf{Inv} \ni \mathtt{i} &::= v\ w \ \lbag v \notin \mathsf{Lam} \rbag \\
\mathsf{Ctx} \ni E, F &::= \square \mid E\ e \mid v\ E \mid \mathtt{if}\ E\ e\ e \\
\mathsf{Val} \ni v, w &::= \mathtt{f} \mid \mathtt{b} \\
\mathsf{Lab} \ni \mathtt{j}, \mathtt{k}, \mathtt{l}
\end{aligned}
$$

Figure 2.5: Dynamic Syntax of FUN

Figure 2.5 defines the dynamic syntax of FUN. The dynamic syntax serves to help specify FUN's reduction relation and is not written by programmers. Side conditions are indicated within $\lbag\ \rbag$ brackets. Expressions are extended with errors that have two labels: $\mathtt{j}$ names the component that specified the invariant and $\mathtt{k}$ names the component that violated the invariant. For now, the specifier is always labeled $\mathcal{L}$ (the language runtime) and the violator is always labeled $\mathcal{P}$ (the program itself). The remaining sets are invalid expressions that should produce errors, evaluation contexts that stipulate left-to-right evaluation order, and values.

Figure 2.6 defines the semantics of FUN via a notion of reduction ($\succ$). A notion of reduction [12] relates a redex (an expression that can reduce) to its contractum (the expression that it reduces to). The $\succ$ rules specify reduction for applications, conditionals (where all values except $\mathtt{ff}$ are treated as true), and invalid expressions. Every invalid expression defined by Inv reduces to an error.

$\boxed{e \succ e}$ for FUN

$$\begin{array}{lll} \textsc{lam-app} & (\lambda x.e)\, v \succ e[v/x] & \\ \textsc{if-tt} & \mathtt{if}\ v\ e_t\ e_f \succ e_t & \lbag v \neq \mathtt{ff} \rbag \\ \textsc{if-ff} & \mathtt{if}\ \mathtt{ff}\ e_t\ e_f \succ e_f & \\ \textsc{err-inv} & i \succ \mathtt{err}^{\mathcal{P}}_{\mathcal{L}} & \end{array}$$

$\boxed{e \longmapsto e}$ for FUN

$$\begin{array}{lll} \textsc{lift} & E[e] \longmapsto E[e'] & \lbag e \succ e' \rbag \\ \textsc{err} & E[\mathtt{err}^k_j] \longmapsto \mathtt{err}^k_j & \lbag E \neq \square \rbag \end{array}$$

Figure 2.6: Semantics of FUN

The notion of reduction alone does not suffice to define an evaluator because it does not consider expressions in *context* [48]. Thus, the notion of reduction must be pushed syntactically through the set of evaluation contexts to form a reduction relation ($\longmapsto$). As shown in the LIFT rule, moving from a notion of reduction to a reduction relation requires an evaluation context E to deterministically choose a redex within a larger expression. Raising an error, which is an effect, is special because it discards the context. This behavior, given by the ERR rule, models how an uncaught exception halts program execution.

FUN (EVAL)

$$\mathsf{Ans} \ni a ::= b \mid ? \mid \mathtt{err}^k_j$$

$\boxed{\mathsf{eval} : \mathsf{Expr} \rightharpoonup \mathsf{Ans}}$ for FUN $\qquad$ $\boxed{\lfloor \cdot \rfloor : \mathsf{Val} \to \mathsf{Ans}}$ for FUN

$$\mathsf{eval}(e) := \begin{cases} \lfloor v \rfloor & e \longmapsto^* v \\ \mathtt{err}^k_j & e \longmapsto^* \mathtt{err}^k_j \end{cases} \qquad \lfloor v \rfloor := \begin{cases} v & v \in \mathsf{Bool} \\ ? & \text{otherwise} \end{cases}$$

Figure 2.7: Evaluator for FUN

Based on the reduction relation, Figure 2.7 defines the evaluator for FUN, relating programs to their final result. Evaluation is defined by the reflexive-transitive closure of the reduction relation ($\longmapsto^*$). Following Plotkin [122], the evaluator masks non-atomic values with ? (an opaque token). This behavior matches most REPLs, where function values are printed as an opaque token.

2.2 THE CLASSIC INDY MODEL

INDY (CORE) extends FUN

$$\mathsf{Expr} \ni e ::= \ldots \mid \mathtt{mon}^{k,l}_j\, e\, e \mid e \Rightarrow f$$

Figure 2.8: Core Syntax of INDY

Extending FUN, Figure 2.8 defines the core syntax for INDY, a model of dependent higher-order contracts [38, 41]. The core syntax extends FUN with two new elements: monitors $\texttt{mon}_j^{k,l}\ e_c\ e_s$ and dependent function contracts $e \Rightarrow f$.

A monitor is used to attach a contract to a value. So, $\texttt{mon}_j^{k,l}\ e_c\ e_s$ attaches the contract produced by e_c to the value of e_s. The value of e_s, i.e., the value to be protected by the contract, is dubbed the *subject* of the contract. Monitors also come with labels naming the parties that agreed to the contract: the contract-defining module j, the server module k, and the client module l. In Racket, all contract attachment forms expand into a single primitive attachment form, similar to `mon`, annotated with the appropriate blame labels.

A dependent function contract describes properties of functions where the codomain contract *depends* on the argument to the subject function. For $e \Rightarrow f$, the contract system ensures f is applied to the argument of the subject function and expects f to return a codomain contract constraining the output. Ordinary function contracts, written $e_d \to e_c$, are macro expressible [47] as $e_d \Rightarrow (\lambda\_.e_c)$.

In addition to dependent function contacts, all other values can be used as contracts in this model. When used as a contract, a function checks first-order properties of the subject. In Racket, these checks correspond to predicates. Booleans correspond to trivial contracts where `tt` permits any value and `ff` forbids all values. In Racket, these contracts correspond to `any/c` and `none/c`, respectively.

INDY (DYNAMIC) extends FUN

$$\mathsf{Ctx} \ni E, F ::= \ldots \mid \texttt{mon}_j^{k,l}\ E\ e \mid \texttt{mon}_j^{k,l}\ v\ E \mid E \Rightarrow f$$

$$\mathsf{Val} \ni v, w ::= \ldots \mid v \Rightarrow f$$

Figure 2.9: Dynamic Syntax of INDY

Figure 2.9 shows the dynamic syntax of INDY, extending the set of evaluation contexts and values. These definitions are entirely standard. Because all values can be used as contracts in this model, there is no need to extend the set of invalid expressions.

Here is an example program with a contract:

$$\texttt{mon}_{\texttt{ctc}}^{\texttt{lib,main}}\ (\texttt{tt} \Rightarrow \lambda x.\lambda y.x = y)\ (\lambda z.z).$$

This contract specifies the behavior of the identity function $(\lambda z.z)$. Suppose this function was applied to 0. Since the domain contract is `tt`, every argument, including 0, is accepted. When the function returns, the output value is checked against the codomain contract $\lambda y.0 = y$, ensuring that the output is equal to zero.[1]

1 Programs in the theoretical part of this dissertation may use language features that are not formally defined. Their meaning should always be clear from context.

$\boxed{e \succ e}$ for INDY

$$\vdots$$

MON-BOOL $\quad \mathtt{mon}_j^{k,l}\, b\, v \succ \mathtt{if}\, b\, v\, \mathtt{err}_j^k$

MON-LAM $\quad \mathtt{mon}_j^{k,l}\, f\, v \succ \mathtt{mon}_j^{k,l}\, (f\, v)\, v$

MON-ARR $\quad \mathtt{mon}_j^{k,l}\, (w \Rightarrow g)\, f \succ \lambda x.\mathtt{let}\, x_j = \mathtt{mon}_j^{l,j}\, w\, x\, \mathtt{in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{let}\, x_k = \mathtt{mon}_j^{l,k}\, w\, x\, \mathtt{in}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{mon}_j^{k,l}\, (g\, x_j)\, (f\, x_k)$

ERR-ARR $\quad \mathtt{mon}_j^{k,l}\, (w \Rightarrow g)\, v \succ \mathtt{err}_j^k \quad \wr v \notin \mathsf{Lam} \int$

Figure 2.10: Semantics of INDY

Figure 2.10 shows the notion of reduction for INDY. Because INDY extends FUN, the notion of reduction extends that of FUN—indicated by ellipses. The four additional rules describe checks performed by each kind of contract. For booleans, the contract succeeds or fails, respectively. For a function f, the result of applying f to the subject is then used as the new contract. If f is a predicate, this rule corresponds exactly to a first-order check because `tt` and `ff` are themselves contracts.

While f may return a boolean, there is nothing that forces it to be one. In particular, it could return another function. This process can go on repeatedly until it reaches a base case (i.e., a boolean). A contract following this pattern is called a *cascading contract*, and can be used to combine arbitrary first-order checks with higher-order contracts:

$$\lambda f.\mathtt{if}\, (\mathtt{arity}\, f = 1)\, (\mathtt{int?} \to \mathtt{int?})\, \mathtt{ff}.$$

This cascading contract checks a first-order constraint, namely that the subject has arity one. If successful, the higher-order contract `int?` $\to$ `int?` protects the subject. Otherwise, the contract fails eagerly.

Finally, MON-ARR describes the indy semantics of dependent function contracts [41]. The key insight is that the contract itself can be inconsistent and therefore must be checked. Consider the following contract:

$$(\mathtt{bool?} \to \mathtt{bool?}) \Rightarrow \lambda f.f\, 0.$$

While the domain states that the input is a function over booleans, the contract itself violates that assumption by applying f to 0 when generating the codomain contract. In this case, INDY signals an error blaming *the contract itself*. To do so, MON-ARR protects the argument twice: once where the client label is j (bound to x_j) and once where the client label is k (bound to x_k).[2] These values are provided to the codomain constructor g and the subject f, respectively, ensuring that any eventual blame information points to the responsible party.

2 Variables present on the right-hand side of a rule that are not present on the left-hand side are assumed to be fresh in the sense of Barendregt's hygiene convention [12].

Part I

THEORETICAL

Three formal models are presented in this part: (1) a modification of INDY where effectful contracts compose reliably; (2) a model of effect-handler contracts that unifies the landscape of effectful contracts; and (3) a model of trace contracts for enforcing multi-call constraints. These models are the foundation upon which later implementations rest.

3 A FOUNDATION FOR EFFECTFUL CONTRACTS

As is, INDY cannot reliably accommodate contracts that perform effects. An effectful contract, when used as the domain of a function, has its effects *duplicated*. Consider the following example:

$$\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\,((\lambda x.\mathtt{print}\ x\ ;\ \mathtt{tt}) \Rightarrow \lambda x.\lambda y.x = y)\ (\lambda z.z).$$

Evaluating the domain contract performs an effect. As the following reduction sequence demonstrates, where redexes are highlighted, `print` is executed twice under the INDY semantics:

$$(\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\,((\lambda x.\mathtt{print}\ x\ ;\ \mathtt{tt}) \Rightarrow \lambda x.\lambda y.x = y)\ (\lambda z.z))\ 0$$

The monitor expression must produce a proxy, via MON-ARR, that checks the arguments against the domain contract and the return value against the codomain contract.

$$\longmapsto\ (\lambda x.\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ ((\lambda x.\lambda y.x = y)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{ctc}}\ (\lambda y.\mathtt{print}\ y\ ;\ \mathtt{tt})\ x))\ ((\lambda z.z)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{lib}}\ (\lambda y.\mathtt{print}\ y\ ;\ \mathtt{tt})\ x)))\ 0$$

The proxy (where x_j and x_k are inlined for brevity) is applied to the argument 0.

$$\longmapsto\ \mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ ((\lambda x.\lambda y.x = y)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{ctc}}\ (\lambda y.\mathtt{print}\ y\ ;\ \mathtt{tt})\ 0)\ )\ ((\lambda z.z)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{lib}}\ (\lambda y.\mathtt{print}\ y\ ;\ \mathtt{tt})\ 0))$$

To produce a codomain contract, the argument is first checked (by MON-LAM) against the domain contract with the contract-defining party (`ctc`) as the client label. At this point in time, 0 is printed.

$$\longmapsto^{+}\ \mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ ((\lambda x.\lambda y.x = y)\ 0)\ ((\lambda z.z)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{lib}}\ (\lambda y.\mathtt{print}\ y\ ;\ \mathtt{tt})\ 0))$$

Once the argument is checked, then the codomain contract is constructed.

$$\longmapsto\ \mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\ (\lambda y.0 = y)\ ((\lambda z.z)\ (\mathtt{mon}_{\mathtt{ctc}}^{\mathtt{main},\mathtt{lib}}\ (\lambda y.\mathtt{print}\ y\ ;\ \mathtt{tt})\ 0)\ )$$

The argument has to be checked against the domain contract (by MON-LAM) once more. This time the client label is `lib`. Again, 0 is printed.

$\longmapsto^+ \mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\,(\lambda y.0 = y)\ ((\lambda z.z)\ 0)$

Now the subject is applied to 0. Since the subject is the identity function, it returns 0.

$\longmapsto \quad \mathtt{mon}_{\mathtt{ctc}}^{\mathtt{lib},\mathtt{main}}\,(\lambda y.0 = y)\ 0$

The returned value is checked against the generated codomain contract. In this case, the contract is satisfied and can be discharged.

$\longmapsto^+ 0$

Effect duplication is a major obstacle for building principled effectful contracts. To understand the source of the problem, consider the right-hand side of the MON-ARR rule from INDY:

$$\begin{aligned}\lambda x.\mathtt{let}\ x_j &= \mathtt{mon}_j^{l,j}\,w\ x\ \mathtt{in}\\ \mathtt{let}\ x_k &= \mathtt{mon}_j^{l,k}\,w\ x\ \mathtt{in}\\ \mathtt{mon}_j^{k,l}\,(g\ x_j)\ (f\ x_k)&.\end{aligned}$$

There are two monitor expressions that contain w, differing only in their client label. If w performs an effect, it happens twice. These monitors cannot be collapsed into a single monitor because the client labels on x_j and x_k must differ.[1] A new semantics for higher-order dependent contracts is needed to accommodate effects.

The key observation that resolves the dilemma is that INDY collapses *interception time* and *crossing time*. Interception time occurs when the contract system intercepts a value from the monitored program, i.e., when a value flows through a contract interception point. Crossing time occurs when an intercepted value moves to another component.

Concretely, consider a proxy created by the contract $w \Rightarrow g$. Every time the proxy is applied it must perform two tasks related to the argument: (1) w must be used to check first-order properties of the argument; and (2) if w is a higher-order contract, proxies must be created for every consumer of the argument. Interception time corresponds to when task (1) occurs and crossing time corresponds to when task (2) occurs.[2] The INDY language has a single monitor form that performs both tasks. Splitting the three-labeled monitor into two separate forms allows greater control over when contract effects happen. Effects occur when one of the forms evaluates but not when the other evaluates.

1 More precisely, alternative semantics violate desirable metatheoretic properties of the contract system. Eliminating the x_j monitor yields a "lax" contract semantics that violates complete monitoring [43]. Reusing x_k for the argument to g and f yields a "picky" contract semantics that violates blame correctness [41].

2 While this claim is usually true, there are exceptions. The `unconstrained-domain->` contract in Racket makes no demand on function arguments. Because such a contract is guaranteed never to blame clients, its proxy can be constructed at interception time.

CONTRACT (CORE) extends FUN

$$\text{Expr} \ni e ::= \ldots \mid \mathtt{mon}_j^{k,l}\, e\, e \mid e \Rightarrow f$$

Figure 3.1: Core Syntax of CONTRACT

Figure 3.1 defines the core syntax of CONTRACT, an extension of FUN that resolves the effect duplication problem with INDY. The core syntax is identical to that of INDY from Figure 2.8. However, the dynamic syntax and semantics differs substantially.

CONTRACT (DYNAMIC) extends FUN

$$\begin{aligned}
\text{Expr} \ni e &::= \ldots \mid \mathtt{mon}_j^k\, e\, e \mid \mathtt{grd}_j^k\, v\, v \mid e \cdot l \\
\text{Inv} \ni i &::= \ldots \mid \mathtt{mon}_j^{k,l}\, w\, v \quad \wr w \notin \text{Con} \int \\
\text{Con} \ni c &::= f \mid b \mid e \Rightarrow f \\
\text{Ctx} \ni E, F &::= \ldots \mid \mathtt{mon}_j^k\, E\, e \mid \mathtt{mon}_j^k\, v\, E \mid E \cdot l \mid E \Rightarrow f \\
\text{Val} \ni v, w &::= \ldots \mid \mathtt{grd}_j^k\, v\, v \mid v \Rightarrow f
\end{aligned}$$

Figure 3.2: Dynamic Syntax of CONTRACT

Figure 3.2 defines the dynamic syntax of CONTRACT. The dynamic syntax adds three constructs to FUN: two-labeled monitors $\mathtt{mon}_j^k\, e_c\, e_s$, guarded values $\mathtt{grd}_j^k\, w\, v$, and label applications $e \cdot l$. Reduction of $\mathtt{mon}_j^k\, w\, v$ corresponds to interception time, when first-order properties of the subject are checked. If the contract performs effects, it does so at interception time. Successful evaluation of a two-labeled monitor yields a guarded value that is waiting for a client label in order to produce a proxy. Reduction of $(\mathtt{grd}_j^k\, w\, v) \cdot l$ corresponds to crossing time and creates a proxy of v for client l. Constructing a proxy does not perform effects.

Figure 3.3 displays the notion of reduction for CONTRACT. The first rule, MON, decomposes a three-labeled monitor into a two-labeled monitor applied to the client label. The remaining rules are responsible for performing the first-order checks of each contract and producing guarded values upon success. Compared to the INDY rules shown in Figure 2.10, the action of MON-BOOL is now divided between MON-BOOL and GRD-TT. The same goes for MON-ARR; it is now divided between MON-ARR and GRD-ARR.

In GRD-TT, there is no need for a proxy, so the subject is produced directly. A proxy is needed for function contracts. In contrast to INDY, the proxy on the right-hand side of GRD-ARR can now exploit the two stages of contract checking. Instead of two monitors containing w, there is now only one, with its result bound to x_g. Effects caused by checking

$\boxed{e \succ e}$ for CONTRACT

$$\vdots$$

$$\begin{array}{rl}
\textsc{mon} & \mathtt{mon}_j^{k,l}\ e_c\ e_s \succ (\mathtt{mon}_j^k\ e_c\ e_s) \cdot l \\
\textsc{mon-lam} & \mathtt{mon}_j^k\ f\ v \succ \mathtt{mon}_j^k\ (f\ v)\ v \\
\textsc{mon-bool} & \mathtt{mon}_j^k\ b\ v \succ \mathtt{if}\ b\ (\mathtt{grd}_j^k\ \mathtt{tt}\ v)\ \mathtt{err}_j^k \\
\textsc{mon-arr} & \mathtt{mon}_j^k\ (w \Rightarrow g)\ f \succ \mathtt{grd}_j^k\ (w \Rightarrow g)\ f \\
\textsc{err-arr} & \mathtt{mon}_j^k\ (w \Rightarrow g)\ v \succ \mathtt{err}_j^k \quad \wr v \notin \mathsf{Lam} \int \\
\textsc{grd-tt} & (\mathtt{grd}_j^k\ \mathtt{tt}\ v) \cdot l \succ v \\
\textsc{grd-arr} & (\mathtt{grd}_j^k\ (w \Rightarrow g)\ f) \cdot l \succ \lambda x.\mathtt{let}\ x_g = \mathtt{mon}_j^l\ w\ x\ \mathtt{in} \\
 & \qquad\qquad \mathtt{let}\ x_j = x_g \cdot j\ \mathtt{in} \\
 & \qquad\qquad \mathtt{let}\ x_k = x_g \cdot k\ \mathtt{in} \\
 & \qquad\qquad \mathtt{mon}_j^{k,l}\ (g\ x_j)\ (f\ x_k)
\end{array}$$

Figure 3.3: Semantics of CONTRACT

w occur only once (while obtaining the value for x_g). In the scope of this `let` binding, two proxies are produced through label application of x_g to j and k. Remember, constructing proxies is a pure operation.

To see the semantics in action, consider how CONTRACT evaluates the example from earlier:

$$(\mathtt{mon}_{ctc}^{lib,main}\ ((\lambda x.\mathtt{print}\ x\ ;\ \mathtt{tt}) \Rightarrow \lambda x.\lambda y.x = y)\ (\lambda z.z))\ 0$$

A three-labeled monitor immediately decomposes (by MON) into a two-labeled one, applied to the client label.

$$\longmapsto\ (\ (\mathtt{mon}_{ctc}^{lib}\ ((\lambda x.\mathtt{print}\ x\ ;\ \mathtt{tt}) \Rightarrow \lambda x.\lambda y.x = y)\ (\lambda z.z))\ \cdot \mathtt{main})\ 0$$

The two-labeled monitor produces (by MON-ARR) a guarded value that is ready to construct a proxy.

$$\longmapsto\ ((\mathtt{grd}_{ctc}^{lib}\ ((\lambda x.\mathtt{print}\ x\ ;\ \mathtt{tt}) \Rightarrow \lambda x.\lambda y.x = y)\ (\lambda z.z)) \cdot \mathtt{main})\ 0$$

Applying the guarded value to a client label, via GRD-ARR, produces a proxy.

$$\begin{array}{ll}
\longmapsto & (\lambda x.\ \mathtt{let}\ x_g = \mathtt{mon}_{ctc}^{main}\ (\lambda x.\mathtt{print}\ x\ ;\ \mathtt{tt})\ x\ \mathtt{in} \\
 & \qquad \mathtt{mon}_{ctc}^{lib,main}\ ((\lambda x.\lambda y.x = y)\ (x_g \cdot \mathtt{ctc}))\ ((\lambda z.z)\ (x_g \cdot \mathtt{lib})))\ 0
\end{array}$$

The proxy (where x_j and x_k are inlined for brevity) is applied to the argument 0.

$\longmapsto$ $\texttt{let}\ x_g = \texttt{mon}^{\texttt{main}}_{\texttt{ctc}}\ (\lambda x.\texttt{print}\ x\ ;\ \texttt{tt})\ 0\ \texttt{in}$
$\texttt{mon}^{\texttt{lib,main}}_{\texttt{ctc}}\ ((\lambda x.\lambda y.x = y)\ (x_g \cdot \texttt{ctc}))\ ((\lambda z.z)\ (x_g \cdot \texttt{lib}))$

Evaluation of the two-labeled monitor (by MON-LAM) prints out 0.

$\longmapsto^+$ $\texttt{let}\ x_g = \texttt{grd}^{\texttt{main}}_{\texttt{ctc}}\ \texttt{tt}\ 0\ \texttt{in}$
$\texttt{mon}^{\texttt{lib,main}}_{\texttt{ctc}}\ ((\lambda x.\lambda y.x = y)\ (x_g \cdot \texttt{ctc}))\ ((\lambda z.z)\ (x_g \cdot \texttt{lib}))$

The guarded value is substituted into the function body.

$\longmapsto$ $\texttt{mon}^{\texttt{lib,main}}_{\texttt{ctc}}\ ((\lambda x.\lambda y.x = y)\ ((\texttt{grd}^{\texttt{main}}_{\texttt{ctc}}\ \texttt{tt}\ 0) \cdot \texttt{ctc}))$
$((\lambda z.z)\ ((\texttt{grd}^{\texttt{main}}_{\texttt{ctc}}\ \texttt{tt}\ 0) \cdot \texttt{lib}))$

Evaluation of each guarded value yields 0 (by GRD-TT) but does *not* print.

$\longmapsto^+$ $\texttt{mon}^{\texttt{lib,main}}_{\texttt{ctc}}\ (\lambda y.0 = y)\ ((\lambda z.z)\ 0)$

The subject is applied to the argument and then the final result is checked.

$\longmapsto^+$ 0

Crucially, `print` is executed just once under CONTRACT—when the two-labeled monitor is reduced. Creating proxies from guarded values, which occurs twice, does not `print`.

4 EFFECT-HANDLER CONTRACTS FORMALLY

Existing contract systems can deal with a wide range of functional properties, but none have the expressive power to systematically enforce properties that require contract-level effects. For example, a library may wish to guarantee that a function is called at most once, necessitating a piece of contract-local state [143]. While the model from Chapter 3 is a good starting point for studying effectful contracts, simply adding primitive effects to it easily runs afoul of desirable metatheoretic properties.

Effect-handler contracts are a uniform mechanism for expressing *well-behaved* effectful contracts. The formal model of effect-handler contracts (Sections 4.2 to 4.4), building atop CONTRACT, consists of a language where all effectful operations are expressed in terms of effect requests and handlers [123], not as primitive operations. In the context of such a language, effect-handler contracts suffice to check a broad class of constraints while satisfying the critical *erasure* property (Section 4.5). Intuitively, erasure means that contracts cannot interfere with a program's computation, other than signaling a contract error and halting execution. Consequently, contracts that are expressible as effect-handler contracts also satisfy erasure.

4.1 EXAMPLES

Before diving into the formal model, it will be helpful to get a taste of effect-handler contracts. This section introduces effect handlers and effect-handler contracts with synthetic examples written in the Effect Racket language. Aside from regular contracts, there are four critical features of Effect Racket: main-effect handlers, main-effect contracts, contract-effect handlers, and contract-handler contracts.

HIGHER-ORDER CONTRACTS The RSA cryptographic algorithm is widely used for secure communication [127]. Crucially, RSA relies on the difficulty of factoring prime numbers.

Figure 4.1 shows an RSA-key-generating function with contracts for the primality constraint. The `rsa` function is higher-order, transforming a prime-generating function into a key-generating function. If the contract system discovers a violation, an error is signaled identifying the contract and blaming the responsible party.[1]

1 For background reading, Chapter 7 of the Racket Guide [58] gives a gentle introduction to contracts in Racket. Chapter 8 of the Racket Reference [59] contains a detailed description of Racket's contract system.

```
(provide
  (contract-out
   [rsa (-> prime-gen/c key-gen/c)]))

(define prime-gen/c (-> (values prime? prime?)))
(define key-gen/c (-> (values pub-key? priv-key?)))
(define (rsa pg) ── implementation ── )
```

Figure 4.1: RSA Key Generation

Given an invalid `pg` function (i.e., one that does not always generate primes), the contract system can identify the source of the violation:

```
> ((rsa (λ () (values 3 4))))
rsa: contract violation
  expected: prime?
  given: 4
  blaming: top level
```

MAIN-EFFECT HANDLERS A key requirement of RSA is that the prime numbers used to generate the keys are random. To generate random primes, a program must be able to acquire ordinary random numbers. A pseudorandom number generator (PRNG) is a deterministic algorithm for generating numbers with properties similar to truly random numbers. The interface to most PRNGs is effectful: generating a random number causes the PRNG's internal state to change.

```
(effect random ())

;; with-random: (→ A) → A
(define (with-random thk)
  (with ((random-handler (seed->state INITIAL-SEED)))
    (thk)))

;; random-handler: State → Handler
(define (random-handler st)
  (handler
    [(random)
     (with ((random-handler (state-next st)))
       (continue* (state->number st)))]))
```

Figure 4.2: Main-Effect Handlers in Racket

Figure 4.2 shows how a PRNG can be implemented using effect handlers. From a programmer's perspective, effect handlers generalize exception handlers by allowing a handler to resume evaluation where an

exception was raised. Resumption is enabled by multi-shot delimited continuations [46]. Generalizing in this way allows effect handlers to simulate a variety of effectful operations—not just exceptions.

First, the `effect` form declares an effect called `random` that takes no arguments. Think of an `effect` declaration as similar to declaring an exception type that can be raised by applying the declared effect as a function. So, `(random)` searches the context for the innermost matching handler and invokes it to perform the effect.

An effect handler is installed via the `with` form, as seen in the `with-random` function. When given a thunk, `with-random` runs the thunk in a context where random-number generation is available. To offer random-number generation, `with-random` applies the thunk inside a `with` form that contains the `random-handler` effect handler, initialized with a PRNG state created from `INITIAL-SEED`. When given the state of a random-number generator, the `random-handler` function returns a handler that interprets requests for random generation.

The `handler` form, modeled after `match`, returns a *main-effect handler*. When an effect request matches the given pattern, originating from the main code of a program (i.e., not contract code), the associated arm is executed. Effect requests that do not match are propagated further up the context. If no matching handler is found in the context, then an exception is raised. The handler returned from `random-handler` services `random` effects by applying the continuation (bound to `continue*`) to the next random number (extracted from the PRNG state). This example program generates two random numbers:

```
> (with-random (λ () (list (random) (random))))
'(0.7769506309660458 0.5790039700066731)
```

Typically, effect handlers come in two flavors: *deep* [23, 123] and *shallow* [77]. A deep continuation resumes with the same handler installed. A shallow continuation resumes without the handler installed. Effect Racket supports both: `continue` is deep and `continue*` is shallow. Shallow continuations are often used when future interpretations of the effect must change—as is the case here. Future `random` requests should be serviced according to the *next* PRNG state, otherwise the random-number generator would always return the same number. The `with` in `random-handler` installs a new handler with the correct behavior.

Effect composition is the key benefit of an effect-handler-based language over a language with primitive effects. Since an effect-handler language expresses effects *uniformly*, it is straightforward to reinterpret them. In particular, a programmer can replace or supplement the PRNG provided by `with-random`, leaving the computation (`thk`) intact.

For example, the handler in Figure 4.3 biases the distribution of random numbers by squaring them. Assuming the original PRNG produces reals in $[0, 1]$, this new handler can be composed with the original PRNG to yield a biased generator.

```
;; with-bias: (→ A) → A
(define (with-bias thk)
  (with (bias-handler) (thk)))

(define bias-handler
  (handler
    [(random)
     (continue (sqr (random)))]))
```

Figure 4.3: Biasing Handler

The following idealized reduction sequence shows roughly how the composition of `with-random` and `with-bias` evaluates in Effect Racket:

```
      (with-random (λ () (with-bias (λ () (random)))))

⟼+ (with ((handler ── st0 ──))
       (with (bias-handler) (random)))

⟼  (with ((handler ── st0 ──))
       (define (continue x)
         (with (bias-handler) x))
       (continue (sqr (random))))

⟼  (define (continue x)
       (with (bias-handler) x))
     (define (continue* x)
       (continue (sqr x)))
     (with ((random-handler (state-next st0)))
       (continue* (state->number st0)))

⟼+ (define (continue x)
       (with (bias-handler) x))
     (define (continue* x)
       (continue (sqr x)))
     (with ((handler ── st1 ──))
       (continue* n))

⟼+ (define (continue x)
       (with (bias-handler) x))
     (with ((handler ── st1 ──))
       (continue n²))

⟼  (with ((handler ── st1 ──))
       (with (bias-handler) n²))

⟼+ n²
```

MAIN-EFFECT CONTRACTS In the presence of I/O effects, the contract for `rsa` does not suffice. A program may accidentally (or intentionally) supply a prime-generating function that reveals more information than it should:

```
(define (bad-pg)
  (define-values (p q) — compute primes — )
  (displayln (format "primes are ~a and ~a" p q))
  (values p q))
```

In this snippet, the prime-generating function prints the secret primes, thus compromising the RSA key. A contract for `rsa` should prohibit effectful procedures such as `bad-pg`.

With *main-effect contracts*, constructed using `->e`, expressing this restriction is easy:

```
(define prime-gen-v2/c
  (and/c prime-gen/c (->e random? (real-in 0 1))))
```

The revised contract is a conjunction of two pieces. To protect a value, `and/c` uses each of the two conjuncts to protect the value, one after another. Consequently, the prime-generating function must satisfy both. While the first conjunct is the original `prime-gen/c` contract, the second one restricts effects. The `random?` contract is a constraint on the function itself, whereas `real-in` is a constraint on the context at call sites. In this example, `(->e random? (real-in 0 1))` ensures that effect requests satisfy `random?` and that a use-site handler passes values to the continuation only if they satisfy `(real-in 0 1)`. Since `random?` yields `#f` on the effect request produced by `displayln`, providing `bad-pg` raises an error under this contract. Note that even built-in effects, such as `displayln`, can be handled in Effect Racket.

```
;; with-log: (→ A) → A
(define (with-log thk)
  (with (log-handler) (thk)))

(define log-handler
  (handler
    [(random)
     (define res (random))
     (displayln (format "random was ~a" res))
     (continue res)]))
```

Figure 4.4: Logging Handler

Main-effect contracts are active only during calls to the subject function and not at any other point—resulting in some surprising behavior. Take the handler given in Figure 4.4 that intercepts all random-number

requests and prints them out. In the same way as `bad-pg`, the handler can be used to expose secret information:

```
(with-random (λ () (with-log (λ () (rsa pg)))))
```

However, this program does not result in a contract violation, even when the contract placed on `pg` is `prime-gen-v2/c`. When `pg` requests a random number, evaluation moves to the body of `log-handler`, outside the dynamic extent of `pg`. Therefore, `prime-gen-v2/c` is no longer active when `log-handler` prints.

This behavior is by design and is necessary for correctly assigning blame. A contract establishes an agreement between a client and a server. According to `prime-gen-v2/c`, the `pg` function is responsible only for ensuring that its code does not perform forbidden effects directly, or indirectly by calling other functions. Client code, including the code that calls `pg` and the code that handles the legitimate effects `pg` performs, is not beholden to this part of the contract. Raising a contract error is wrong, even if `log-handler` prints.

The `->e` contract combinator is not expressive enough for some situations. In particular, `->e` cannot express properties where there is a dependency on the effect request in the contract for the handler response.

Consider an adaptation of `random` with a parameter that yields a random integer between 0 and k (inclusive):

```
(effect random-int (bound))
```

Guaranteeing that the random number is within bounds calls for `->de`, the dependent main-effect contract combinator:

```
(->de any/c
      (λ (req)
        (match req
          [(random-int bound) (integer-in 0 bound)])))
```

The `->de` contract allows the contract on the handler response to depend on the effect performed. In this example, the second component of `->de` is supplied with the `random-int` request itself, including the bound. It matches on the effect request and returns a contract that ensures the random value is within range.

CONTRACT-EFFECT HANDLERS The `prime-gen-v2/c` contract guarantees that the thunk always receives a real number from the PRNG handler in response to its requests, but it gives no assurance that these real numbers are even somewhat random. A PRNG function that always returns $\frac{1}{2}$ does not cause an error, but it does yield a useless prime generator. Statistical tests exist to detect faulty PRNGs [17, 89], and a contract can employ them to detect bad PRNG implementations.

Consider a simple test that ensures two consecutive random numbers are different.[2] The first step to realize such a check is to implement contract-local state with a *contract-effect handler*. A contract-effect handler is created using the `contract-handler` form.

```
(effect diff (prev-val))

;; with-diff: (→ A) → A
(define (with-diff thk)
  (with ((diff-handler -1)) (thk)))

;; diff-handler: Real → Contract-Handler
(define (diff-handler prev)
  (contract-handler
    [(diff cur)
     (values (not (= prev cur))
             (diff-handler cur))]))
```

Figure 4.5: Contract-Effect Handler for Random Number Difference

Figure 4.5 displays a contract-effect handler for keeping track of the previous value returned by a PRNG. When executed inside `with-diff`, contracts can use this test to determine whether the most recently generated random number differs from the previously generated one. The `contract-handler` form is a restricted version of `handler` that interprets only effects requested in the dynamic extent of a contract check. Critically, `contract-handler` affects only contract-checking code because it can transfer values only to contract code.

Additionally, a `contract-handler` does not get to directly invoke the delimited continuation of the effect request. Instead, the handler is expected to return a pair of values: the effect result and a new handler to replace the current one.[3] Direct access to the delimited continuation would permit tampering with the program's result and would allow interference between program code and contract code, violating erasure. For example, a handler could ignore the continuation completely and return an arbitrary value. This flexibility is necessary for main-effect handlers that implement exceptions but is not desirable for contract-effect handlers.

2 Typically, a contract error signals a definite violation of the specification. By contrast, this test signals that the random-number generator is likely faulty. For the implementation of `random`, the contract is reasonable. However, a correct implementation of `random-int` may plausibly yield two of the same numbers consecutively. In such a setting, this contract is not suitable.

3 This restriction is similar to that of a runner [1] where, informally, a handler *may* invoke the continuation at most once in tail position. Here, the handler *must* invoke the continuation exactly once in tail position.

Despite its limitations, `contract-handler` is still pretty capable. Adapting `prime-gen-v2/c` yields a contract with the desired test:

```
(define diff-real?
  (and/c real? diff))

(define prime-gen-v3/c
  (and/c prime-gen/c (->e random? diff-real?)))
```

Here, `diff-real?` requests an effect whose purpose is to check whether the latest value differs from the most recent one. Performing the `diff` effect simultaneously checks that the latest value is acceptable and updates the contract-local state. In this context, `diff` is an effectful predicate and can therefore be used as a contract. With `prime-gen-v3/c` and its corresponding effect handler installed, a constant PRNG signals a contract error.

The `prime-gen-v3/c` example illustrates why contracts themselves may need to perform effects. Mutable state is essential for tracking the previous value generated by the PRNG. In this example, state must also *persist* across multiple calls to the prime-generating function. If `prime-gen-v3/c` reset its state on every invocation of the prime-generating function, then more faulty PRNGs would pass the contract.

CONTRACT-HANDLER CONTRACTS For unit testing, the author of `rsa` may want to use a pre-determined pool of numbers for random generation instead of a PRNG. Suppose `with-pool` is a function that, when given a thunk and a list of random numbers, applies the thunk in a context where random numbers are generated from the pool:

```
> (with-pool (λ () (list (random))) '(0.5 0.1 0.7))
'(0.5)
> (with-pool (λ () (list (random) (random))) '(0.5 0.1 0.7))
'(0.5 0.1)
```

It is important that the number of times a thunk requests a random number does not exceed the size of the pool. To enforce that property, a piece of contract-local state is needed to keep track of how many times `random` is invoked inside a thunk. However, that contract-local state must be associated with the dynamic extent of the thunk given to `with-pool`. State persisted across the entire program, implemented using a contract-effect handler, does not work. A *contract-handler contract*, constructed via `with/c`, is needed to install a contract-effect handler during the dynamic extent of the thunk given to `with-pool`.

Figure 4.6 sketches the outline of `with-pool`. The contract attached to `with-pool`, in particular the `pool/c` component, ensures that `thk` does not exhaust the fixed-size pool of random numbers. Because the constraint on the thunk depends upon the size of the pool, the dependent-function combinator `->i` must be employed. The `->i` form first takes in a list of named arguments and associated contracts. If a domain con-

```
(provide
  (contract-out
   [with-pool
    (->i ([thk (xs) (pool/c (length xs))]
          [xs list?])
         [res any/c])]))

(define (with-pool thk xs)
  (with ((random-pool-handler xs))
    — implementation —))

;; random-pool-handler: List → Handler
(define (random-pool-handler xs)
  (handler
    [(random)
     (with ((random-pool-handler (rest xs)))
       (continue* (first xs)))]))
```

Figure 4.6: Random-Number Pool

tract depends on other arguments, then it must be preceded by a list of dependencies. After the domain contracts, there is one final contract for the codomain.

Figure 4.7 displays the definition of `pool/c` and auxiliary functions. At a high level, `pool/c` needs its own piece of state to keep track of how many times `random` is invoked. That piece of state is supplied by the `with/c` contract-handler contract. During the dynamic extent of the thunk that the `pool/c` contract guards, `with/c` installs the given contract handler. In this case, `rem-handler` stores the number of values remaining in the random-number pool. So, `with-pool` executes `thk` in a context where *contracts* can perform the `remaining` effect that is then interpreted by `rem-handler`.

On its own, a contract-handler contract cannot signal a violation. Rather, it supports other contracts that can raise an exception. Here, the `->e` conjunct of `pool/c` uses `rem?` to ensure that, if the thunk requests a random number, the pool still has elements remaining.[4] If so, the request is propagated. Otherwise, an error is raised.

SUMMARY Taking stock, there are the four primary constructs of Effect Racket: (1) `handler` produces a main-effect handler interpreting only effects performed by ordinary code in the body of a `with` expression; (2) `contract-handler` produces a contract-effect handler inter-

4 The order of conjuncts in `pool/c` is relevant. Since `rem?` requires that `rem-handler` is installed, it must come earlier in the list of conjuncts than `with/c`. Because `and/c` applies contracts left-to-right, the right-most conjunct creates the outermost proxy.

```
(effect remaining ())

;; pool/c: Natural → Contract
(define (pool/c k)
  (and/c (->e rem? real?) (with/c (rem-handler k))))

;; rem?: Contract
(define rem?
  (if/c random? (λ (x) (positive? (remaining))) any/c))

;; rem-handler: Natural → Contract-Handler
(define (rem-handler k)
  (contract-handler
    [(remaining)
     (values k (rem-handler (sub1 k)))]))
```

Figure 4.7: Random-Number Pool Contract

preting only effects performed by contract-checking code in the body of a `with` expression; (3) `(->e c1 c2)` produces a main-effect contract that ensures effects performed during the application of the protected function satisfy `c1` and values received from the handler satisfy `c2`; and finally (4) `(with/c h)` produces a contract-handler contract that handles (using `h`) effect requests during the application of the protected function within contract-checking code.

4.2 CORE SYNTAX

The remainder of this chapter presents EFFECT, a model of a language with effect-handler contracts, extending CONTRACT. This model is intended to capture the essence of Effect Racket.

EFFECT (CORE) extends CONTRACT

Expr ∋ e ::= ... | $\texttt{with}^{\mathcal{M}}$ e e | $\texttt{with}^{\mathcal{C}}$ e e | p | do e | e $\mathcal{M}$ f | $\mathcal{C}$ f

Pair ∋ p ::= ⟨e, e⟩

Figure 4.8: Core Syntax of EFFECT

Figure 4.8 gives the core syntax of EFFECT. The expression $\texttt{with}^{\mathcal{M}}\ e_h\ e_b$ runs a main-effect handler that interprets effects performed by ordinary code (not contract code) in the body expression e_b using the handler e_h (i.e., `handler`). The expression $\texttt{with}^{\mathcal{C}}\ e_h\ e_b$ runs a contract-effect handler that interprets effects performed by contract-checking code (not ordinary code) in the body expression e_b using contract handler

e_h (i.e., `contract-handler`). Pairs are included so that contract handlers can return multiple values. An effect is performed using `do` e, where the value of e is passed to the nearest handler.

Mirroring the handlers, effect-handler contracts also demand two constructs, one per level. Both of these forms monitor a function that may request effects. The expression $e\ \mathcal{M}\ f$ produces a dependent main-effect contract (i.e., `->de`), where the constraint on handler responses can depend on the effect request. It ensures that effects requested during the application of a subject function satisfy e and values received from the handler satisfy the contract returned by f. The expression $\mathcal{C}\ f$ produces a contract-effect contract (i.e., `with/c`) that installs the given contract-effect handler during the application of a subject function.

4.3 DYNAMIC SYNTAX

EFFECT (DYNAMIC) extends CONTRACT

$$
\begin{aligned}
\mathsf{Expr} \ni e &::= \ldots \mid \mathtt{mark}_j^{k,l}\ (v\ \mathcal{M}\ f)\ e \\
\mathsf{Inv} \ni i &::= \ldots \mid \mathtt{with}^{\mathcal{M}}\ v\ e \ \ \wr v \notin \mathsf{Fun} \int \mid \mathtt{with}^{\mathcal{C}}\ v\ e \ \ \wr v \notin \mathsf{Fun} \cup \mathsf{Pair} \int \\
\mathsf{Con} \ni c &::= \ldots \mid e\ \mathcal{M}\ f \mid \mathcal{C}\ f \\
\mathsf{Val} \ni v &::= \ldots \mid v\ \mathcal{M}\ f \mid \mathcal{C}\ f \mid \langle v, w \rangle \\
\mathsf{Ctx} \ni E &::= \ldots \mid \mathtt{with}^{\mathcal{M}}\ E\ e \mid \mathtt{with}^{\mathcal{M}}\ f\ E \mid \mathtt{with}^{\mathcal{C}}\ E\ e \mid \mathtt{with}^{\mathcal{C}}\ f\ E \\
&\quad \mid \mathtt{with}^{\mathcal{C}}\ \langle v, v \rangle\ E \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \mathtt{do}\ E \mid E\ \mathcal{M}\ f \\
&\quad \mid \mathtt{mark}_j^{k,l}\ (v\ \mathcal{M}\ f)\ E \\
\mathsf{Ctx}^{\mathcal{M}} \ni E^{\mathcal{M}} &::= \text{—— } \textit{similar to } E \text{ ——} \mid \mathtt{mon}_j^k\ E^{\mathcal{M}}\ e \mid \mathtt{with}^{\mathcal{C}}\ E^{\mathcal{M}}\ e \\
\mathsf{Ctx}^{\mathcal{C}} \ni E^{\mathcal{C}} &::= \text{—— } \textit{similar to } E \text{ ——} \mid \square \mid \mathtt{mon}_j^k\ E\ e \mid \mathtt{with}^{\mathcal{C}}\ E\ e
\end{aligned}
$$

Figure 4.9: Dynamic Syntax of EFFECT

Figure 4.9 shows the dynamic syntax of EFFECT—needed for articulating the reduction relation. This syntax includes mark expressions $\mathtt{mark}_j^{k,l}\ (v\ \mathcal{M}\ f)\ e$. A mark ensures that effects requested by e, and their fulfillment, satisfy the contract $v\ \mathcal{M}\ f$. In other words, effect requests "passing through" the mark should satisfy the contract. These marks are installed by main-effect contracts.

Three different kinds of evaluation context are defined, each with a different role. Main-effect handlers respond to effect requests in ordinary code, while contract-effect handlers respond to effect requests in contract-checking code. Evaluation contexts provide a formal means to detect whether evaluation is occurring in ordinary code or contract-checking code. $\mathsf{Ctx}^{\mathcal{M}}$ is the set of *main-executing contexts* containing reg-

ular code that is handled by a main-effect handler. $\mathsf{Ctx}^{\mathcal{C}}$ is the set of *contract-executing contexts* describing the dynamic extent of contract code that is handled by a contract-effect handler. Ctx is the set of all contexts and is the union of the previous two.

Contract code executes in two syntactic positions: e_c in $\mathtt{mon}^k_j\, e_c\ e_s$ and e_h in $\mathtt{with}^{\mathcal{C}}\, e_h\ e_b$. While the former is more obviously a contract-executing position, the latter demands a closer look. Recall that $\mathtt{with}^{\mathcal{C}}$ interprets effect requests that originate in contract code. By implication, e_h may receive and apply higher-order values originating from contract code. Therefore, it *must* be considered contract code. The definition of evaluation contexts reflects this reasoning. In particular, $\mathsf{Ctx}^{\mathcal{M}}$ omits (indicated by inverted colors) productions of the shape $\mathtt{mon}^k_j\, \mathsf{E}^{\mathcal{M}}\ e$ and $\mathtt{with}^{\mathcal{C}}\, \mathsf{E}^{\mathcal{M}}\ e$. Therefore, if a redex is in contract-checking code, then the evaluation context will not match $\mathsf{E}^{\mathcal{M}}$. Similarly, $\mathsf{Ctx}^{\mathcal{C}}$ omits the production for $\Box$ and specifies an unrestricted context E in contract-checking positions $\mathtt{mon}^k_j\, \mathsf{E}\ e$ and $\mathtt{with}^{\mathcal{C}}\, \mathsf{E}\ e$. Consequently, a fully formed $\mathsf{E}^{\mathcal{C}}$ context is forced to have a hole located within a contract-checking position.

4.4 SEMANTICS

$\boxed{e \succ e}$ for EFFECT

$$
\begin{array}{rr}
 & \vdots \\
\textsc{with-m} & \mathtt{with}^{\mathcal{M}}\, f\ v \succ v \\
\textsc{with-c} & \mathtt{with}^{\mathcal{C}}\, w\ v \succ v \\
\textsc{do-m} & \mathtt{with}^{\mathcal{M}}\, f\ \mathsf{E}^{\mathcal{M}}[\mathtt{do}\ v] \succ \mathsf{E}[f\ y_0\ (\lambda z_0.\mathtt{with}^{\mathcal{M}}\, f\ \mathsf{E}^{\mathcal{M}}[e])] \\
 & \quad \wr \mathsf{E}^{\mathcal{M}} \text{ unhandled}, \\
 & \quad \mathsf{E} = \uparrow_0 \mathsf{E}^{\mathcal{M}}[\mathtt{do}\ v], \\
 & \quad e = \downarrow_0 \mathsf{E}^{\mathcal{M}} \int \\
\textsc{do-c-pair} & \mathtt{with}^{\mathcal{C}}\, \langle v, w\rangle\ \mathsf{E}^{\mathcal{C}}[\mathtt{do}\ v'] \succ \mathtt{with}^{\mathcal{C}}\, w\ \mathsf{E}^{\mathcal{C}}[v] \\
 & \quad \wr \mathsf{E}^{\mathcal{C}} \text{ unhandled} \int \\
\textsc{do-c-fun} & \mathtt{with}^{\mathcal{C}}\, f\ \mathsf{E}^{\mathcal{C}}[\mathtt{do}\ v] \succ \mathtt{with}^{\mathcal{C}}\, (f\ v)\ \mathsf{E}^{\mathcal{C}}[\mathtt{do}\ v] \\
 & \quad \wr \mathsf{E}^{\mathcal{C}} \text{ unhandled} \int
\end{array}
$$

$\boxed{e \longmapsto e}$ for EFFECT

$$
\begin{array}{rr}
 & \vdots \\
\textsc{err-do} & \mathsf{E}[\mathtt{do}\ v] \longmapsto \mathsf{E}[\mathtt{err}^{\mathcal{P}}_{\mathcal{L}}] \ \wr \mathsf{E} \text{ unhandled} \int
\end{array}
$$

Figure 4.10: Semantics of EFFECT

Figure 4.10 presents the handler-related reduction rules for EFFECT. The WITH-M and WITH-C rules state that when the body of a handler is a value, the computation has run its course and the handler is eliminated. Otherwise, one of the DO rules may apply. Both DO-M and DO-C use the special evaluation contexts from Figure 4.9 to ensure that the requested effect originates from either main-program code or contract code. The DO-M rule specifies main-program handlers as deep, i.e., the continuation reinstalls the same handler when invoked. As will be shown later, the evaluation context $E^{\mathcal{M}}$ may contain marks deposited by main-effect contracts. These contracts constrain main-effect requests and main-effect-handler responses. Therefore, DO-M must arrange for these checks to occur by reading from the marks.

$$\frac{\texttt{with}^{\mathcal{M}}\ f\ E^{\mathcal{M}} \not\sqsubseteq E \qquad \texttt{with}^{\mathcal{C}}\ v\ E^{\mathcal{C}} \not\sqsubseteq E \quad \lceil v \in \mathsf{Fun} \cup \mathsf{Pair} \rfloor}{E\ \mathsf{unhandled}} \qquad \frac{\exists E'.\ F = E'[E]}{E \sqsubseteq F}$$

Figure 4.11: Auxiliary Judgment for Semantics

The DO rules rely on the E unhandled judgment, shown in Figure 4.11, to ensure that the innermost matching handler is triggered. Omission of these side conditions causes the semantics to be non-deterministic.

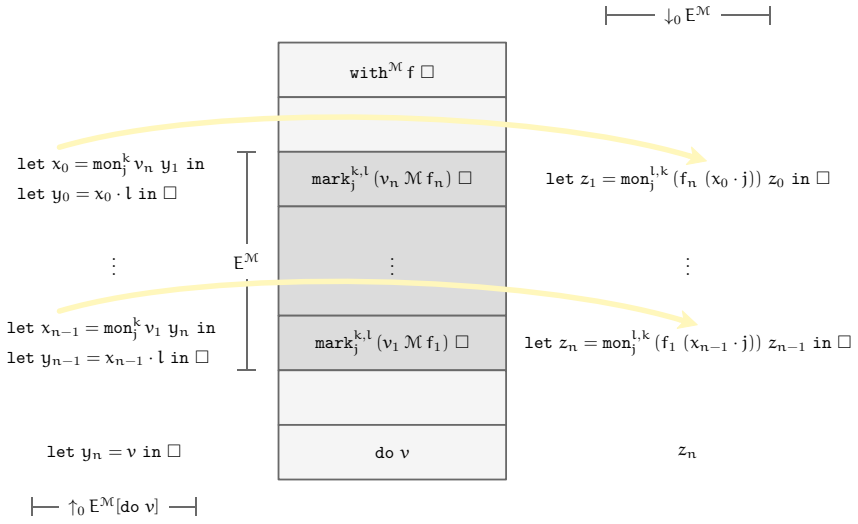

Figure 4.12: Metafunctions for EFFECT Graphically

Two metafunctions, $\uparrow_n$ (pronounced "up") and $\downarrow_n$ (pronounced "down"), collect the contracts for main-effect requests and handler responses, respectively. The action of these metafunctions is shown graphically in Figure 4.12. For precisely the same reason as stated in Chapter 3, these metafunctions must produce a collection of `let` bindings to avoid effect duplication, given the combination of dependency and the potential for effects. Because there can be an arbitrary number

of marks on the context, properly arranging the checks is far more challenging than in the CONTRACT model. At the center of the graphic is the redex of DO-M, containing within it the evaluation context $E^{\mathcal{M}}$. The left-hand side, which should be read from *bottom to top*, shows how $\uparrow_n$ builds an evaluation context that produces `let` bindings for checking effect requests. The right-hand side, which should be read from *top to bottom*, shows how $\downarrow_n$ builds an expression for checking handler responses. Dependencies are marked as yellow arrows, going from the binding created by $\uparrow_n$ to references in $\downarrow_n$. These metafunctions are used in a well-scoped manner because the contractum of DO-M places the $\downarrow_n$ expression in the scope of bindings produced by $\uparrow_n$.

$\boxed{\uparrow_n : \mathsf{Expr} \to \mathsf{Ctx}}$

$$\uparrow_n \texttt{do}\ v \coloneqq \texttt{let}\ y_n = v\ \texttt{in}\ \square$$
$$\uparrow_n \texttt{mark}^{k,l}_j\ (w\ \mathcal{M}\ f)\ e \coloneqq (\uparrow_{n+1} e)[\texttt{let}\ x_n = \texttt{mon}^k_j\ w\ y_{n+1}\ \texttt{in}\ \texttt{let}\ y_n = x_n \cdot l\ \texttt{in}\ \square]$$

$\boxed{\downarrow_n : \mathsf{Ctx} \to \mathsf{Expr}}$

$$\downarrow_n \square \coloneqq z_n$$
$$\downarrow_n \texttt{mark}^{k,l}_j\ (w\ \mathcal{M}\ f)\ E \coloneqq \texttt{let}\ z_{n+1} = \texttt{mon}^{l,k}_j\ (f\ (x_n \cdot j))\ z_n\ \texttt{in}\ \downarrow_{n+1} E$$

Figure 4.13: Metafunctions for EFFECT

Figure 4.13 gives the formal definitions of these two metafunctions. The formal definitions show that the index placed on the metafunctions plays a role in generating appropriately indexed variables. Careful management of the indices ensures that bindings and references are properly aligned. Additionally, note that the bottom-to-top reading of $\uparrow_n$ implies that the innermost element of the redex (i.e., `do` v) corresponds to the outermost frame of the resulting evaluation context. To achieve this reversal, the formal definition of $\uparrow_n$ fills the hole of the recursive result, flipping the evaluation context inside-out.

The DO-C rules specify contract handlers that have no control over the continuation. Furthermore, the reduction of contract handlers necessitates two rules to distinguish two cases. Specifically, the expression e_h in $\texttt{with}^{\mathcal{C}}\ e_h\ e_b$ can reduce to either a function or a pair (i.e., multiple `values`). In DO-C-PAIR, the first component is placed into the evaluation context, which is the continuation of the effect request, and the second component becomes the next handler. In DO-C-FUN, the function is applied to the effect request with the expectation that this new contract expression eventually reduces to a pair, and then the previous rule will apply. Like MON-LAM, this rule ensures that contract code is always executed in a syntactic position that is recognized by $\mathsf{Ctx}^{\mathcal{C}}$ contexts.

$$\vdots$$

MON-M	$\texttt{mon}_j^k\ (w\ \mathcal{M}\ f)\ g \succ \texttt{grd}_j^k\ (w\ \mathcal{M}\ f)\ g$
ERR-M	$\texttt{mon}_j^k\ (w\ \mathcal{M}\ f)\ v \succ \texttt{err}_j^k\ \lceil v \notin \mathsf{Lam} \rfloor$
GRD-M	$(\texttt{grd}_j^k\ (w\ \mathcal{M}\ f)\ g) \cdot l \succ \lambda x.\texttt{mark}_j^{k,l}\ (w\ \mathcal{M}\ f)\ (g\ x)$
MARK	$\texttt{mark}_j^{k,l}\ (w\ \mathcal{M}\ f)\ v \succ v$
MON-C	$\texttt{mon}_j^k\ (\mathcal{C}\ f)\ g \succ \texttt{grd}_j^k\ (\mathcal{C}\ f)\ g$
ERR-C	$\texttt{mon}_j^k\ (\mathcal{C}\ f)\ v \succ \texttt{err}_j^k\ \lceil v \notin \mathsf{Lam} \rfloor$
GRD-C	$(\texttt{grd}_j^k\ (\mathcal{C}\ f)\ g) \cdot l \succ \lambda x.\texttt{with}^{\mathcal{C}}\ f\ (g\ x)$

Figure 4.14: Semantics of EFFECT (Continued)

Finally, Figure 4.14 presents the reduction relation rules governing both kinds of effect-handler contracts. The MON rules ensure the first-order constraint that the subjects are functions. If they are functions, the contracts act in a higher-order manner via GRD-M and GRD-C. The GRD-M rule installs a mark that constrains effects performed during applications of g. Actually checking these contracts, as seen previously, is delegated to DO-M. Once the dynamic extent of a mark expression ends, the mark can be eliminated via the MARK rule. The GRD-C rule simply wraps the subject in a contract-effect handler, where f becomes the handler function.

4.5 METATHEORETIC PROPERTIES

The key property of interest for the model is *contract erasure*. Contracts are intended to detect violations of specifications and, therefore, the output of a correct program should not depend on their presence or absence. In short, contracts must not interfere with program execution—other than possibly signaling an error.

$\mathcal{E}[\![\cdot]\!] : \mathsf{Expr} \to \mathsf{Expr}$

$$\mathcal{E}[\![x]\!] \coloneqq x$$
$$\vdots$$
$$\mathcal{E}[\![\texttt{mon}_j^{k,l}\ e_c\ e_s]\!] \coloneqq \mathcal{E}[\![e_s]\!]$$
$$\mathcal{E}[\![\texttt{with}^{\mathcal{C}}\ e_h\ e_b]\!] \coloneqq \mathcal{E}[\![e_b]\!]$$

Figure 4.15: Erasing Metafunction

The erasure theorem requires an erasure metafunction $\mathcal{E}[\![\cdot]\!]$, defined in Figure 4.15, to erase contract monitors and contract handlers.

Theorem (Erasure). If $\mathrm{eval}(e) = b$ then $\mathrm{eval}(\mathcal{E}[\![e]\!]) = b$.

Proof. By simulation. See Chapter A. □

Non-termination is the one contract effect in the model that can affect a program's behavior. So long as contracts contain code in a Turing-complete language, this effect is unavoidable. As stated, Erasure holds because the antecedent rules out non-terminating contracts.

Establishing the erasure theorem is straightforward in a pure setting, yet difficult to achieve in a language with effects. Erasure means contract code cannot interfere with the main program, directly or indirectly, via effects. A language with effect-handler contracts poses the additional problem of granting contract code the right to perform effects, while imposing enough constraints such that erasure holds.

Designing a language semantics satisfying contract erasure requires balancing expressive power with preventing interference. The EFFECT model achieves that delicate balance. Limiting the expressive power any further makes programming inconvenient and would disallow existing use cases. Adding more power potentially violates erasure.

Consider a naive design where the reduction relation for handlers merges the two levels of effect handling:

$$\texttt{with}\ f\ E[\texttt{do}\ v] \succ f\ v\ (\lambda z.\texttt{with}\ f\ E[z])\quad \wr E\ \text{unhandled}\int.$$

Instead of restricting the evaluation context in the body of the handler, this rule uses the unrestricted context E. Such a rule violates contract erasure as the following program demonstrates:

$$\texttt{with}\ (\lambda x.\lambda y.x)\ (\texttt{mon}_j^{k,l}\ (\texttt{do}\ \texttt{ff})\ \texttt{tt}).$$

The original program evaluates to `ff`, but erasing the contract yields a program whose value is `tt`. Modifying DO-M to use an unrestricted evaluation context, or modifying DO-C-FUN to give direct access to the continuation, both result in violations of erasure because they produce a rule equivalent to the one above.

Introducing "main-handler contracts" with the rule

$$\texttt{mon}_j^{k,l}\ (\mathcal{M}\ f)\ g \succ \lambda x.\texttt{with}^{\mathcal{M}}\ f\ (g\ x)$$

also violates erasure. Here is a counterexample:

$$\texttt{with}^{\mathcal{M}}\ (\lambda y.\lambda z.y)\ ((\texttt{mon}_j^{k,l}\ (\mathcal{M}\ (\lambda y.\lambda z.\texttt{ff}))\ (\lambda x.\texttt{do}\ x))\ \texttt{tt}).$$

The original program evaluates to `ff` because the contract-installed handler is invoked, while the outer handler stays inactive. The erased variant, however, yields `tt` because the outer handler is invoked in the absence of the contract-installed one.

5 TRACE CONTRACTS FORMALLY

Effect-handler contracts are, by design, as general as possible. As a practical tool, though, they are too low level. Higher-level effectful contracts are needed for real applications. For instance, library documentation often comes with sequence diagrams, protocol descriptions, and other temporal specifications, dictating how a library may be used. Unix's I/O API is a standard example: "open a file before reading from it." A framework for computing static-analysis passes may require that it be given monotone transfer functions. A GUI framework may allow the registration of callback objects and promise to call them back in the order of registration.

Trace contracts are one kind of effectful contract that address these kinds of specifications. Namely, they permit the functional specification of constraints across multiple function and method calls. A *trace* reifies the sequence of values that flow through interception points of a contract system, such as function calls. A *trace contract* inspects this reified trace with a predicate that decides whether a property holds.

This chapter introduces trace contracts by example (Section 5.1) and describes a model of trace contracts, supplying its syntax (Section 5.2) and semantics (Section 5.3). Formally modelling trace contracts clarifies the central challenge: on the one hand, specifications should remain functional, while on the other hand, collecting a trace of values necessarily involves mutable state. Managing this state while maintaining ordinary contract composition is the key design criterion.

5.1 EXAMPLES

In 2020, a developer reported a bug to Racket's mailing list about the `current-memory-use` function [73]. The documentation states that the function "returns an estimate of the total number of bytes allocated since start up, including bytes that have since been reclaimed by garbage collection" [59]. Given this description, one might expect that the series of return values from `current-memory-use` would increase over time. However, a memory-consumption plot for a long-running system showed periodic dips.

In a language with a conventional type system, such as Java, this function would have the following signature:

```
// Returns the number of bytes allocated since start up,
// including those deallocated during garbage collection.
int currentMemoryUse();
```

The comment mentions two unchecked constraints. First, the function's result cannot be negative, so `int` is imprecise. In Racket, the API author could improve on this type with the `(-> natural?)` contract. Second, the documentation implies that every call returns a number that is greater than or equal to the result of all previous calls.

```
(provide
  (contract-out
   [current-memory-use
    (trace/c ([y natural?])¹
      (-> y)²
      (full (y) sorted?)³)]))
```

Figure 5.1: Basic Trace Contract

Figure 5.1 demonstrates that trace contracts can express both constraints directly. As the superscripts indicate, a trace contract consists of three parts: (1) a sequence of *trace variable declarations*, including one contract associated with each variable; (2) a contract expression, dubbed the *body contract*; and (3) a sequence of *predicate clauses*.

The simple trace contract in Figure 5.1 comes with a single trace variable, `y`, associated with `natural?`. The body contract is `(-> y)`, which specifies ordinary single-call constraints placed on values protected by the trace contract. When a client module calls `current-memory-use`, the contract system ensures that the returned value is a natural number, expressing the first constraint. If the value is a natural number, it is collected in a trace data structure associated with `y`. Additionally, the trace contract specifies a `full` predicate clause that depends on `y`. For `full`, the trace data structure is represented with a stream. Every time the contract system collects a value in the `y` trace, it applies the function specified in the predicate clause (i.e., `sorted?`) to the stream of values. The trace contract fails if `sorted?` returns `#f`, indicating a dip in the sequence.

Note that `sorted?` is a pure function in the host language, just like ordinary first-order behavioral contracts. One immediate advantage is that a developer can test contracts like any other piece of code—an important property considering that all code, including specification code, may have bugs.

With this contract in place, violations are detected as soon as they occur and blame information points to the party responsible:

```
> (current-memory-use)
100
> (current-memory-use)
current-memory-use: broke its own contract
    produced: 0
    blaming: current-memory-use
```

In this interaction, `current-memory-use` returns `0` on the second call, causing a contract error. Since the problematic value was collected from the module that defined `current-memory-use`, the function itself is to blame. Developers confronted with this error message can immediately report a bug in the library, knowing with confidence that their code is not responsible for the fault.

In its current form, the `current-memory-use` contract comes with a steep performance cost. While any contract can slow down a program, naive trace contracts can be especially expensive because they execute code every time a value is added to a trace. Programmers should be mindful of this expense. In particular, `sorted?` iterates through the entire `y` trace every time a new value is collected. Checking this trace contract is quadratic in the number of calls to `current-memory-use`. To reduce this overhead, a trace-contract system must give developers fine-grained control over how the trace is represented.

Developers must be able to choose a custom representation of the trace instead of the stream data structure. This choice involves three steps: (1) decide on a data structure; (2) pick an initial value; and (3) supply an operation that incorporates a value into the existing trace representation or signals a failure. The **`accumulate`** predicate clause gives developers this control, and the data structure is dubbed the *accumulator*. The *accumulating function* given to **`accumulate`** receives two values: the current accumulator and the newly collected values. It returns the new accumulator on success or a designated failure value otherwise. For the running example, it suffices to use a single number as the accumulator. A simple comparison between any collected value and the accumulator is enough to enforce the promised behavior.

```
(trace/c ([y natural?])
  (-> y)
  (accumulate 0
    [(y) (λ (acc cur) (if (<= acc cur) cur (fail)))]))
```

Figure 5.2: Trace Contract with Accumulator

Figure 5.2 improves on the previous trace contract using an accumulator. The **`accumulate`** clause specifies an initial accumulator value of `0` and an accumulating function. When the `y` trace receives a new value, the accumulating function is applied to the current accumulator and the latest value. If the current accumulator is less than or equal to the new value, then the new value is returned and becomes the next accumulator.[1] Otherwise, the function's result is `(fail)`, the designated failure value.

1 If `current-memory-use` were to return a non-numeric result, an error would be raised even without the **`natural?`** check on `y` because `<=` expects two numbers. The error message, however, would blame the contract itself for violating the contract on `<=`, instead of `current-memory-use`. So, the **`natural?`** check should remain.

Every trace contract can be expressed with `accumulate` instead of `full`.[2] While `full` is a useful tool to understand trace contracts conceptually, in practice programmers should almost always use `accumulate` combined with an efficient data structure.

5.2 SYNTAX

TRACE (CORE) extends CONTRACT

$$\text{Expr} \ni e ::= \ldots \mid f \mathbin{\mathcal{T}} g$$

Figure 5.3: Core Syntax of TRACE

Figure 5.3 defines the core syntax of TRACE, a language with trace contracts extending the CONTRACT model from Chapter 3. The core syntax contains one new form $f \mathbin{\mathcal{T}} g$, a trace contract containing body-contract constructor f and accumulating function g. A body-contract constructor is a function that, when provided with a *collector*, returns the body contract. A collector is a contract that gathers values flowing through interception points specified in the body contract and ensures the trace predicate remains satisfied. Collectors rely on mutable state to update the accumulator.

TRACE (DYNAMIC) extends CONTRACT

$$\begin{aligned}
\text{Config} \ni \Sigma &::= \langle e, \sigma \rangle \\
\text{Expr} \ni e &::= \ldots \mid \ell \mathbin{\mathcal{O}} f \mid \ell \leftarrow e \\
\text{Con} \ni c &::= \ldots \mid f \mathbin{\mathcal{T}} g \mid \ell \mathbin{\mathcal{O}} f \\
\text{Val} \ni v &::= \ldots \mid f \mathbin{\mathcal{T}} g \mid \ell \mathbin{\mathcal{O}} f \\
\text{Ctx} \ni E &::= \ldots \mid \ell \leftarrow E \\
\text{Store} \ni \sigma &\coloneqq \text{Loc} \rightharpoonup \text{Val} \\
\text{Loc} \ni \ell &
\end{aligned}$$

Figure 5.4: Dynamic Syntax of TRACE

Figure 5.4 defines the dynamic syntax of TRACE. Because trace contracts need mutable state, the semantics is defined over configurations containing stores. Expressions include $\ell \mathbin{\mathcal{O}} f$ for collector contracts (where ℓ is the cell containing the accumulator and f is the accumulating function) and $\ell \leftarrow e$ for writing to the store.

2 In fact, `full` is syntactic sugar over an `accumulate` clause with a stream accumulator.

5.3 SEMANTICS

$$
\begin{array}{rl}
 & \vdots \\
\textsc{mon-t} & \langle \mathsf{E}[\mathtt{mon}^{k}_{j}\,(f\,\mathcal{T}\,g)\,v], \sigma\rangle \longmapsto \langle \mathsf{E}[\mathtt{mon}^{k}_{j}\,(g\,(\ell\,\mathcal{O}\,f))\,v], \sigma[\ell \mapsto \mathtt{tt}]\rangle \\
 & \qquad\qquad \langle \ell \notin \mathrm{dom}\,\sigma \rangle \\
\textsc{mon-o} & \langle \mathsf{E}[\mathtt{mon}^{k}_{j}\,(\ell\,\mathcal{O}\,f)\,v], \sigma[\ell \mapsto v_{\ell}]\rangle \longmapsto \langle \mathsf{E}[\mathtt{mon}^{k}_{j}\,(\ell \leftarrow f\,v_{\ell})\,v], \sigma\rangle \\
\textsc{mut-t} & \langle \mathsf{E}[\ell \leftarrow v], \sigma\rangle \longmapsto \langle \mathsf{E}[\mathtt{tt}], \sigma[\ell \mapsto v]\rangle \quad \langle v \neq \mathtt{ff} \rangle \\
\textsc{mut-f} & \langle \mathsf{E}[\ell \leftarrow \mathtt{ff}], \sigma\rangle \longmapsto \langle \mathsf{E}[\mathtt{ff}], \sigma[\ell \mapsto \mathtt{ff}]\rangle \\
\textsc{lift} & \langle \mathsf{E}[e], \sigma\rangle \longmapsto \langle \mathsf{E}[e'], \sigma\rangle \quad \langle e \succ e' \rangle \\
\textsc{err} & \langle \mathsf{E}[\mathtt{err}^{k}_{j}], \sigma\rangle \longmapsto \langle \mathtt{err}^{k}_{j}, \sigma\rangle \quad \langle \mathsf{E} \neq \square \rangle
\end{array}
$$

Figure 5.5: Semantics of TRACE

The semantics of the trace-contract model is presented in Figure 5.5. There are two rules relating to contracts and two rules for writing to the store. The MON-T rule first allocates a new location for storing the accumulator, initialized with `tt`. Then, it creates a collector and provides it to the body-contract constructor. The MON-O rule produces code that updates the accumulator according to the newly collected value. If the new accumulator is not `ff`, then MUT-T yields `tt` after updating the store, and the contract check succeeds. Notably, MUT-T does not yield v because that may result in an incorrect contract check in MON-O. If the new accumulator is `ff`, then MUT-F yields `ff`, and the contract check fails. Finally, the LIFT and ERR rules from CONTRACT must be adjusted to accomodate stores.

A deliberate choice has been made here to extend CONTRACT and not EFFECT. Given that effect-handler contracts are a supposedly unified mechanism, one may wonder whether trace contracts are expressible using just the tools described in Chapter 4. The answer is yes. Formalizing this claim, however, would involve defining a compiler from TRACE to EFFECT, and then proving that compiler correct. Such a proof is possible but would not be enlightening.[3] It is at this point—where the complexity of the model grows faster than the insight it delivers—that implementations come into play. The next part answers the expressiveness question with *macros*. Several effectful contracts, including trace contracts, will be given macro translations in Effect Racket.

3 The original trace-contract publication [108] has a similar compiler-correctness proof, although for a different target language.

Part II

CONSTRUCTIVE

Following the formal models, Effect Racket realizes effect-handler contracts within an entire handler-based language. Three kinds of effectful contract, within the context of stock Racket, are presented as practical alternatives: parameter contracts, trace contracts, and attribute contracts.

6 EFFECT RACKET

Moving beyond models calls for a production-level language with linguistic constructs for realizing effect-handler contracts easily and a well-developed higher-order contract system. Racket is such a programming language [52, 56, 59, 60].

This chapter presents Effect Racket, a flavor of Racket with effect handlers and a full contract system, aimed at capturing the features described in Chapter 4. Following the precedent of Typed Racket, the language is implemented as a library [140] and validates that the model from Chapter 4 can be realized. Section 4.1 already showed some synthetic programs written in Effect Racket. Here there will be somewhat more realistic examples (Section 6.1) and a discussion of how the implementation itself works (Section 6.2).

6.1 EXAMPLES

Effect Racket implements features from the formal model of effect-handler contracts to support experimentation. The language includes main-effect handlers, main-effect contracts, contract-effect handlers, and contract-handler contracts.

MAIN-EFFECT HANDLERS As an introductory example, consider implementing ML's first-class mutable references using effect handlers. References come with a `ref` constructor and two elimination forms:

```
(effect ref (v))
(effect ref-get (r))
(effect ref-set (r v))
```

In Effect Racket, each form demands the declaration of a corresponding effect: one for allocating a reference cell, one for getting its value, and yet another for assigning to a cell. Declaring an effect makes the effect name available both for requesting the effect and, within a handler, interpreting the effect. The `effect` form in similar to a `struct` declaration, but instead of creating an instance of a struct, applying the effect name performs the given effect.

Figure 6.1 displays code for the corresponding effect handler, with one clause per declared effect. All other effects are propagated automatically. Furthermore, the `handler` form binds two identifiers to delimited continuations: `continue` for resuming in a deep manner and `continue*` for resuming in a shallow manner. With a deep handler, applying the continuation automatically reinstates the current handler.

```
;; ref-handler: Store → Handler
(define (ref-handler store)
  (handler
    [(ref init)
     (define-values (r store*)
       (store-alloc store init))
     (with ((ref-handler store*))
       (continue* r))]

    [(ref-get r)
     (continue (store-get store r))]

    [(ref-set r v)
     (with ((ref-handler (store-set store r v)))
       (continue* (void)))]))
```

Figure 6.1: Mutable References in Effect Racket

With a shallow handler, it does not. Otherwise, the handler uses standard techniques for implementing a store in this setting [23, 124].

Any language in the Racket ecosystem, including Effect Racket, is easily equipped with a REPL. By running an Effect Racket program, the definitions of effects and handlers become available for interactive experimentation:

```
> (with ((ref-handler empty-store))
    (define r (ref 0))
    (ref-set r (add1 (ref-get r)))
    (ref-get r))
1
```

A handler is installed using the `with` form. In the context of the `with` expression body, it is possible to allocate a reference cell, to increase its value by `1`, and then to retrieve its contents.

MAIN-EFFECT CONTRACTS Suppose a programmer wishes to write a library function that guarantees a frame condition. In particular, the function guarantees that it manipulates only a specific reference cell during the dynamic extent of a call. A good name for this contract would be `mutates-only/c`.

Figure 6.2 shows how a library's interface may state that guarantee. The `ref-restore` function is intended to run `f` applied to `r`, restore the contents of `r` to its original value, and return the value of `r` that `f` stored there. Protecting the function is a dependent contract that governs two arguments, `r` and `f`, and promises nothing about its result. The novel piece is the contract for `f`, which says that it may mutate *only* `r`.

```
(provide
  (contract-out
   [ref-restore
    (->i ([r reference?]
          [f (r) (mutates-only/c r)])
         [res any/c])]))
```

Figure 6.2: Contract Restricting Mutation

```
;; mutates-only/c: Ref → Contract
(define (mutates-only/c r-ok)
  (define (effect-ok? e)
    (match e
      [(ref-set r _) (equal? r r-ok)]
      [_ #t]))
  (->e effect-ok? any/c))
```

Figure 6.3: Frame Contract

Shown in Figure 6.3 is a frame contract implemented as a main-effect contract. The `mutates-only/c` function takes a reference cell as an argument and returns a main-effect contract that permits writing only to the given cell and no other. The two-part `->e` contract tells a reader that requested effects must satisfy the `effect-ok?` predicate and that values returned by the handler can be anything. According to `effect-ok?`, any write effect must be to a reference cell that is `equal?` to `r-ok`. All other effects are permitted.

CONTRACT-EFFECT HANDLERS An affine contract guarantees that a function is called at most once by keeping track of how many times it has previously been called using mutable state.

Figure 6.4 shows how an affine contract can be defined in Effect Racket. The —∘ contract must allocate a reference for keeping track of whether a function has previously been called. The presented code realizes this constraint using the `self/c` combinator, which enables cascading contracts. When protecting a subject `v`, `self/c` first applies a function to `v` and then uses the result to protect `v`. Here, the function given to `self/c` returns the expected function contract. The purpose of `self/c` in this definition is to ensure that a new reference is initialized for each protected function. Creating the reference outside `self/c` is incorrect because it results in a single global reference cell for the entire program. For —∘, the value `v` is not needed by `self/c` and is discarded. Using the precondition-checking feature of `->i`, annotated with `#:pre`, the reference is used to determine if the call should succeed and, if it does, the contents of the reference is decremented.

```
;; —∘: Natural Contract Contract → Contract
(define (—∘ n dom cod)
  (self/c
    (λ _
      (define r (ref n))
      (->i ([x dom])
           #:pre () (unused? r)
           [res cod])))))

;; unused?: Reference → Boolean
(define (unused? r)
  (define m (ref-get r))
  (cond
    [(zero? m) #f]
    [else (ref-set r (sub1 m)) #t]))
```

Figure 6.4: Affine-Function Contract with Effect Racket

Since the contract relies on reference cells at the contract level, it is necessary to execute the program within a contract handler that supports mutable references. The implementation of mutable references from Figure 6.1 must be adapted to the `contract-handler` form instead of the `handler` form, but doing so is straightforward.

CONTRACT-HANDLER CONTRACTS A function is *reentrant* if it can call itself recursively (directly or indirectly). A contract-handler contract can check for non-reentrancy by prohibiting recursive calls during a function call's dynamic extent. Implementing such a constraint requires both a `contract-handler` to mark the dynamic extent of a function call and a contract-handler contract.

Figure 6.5 displays a contract that enforces non-reentrancy. When a client applies `f`, the precondition requests a `non-reentrant` effect. If this request yields `#f`, the function is already running; otherwise, the `#:fail` option, which provides a default value if no matching handler is installed, returns `#t`. Once the precondition check passes, the second wrapper sets up a contract-handler contract using `with/c`. Thus, if `f` were to call itself, the contract would raise an exception because the installed `non-reentrant-handler` supplies `#f`.

6.2 IMPLEMENTATION OVERVIEW

The implementation of Effect Racket consists of about 1100 source lines of code (SLOC). Most of these lines compose elements from existing libraries. For example, effect handlers themselves are implemented as thin wrappers around Racket's existing library of delimited control op-

```
(provide
  (contract-out
   [f (and/c
       (with/c non-reentrant-handler)
       (->i ([x any/c])
            #:pre () (non-reentrant #:fail #t)
            [res any/c]))]))

(effect non-reentrant ())

(define non-reentrant-handler
  (contract-handler
    [(non-reentrant)
     (values #f non-reentrant-handler)]))
```

Figure 6.5: Contract for Non-Entrancy

erators [60]. Other pieces of the implementation ensure that Racket's effectful primitive operations are inaccessible to programs in Effect Racket. After all, a main-effect contract would be meaningless if certain primitive effects cannot be reinterpreted.

One critical aspect of the implementation concerns a key assumption from the model in Chapter 4, which mandates that handlers can detect whether an effect request originates from main code or contract code. Formally, this idea is encoded via special evaluation contexts. As it turns out, Racket's contract system already provides a mechanism for determining whether code is executing inside a contract [3]. Specifically, contracts set up continuation marks [30] that delineate contract-specific code from user code. Thus, the effect handler forms inspect the delimited continuation and look for this mark to determine whether the effect should be handled. As a result, Effect Racket does not require any modifications to Racket's existing contract system.

As a language in Racket's ecosystem, Effect Racket inherits the module system too, which raises the question of interoperability. In addition to full interoperability with other Effect Racket modules, the language has a shallow form of interoperability with plain Racket modules. First-order values can flow freely from an Effect Racket module to a foreign module and back; higher-order values are wrapped in an opaque structure so they become unusable until returned to Effect Racket, where they are unwrapped.

In summary, the implementation effort reveals that the addition of effectful contracts to an effect-handler-based language is straightforward, except for effect stratification. Assuming that the erasure property is desirable, an implementer must add a mechanism that demarcates the dynamic extent of contract-checking code.

7 PARAMETER CONTRACTS

Effect Racket is a useful demonstration that the ideas from Chapter 4 can be realized. As a practical tool that Racket programmers can employ in their software, however, it is insufficient. Developers should be able to get some benefits of effect-handler contracts without being forced to change the ambient language. In that spirit, this chapter presents a backwards-compatible extension to Racket's contract system that covers contract-handler contracts.

7.1 EXAMPLES

Contract-handler contracts can set up information during a dynamic extent that other contracts can later reference. In Racket, information is associated within a dynamic extent via parameters [64]. A parameter can store a value for the dynamic extent of an expression's evaluation; no matter how evaluation evolves, the original value is placed back into the parameter when the dynamic extent ends—even via an exception or continuation jump. The Racket implementation of parameters uses continuation marks [30]. A *parameter contract* is a specialized form of contract-handler contract that sets the value of a parameter during the dynamic extent of a function call. This section illustrates parameter contracts with two examples, one enforcing that `yield` is called only within a generator, and the other ensuring termination.

GENERATOR AND YIELD A *generator* is a thunk that may call the one-argument `yield` procedure in its dynamic extent. When it does so, the evaluation of the generator is suspended, and the value handed to `yield` becomes the result of the generator. When the generator is called again, the suspension is resumed until the next call to `yield`.

```
(define evens
  (generator
    (λ ()
       (for ([k (in-naturals)])
         (yield (* 2 k))))))
```

Figure 7.1: Generating Even Naturals

Figure 7.1 shows a simplistic example of a generator that produces all even natural numbers. A generator is created by passing a thunk to the `generator` function. The key constraint is that `yield` should

be invoked only during the dynamic extent of the thunk. Without effectful contracts, this constraint is documented informally, or possibly checked using interspersed defensive checks.

```
(provide
  (contract-out
   [generator
    (->i ([thunk (->i () #:param gen #t any/c)])
         [result generator?])]
   [yield
    (->i ([v any/c]) #:pre (gen) [result any/c])]))

(define gen (make-parameter #f))
```

Figure 7.2: Contracts for Generator and Yield

Figure 7.2 displays a parameter contract that expresses this constraint on the generator interface. The `->i` contract on `generator` is a parameter contract because it contains the `#:param` clause, initializing the parameter `gen`. Symmetrically, a precondition clause on `yield` checks the context to ensure that the `gen` parameter is set to `#t`, indicating that it is being evaluated during generation.

TERMINATION The literature on static analysis occasionally relies on termination checking, often encoded via the size-change property (SCP) [92]. Nguyễn et al. [112] present an ad hoc contract for checking the SCP. Roughly, the contract keeps track of a size-change graph that includes information about non-descending paths of argument sizes.

```
(define-syntax (total-> stx)
  (syntax-parse stx
    [(_ arg-ctc ... res-ctc)
     #:with (arg ...) (generate-temporaries #'(arg-ctc ...))
     #'(self/c
         (λ _
           (define sg (make-parameter empty-graph))
           (->i ([arg arg-ctc] ...)
                #:pre (graph-update sg (list arg ...))
                #:param (graph-update sg (list arg ...))
                [result res-ctc])))]))
```

Figure 7.3: Contract for Total Functions

Shown in Figure 7.3 is a parameter contract that expresses the termination contract with a parameter to update the size-change graph. The `total->` macro produces a termination contract that expands into `->i`.

As with affine contracts, the contract uses `self/c` to initialize the `sg` parameter at the appropriate time. This parameter initially contains the empty size-change graph. When called, the `total->` contract's precondition first checks whether updating the size-change graph with the new arguments would violate the SCP.[1] If so, `graph-update` returns `#f` and the program signals a contract violation. Otherwise, the `#:param` option extends the size-change graph with new information about the arguments. Due to the halting problem [144], this contract is necessarily overapproximate. However, any function that satisfies the contract is guaranteed to terminate.

```
(define ack
  (invariant-assertion
   (total-> integer? integer? integer?)
   (λ (m n)
     (cond
       [(= 0 m) (+ 1 n)]
       [(= 0 n) (ack (- m 1) 1)]
       [else (ack (- m 1) (ack m (- n 1)))]))))
```

Figure 7.4: Ackermann Function

Figure 7.4 shows the Ackermann function with termination checking enabled. The key element is the `invariant-assertion` construct, which attaches a contract to a function and enforces that it is checked on every call site, including recursive ones. Ordinary contract attachment forms, such as `contract-out` or `define-contract`, are active only at the boundary between components. These forms are not appropriate for termination checking because an arbitrary number of recursive calls can occur within a single component.

7.2 IMPLEMENTATION OVERVIEW

The addition of parameter contracts to Racket's contract system consists of a 230 SLOC patch to the `->i` combinator code. Key to the implementation is Racket's expressive support for proxy values [134]. Proxy values are able to maintain expected invariants, such as equality between the original value and the proxy. Importantly, procedure proxies support manipulating continuation marks upon application. The patch to `->i` takes advantage of a special internal value that serves as the continuation-mark key for all parameterizations [57]. The value of

1 On a recursive call, a new size-change graph is created that records the size relationship between each argument and its value in the recursive call. This graph is added to the current set of size-change graphs and is closed under sequential composition. If there is an idempotent graph that lacks a self-descending edge, then the SCP fails. See Nguyễn et al. [112] for a full explanation.

this key is a mapping between parameters and assignments. Another internal function updates this mapping. When the `#:param` option is set, the modified `->i` contract generates code that installs an updated value for the parameter continuation-mark key. When the `#:param` option is missing, the contract does not generate this code. In short, parameter contracts are a pay-as-you-go construct.

Like Effect Racket, the modification of Racket's contract system can guide the effort of others in adding parameter contracts to existing contract systems. If the underlying language comes with a mechanism such as continuation marks, the effort is fairly minimal. Otherwise, the implementer may wish to consider adding a continuation-mark mechanism because it has proven useful for a variety of different applications [26, 29, 30, 57].

7.3 TRANSLATING TO EFFECT RACKET

```
(effect env-get (key))

(define (parameterize/c key val)
  (make-contract
   #:late-neg-projection
   (λ (blm)
     (λ (subj neg)
       (λ args
         (with ((env-contract-handler key val))
           (apply subj args)))))))

(define (env-contract-handler key val)
  (contract-handler
   [(env-get (== key))
    (values val (env-contract-handler key val))]))
```

Figure 7.5: Parameter Contracts in Effect Racket

Figure 7.5 shows a basic implementation of parameter contracts as a macro in Effect Racket. It begins with an effect declaration for `env-get` that allows contract code to retrieve the value associated with a given key. This key plays the role of Racket's parameters.

The `parameterize/c` contract ensures that the function it protects is augmented with a handler that creates, in contract code, the key–value association. Its definition drops into the low-level contract-defining interface for Racket (`make-contract`) to produce a value with the desired handler installed. The curried `#:late-neg-projection` function is given three inputs: a first-class blame object, the subject value, and a negative blame-party label. It is expected to return a proxy of the sub-

ject function. Here, the proxy applies the subject to its argument with `env-contract-handler` installed. This handler matches `env-get` effect requests with the correct key—responding with the associated value.

There are two major simplifications in this implementation. First, unlike the real implementation, `parameterize/c` cannot have a parameter whose value depends on the arguments to a function. Second, the wrapper returned by the projection function in `parameterize/c` should ideally be a chaperone [134] of the original value. Chaperones are a kind of restricted proxy with additional guarantees that are desirable when possible. Handlers in Effect Racket do not use continuation marks, so the technique used in Section 7.2 is not possible here.

8 TRACE CONTRACTS

While Chapter 5 illustrates trace contracts with synthetic examples suitable to understand the model, a practical implementation has to contend with a far greater amount of complexity than is worth formalizing. Developers need features. This chapter dives further into the Racket implementation of trace contracts, including practical aspects of the library and how it works under the hood. A simplified macro implementation in Effect Racket is given in the final section, further validating that effect-handler contracts are a flexible mechanism.

8.1 EXAMPLES

The examples in this section demonstrate trace contracts for functions that demand sophisticated accumulator data structures and different state initialization strategies.

CHECKING ALL CALLS TO ONE FUNCTION Consider a compiler pass that computes a live-variable analysis via fixed-point iteration [86]. The interface to such an analysis, using ordinary contracts, may look similar to Figure 8.1.

```
(provide
  (contract-out
   [live-vars (-> (-> set? set?) label? set?)]))
```

Figure 8.1: Live Variables Contract

Given a monotonically increasing transfer function and a program label, `live-vars` returns the set of live variables at that label [114]. To be correct, the transfer function must be monotonically increasing, but this constraint is not enforced in Figure 8.1. An incorrectly computed least fixed point can cause a silent failure and may be difficult to debug.

A trace contract can enforce monotonicity, as seen in Figure 8.2. The `mono/c` function consumes two contracts and a comparison function; it returns a function contract that checks monotonicity with respect to the given comparison function. When a client module imports `live-vars` and invokes it, the `mono/c` contract is attached to the supplied transfer function. This contract stipulates that the transfer function takes and returns sets and is monotone with respect to set inclusion. During fixed-point iteration, the trace contract observes all input–output pairs and builds an extensional representation of the transfer function.

```
(provide
  (contract-out
   [live-vars (-> (mono/c set? set? subset?) label? set?)]))

;; mono/c: Contract Contract Comparator → Contract
(define (mono/c dom/c cod/c leq?)
  (trace/c ([x dom/c] [y cod/c])
    (-> x y)
    (accumulate (red-black-tree leq?)
      [(x y) (mono-func leq?)])))
```

Figure 8.2: Monotonicity Contract

Violations of monotonicity are detected by ensuring that no two input–output pairs fail the property.

In this example, the subclause within `accumulate` depends on two traces, `x` and `y`. When a subclause depends on more than one trace, the contract system waits until at least one new value has been collected for each trace before applying the function. If a trace receives more than one value before the other traces are ready, then all but the last are discarded.[1]

While a stream containing all input–output pairs would work, it would be highly inefficient. An order-aware data representation can reduce the time needed from $O(n^3)$ to $O(n \log n)$, where n is the number of calls to the transfer function. One possible choice is an immutable red-black tree [106, 115] as it can quickly determine the immediate predecessor and successor of an element in an ordered set.[2] Every time the trace contract monitors a new value, it initializes a new accumulator. If `live-vars` is invoked twice, two separate accumulators are created, one for each transfer function.

Figure 8.3 displays the curried accumulating function that finishes the definition of `mono/c`. When the transfer function returns, `mono-func` is applied to the current accumulator `acc`, the latest input `x`, and the latest output `y`. It determines the transfer function's predecessor and successor results for `x` and, if they exist, checks that they properly relate to the current output `y`. Just two comparisons are enough; by transitivity there are no other monotonicity violations. If successful, `mono-func` returns the next accumulator, relating the new input–output pair in the augmented red-black tree.

1 Other choices are expressible by having multiple `accumulate` subclauses with one dependency each. The accumulator would store collected values and then the accumulating function would determine the policy.

2 Ordinarily, red-black trees work only for a total order and not a partial order such as set inclusion. However, since fixed-point iteration explores a chain of comparable elements, a red-black tree is acceptable. A general-purpose contract for monotonicity that supports partial orders would require a different data structure.

```
;; mono-fun: Comparator → (Dict Any Any → (Or Dict Fail))
(define (mono-func leq?)
  (λ (acc x y)
    (cond
      [(dict-has-key? acc x)
       (if (equal? y (dict-ref acc x)) acc (fail))]
      [else
       (define pred-y (dict-pred acc x))
       (define succ-y (dict-succ acc x))
       (if (and (=> pred-y (leq? pred-y y))
                (=> succ-y (leq? y succ-y)))
           (dict-set acc x y)
           (fail))])))
```

Figure 8.3: Monotonicity Accumulating Function

GLOBAL INITIALIZATION OF TRACES The following warning from Racket's documentation tells developers about an essential constraint that the language does not enforce:

> If a key in an `equal?`-based hash table is mutated (e.g., a key string is modified with `string-set!`), then the hash table's behavior for insertion and lookup operations becomes unpredictable.

Time and again, programmers fail to heed this warning, experience arbitrary program behavior, and have a difficult time debugging.

```
(provide
  (contract-out
   [hash-set hash-set/c]
   [string-set! (-> mutable/c natural? char? void?)]))

(define-values (hash-set/c mutable/c)
  (trace/c ([t any/c])
    #:global
    (values (-> hash? (list/t 'set t) any/c void?)
            (list/t 'mut t))
    (full (t) not-interfere?)))
```

Figure 8.4: Contract for Hash Mutation

Figure 8.4 shows a trace contract that can enforce this property. First, the body contract produces two values using the `values` function to return multiple results [8]. Because the property relates different functions, i.e., `hash-set` and `string-set!`, their contracts need to be created

```
;; not-interfere?: Stream → Boolean
(define (not-interfere? xs)
  (match xs
    [(stream) #t]
    [(stream* `(mut ,x) xt) (not-interfere? xt)]
    [(stream* `(set ,x) xt)
     (and (not (stream-member? xt `(mut ,x)))
          (not-interfere? xt))]))
```

Figure 8.5: Predicate for Hash Mutation

within the same `trace/c`. Second, the `#:global` option causes the state of the trace contract to be initialized at *definition* time, not the usual *attachment* time. Without `#:global`, the `hash-set/c` and `mutable/c` contracts would be initialized separately and could never interact. Finally, the `list/t` function alters the given collector to tag incoming values with a symbol. Here, the symbol is used to indicate the operation. The `not-interfere?` predicate in Figure 8.5 ensures that no key is modified after it is inserted into a hash table.[3]

8.2 BLAME AND SUSPECTS

When a contract system discovers a contract violation, it raises an exception that includes a witness value and a pointer to the responsible component. As Lazarek et al. [91] show in the context of behavioral contracts, blame assignment comes with enough information to almost always locate the actual source of a bug. They simulate tens of thousands of buggy programs by introducing a targeted fault via mutation. In most cases, following blame assignment leads to the source of the bug. The few hundred cases where blame fails to identify the bug is precisely due to a lack of multi-call contracts. One of their examples is the DUNGEON program. Strengthening the behavioral contract to a trace contract for DUNGEON provides exactly the needed blame information. See Section 12.1 for a description of the DUNGEON benchmark.

Trace contracts also complicate the situation, however. By default, blame goes to the party that added a value to the trace just before the predicate fails. Since all prefixes of the trace satisfied the predicate, this blame assignment seems to make sense. Yet, debugging real scenarios suggests that the usual metatheoretic properties of a contract system (blame correctness [38] and complete monitoring [43]) may not be strong enough design guidelines for trace contracts.

3 A trace contract is one solution to eliminating this source of undefined behavior, but there are others. For instance, Racket comes with an `immutable?` predicate that recognizes some immutable values. The keys of hash tables could be constrained to immutable values, thereby imposing no performance overhead on mutable operations.

To make this discussion concrete, imagine a scenario with five components (A, B, C, D, E), where each contributes a number to a trace in increasing order ($\leqslant$). Here is an execution:

Component	A	B	C	E	D
Contribution	1.41	2.71	3.14	5.05	4.67

A trace contract would blame D because it contributes 4.67, causing the $\leqslant$ relation to fail. But E might have made a call to the API out of order, and blaming just D does not even indicate a suspicion that some other component could be at fault. It is often useful to know the source of *all* values in a trace. After all, the idea behind trace contracts is to subject multi-function interactions to contractual obligations.

A careful reader may argue that the problem is not with the blame assignment system but with the predicate. Rather, the contract should enforce that E provides a real number in $[4, 5]$ and $\leqslant$ simply does not capture the specification to a sufficient degree. This claim is already true about behavioral contracts because a predicate may always be weaker than the intended property. And if the predicate is weaker than the intended property, the contract system may blame the wrong party.

This argument overlooks the key premise of contract-system design: blame assignment must help developers narrow the search space for bugs, *regardless of the strength of the predicate*. To explain this idea rigorously, Lazarek et al. [91] turn folk wisdom into two properties: *blame trail* and *search progress*. The blame trail property states that either (1) blame is assigned to the buggy component or (2) blame can be shifted to another component by strengthening contracts. The search progress property states that blame shifting always points to a component closer to the bug than before the modification.

For trace contracts, both properties can be violated in practice. In the example, strengthening contracts on D may not shift the blame, meaning the blame trail property is violated. When strengthening a trace predicate, the violating trace may decrease in length but there is no reason to think *a priori* that the last contributor to a trace is always closest to the source of a bug, violating the search progress property. In short, the current blame assignment scheme points to the broken contract, but more information is needed to help developers identify the fault.

To address this problem, the implementation comes with three different ways of expressing blame assignments. Let a *suspect* be any party that contributes to a trace. Here are the three supported mechanisms for expressing blame in terms of suspects:

1. By default, the `trace/c` implementation does not report suspects. Instead, the error message merely mentions the violated contract and the violating party.

2. The `setof-suspect` option forces the trace-contract system to track the set of all suspects and report that information when assigning blame. Often, there are just two parties to a contract.

3. The `listof-suspect` option causes the trace-contract system to report the exact sequence of suspects, one per value in the trace. This option supplies the most information, but it requires a large amount of memory and may produce long error messages.

8.3 ADDITIONAL FUNCTIONALITY

Working with the trace-contract system pointed to limitations in the existing behavioral-contract system. In particular, additional combinators are needed to support the specification of interception points relevant to trace contracts. Fortunately, these practically important combinators are orthogonal additions to the base system. The trace-contract library comes with additional functions for manipulating interception points, resetting state explicitly, transforming collectors, and augmenting error messages with additional information.

Unlike behavioral contracts, trace contracts occasionally need to note events even in the absence of an informative value flow. For example, when a function receives no arguments, there is no natural interception point. The trace-contract library supplies some combinators to create interception points for such situations, e.g., `apply/c` and `return/c`.

Collector transformers wrap a collector and compute the value to be added to a trace from the given one. An example is `list/t`, which allows a programmer to tag values before they are appended onto a trace. Typically, this tag adds information about the interception point.

In practical situations, the `fail` function may have to perform more tasks than just informing the contract system of a failure. A software system may have to recover from a contract failure, and in those cases, a failure should reset accumulators to certain values. The author of a trace contract may also wish to add information about the rationale behind a failure. To this end, the trace-contract system supports the augmentation of error messages.

8.4 IMPLEMENTATION OVERVIEW

An implementation effort also informs designers of what is needed in a target host language to add a new feature. While the use of Racket's macro system greatly facilitates the addition of macro-expressible features, it should not be much more effort to extend existing compilers directly with trace contracts, provided the target language supports certain features. The experience implementing trace contracts in Racket suggests a few criteria.

MONITORING HIGHER-ORDER VALUES A trace is a data structure representing the sequence of values collected from various interception points. In the context of a functional language, function calls and returns are obvious interception points. Similarly, in an object-oriented language, this same role is played by methods. Generally speaking, an implementer's first business is to decide where to enable interception and how to enable the monitoring of value flow. In a higher-order language, a contract system cannot determine statically where a particular call or return takes place. Thus, the target language's runtime must support monitoring value flows. The Racket implementation employs proxy values [134] for interception. With proxies, it is straightforward to perform interception in the presence of higher-order values.

Proxies are not the only option. For instance, the weaving mechanism from aspect-oriented programming [85] can be used for a similar purpose. Roughly speaking, weaving injects code into a program at specifiable program points. Although weaving is powerful, it is not clear whether weaving can efficiently and effectively intercept values in a higher-order language, as needed by the trace-contract design.

MUTATION WITHIN CONTRACTS Trace-contract checking is effectful. When a collector receives a value, it mutably adds this value to a trace. That being said, the underlying component subject to the trace contract may be purely functional. Hence, the underlying language must allow side effects in contracts, even though trace predicates themselves are pure functions.[4]

Trace contracts are expressible as shorthand in an underlying language with higher-order contracts and mutable data structures. In the terminology of Felleisen [47], the new feature is macro expressible. Felleisen also shows that imperative assignment increases the expressive power of a pure host language. By implication, trace contracts are *not* expressible in a pure setting.

INTERCEPTION AND CROSSING TIMES A trace-contract system assumes that crossing and interception time in the target contract system are separate. As it turns out, the implementation of trace contracts exposed this lack of separation in Racket's contract system. Racket fails to separate the two points in one combinator: the depended-upon argument contract in `->i` [40]. A change to Racket's contract system allows trace contracts to distinguish these boundary crossings, meaning that a collector may ignore arguments passing through a boundary that has indy-party blame. This solution is sufficient to eliminate the duplicate-collection problem without major changes to the contract system.[5]

4 Since collectors mutate traces, checking a collector is not idempotent. While idempotence is sometimes considered an important property of contract systems [36, 53], it often fails to hold for a variety of reasons. For example, Owens [118] and Hinze et al. [78] observe violations of idempotence in several useful contexts.

5 Thanks to Robby Findler for help with this change to Racket's contract system.

MACROS NOT NEEDED An implementer can easily add trace contracts to a language with a rich macro system, such as a Racket. While macros are a convenient implementation mechanism for trace contracts, they are not a requirement. The implementer of a language such as SML, which elaborates surface syntax into a small kernel, can add trace contracts with a similar addition to the front-end elaborator.

8.5 TRANSLATING TO EFFECT RACKET

```
(define-syntax #%module-begin
  (syntax-parser
    [(_ ?body ...)
     #'(with ((ref-ctc-handler empty-store))
         ?body ...)]))

(define-syntax trace/c
  (syntax-parser
    [(_ ?var:id ?body:expr ?acc-fn:expr)
     #'(let ([f ?acc-fn])
         (self/c
          (λ _
            (let ([?var (collector/c (ref #t) f)])
              ?body))))]))

;; collector/c: Reference (Any Any → Any) → Any
(define (collector/c r f)
  (λ (v)
    (define acc (f (ref-get r) v))
    (ref-set r acc)
    acc))
```

Figure 8.6: Trace Contracts in Effect Racket

A simplified version of trace contracts can be translated to Effect Racket, as shown in Figure 8.6. Trace contracts require state, so programs must be wrapped in a top-level contract handler (hence the `#%module-begin` macro) providing the same interface as the mutable reference implementation alluded to in Section 6.1. In Racket, redefining `#%module-begin` allows for a language or library to wrap an entire module, often adding extra functionality [140]. For the definition of `trace/c` itself, `self/c` is used to initialize the mutable reference that holds the current accumulator. A collector contract is created that, upon protecting a value, updates the stored accumulator using the accumulating function `f`. The new accumulator, if it is truthy, causes the contract check to pass.

9 ATTRIBUTE CONTRACTS

Parameter and trace contracts are suitable in a broad range of situations. However, there is an important set of properties that is not served well by either kind of contract. When the constraints on a value are spread across several functions, perhaps in different modules, trace contracts are not a natural solution. A cousin of trace contracts, called *attribute contracts*, allows for a piece of state to be associated with a value and manipulated easily across many functions.

9.1 EXAMPLES

Here is an excerpt from the documentation of Racket's drawing library-for the `set-bitmap` method of the `bitmap-dc` object:

> Installs a bitmap into the drawing context (DC), so that drawing operations on the bitmap DC draw to the bitmap. A bitmap is removed from a DC by setting the bitmap to `#f`. A bitmap can be selected into at most one bitmap DC, and only when it is not used by a control (as a label) or in a `pen%` or `brush%` (as a stipple).

Certainly this stipulation could be expressed as a trace contract with globally initialized traces. However, the contract would be unwieldy because the trace would have to be manipulated such that different bitmap instances do not interact with one another. Instead of keeping state in a trace, attribute contracts keep state in the proxy protecting a value. Contracts mutate and check this piece of state, keeping track of information about how the value may be used, without the underlying value being aware of this information. From the perspective of the underlying implementation, attribute contracts manipulate *ghost state*, i.e., state associated with a value that only the contract system can retrieve and update.

To enforce this property, an attribute contract can associate with each bitmap the set of values (i.e., drawing contexts, pens, brushes) that it is installed into. So, calling the `set-bitmap` method of a drawing context with a bitmap adds the drawing context to the bitmap's set of DCs. This attribute can then be read and updated by other methods.

Figure 9.1 shows a contract that is intended to protect newly constructed bitmaps. In addition to ordinary method contracts inside `object/c`, the contract uses `attribute/c` to associate a bitmap with a key–value pair. The key is a first-class value, called an *attribute*, that in this case is referred to as `bmp-key`. Initially, the value of this attribute is

```
(define bmp-key (make-attribute))
(define bitmap/c
  (and/c
   (attribute/c bmp-key (set))
   (object/c  —— method contracts ——)))
```

Figure 9.1: Contract for Bitmap Constructor

the empty set, representing that the bitmap is not installed anywhere. Once the attribute is initialized for bitmap objects, contracts can update and query the attribute.

```
;; bitmap-install: Any → Contract
(define (bitmap-install/c x)
  (attribute-update/c bmp-key (λ (s) (set-add s x))))

;; bitmap-uninstall: Any → Contract
(define (bitmap-uninstall/c x)
  (attribute-update/c bmp-key (λ (s) (set-remove s x))))

;; bitmap-exclusive: Any → Contract
(define (bitmap-exclusive/c x)
  (and/c (attribute-satisfies/c bmp-key set-empty?)
         (bitmap-install/c x)))
```

Figure 9.2: Bitmap Attribute Contracts

Figure 9.2 defines several functions for updating attributes. Two contracts, `bitmap-install/c` and `bitmap-uninstall/c`, add or remove an element from the set associated with the `bmp-key` attribute using `attribute-update/c`. The `bitmap-exclusive/c` contract checks that the given bitmap is not installed anywhere and updates the attribute such that the set contains only `x`. Note that the quoted documentation implies that a bitmap can be used as the stipple of multiple pens and brushes. The exclusivity restriction is present only when it comes to installing a bitmap in a drawing context.

Any object that has a method that installs a bitmap needs to be adjusted to check and then update the bitmap's attribute. Shown in Figure 9.3 are contracts for the bitmap drawing context and pen that demonstrate how to turn the informal constraint into a checked contract. These two contracts are similar but are not the same. For `bitmap-dc%/c`, the `set-bitmap` method gets a dependent function contract where the first argument is the target of the method call, i.e., the bitmap drawing context. Using `uninstall-self/c`, the contract extracts out the bitmap currently installed in the drawing context. This

```
(define bitmap-dc%/c
  (class/c
   — method contracts —
   [set-bitmap
    (->i ([self (uninstall-self/c
                 (λ (self) (send self get-bitmap)))]
          [bmp (self) (or/c #f (bitmap-exclusive/c self))])
         [res void?])]))

(define pen%/c
  (class/c
   — method contracts —
   [set-stipple
    (->i ([self (uninstall-self/c
                 (λ (self) (send self get-stipple)))]
          [bmp (self) (or/c #f (bitmap-install/c self))])
          [res void?])]))

;; uninstall-self/c: (Any → Any) → Contract
(define (uninstall-self/c get)
  (self/c
    (λ (self)
      ((or/c #f (bitmap-uninstall/c self))
       (get self)))))
```

Figure 9.3: Bitmap Drawing Context and Pen Contracts

value is either `#f` or an actual bitmap, in which case the bitmap is uninstalled. The `bitmap-exclusive/c` contract on the bitmap argument ensures that the bitmap is exclusively installed in the drawing context. A similar contract is placed on pens, although exclusivity of the bitmap is not required.

9.2 IMPLEMENTATION OVERVIEW

While attribute contracts resemble trace contracts, in the sense that they are effectful, their implementations in Racket do not share code. Attribute contracts require a mechanism to store information alongside a value without otherwise affecting it. Languages such as Clojure [76] have a well-known feature called metadata that allows for arbitrary maps to be associated with values. In Clojure, metadata is used frequently enough that it has dedicated reader syntax.

Racket does not have the same metadata facility, but it can be simulated using *impersonator properties*. In Racket, impersonators [134] are the most general term for a proxy value. Impersonators can have prop-

erties that carry information. In the contract system, this feature is primarily used to support the `value-contract` function for extracting the contract from a protected value.

Attribute contracts use impersonator properties to store, retrieve, and update information for contract checking. Not all values, such as strings, can be impersonated. Such values cannot be used with attribute contracts. For this reason, the mutable key example from Section 8.1 cannot be expressed as an attribute contract.

9.3 TRANSLATING TO EFFECT RACKET

```
(define-syntax #%module-begin
  (syntax-parser
    [(_ ?body ...)
     #'(with ((ref-contract-handler empty-store))
         ?body ...)]))

(define (attribute/c key val)
  (make-contract
   #:late-neg-projection
   (λ (blm)
     (λ (subj neg)
       (meta-set subj key (ref val))))))

(define (attribute-update/c key proc)
  (λ (subj)
    (define r (meta-ref subj key))
    (ref-set r (proc (ref-get r)))
    #t))

(define (attribute-satisfies/c key pred)
  (λ (subj)
    (pred (ref-get (meta-ref subj key)))))
```

Figure 9.4: Attribute Contracts in Effect Racket

An implementation of attribute contracts in Effect Racket is shown in Figure 9.4. Similar to the translation of trace contracts in Section 8.5, the program needs a top-level contract handler for mutable references. The definition of `attribute/c` uses `make-contract` to produce a value with the desired attribute attached. Attributes are set and retrieved from values via the `meta-set` and `meta-ref` functions, respectively. These functions come from an auxiliary library that provides a Clojure-like metadata facility for Racket. The `attribute-update/c` and `attribute-satisfies/c` contracts update and check the attributes of values, respectively, using the same metadata functions.

Part III

PRACTICAL

Implementations do not exist in a vacuum but within a larger software ecosystem. Effectful contracts interact with, and benefit from, features in the host language. This part of the dissertation looks at how trace contracts can be combined with other language constructs in powerful ways.

10 DOMAIN-SPECIFIC NOTATIONS

Contract systems stand to benefit from domain-specific notations for specifications. In other words, it should be easy to plug in any number of domain-specific specification languages into a contract system. Effectful contracts in general, and trace contract in particular, improve in readability when paired with domain-specific notations. This chapter presents several illustrative examples of how to combine trace contracts with domain-specific languages (DSLs) tailored for writing *specifications* (Section 10.1). These domain-specific notations are defined using Racket's `syntax-spec` system (Section 10.2), which brings about many advantages (Section 10.3).

10.1 EXAMPLES

The examples in this section demonstrate the integration of trace contracts with domain-specific notations. Each contract builds in complexity, starting from simple regular expressions, all the way up to entirely graphical means of expressing specifications.

SPECIFYING WITH REGULAR EXPRESSIONS Racket comes with a built-in library, `racket/draw`, for drawing images. The library provides a thin wrapper around a low-level graphics API written in C. As such, the wrapper must protect against client behavior that would induce undefined behavior at the C level. One instance of undefined behavior occurs with drawing context (DC) objects.

To produce an image with `racket/draw`, a developer must first choose a DC representing the desired output device. There are many such contexts that all share a common interface. Part of this interface is a collection of methods for managing the pages of a document: `start-doc`, `start-page`, `end-page`, `end-doc`. Clients must call these methods in a particular order. It does not make sense to call, e.g., `end-doc` before `start-doc`. Moreover, all drawing commands must occur within a page.

Here is a regular expression that describes a valid *complete* sequence of method calls:

$$\texttt{start-doc}, (\texttt{start-page}, \texttt{draw}^*, \texttt{end-page})^*, \texttt{end-doc}$$

This regular expression is not yet suitable for trace-contract monitoring. A trace contract also checks every *incomplete* sequence of method calls, not just the complete sequence. So, this regular expression has to be adapted to accept any prefix of the complete sequence.

```
(define-re SINGLE-PAGE
  (seq/close 'start-page (star 'draw) 'end-page))

(define-re DC-RE
  (seq/close 'start-doc (star ,SINGLE-PAGE) 'end-doc))
```

Figure 10.1: DC Regular Expressions in Racket

Shown in Figure 10.1 is an adapted version of the regular expression using a library for constructing automata. The `define-re` form compiles the given regular expression to a finite-state automaton. Within `define-re`, the `seq/close` form denotes a regular expression that accepts, not just the given sequence, but any prefix of that sequence.

```
(provide
  (contract-out
   [make-ps-dc (-> (dc/c DC-RE))]))

(define (dc/c mach)
  (trace/c ([s symbol?])
    (object/c
      [start-doc  (apply/c [s 'start-doc])]
      [start-page (apply/c [s 'start-page])]
      [draw-point (apply/c [s 'draw])]
      [end-page (apply/c [s 'end-page])]
      [end-doc  (apply/c [s 'end-doc])])
    (accumulate mach
      [(s) (λ (acc x)
             (define acc* (acc x))
             (if (machine-accepting? acc*) acc* (fail)))])))
```

Figure 10.2: Contract for Drawing Contexts

Figure 10.2 shows a trace contract that enforces the protocol defined by `DC-RE`. Given a finite-state automaton, `dc/c` produces a contract for a DC where the method call sequence is governed by the regular expression. In the body of `dc/c`, a trace contract is wrapped around an object contract specifying each of the DC methods. There is only a single trace, `s`, that contains symbols corresponding to method calls. The `apply/c` combinator provides the collector with a constant value each time a protected method is called. To check the protocol, the trace predicate uses the state of the automaton as the accumulator. So long as the automaton is accepting, the contract is satisfied. The trace contract is then used in the codomain of `make-ps-dc`, a function producing PostScript (PS) drawing contexts.

```
(provide
  (contract-out
   [make-eps-dc (-> (dc/c EPS-RE))]))

(define-re EPS-RE
  (seq/close 'start-doc ,SINGLE-PAGE 'end-doc))
```

Figure 10.3: Encapsulated PostScript Contract

However, there is more than one kind of DC. In particular, an Encapsulated PostScript (EPS) drawing context has a slightly different constraint than an ordinary PS context. Since an EPS file is intended to be embedded in a larger document, it can only have a single page. Supporting EPS is easy since `dc/c` abstracts over the regular expression. Checking a different protocol requires passing in a different finite-state automaton to `dc/c`, as shown in Figure 10.3.

VALUE-DEPENDENT PROTOCOL Imagine a board-game framework that pits player components against one another. A natural implementation of a player is as an object with methods that correspond to game stages. Each player expects that these methods are called in a certain order, which may depend on the state of the game. Methods calls must satisfy a value-dependent, multi-function, temporal property.

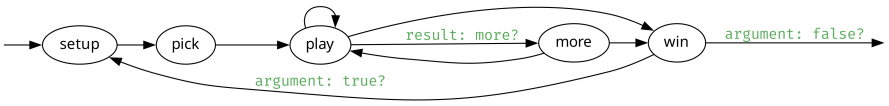

Figure 10.4: Game Protocol Diagram

Figure 10.4 displays a state-transition diagram for an automated board-game player. States in this diagram indicate the order in which the referee component must call the player's methods. Labeled edges represent transitions that depend on either an argument value or a return value. Unlabeled edges represent independent transitions. Since there are several possible transitions for some states, this protocol demands a non-deterministic finite-state automaton (NFA).

The diagram dictates that players must implement five methods:

1. A `setup` method that delivers the game pieces.

2. A `pick` method that asks a player to choose some game objectives.

3. A `play` method that grants a player the right to take a turn. The result is either a request to perform an *action* on the game state or a request for *more* game pieces.

```
(provide
  (contract-out
   [player-factory (-> strategy/c player/c)]))

 (define PLAYER-NFA
   (nfa {setup} {setup pick play more win done}
     [setup (list 'setup _) {pick}]
     [pick  (list 'pick _)  {play}]
     [play  (list 'play (? action?)) {play win}]
     [play  (list 'play (? more?))   {play more win}]
     [more  (list 'more _) {play win}]
     [win   (list 'win #t) {setup}]
     [win   (list 'win #f) {done}]))

(define player/c
  (trace/c ([x any/c])
    (object/c
      [setup (->m game-map? (list/t 'setup x))]
      [pick  (->m set? (list/t 'pick x))]
      [play  (->m state? (list/t 'play x))]
      [more  (->m list? (list/t 'more x))]
      [win   (->m (list/t 'win x) any/c)])
    (accumulate PLAYER-NFA
      [(s) (λ (acc x)
             (define acc* (acc x))
             (if (machine-accepting? acc*) acc* (fail)))])))
```

Figure 10.5: Player Contract

4. If the referee gets this second kind of request in response to `play`, it may invoke the player's `more` method. It may also skip this call, depending on the game state.

5. The player is granted turns and more pieces until the referee discovers an end-game condition and then informs the player whether it won or lost. The player may participate in the next game only if `win` is called with `#t`.

A trace-contract specification for the protocol is shown in Figure 10.5. In this particular software system, a factory function creates automated players from a strategy and returns player objects that implement the above five methods. The contract on this factory method attaches a trace contract to each player object. As a result, every player object must obey the order of method calls specified in Figure 10.4.

The protocol is specified as an NFA over an alphabet containing the names of methods, along with the specific arguments or return values

of two of them: `play` and `win`. The trace contract, as before, simulates the automaton with `accumulate`. For `setup`, `pick`, and `more`, the transition is independent of run-time values. For `play` and `win`, the transition is value dependent. The `play` method uses the `action?` and `more?` predicates to determine the next set of states. It does so using Racket's `(? p)` pattern that match a value if the predicate `p` holds. The `win` method determines the successor state based on its boolean-valued argument.

CONCISE OBJECT CONTRACTS The MapIterator property from Java's API [117] states that an `Iterator`, created from a `Collection`, which is itself derived from a `Map`, cannot be used *after* the `Map` has been mutated. The word "after" suggests a temporal relationship—a need to look back at whether a mutating method of `Map` has previously been called. Past-Time Linear Temporal Logic (PLTL) [96] is one suitable choice for stating the MapIterator property:

```
(define-pltl map-iterator-violation
  (∧ 'next (⧫ 'mutate)))
```

PLTL comes with the primitive temporal operators • ϕ (pronounced "previous") and ϕ S ψ (pronounced "since"). Otherwise, PLTL looks like first-order logic. All other temporal operators, for example ⧫ ϕ (pronounced "once"), are derived notions. With these operators, an *invalid* sequence of method-call events is easy to express: `next`∧⧫`mutate`.

```
(define map-iterator/c
  (object-trace/c
   #:refutes (pltl map-iterator-violation)
   [next (->m any/c) 'next]))

(define map-collection/c
  (object-trace/c
   [iterator (->m map-iterator/c)]))

(define map/c
  (object-trace/c
   [keys   (->m map-collection/c)]
   [remove (->m any/c void?)       'mutate]
   [set    (->m any/c any/c void?) 'mutate]))
```

Figure 10.6: First Attempt: Checking the MapIterator Property

Instead of employing `trace/c`, a developer can use `object-trace/c` with a PLTL formula to create an enforceable contract. Figure 10.6 shows a first attempt at a contract for MapIterator. The three displayed definitions express a specification involving three methods across two classes, with an intervening class that bridges the gap. Two methods from the `Map` class (`remove` and `set`) mutate the `Map` object. The

intent with these contracts is that any iterator derived from a collection of keys becomes unusable when the underlying map is mutated. The `#:refutes` clause requests that the PLTL formula named `map-iterator-violation` is translated into a predicate over the trace of method-call events. If the contract ever discovers that the formula is satisfied, then the system signals a violation. In addition, each method signature can specify a value to be added to a trace associated with the object when the method is invoked. For these contracts, those values are the symbols `'next` and `'mutate`.

As is, these contracts are not quite right. An `object-trace/c` contract knows only about method calls for the current object. Therefore, `map-iterator/c` sees only calls to the `next` method and is unaware of any mutations to the underlying map.

```
(define (map-iterator/c map/c)
  (object-trace/c
   #:refutes (pltl map-iterator-violation)
   #:include map/c
   [next (->m any/c) 'next]))

(define (map-collection/c map/c)
  (object-trace/c
   [iterator (->m (map-iterator map/c))]))

(define map/c
  (object-trace/c
   [keys   (->m (map-collection/c this-contract))]
   [remove (->m any/c void?)       'mutate]
   [set    (->m any/c any/c void?) 'mutate]))
```

Figure 10.7: Checking the MapIterator Property

Figure 10.7 displays an adaptation such that `map-iterator/c` can accumulate information about other relevant method calls. To express this relationship, the contracts for collection and iterator objects become *functions* that accept `map/c` itself. The `object-trace/c` form uses the keyword `this-contract` which, similar to `this` for objects, refers to the `object-trace/c` contract itself. Even though the `map-collection/c` function just passes this contract through to `map-iterator/c`, the latter actually makes use of it. The `#:include` clause informs the contract system that events from the trace for `map/c` are to be injected into the trace for `map-iterator/c`. Consequently, `map-iterator-violation`, which refers to the `'mutate` event, now works correctly when placed in the `#:refutes` clause.

These contracts look simple, but the underlying property is sophisticated. Consider the scenario where an iterator is created, the underlying map is mutated, and then another iterator is created. It should be the case that the first iterator is invalidated, while the second one remains usable. In other words, these two iterators require independent pieces of state. Some events, such as mutations to the underlying map, are shared with both. Other events, such as calls to `next`, are particular to a specific iterator and are not shared. All this careful state management is handled *automatically*. Conversely, manually constructing defensive checks to correctly keep track of this information is difficult.

MIXING DOMAIN-SPECIFIC NOTATIONS Interfaces should be expressed as directly as possible, using the notation most appropriate for the property at hand. Sometimes a single interface benefits from mixing several domain-specific notations.

Consider an idealized TCP interface, similar to the one found in Java, which consists of four classes: a TCP manager, a TCP listener, an input stream, and an output stream. A TCP manager, when given a port number, is responsible for creating a TCP listener that deals with the given port. The purpose of a TCP listener is to accept client connections. For each connected client, a TCP listener produces an input and output stream for receiving and sending data, respectively.

Many languages, including Java, guarantee the following properties of TCP sockets: (1) StreamClose states that input and output streams can be used until either the stream itself is closed or the underlying TCP listener is closed; and (2) PortExclusivity states that at any point in the program, there can be at most one active TCP listener per port. The Java standard library uses defensive checks to enforce these properties.

Contracts can express both properties concisely, and allows for a second or third TCP implementation to be monitored easily. The two properties, though, warrant different notations. The StreamClose property can be expressed as a regular expression:

```
(define-re port-re
  (seq
    (star 'use)
    (opt (union 'close-stream `(close-listener ,_)))))
```

This regular expression accepts traces of method-call events that use a stream (input or output) until it has been closed, or the underlying listener is closed. There is one unusual feature here, which is the close listener event. It carries the port number of the corresponding listener, which is simply ignored in `port-re` using a wildcard pattern. Why the close listener event carries a port number will be clear in a moment.

The PortExclusivity property is not easily expressed as a regular expression. Another formalism, known as quantified event automata (QEA [14]), is a good match.

```
(define manager-qea
  (qea
   (∀ port)
   (start ready)
   [-> ready `(listen ,port) listening]
   [-> listening `(close-listener ,port) ready]))
```

Figure 10.8: QEA for PortExclusivity

Figure 10.8 shows a QEA for PortExclusivity. A QEA is similar to an ordinary finite-state automaton, but it is enriched with quantifiers. By quantifying over a variable, a QEA describes a *family* of automata: one per instantiation of quantified variables. Intuitively, `manager-qea` constructs a two-state automaton for every possible port number. These automata are created on demand, though, to avoid unnecessary overhead. Each automaton starts in the `ready` state and transitions to the `listening` state when a new TCP listener is created on that port. When the listener is closed, the automaton moves back to the `ready` state. Each listener event must carry a port number, so the event can be dispatched to the appropriate automaton.

Using **`object-trace/c`**, these logical specifications can be connected to the TCP interface itself, as shown in Figure 10.9. Two differences from the example in Figure 10.7 are worth pointing out.

First, `tcp-manager/c` uses the dependent method contract **`->dm`** [133] to produce an event that is dependent on the argument to `listen`. This contract form, similar to **`->i`**, permits naming the arguments and result—binding those names for use in the event constructor. Here, the listen event contains the `port` argument.

Second, `tcp-listener/c` uses `#:extend` to contribute events to the parent `manager/c` contract. As shown in Figure 10.7, `#:include` allows events to flow from "ancestor" to "descendant" object. By contrast, the `#:extend` keyword allows events to flow in the other direction, from "descendant" to "ancestor" object. Specifically, the TCP manager (i.e., the parent) must be aware if a TCP listener (i.e., the child) is closed on a particular port, because then that port becomes available for listening once again according to PortExclusivity. Using `#:extend manager/c` tells the contract to communicate close-listener events from the listener to the manager. It turns out that StreamClose also requires information to flow in the other direction because input and output streams (i.e., the children) must be closed if the underlying listener (i.e., the parent) is closed. This direction is covered using `#:include listener/c`. In sum, this example demonstrates that **`object-trace/c`** can accommodate information flow in both directions, even within the same interface, and can mix distinct domain-appropriate notations to check all desired properties.

```
(define (input-stream/c listener/c)
  (object-trace/c
   #:satisfies port-re
   #:include listener/c
   [read-byte (->m byte?) 'use]
   [peek-byte (->m byte?) 'use]
   [close     (->m void?) 'close-stream]))

(define (output-stream/c listener/c)
  (object-trace/c
   #:satisfies port-re
   #:include listener/c
   [write-byte (->m byte? void?) 'use]
   [close      (->m void?)       'close-stream]))

(define (tcp-listener/c manager/c port)
  (object-trace/c
   #:extend manager/c
   [close  (->m void?) `(close-listener ,port)]
   [accept (->m (values (input-stream/c this-contract)
                        (output-stream/c this-contract)))]))

(define tcp-manager/c
  (object-trace/c
   #:satisfies manager-qea
   [listen (->dm ([port (integer-in 0 65535)])
                 [res (tcp-listener/c this-contract port)])
           `(listen ,port)]))
```

Figure 10.9: The TCP Interface

HYBRID SYNTAX In the board-game example, a state-machine diagram (Figure 10.4) was manually translated into DSL code (Figure 10.5) for use in a trace contract. Of course, there is no guarantee that the code corresponds to the diagram. Worse, the diagram and the code are likely to get out of sync as the library evolves.

Ideally, the state-machine diagram itself would be a part of the code. With hybrid textual–graphical syntax [4], or *hybrid syntax* for short, a library author can describe a protocol graphically and have the corresponding run-time-checking code generated automatically.

Consider a different example, though. The authentication protocol for a REST API may constrain an object with three methods: `auth`, `req`, `done`. This protocol imposes the following constraints on method calls: (1) the `auth` method to sends credentials and receives an authentication token in response; (2) the `req` method, with an endpoint URL and

a valid authentication token, requests data; and (3) the `done` method ends the authenticated session.

```
;; [Sequenceof Message] -> Boolean
;; Returns whether the sequence of messages satisfies the
;; authentication protocol.
(def satisfies-auth-protocol?
```

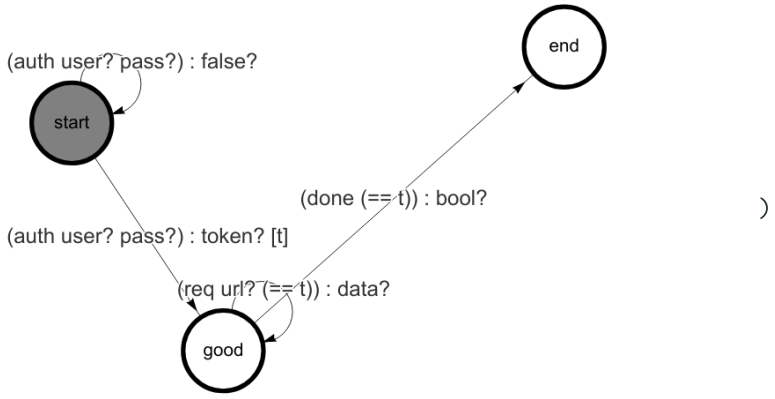

Figure 10.10: Hybrid Syntax for an Authentication Protocol

Figure 10.10 shows how hybrid syntax can express this protocol directly as a diagram. The automaton *graphical form* allows a developer to interactively construct an automaton directly inside a larger textual file. At compile time, the graphical form elaborates to a predicate that determines if a sequence of method calls satisfies the protocol. An eagle-eyed reader may notice that the language here is Clojure and not Racket. The hybrid-syntax system, for practical reasons, was built in ClojureScript [5]. Such a file can be imported within Racket via a lightweight bridge [2].

The automaton consists of three states: `start`, `good`, and `end`. The shaded gray background of `start` shows that it is the starting state. Each state indicates, via the transitions emanating from it, the set of methods a client module can call. For example, in the `good` state, a client module can call either the `req` or `done` method. A transition is labeled with a method name plus predicates for the arguments and result. If the arguments and result satisfy the predicates specified on the transition, then the automaton moves to the next state. If no such transition exists, then the protocol must have been violated, and the contract violation is reported.

In Figure 10.10 the transition corresponding to a successful authentication binds the returned token to the variable `t` (shown in square brackets). The scope of this binding includes all downstream transitions. Any transition in scope can then use this variable in predicates. For example, the expression `(== t)` constructs a predicate that determines if a value is equal to the token.

```
;; [Sequenceof Message] -> Boolean
;; Returns whether the sequence of messages satisfies the
;; Android MediaPlayer protocol.
(def satisfies-android-protocol?
```

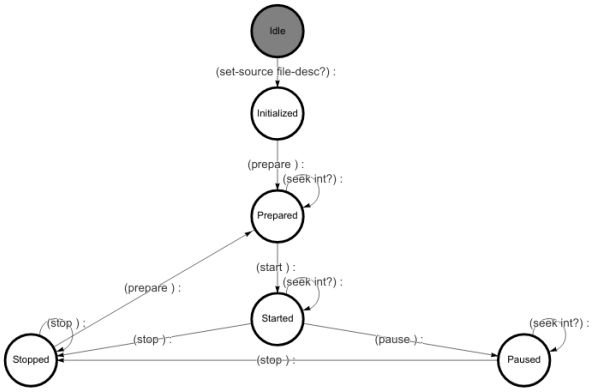

Figure 10.11: Android MediaPlayer with Hybrid Syntax

The diagram presented in Figure 10.10 is just one instance of a general-purpose graphical form. To demonstrate its versatility, Figure 10.11 shows a slightly simplified implementation of the Android MediaPlayer API [69] protocol, written with the same graphical form.

Interacting with the graphical form, a programmer performs GUI gestures to create new states, delete existing ones, add or delete transitions, edit the source and destination of a transition, turn states into starting or accepting states, rename states (via a text box), edit the predicates labeling a transition (via a text box), and change what variables are bound. These gestures are intuitive. For example, creating a new transition merely requires clicking and dragging from the source state to the destination state. Altering the properties of a transition involves selecting the transition and clicking the edit button.

The graphical form's elaborator analyzes code on the transitions to determine the necessary binding structure. Specifically, the elaborator creates a separate function for each transition with the appropriate parameters, and provides the run-time system enough information to supply the correct arguments to each function. Syntax and type errors in the specification are raised at compile time. For example, if a transition predicate specified a dependency on a variable that is not in scope, the elaborator would signal a compile-time error.

10.2 IMPLEMENTATION OVERVIEW

Implementing new domain-specific notations necessitates a means of syntactically extending a language. This section discusses how the notations in the previous section were created with the `syntax-spec` metaprogramming system [10].

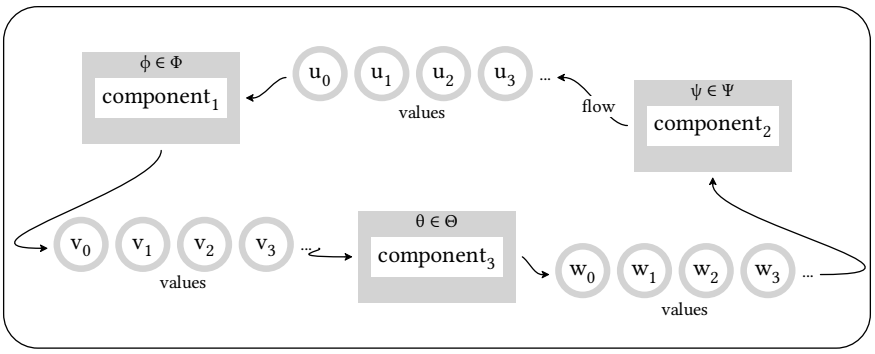

Figure 10.12: System Diagram for Domain-Specific Contracts

Figure 10.12 summarizes how domain-specific notations work within a contract system. Each software component is combined with a logical formula (the gray frame). The component's developers choose the most appropriate logic to use, based on the property to be enforced. These logic languages come with translators that turn these specifications into predicates, which then become pieces of trace contracts. In the diagram, the enforcement mechanism comes into play with the sequences of values that components exchange. The contract code checks the sequence of values that a component receives and sends out. Gray borders on values in Figure 10.12 indicate that values may acquire a proxy layer that contains specification-checking code from the component's interface [56, 134].

A metaprogramming system enables programmers to extend their language with new linguistic constructs and collect them in libraries or frameworks. Ideally, it permits blurring the distinction between built-in constructs and user-defined ones. Importantly, metaprogramming facilities exist *inside* the programming language. Depending on the chosen language, metaprogramming facilities can be used in the small and large: from quick syntactic abbreviations, all the way up to full feature-rich special-purpose DSLs.

Different languages have vastly different metaprogramming mechanisms. Some languages provide lightweight syntactic metadata: decorators in Python and annotations in Java. In other languages, such as Ruby, the syntax is sufficiently flexible, and the semantics sufficiently dynamic, that a DSL can be built without any explicit language support. Another major class of metaprogramming facilities are those based on (syntax-tree) *macros* that perform compile-time rewriting. Before code generation, the compiler applies these rules to obtain the program in a core syntax—a process called *macro expansion*. Languages in the Lisp family, such as Clojure [76], are especially well-known for macros, but others including Rust and Scala have adopted them too. Parenthetical syntax is not necessary [61].

Racket is well-suited for defining domain-specific notations as it provides many tools for language-oriented programming [52]. The `syntax-spec` metaprogramming library is one such tool. Two features of `syntax-spec` make it suitable for creating new notations. First, developers can declaratively specify the language grammar, making it straightforward to develop and maintain. Second, `syntax-spec` is *binding aware*, meaning it understands the scope of variable declarations. It can reliably provide services such as rename refactoring and determine the free variables of an expression.

$$\text{Formula} \ni \phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists x.\phi \mid \bullet\,\phi \mid \phi\ S\ \phi$$

```
(syntax-spec
 (nonterminal pltl-formula
  #:allow-extension pltl-macro
  #:binding-space pltl-space
  (neg f:pltl-formula)
  (disj2 f1:pltl-formula f2:pltl-formula)
  (conj2 f1:pltl-formula f2:pltl-formula)
  (exists x:pltl-var f:pltl-formula)
  #:binding (scope (bind x) f)
  (previous f:pltl-formula)
  (since f1:pltl-formula f2:pltl-formula)
  p:pltl-pat))
```

Figure 10.13: Grammar of PLTL (Simplified)

Figure 10.13 shows the Backus–Naur form (BNF) representation of the PLTL grammar and the `syntax-spec` definition for the same grammar in Racket. These grammars directly correspond to each other and it is easy to go back and forth. In order to make the `syntax-spec` grammar binding aware, a developer adds *binding specifications*, such as the one for the `exists` form. Here, the binding specification states that in $\exists x.\phi$, the variable x is bound in ϕ.

Implementing the static semantics of a domain-specific notation is almost as simple as declaring its grammar. Take the monitorability criteria of PLTL [16], which says that a well-formed PLTL formula can be turned into a predicate over a trace if it satisfies the judgment on the left side of Figure 10.14. The equivalent Racket code is on the right side of the figure. The Racket implementation pattern matches on the syntax using the `syntax-parse` system [33]. Each rule in the monitorability judgment corresponds to a branch in the match. In the PLTL pattern case, the function uses the free variables (`fvs`) feature of `syntax-spec`.[1]

1 A straightforward addition to this static semantics is a monitorability search procedure. Some tools, such as MonPoly [16], attempt to rewrite non-monitorable formulas into monitorable ones if possible.

$$\frac{\rho \vdash \phi \qquad \rho \vdash \psi}{\rho \vdash \phi \vee \psi}$$

$$\frac{\begin{array}{c}\rho_\phi \vdash \phi \\ \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi\end{array}}{\rho_\phi \cup \rho_\psi \vdash \neg\phi \wedge \psi}$$

$$\frac{\begin{array}{c}\rho_\phi \vdash \phi \\ \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi\end{array}}{\rho_\phi \cup \rho_\psi \vdash \neg\phi \,S\, \psi}$$

$$\frac{\begin{array}{c}\rho_\phi \vdash \phi \\ \rho_\psi \vdash \psi \qquad \rho_\phi \subseteq \rho_\psi\end{array}}{\rho_\phi \cup \rho_\psi \vdash \phi \,S\, \psi}$$

$$\frac{\rho_\phi \vdash \phi \qquad \rho_\psi \vdash \psi}{\rho_\phi \cup \rho_\psi \vdash \phi \wedge \psi}$$

$$\frac{\rho \cup \{x\} \vdash \phi}{\rho \vdash \exists x.\phi} \qquad \frac{\rho \vdash \phi}{\rho \vdash \bullet\, \phi}$$

$$\frac{\rho = \textsc{fv}(p)}{\rho \vdash p}$$

```
(define (monitorable? stx)
  (syntax-parse stx
    [(disj2 f g)
      (and (equal? (fvs #'f)
                   (fvs #'g))
           (monitorable? #'f)
           (monitorable? #'g))]
    [(conj2 (neg f) g)
     (and (subset? (fvs #'f)
                   (fvs #'g))
          (monitorable? #'f)
          (monitorable? #'g))]
    ── similar cases ──
    [(conj2 f g)
     (and (monitorable? #'f)
          (monitorable? #'g))]
    [(exists (x:id ...) f)
     (monitorable? #'f)]
    ── similar cases ──
    [(neg _) #f]
    [p #t]))
```

Figure 10.14: Static Semantics for PLTL (Simplified)

Finally, if a formula satisfies the static semantics, it can be translated into a predicate. Figure 10.15 shows a simple translation for PLTL that translates each formula into a tree of structs, which is then run using the PLTL interpreter provided by the library.

While Racket and `syntax-spec` make realizing domain-specific notations easy, the general architecture presented in Figure 10.12 should apply to many programming languages with varying degrees of ease. A programming language with powerful metaprogramming facilities reduces the notational overhead on programmers; without such facilities, an implementer may have to resort to shallow or deep embedding techniques [66], which may impose an extra burden on developers.

10.3 BENEFITS OF DOMAIN-SPECIFIC NOTATIONS

While the use of `syntax-spec` greatly facilitates the declaration and translation of new notations, this very simplicity obscures a number of aspects that come "for free" but may have to be realized separately in alternative implementations. When implemented with `syntax-spec`,

```
(define-syntax (translate stx)
  (syntax-parse stx
    [(_ (neg f))            #'(neg-s (translate f))]
    [(_ (disj2 f g))        #'(disj2-s (translate f) (
       translate g))]
    ── similar cases ──
    [(_ (exists (x ...) f)) #'(exists-s (set 'x ...) (
       translate f))]
    [(_ x:id)               #'x]
    [(_ p)                  #'(translate-pattern p)]))
```

Figure 10.15: Translator for PLTL (Simplified)

domain-specific notations signal syntax errors at the appropriate level, have direct support from the IDE, and are syntactically extensible.

STATIC ERRORS A `syntax-spec` language can automatically synthesize reasonable syntax errors for the new notation. For example, ill-formed PLTL formulas are identified like this:

```
> (define-pltl ϕ
    (S 'next))
expected more terms in: (S 'next)
```

In ϕ, the S operator expects another operand, so the error points out an arity mismatch.

Similarly, a `syntax-spec` implementation can issue an informative message when static checking fails. As mentioned in the preceding section, a syntactically well-formed PLTL formula may be *invalid* because it cannot be translated into a predicate (as is):

```
> (define-pltl ψ
    (∃ (t) (¬ (• t))))
not monitorable in: (¬ (• t))
```

Even if a PLTL formula is well-formed and valid, it can still be misused:

```
> (+ ψ 13)
cannot use PLTL formula here: ψ
```

To realize these benefits, the `define-pltl` form creates definitions in a separate *binding space*, as required by the one-line `#:binding-space` option in Figure 10.13. Languages, including domain-specific notations, are often composed of several syntactic categories that are mixed at particular interface points. For example, Java has a syntactic category of expressions and types that are interleaved in a certain way; definitions created in one do not exist in the other. The `syntax-spec` implementation of PLTL creates its own binding space, distinct from the one for expressions, meaning that a reference to a PLTL name from an expression context signals a static error.

USER EXTENSIBILITY PLTL comes with just two primitive temporal operators. Although derived operators add no expressive power to the logic, they are necessary for writing concise specifications. While the PLTL library could provide direct support for such derived operators, doing so has the serious downside of complicating the static semantics, the translator, and the runtime system, which would have to deal with each kind of operator separately.

Instead, a `syntax-spec` language can be made syntactically extensible with the `#:allow-extension` option. This option opens up the `pltl-formula` nonterminal for simple macro extensibility. A user can define additional operators with ease:

```
(define-pltl-rule ⊤
  (? (λ _ #t)))

(define-pltl-rule (⧫ ϕ)
  (S ⊤ ϕ))
```

The `syntax-spec`-generated `define-pltl-rule` form defines a syntactic transformation that rewrites all instances of ⊤ and ⧫ ϕ in a PLTL formula: ⊤ is rewritten to a predicate that always returns `#t`, meaning a machine that succeeds on all traces of collected events; and ⧫ ϕ is rewritten to a formula that requires nothing to be true until ϕ holds.

These syntactic transformations are seamlessly integrated into PLTL. For example, a user can write a utility library with these abbreviations and export them for use in other modules. In short, user-extensible transformations make the PLTL library flexible to use and simplify its implementation tremendously.

IDE INTEGRATION The `syntax-spec` library facilitates integrating new notations with Racket IDEs, including DrRacket [54] and Racket Mode for Emacs [75].

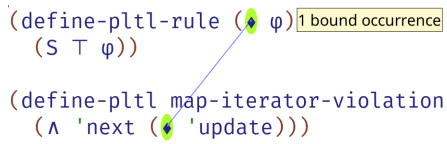

Figure 10.16: Binding Arrows for PLTL

Figure 10.16 shows one such service. When a user hovers over an identifier, arrows are drawn from the binding occurrence of an identifier to every reference. Here, the definition of ⧫ points to its use in the formula. The IDE has a rich understanding of the binding structure of a language, which can be used to support reliable program transformations including rename refactorings.

11 A CASE STUDY IN COMPOSITION

Imagine developing a channel-based concurrent application that conforms to a complex protocol specification. Trace contracts can help enforce such protocols, increasing confidence that the software is correct. To do so, a programmer must first translate the protocol into a predicate over channel messages. The predicate may be written as a ordinary function in the host language or using a domain-specific specification language as discussed in Chapter 10. Then, the programmer selects communication channels of interest and attaches trace contracts to monitor the message flow.

Importantly, trace contracts *compose* with other contracts via combinators, including ordinary channel contracts as defined by Racket's contract library. Composition is an essential characteristic of well-designed contract systems [42], avoiding the need to manually define a special primitive form of channel-protocol contracts. Protocol enforcement over channels simply decomposes into two orthogonal contract mechanisms.

This chapter starts with an illustration of the approach with a simple client–server example (Section 11.1) before scaling up to correctness properties of the two-phase commit algorithm (2PC) for distributed systems (Sections 11.2 and 11.3).

11.1 CLIENT–SERVER EXAMPLE

Consider a concurrent application with one client and one server that communicate via a channel. For the purposes of this example, the details of the channel itself do not matter; it could be a synchronous channel or an asynchronous channel. A simple protocol may state that when either process receives a number n, it sends to the other process the number $n + 1$.

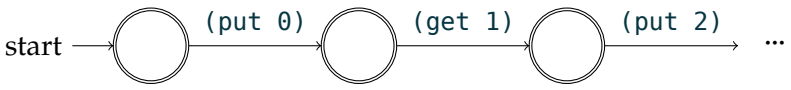

Figure 11.1: Client–Server Protocol Diagram

Figure 11.1 shows a state machine diagram for an interaction where the client kicks off communication with 0. The alphabet of the machine consists of channel actions (i.e., `put` or `get`) paired with the value that crosses the channel. A similar state machine can be constructed for the server, where the `put` and `get` actions are swapped.

```
(provide
  (contract-out
   [client-start (-> client/c any/c)]
   [server-start (-> server/c any/c)]))

(define (client-start ch)
  (for/fold ([acc 0])
            ([k (in-range ROUNDS)])
    (channel-put ch acc)
    (add1 (channel-get ch))))

(define (server-start ch)
  (for ([_ (in-range ROUNDS)])
    (channel-put ch (add1 (channel-get ch)))))
```

Figure 11.2: Client and Server Functions

Figure 11.2 shows the implementation of two functions following the protocol for several rounds of communication. The contracts for `client-start` and `server-start` state that they are functions whose channel arguments must satisfy `client/c` and `server/c`, respectively, and guarantee nothing about the return value.

```
(define client/c
  (trace/c ([n natural?])
    (channel*/c (list/t 'put n) (list/t 'get n))
    (accumulate —— check state machine ——)))

(define server/c
  (trace/c ([n natural?])
    (channel*/c (list/t 'put n) (list/t 'get n))
    (accumulate —— check state machine ——)))
```

Figure 11.3: Contracts for Client–Server Communication

Figure 11.3 defines `client/c` and `server/c` as trace contracts over a `channel*/c` contract. The `channel*/c` contract constrains inputs and outputs of a channel. For instance, `(channel*/c even? odd?)` stipulates that the channel must carry even numbers outbound and odd numbers inbound. The `client/c` trace contract ensures that when `client-start` sends a number `k` across its channel, the value `(list 'put k)` is added to trace `n` and checked with the state machine.

This example demonstrates how compositionality separates concerns. The `channel*/c` contract is responsible for monitoring values flowing across a channel, while `trace/c` is responsible for enforcing a temporal sequence on those values.

11.2 BACKGROUND ON TWO-PHASE COMMIT

Two-phase commit (2PC) is a fundamental protocol in distributed systems. It is designed to address the distributed commit problem: given a pending transaction and a process group, ensure that either all the processes commit to the transaction or that none of them do. 2PC is mediated by a distinguished *leader* and assumes that the rest of the group (dubbed the *followers*) is static. As its name suggests, the basic 2PC protocol proceeds in two phases:

VOTING PHASE The leader sends the same QUERY to all processes in the group. Upon receiving a QUERY, followers send back a VOTE, either YES or NO.

COMPLETION PHASE Upon receving VOTES from every follower, the leader sends the same DECISION to all processes in the group. If the leader received any NO vote, then the DECISION is ABORT. Otherwise, the DECISION is COMMIT. Upon receiving a DECISION, followers send back an acknowledgement ACK.

Consider a naive monitor design that requires a programmer to (1) implement *oracle processes* that are assumed to be correct and (2) monitor that the output of the real processes exactly matches that of the oracles. Such a methodology imposes far too many restrictions on implementations. It is an *overspecification*. There are perfectly legal extensions of 2PC that do not exactly match one another, nor do they exactly match the basic protocol. For instance, if the network is unreliable, it would be natural for processes to send redundant messages to compensate for drops. If the oracles implement basic 2PC, then redundant messages would be rejected as a breach of protocol. Even if the oracle monitors implement a fault-tolerant variant of 2PC (e.g., with redundant messages), it is still too *prescriptive* because implementations would need to use the exact same array of fault-tolerance measures (e.g., the same amount of redundancy). Furthermore, the monitors should not require that the channels they monitor be used only for 2PC, lest they become impractical to install into real environments.

Supposing one proceeded with the naive-oracle approach anyway, the question arises about what it means for processes to "match" the monitors. A common technique for proving that two programs (e.g., the specification and the implementation) "match" is to develop a *simulation argument* that relates sequences of observable behaviors between the two programs. This method is ubiquitous in the study of compiler correctness [94] and cryptography [22]. Most commonly, traces are proven to be either equivalent or inclusive. Enforcing trace inclusion dynamically ensures that the trace, at each point in time, is always among the set of legal traces for a protocol invocation.

Another question to consider is the matter of *liveness*. Usually specifications abstract over implementation details and, as such, tend to ben-

efit from richer assumptions than their real-world counterparts. For instance, viewing the basic 2PC protocol in a world without failures and perfect networking, then the protocol is live—it can always make progress. By relaxing those assumptions, as required by the real world, it is no longer live. Although there are tools that can verify liveness properties, dynamic enforcement is concerned solely with safety properties. As such, the specification must phrase all constraints as safety properties while remaining consistent with the protocol.

To recap, the design constraints are as follows: (1) allow some flexibility, e.g., with respect to redundant and irrelevant messages; and (2) stay within safety properties.

11.3 IMPLEMENTING TWO-PHASE COMMIT

This section describes, at a high level, contracts that enforce the 2PC protocol. The implementation composes channel and trace contracts, where the trace predicate uses a domain-specific notation based on state machines. Contracts can enforce the protocol both at the leader, which sees all messages, and at followers, which see only a subset of all messages. Since 2PC is quite a sophisticated protocol, this case study demonstrates that these techniques can scale to larger examples.

$$
\begin{aligned}
\mathsf{Msg} \ni m &::= q \mid o \mid d \mid a \\
\mathsf{Query} \ni q &::= \texttt{(query } r\ v\texttt{)} \\
\mathsf{Vote} \ni o &::= \texttt{(vote } r \texttt{ yes)} \mid \texttt{(vote } r \texttt{ no)} \\
\mathsf{Dec} \ni d &::= \texttt{(decision } r \texttt{ commit)} \mid \texttt{(decision } r \texttt{ abort)} \\
\mathsf{Ack} \ni a &::= \texttt{(ack } r \texttt{ commit)} \mid \texttt{(ack } r \texttt{ abort)} \\
\mathsf{Round} \ni r &\coloneqq \mathbb{N} \\
\mathsf{Val} \ni v &
\end{aligned}
$$

Figure 11.4: Grammar of Messages

To monitor 2PC, it is first necessary to define a notion of safety that, once specified, can be enforced using a domain-specific notation. Figure 11.4 defines the grammar of messages that may pass between processes via channels.

Definition (Safety). A trace is *safe* if and only if its relevant messages satisfy Gapless Ordering, Cooperation, Unanimity, and Consistency.

Safety relies on a notion of relevance and four subordinate properties. These definitions will be given in turn.

Definition (Relevancy). For an instance of 2PC with round identifier r and process group g, a message is *relevant* if and only if it is a 2PC message with round r that is from or to a process in g.

Under this definition of relevancy, monitors filter out "noise" when checking traces. Notice that redundant messages *are* relevant and are considered when checking traces.

Definition (Gapless Ordering). Consider a partial order on messages based on the reflexive-transitive closure of the following relationships between message types: QUERY $\leqslant$ VOTE $\leqslant$ DECISION $\leqslant$ ACK. For any message types t, t' where $t' \leqslant t$, if a fresh message m of type t appears in the sequence, then a fresh message m' of type t' appears before m. A fresh message is one that has not appeared before in the sequence.

Definition (Cooperation). The leader sends messages to all followers and waits on messages from all followers. Followers send messages to the leader and wait on messages from the leader, meaning (1) a follower's VOTE is preceded by a QUERY from the leader, (2) a leader's DECISION is preceded by a VOTE from each follower, and (3) a follower's ACK is preceded by a DECISION from the leader. Furthermore, the leader does not consider the protocol complete until it has received an ACK from each follower.[1]

Definition (Unanimity). The final decision is COMMIT if and only if all VOTES are YES.

Definition (Consistency). Messages of the same type, and from the same process, must have equal values.

Monitors enforcing 2PC for leader and follower processes, based on these formal definitions, were implemented using trace contracts. These contracts look similar to the example from Figure 11.3. except the trace predicate itself is far more complicated. Messages are tagged with a direction (send or receive) and a process ID (source or destination). For instance, the message `(put 0 (query r val))` in a trace at the leader indicates that the leader is sending a `query` to follower 0. The message would appear on 0's trace as `(get leader (query r val))`.

Violating safety is a fatal error, immediately stopping the offending process. In principle, a process might *receive* a message that violates safety. However, because outbound messages are monitored and violations are fatal, the system *fails early*, and a receiving process never has the chance to see an unsafe message.

The following propositions are not used to define or enforce safety but can be helpful to keep in mind when examining traces. In particular, given a trace, they allow one to reason about the safety of minor variations of that trace.

1 Note that this definition does not say that requests are followed by responses because that stipulation is a liveness property.

Proposition (Prefix Closure). Any prefix of a safe trace is safe.

Proposition (Safety Preserved Under Projection). Consider the projection from a trace at the leader to a trace at one of its followers, defined by mapping all messages to (from) that follower to messages from (to) the leader. If a leader trace is safe, then so is its projection at the follower.

The remainder of this section demonstrates how the trace contract determines conformance with the 2PC protocol via a series of example traces, some of which satisfy the protocol, and others that do not. Each of these example traces is assumed to be observed at the leader.

```
1 (put 0 (query r 42))
2 (get 0 (vote r yes))
3 (put 0 (decision r commit))
4 (get 0 (ack r commit))
```

Figure 11.5: Basic 2PC (Two Processes)

For the 2PC instance r involving the process group {leader, 0} the trace in Figure 11.5 is safe. This trace exhibits the simplest case of "basic" 2PC, with one follower that VOTES YES (Line 2), allowing the leader to COMMIT (Line 3). The trace consisting of solely Lines 2–4 is unsafe since it violates Gapless Ordering.

```
1 (put 0 (query r 42))
2 (put 1 (query r 42))
3 (get 1 (vote r yes))
4 (get 0 (vote r no))
5 (put 1 (decision r abort))
6 (put 0 (decision r abort))
7 (get 0 (ack r abort))
8 (get 1 (ack r abort))
```

Figure 11.6: Basic 2PC (Three Processes)

For the 2PC instance r involving the process group {leader, 0, 1}, the trace in Figure 11.6 is safe. Building on the previous example, another follower is added that VOTES NO (Line 4). As such, the leader decides to ABORT (Lines 5–6). Once again, this trace adheres strictly to "basic" 2PC. Safety is preserved if Lines 3 and 4 are swapped, or Lines 5 and 6, or Lines 7 and 8.

For the 2PC instance r involving the process group {leader, 0}, the trace in Figure 11.7 is safe. In this trace, the 2PC protocol has not yet finished; the leader is still in the completion phase, since it has not yet received all ACKS. Line 2 is an example of a *redundant messsage*: there is

```
1 (put 0 (query r 42))
2 (put 0 (query r 42))
3 (put 0 (decision r' commit))
4 (put 0 foo)
5 (get 0 (vote r yes))
6 (put 0 (decision r commit))
7 (get 0 (vote r yes))
```

Figure 11.7: 2PC in Completion Phase

already a QUERY to 0 in the trace (Line 1), but the two are consistent, so it is safe. Lines 3–4 are examples of *irrelevant messages*: the DECISION on Line 3 concerns a different instance of 2PC (R'), and the FOO on Line 4 is not a 2PC message at all. The VOTE on Line 7 is an example of a *stale* message. After receiving the VOTE on Line 5, the leader transitions to the completion phase, at which point all VOTES are stale. Informally, a redundant message is stale when it can no longer affect the outcome of the protocol (e.g., a follower receives another query after it has already cast a VOTE). More formally, a relevant message is stale if and only if (1) it is a QUERY sent or received during the completion phase, or (2) it is a VOTE sent or received during the completion phase, or (3) it is sent or received after all ACKS have been received by the leader.

```
1 (put 0 (query r 42))
2 (put 1 (query r 42))
3 (get 0 (vote r yes))
4 (put 0 (decision r commit))
```

Figure 11.8: Cooperation Violation

For the 2PC instance r involving the process group {leader, 0, 1}, the trace in Figure 11.8 is unsafe. It is unsafe because Line 4 violates Cooperation; the leader sends a DECISION without receiving VOTES from all followers (follower 1 is missing). Because follower 1 did not vote, the system may be left in an inconsistent state if follower 1 is unable to successfully commit.

```
1 (put 0 (query r 42))
2 (put 1 (query r 43))
```

Figure 11.9: Consistency Violation

For the 2PC instance r involving the process group {leader, 0, 1}, the trace in Figure 11.9 is unsafe. It is unsafe because Line 2 violates Consistency: there is another QUERY (Line 1) with a different value.

```
1 (put 0 (query r 42))
2 (put 1 (query r 42))
3 (get 1 (vote r yes))
4 (get 0 (vote r no))
5 (put 1 (decision r commit))
```

Figure 11.10: Unanimity Violation

For the 2PC instance r involving the process group {leader, 0, 1}, the trace in Figure 11.10 is unsafe. Notice how Line 5 violates Unanimity: there is a NO VOTE (Line 4) but the leader decides to COMMIT (Line 5).

Part IV

EMPIRICAL

Given the implementations and case studies of effectful contracts seen thus far, it is worth considering how well they might work in the real world. Two concerns, regarding the run-time performance and learning curve of effectful contracts, are addressed in the upcoming chapters.

12 PERFORMANCE ASSESSMENT

Imposing contracts on components impacts the performance of the overall system; effectful contracts increase this cost. Hence, a key question is how much a system is affected in realistic scenarios. This chapter presents an evaluation of the performance cost of trace contracts and attribute contracts.[1] For performance, the relevant question is what kind of *fixed cost* the mechanism imposes on programs, not the *variable cost* of the programmer-defined predicates. A performance evaluation cannot answer questions concerning the variable cost of trace predicates. Effectful contracts, like ordinary contracts, are property agnostic. Thus, the variable cost of run-time enforcement depends largely on the property being checked. This variable cost is solely under the purview of the programmer and not the contract system. State initialization, mutation, and calls to predicates are all included in the fixed cost. Benchmarks (Section 12.1) measuring the fixed-cost performance of effectful contracts show a tolerable performance overhead (Section 12.2).

12.1 BENCHMARKS

Nine benchmarks represent real-world uses of Racket that offer opportunities for adding effectful contracts. The MEMORY benchmark turns the `current-memory-use` example from Chapter 5 into a pathological stress test. The FUTURE benchmark consists of a large existing Racket library equipped with attribute contracts, plus an application that stresses the functionality. Four of the benchmark programs (DUNGEON, JPEG, LNM, TETRIS) are variants on programs from the standard gradual typing benchmark suite [71]. Three benchmarks (DATAFLOW, FISH, TICKET) are programs developed for university courses. All benchmarks have been modified so that they do not measure I/O operations.

DATAFLOW computes a constant propagation analysis for a simple imperative language. A trace contract, similar to one from Chapter 8, checks the monotonicity of a transfer function during fixed-point iteration.

DUNGEON randomly generates a maze. A trace contract on the random-number generator ensures that it does not exhaust a fixed pool of random numbers. The contract must keep track of how many

1 Parameter contracts are omitted because an existing performance evaluation [57] rigorously measures the cost of continuation marks—the run-time mechanism underlying parameter contracts.

times the `random` function is called, so its accumulator is just a natural number and the check is cheap.

FISH runs a "That's My Fish" board-game tournament. There are two trace contracts: a referee contract and a player contract.

The referee contract ensures that the referee calls back players in the specified order unless the game state does not permit the player to take a turn. The contract is a promise made by the referee to all players. To enforce this promise, the contract is placed on the referee's list of player objects. A collector receives a new value every time the referee calls the `take-turn` method on any player. The trace contract then checks for conformance with the promised callback order on the *players*, including skipping over players that are momentarily prohibited from taking a turn.

The player contract enforces a sequence property on its method calls. In other words, the player components ensure that their individual *methods* are called in a specific order. This contract is similar to the value-dependent temporal protocol from Figure 10.4. It is independent of, and orthogonal to, the referee contract.

FUTURE visualizes the performance of a program using futures. Futures are a run-time mechanism for incrementally adding parallelism to programs [137]. The future visualizer [136] uses a version of Racket's drawing library that has been equipped with attribute contracts to enforce multi-call properties. A full list of these properties is enumerated in Chapter B. Some properties were originally monitored by the drawing library using ad-hoc checks, but others were not checked at all.

JPEG parses a JPEG input stream and writes it to an output stream. A trace contract guarantees that operations on the output stream occur in the correct order. Like the board-game example in Figure 10.4, it checks every stream-related function call against a finite automaton. Formulating the trace contract involves creating several contracts that share the same accumulator (the state of the finite automaton) using `#:global`.

LNM draws plots of performance measurements from gradually typed programs. Like FUTURE, this benchmark uses the variant of Racket's drawing library equipped with attribute contracts.

MEMORY reports memory consumption including garbage-collected blocks. The trace contract ensures that `current-memory-use` returns increasing numbers over time; it is called 10,000 times in a tight loop, the results of which are graphed on a line chart using Racket's plot [142] library.

TETRIS simulates and displays a recording of a Tetris playthrough. This benchmark also uses the variant of Racket's drawing library equipped with attribute contracts.

TICKET runs a "Ticket to Ride" board-game tournament. Like FISH, TICKET has both a referee and a player contract. The referee contract enforces a promise that the referee calls back players in the specified order. This trace contract is significantly simpler than the one for FISH, because every player can execute an action in every game state. The player-side trace contract enforces the correct sequence of method calls. The board-game example presented in Figure 10.4 is a simplified version of this contract.

12.2 RESULTS

I performed the experiments on a dedicated Linux machine with an Intel Xeon E3 processor running at 3.10 GHz with 32 GB of RAM and with Racket 8.6 CS. Each benchmark configuration was repeated 100 times with a maximum timeout of two minutes.

Benchmark	SLOC	Protects	Checks
DATAFLOW	502	1	584
DUNGEON	589	0	538,000
FISH	1,452	2,698	63,175
FUTURE	1,721	16,360	234,444
JPEG	1,481	0	54,556
LNM	564	168	3,248
MEMORY	59	0	10,000
TETRIS	334	6,807	125,570
TICKET	1,427	384	15,794

Table 12.1: Basic Properties

Table 12.1 first lists the number of essential lines of source code (SLOC) for each program, including the contract and its auxiliary functions. None of the trace contracts require much code. FISH and TICKET contain the most complex ones but the others are relatively simple. Even the most complex trace contracts are concise. Since predicates are ordinary code, they can make use of existing data structure libraries and those libraries serve as workhorses in many cases.

The Protects column reports the number of times a trace contract monitors a new value during the steady state of a program's execution. In other words, it is the number of times a trace contract evaluates its body-contract expression. Each time, there is some overhead due to

allocating references for accumulators and creating collector contracts. Some benchmarks have a zero entry because all the contracts are initialized before the main body of the program begins, for example, when dependencies are being loaded. The Checks column states the number of times each trace or attribute predicate is checked.

Benchmark	Disabled	Enabled	Predicate	Overhead
DATAFLOW	83 ± 3	87 ± 2	274 ± 3	5%
DUNGEON	2441 ± 38	2715 ± 46	2713 ± 33	11%
FISH	7780 ± 70	8340 ± 82	8366 ± 80	7%
FUTURE	6075 ± 54	7083 ± 83	7502 ± 86	17%
JPEG	276 ± 5	303 ± 6	316 ± 6	10%
LNM	522 ± 8	532 ± 9	534 ± 9	2%
MEMORY	141 ± 4	164 ± 4	164 ± 4	16%
TETRIS	3040 ± 24	3566 ± 36	3927 ± 43	17%
TICKET	13062 ± 149	13186 ± 170	13199 ± 182	1%

Table 12.2: Performance Measurements

Table 12.2 shows the timing measurements. Benchmarks were executed at two levels: Disabled where trace contracts are disabled, and Enabled where they are enabled. These measurements are the mean number of milliseconds it takes to run each benchmark, averaged over 100 samples, along with the standard deviation. As mentioned, the performance evaluation is primarily concerned with the fixed cost of effectful contracts. For Enabled, each predicate is replaced with a trivial one that always returns `#t`. The Predicate column lists the performance numbers where effectful contracts are enabled and the predicate actually checks the desired property. In other words, it also measures the variable cost of the contracts. Such predicates are straightforward implementations and are not heavily optimized. Finally, the Overhead column shows the percent overhead of Enabled compared to Disabled.

The overhead of the contract mechanism is relatively low, somewhere between 1% and 17%. As is, some benchmarks basically simulate worst-case scenarios. For example, MEMORY just calls a simple function in a tight loop, so contract checking takes up a large portion of total execution time. By contrast, benchmarks that are closer to real-world programs, such as TICKET, incur a low overhead. Thus, the evidence suggests that the contract mechanisms do not exhibit pathological performance. These measurements do not exercise an industrial-strength implementation of effectful contracts but rather a direct translation of the design. With some additional performance engineering, it is likely to perform even better.

13 TEACHING SOFTWARE SPECIFICATION

Effectful contracts, even in their most principled form, are worthless if developers cannot put them into practice. A broad-spectrum evaluation of effectful contracts must consider whether humans can make use of them. To that end, this chapter describes a redesign of Northeastern's Logic and Computation course focusing on the art of *correct* software design using contracts, including effectful ones.

Software correctness is well-recognized as essential in many fields. Showing that software is correct can be split into two activities: *specification* and *verification*. Specification formally characterizes (some aspect of) the intended behavior of software, whereas verification shows, using various methods, that software conforms to its specification. While these two activities may seem equally important, in practice, time spent articulating a specification is often dwarfed by time spent working on verification. This imbalance can also be present in courses about software correctness. If specifications and theorems are written by instructors, then students get little to no practice with the first, crucial step of creating correct software. If the theorems are relatively simple, then the complexity of the programs students reason about might never exceed single-digit line counts, leading them to wonder how this experience relates to real software engineering.

Rather than thinking of specifications as instructor-written problems that students solve by carrying out proofs, specification can be made the primary activity, avoiding most work on proofs. Indeed, while writing correct software requires both specification and verification, arguably it is specification that is more challenging and critical. Without a specification, any proof, successful or not, is worthless. Even projects that are never subject to formal verification can benefit from clear descriptions of functional correctness, security properties, or other behavioral invariants. For this reason, specification should be a core part of any undergraduate curriculum, and it should come as early as in the first year.

The course redesign is based on two intertwined components. First, contracts and property-based testing are used to capture sophisticated (type) invariants and logical properties. Contracts can express dependent-type invariants, introducing students to the power of type-based reasoning without any background on the complex topic of dependent type systems. Property-based tests (PBT) [28] also provide a relatively low-barrier means of expressing complex invariants about code. One way to understand this approach to PBT is that students

practice writing theorems, but proofs are deferred to the approximation produced by a PBT engine.

Second, students use a pedagogic functional language built especially for the course, dubbed the Logical Student Language (LSL), that includes not only constructs for contracts and property-based testing, but also linguistic support for non-trivial programs. For example, one assignment has students implement a distributed snapshot algorithm and express snapshot consistency as property-based tests, relying on message-passing concurrency and an accompanying visualizer. Linguistic support allows such problems to be tractable, even for students in their second semester of programming, by eliminating the incidental complexity that typically arises without full control over the language.

As part of this new course, students were introduced to a simplified version of trace contracts and used them to enforce temporal properties in their programs. For instance, one assignment had students build an idealized memory allocator and write down memory-correctness properties as predicates over traces of memory-relevant events. Students had no issue completing this assignment and some reported in a post-course interview that they found the memory-correctness project particularly helpful. First-year students were able to write non-trivial trace contracts, suggesting that more experienced developers should be able use trace contracts as well.

13.1 BACKGROUND

For the last two decades, Northeastern's Logic and Computation course has aimed to connect logical reasoning with systematic program design. Historically, the course functioned as an introduction to *both* specification and verification, first using the ACL2s theorem prover [37], and more recently using the Lean theorem prover [35]. The only prerequisites for the course are a single semester of programming and discrete mathematics; it was originally envisioned, and is still often realized, as a second-semester freshman course.

Unfortunately, a phenomenon familiar to those teaching formal methods has stymied this effort: time spent on verification far surpasses time spent on specification. In an educational context, this results in the class being either too hard for students to learn, or structured such that the majority of the semester is dedicated to proofs of relatively uninteresting theorems.

Originally, the course was taught using the custom language Dracula, a frontend to the ACL2 theorem prover [45, 119]. This tool was used for Logic and Computation as well as an upper-level course on software engineering at another institution. While the goal was theorem proving, Dracula also included a mechanism for property-based testing (called DoubleCheck). In a paper summarizing the first few years of this experiment, Page et al. [119] report that, across two uni-

versities, only ten to twenty percent of students ended up being able to use ACL2 via Dracula effectively. Interestingly, the authors also report that DoubleCheck, their property-based testing framework, "helps students with one of the trickiest task[s] of using logic in programming, namely, the transition from ideas to formal statements" [119]. While the authors had plans, never fully realized, to improve Dracula to address some issues, the latest research [82] indicates that beginners still face challenges using interactive theorem provers.

More recently, when the course has been taught using Lean, students learned to write proofs about natural numbers and lists, but little more. Attempting to shift this balance seems impossible as significant practice with basic theorems over simple inductive types is necessary to build the skills needed to complete large proofs.

It is tempting to think automation might help, but after the experiments with Dracula, the course was taught for many years using ACL2s, which has world-class automation support. This does not seem to help. In particular, knowing how to drive sophisticated automatic provers, especially intermediate lemma identification, may be a more advanced skill than constructing proofs manually—even if the resulting proofs are much smaller.

While the intent of the class was always to introduce students to formal reasoning *about software*, perversely, students using Lean largely reported that the class had no bearing on software development whatsoever (Section 13.5). A frank assessment of this decades-long experiment is that at best students are introduced to the idea of mechanized proof, unconnected to the software work they do in other courses or on internship, and at worst they learn a few details about an esoteric tool they will never use again.

13.2 REDESIGN OVERVIEW

REVIEW OF PROPOSITIONAL LOGIC (WEEK 1) The course begins with a review of propositional logic, but turned into computation: propositional formulas become boolean-valued functions, and truth tables become exhaustive sets of unit tests. This exercise introduces students to the idea that logical reasoning can be viewed as computation. Additionally, students see how translating formulas into code, and truth tables into unit tests, can uncover mistakes just by running the tests.

ATOMIC DATA (WEEKS 2–3) The second unit explores the idea of specifications as code, but in the setting of atomic data types such as numbers and strings. Students translate these informal descriptions of how functions should work into boolean expressions that relate inputs and outputs.

INCREASING COMPLEXITY (WEEKS 4–8) The third unit of the class explores both data and properties of increasing sophistication. Students deal with recursive data, higher-order functions, and data abstraction via contracts. Additionally, this unit transitions the homework assignments from introductory practice to somewhat complex projects (Section 13.4).

ADVANCED TOPICS (WEEKS 9–12) The latter part of the semester covers mutation and aliasing along with an assignment building an idealized memory allocator. Pairing the lecture content, which presents aliasing puzzles, with a large project where aliasing is a central theme allows students to explore the concepts deeply even in a relatively short period of time. Enforcing memory invariants serves to enrich the content and helps with debugging.

This unit of the course also introduces concurrency with a simple message-passing implementation and accompanying visual debugger, all built into LSL. Students first implement a warm-up homework using message passing and, once graded, implement the Chandy–Lamport concurrent snapshotting algorithm. As with the memory allocator, the project is small enough to be achievable, but complex enough that students realize that correctly implementing it is much easier if they write down invariants and ensure they are preserved as the code runs.

SPECIAL TOPICS (WEEKS 13–14) The last two weeks are left for special topics. Typically, the special topic is about interactive theorem proving, though different instructors may choose different topics. The structure of Logic and Computation, and the placement of the last exam within the semester, means that this content is not subject to any assessment; it serves as bonus material for students who are interested in further enrichment. The last week is a nice place to introduce tools like Lean, as students who are interested can learn about them but the central learning outcomes of the course are not affected.

13.3 THE LOGICAL STUDENT LANGUAGE

Creating a new programming language, especially a pedagogic one, is a substantial undertaking. The effort is worthwhile if it reduces friction, as LSL does in two ways. First, LSL is a superset of the pedagogic language used in Fundamentals 1, Northeastern's first-semester course on program design (prior to Fall 2025). Students enter the course with a full semester of experience in its syntax and semantics, allowing them to immediately acquire new skills without rehashing functional-programming basics. Second, LSL is integrated with the DrRacket [54] pedagogic IDE—also the same one used in their first course. Thus, the workflow students are already familiar with is completely unchanged.

LSL also augments DrRacket to support certain learning objectives. For example, it integrates with Tyche [68], an interface for inspecting random generators, to help discover weak property-based tests.

LSL IN A NUTSHELL LSL is an extension of the Intermediate Student Language (ISL) from How to Design Programs (HtDP) [51]. ISL is a simple pedagogic functional programming language. Traditional introductions to programming teach language features (e.g., variables, assignment, conditionals, loops) and leave program design as an implicit skill to be acquired through practice. HtDP treats the craft of constructing correct and maintainable software explicitly. The language becomes merely a vehicle through which these universal skills are taught. As such, HtDP provides a sequence of increasingly sophisticated pedagogic languages, including ISL, to maximize learning and minimize distractions [49]. For example, error messages are tailored to freshmen students by employing familiar terminology instead of technical jargon [97].

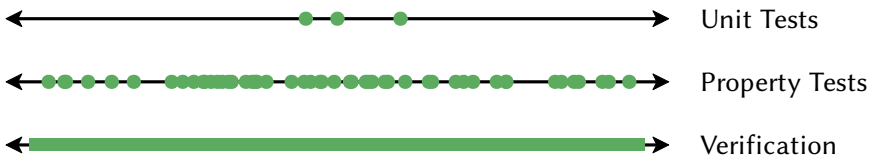

Figure 13.1: Gradual Specification

LSL adds contracts and property-based testing to ISL, enabling the *gradual* approach to specification shown in Figure 13.1. HtDP focuses on unit tests, or what amounts to pointwise specifications. Such tests do not cover much of the input space but are a necessary first step. Property-based tests randomly generate inputs, enabling coverage of a greater region of the input space. Getting the properties and generators right can take some work but yields tangible benefits.

The choice of contracts over types is due to three unique features of contracts. First, contracts consist of ordinary code and as such allow expressive specifications without the cognitive overhead of a sophisticated type system. In class, students regularly write specifications that are dependent, temporal, and intensional. Designing a type system to accommodate all of these properties, let alone one that is appropriate for beginners, seems infeasible. Contracts allow students to freely explore the space of possible specifications, building on their existing programming intuitions instead of putting them aside.

Second, contracts are a *precise* mechanism for enforcing properties. To remain decidable, type systems, or any static technique, must approximate program behavior. Since contracts monitor programs at run time, a violation guarantees that there is a true inconsistency between the specification and a program execution.

Third, and relatedly, contracts supply a concrete counterexample when a program violates its specification. Each error message highlights the violated contract and comes with a *witness* to the violation. Students can use this witness to debug their program or specification. In contrast, type systems supply error messages in terms of failure to satisfy a particular syntactic discipline—easy for experienced programmers to resolve but less so for novices.

BASIC CONTRACTS IN LSL In HtDP, students follow the Design Recipe, a checklist that includes a step for writing a comment above each function definition with its signature.

Here is an example function in the style of HtDP:

```
;; only-non-negative: (List Integer) → (List Natural)
(define (only-non-negative lon)
  (filter (λ (n) (or (zero? n) (positive? n))) lon))
```

Comments are inert, but their simplicity and flexibility are useful because students can write informal signatures that may be challenging to express formally (e.g., signatures with refinements or untagged unions). These specifications are perfectly reasonable, even if they thwart conventional type systems. Eventually, it becomes valuable to consider how to make such statements formal. The same example can be realized in LSL with the following contract:

```
(: only-non-negative (-> (List Integer) (List Natural)))
(define (only-non-negative lon)
  (filter (λ (n) (or (zero? n) (positive? n))) lon))
```

The `:` annotation associates a contract with the given function name, monitoring the specification at run time. These annotations mirror the signature conventions in HtDP, providing a smooth path from informal prose to formal specifications.[1] Violations of the contract raise a run-time exception.

Without a program or test cases to exercise the functionality of `only-non-negative`, the contract serves no purpose other than documentation. One way to test if the function satisfies the contract is to use property-based testing. Every contract built into LSL, including higher-order ones, comes with a generator and shrinker [88]. Thus, students are able to run property-based tests for their specifications with a single line: `(check-contract only-non-negative)`.

In this example, the `check-contract` form generates random lists of integers based on the domain contract for the given function, calls the function repeatedly with these arguments, and ensures that the codomain contract holds. Close integration between the contract system and random-generation capabilities makes PBT convenient to use for first-year students.

1 Crestani and Sperber [32] explore similar ideas, but LSL is a much larger departure from the teaching languages than the extensions described in their work.

If a property-based test encounters an inconsistency, it reports a concrete, shrunk counterexample as an error message:

```
discovered a counterexample
  counterexample: (only-non-negative (list -1))
  error:
    only-non-negative: contract violation
      expected: Natural
      given: -1
      blaming: anonymous-module (as server)
```

Following DrRacket's printing conventions, the counterexample can be copy-and-pasted into the REPL to reproduce the error. The error message also comes with *blame information* pointing to the party that violated the contract [56]. The `(as server)` parenthetical indicates that the function itself violated the contract. If a caller of the function provided an invalid argument instead, then the error would display the `(as client)` parenthetical.

Additionally, DrRacket highlights the violated contract in the program's source. Tight cooperation between LSL and the DrRacket IDE permits affordances that are especially useful for first-year students. Such affordances are one advantage specialized pedagogic languages have over ordinary libraries.

```
(define (longer-than-in? los)
  (λ (x)
    (andmap (λ (s) (>= (string-length x) (string-length s)))
            los)))

(: longest-string
   (Function (arguments [los (NEList String)])
             (result (AllOf String (longer-than-in? los)))))
(define (longest-string los) — implementation — )
```

Figure 13.2: Dependent Function Contract in LSL

Strengthening the signature of a function is one way to increase confidence in the correctness of code. Figure 13.2 displays a dependent function contract that ensures the `longest-string` function returns a string whose length is greater than or equal to all others in the input list. This dependent function contract has two pieces: one constraining the argument and another constraining the return value. The argument is expected to be a non-empty list of strings and is bound to the variable `los` for use in the result contract. The result is expected to be a string that satisfies `(longer-than-in? los)`, ensuring that the string is longer than any other in `los`. Here, `AllOf` represents the conjunction of several contracts, just as `and/c` does in Racket.

A programmer can also define custom contracts, the most basic of which is an *immediate* (or *flat*) contract [56]. An immediate is a first-order check: a predicate for determining whether values satisfy the contract. Immediates stand in contrast to higher-order contracts, like function contracts, that must interpose on all future interactions with a value. LSL comes with many built-in immediates, and students can define their own.

```
(define-contract Even
  (Immediate
    (check (λ (x) (and (integer? x) (even? x))))
    (generate
      (λ (fuel)
        (* 2 (contract-generate Integer fuel))))
    (shrink
      (λ (x)
        (let ([y (/ x 2)])
          (if (even? y) y (sub1 y)))))))
```

Figure 13.3: Contract for Even Numbers

Figure 13.3 defines a contract for even numbers that also includes a generator and shrinker for property-based testing. The `check` clause contains the mandatory predicate. The `generate` clause expects a function that takes `fuel`, a natural number corresponding to how hard the generator should try to construct a value, and returns a value that *must* satisfy the contract. This particular generator delegates to the generator for `Integer`, via `contract-generate`, to return an even number. The `shrink` clause takes a function that returns, if possible, a smaller value than the one given.

Contracts can also handle the kinds of data definitions seen in languages with algebraic data types. Here is the definition of a binary tree parameterized by the type of element:

```
(define-struct leaf [value])
(define-struct node [left right])
(define-contract (Tree X)
  (OneOf (Struct leaf [X])
         (Struct node [(Tree X) (Tree X)])))
```

Property-based testing propagates automatically: if `X` supports generation, so does `(Tree X)`.

ENFORCING DATA ABSTRACTION Data abstraction is a technique whereby implementation details of a particular data type are hidden from some pieces of code but not others. Specifications about data abstraction fit well in Logic and Computation. First, data abstraction is easily motivated as a desirable software-engineering principle. Robust

system design relies on information hiding to decrease the coupling between components [121]. Second, most programming languages offer data-abstraction mechanisms—whether through access modifiers, existential types, or name mangling. Students should be familiar with the concept and understand why it matters. Finally, data abstraction provides a nice on-ramp to properties that are not just about correctness. Many important properties fall into this category, from security properties to resource constraints.

In type systems, data abstraction is enforced via universal and existential types [102]. The key difference between the two rests in *which* pieces of code must treat data abstractly. With universal types, implementation details are hidden from the server component. With existential types, implementation details are hidden from the client component. Dynamically enforced analogues, universal and existential contracts [72, 98], seal and unseal values to ensure programs treat certain values abstractly. Here is an example of an existential contract:

```
(define-struct counter-pkg (make incr get))

(define-contract Counter
  (Exists (T)
    (Struct counter-pkg [(-> T) (-> T T) (-> T Natural)])))
```

The `Counter` contract specifies the signatures of three functions contained in a `counter-pkg` structure, where `T` refers to an abstract type. There can be several packages implementing `Counter` that keep the representation of `T` hidden from clients.

Existential contracts are a *mechanism* and not the only one that may be used to achieve data abstraction. Most first-year students taking Logic and Computation concurrently take a course on object-oriented programming in Java. After discussing data abstraction via existential contracts, students implement pure objects in LSL using structs with functions (i.e., methods) as fields. In such an encoding, closures are a data-hiding mechanism [105]. Seeing two different forms of data abstraction allows for a comparison of their respective tradeoffs [31].

MUTATION Eventually, students must be exposed to effectful programs, and in particular, programs with mutable state. Students should have a deep understanding of mutable state, be made aware of how mutation complicates reasoning about programs, and have the tools needed to manage that additional complexity.

As in most programming languages, LSL supports both variable and value mutation. Variables are mutated with the `set!` form:

```
(: n Integer)
(define n 1)
(set! n 2)
```

When a variable is mutated, the contract associated with the variable is checked again. Value mutation is available for structs but only when they are explicitly declared as mutable:

```
(define-mutable-struct posn [x y])
(: p (Struct posn [Integer Integer]))
(define p (posn 1 1))
(set-posn-x! p 2)
```

The contract on a struct's field is checked when that field is mutated. Making variable mutation and value mutation syntactically distinct helps make the concepts distinct in students' minds.

TEMPORAL PROPERTIES Most contracts fail to account for an important aspect introduced by mutation. Equality of immutable data is best characterized by structural equality, but equality of mutable data is best characterized by pointer equality [9]. Consequently, mutable values have a discernible identity even if their fields change. Fully reasoning about mutation necessitates reasoning about how mutable values change over time. From low-level libraries such as the Unix file API in C, to high-level libraries such as the networking abstractions in Java, temporal constraints crop up all the time.

LSL can readily express temporal properties. Consider a `fresh` function that is intended to return a different natural number every time it is called. A simplified version of a trace contract can enforce freshness:

```
(: ids unique-list?)
(define ids empty)

(: fresh (-> (AllOf Natural (Record ids))))
(define (fresh) — elided — )
```

The `(Record ids)` contract mutates the variable `ids` by appending the returned value from `fresh` onto the current value of `ids` each time the function is called. In other words, `ids` contains the *trace* of return values from `fresh`. When a variable is mutated, here `ids` via `Record`, its associated contract is checked again. Thus, `unique-list?` is checked on a sequence containing every value returned from `fresh`. This check ensures that `fresh` never yields a number that it previously returned.

Students are also introduced to state machines and checking temporal properties using them. The `Record` contract supports accumulating functions, just as ordinary trace contracts do, to improve the efficiency of checking properties. Shown in Figure 13.4 are a pair of contracts that ensure the `g` and `h` functions are interleaved. There is a global accumulator, called `state`, that is updated according to the `g-call` and `h-call` accumulating functions. The states are `'?` (any function can be called next), `'g` (`g` must be called next), and `'h` (`h` must be called next). If the accumulating function returns `#f`, then the contract on `state` fails.

```
(: state (OneOf (Constant '?) (Constant 'g) (Constant 'h)))
(define state '?)

(define (g-call s _)
  (cond [(equal? s '?) 'h]
        [(equal? s 'g) 'h]
        [(equal? s 'h) #f]))

(define (h-call s _)
  (cond [(equal? s '?) 'g]
        [(equal? s 'g) #f]
        [(equal? s 'h) 'g]))

(: g (-> (AllOf Integer (Record g-call state)) Integer))
(define (g x) — body of g — )

(: h (-> (AllOf Integer (Record h-call state)) Integer))
(define (h y) — body of h — )
```

Figure 13.4: Trace Contract with State Machine

13.4 HOMEWORK ASSIGNMENTS

Table 13.1 shows the complete list of assignments in the course. Assignments come from a range of domains: games, security, low-level programming, and distributed systems.

GENERATING MAZES Students are asked to write and property-test transformations that convert between two maze representations and develop generators that randomly construct solvable mazes.

This assignment has two goals. First, round-trip properties, especially between non-bijective data representations, are common and useful in real-world applications of PBT [67]. Second, naive random generation is often ineffective. Generating mazes by randomly generating lists of cells does not produce solvable mazes. Hence, students must develop smart generators that guarantee structural properties of mazes.

One way to represent a maze is as a list that contains cell types (e.g., empty, wall, exit) paired with positions. This *sparse* representation of a maze assumes missing cells are walls and the width (height) is inferred from the maximum x (y) coordinates. Working with a human-friendly data representation, such as the following *dense* representation, is desirable:

```
'((X X P)
  (E X _)
  (_ _ _))
```

WEEK	TITLE
1	Propositional Logic as Code
2	Executable Specifications for Simple Functions
3	More Executable Specifications for Simple Functions
4	Cryptography and Timing Attacks
5	Constant Folding for a Stack-Machine Compiler
6	Generating Mazes
7	SAT Solver using Four Representations of Booleans
8	Aliasing Puzzles and PBT Memos
9	Introduction to Message-Passing Concurrency
10	Manual Memory Allocator
11	Chandy–Lamport Snapshot Algorithm
12	Using Another PBT Library

Table 13.1: Homework Assignments

Students write conversion functions to and from the dense representation and test the round-trip property: converting a maze to and from a dense representation yields an equivalent maze. In doing so, one realizes that every maze has a unique dense representation but not a sparse one. Although not explicitly stated in the assignment, a correct solution must define a non-trivial equality over mazes to faithfully test the round-trip property.

CRYPTOGRAPHY: TIMING ATTACKS Given a function that checks passwords, yet is susceptible to a timing side-channel attack, students write a property that detects this vulnerability. They then modify the function so that it is no longer vulnerable to this kind of attack.

Intensional properties, which reflect *how* a computation proceeds and not just what it computes, are often overlooked in discussions about software specification. Yet there are many domains where intensional properties are as essential as extensional ones. Security researchers are particularly attuned to such considerations, where side-channel attacks can leak private information. One approach to formally modeling intensional properties is to allow programs to request intensional information at run time. Once intensional aspects are reified, contracts and tests can check them like any other ordinary property.

For this homework, students are given an insecure `password=?` function that returns as soon as the two input passwords differ. Thus, the time it takes `password=?` to execute is proportional to the length of the correct prefix of a given password attempt. This leak can be used to infer what the password should be in far fewer tries than what brute force guessing requires [21]. First, given a way to measure the num-

ber of character comparisons a function makes (i.e., a deterministic proxy for time), students define a property for secure password checking. The original `password=?` function should fail this property. Second, students adapt `password=?` such that it still works correctly, but also passes the timing specification. Implementing this modification of the program requires wasting a precise amount of time on redundant character comparisons.

MANUAL MEMORY ALLOCATOR For the memory allocation homework, students implement three functions (`malloc`, `free`, `defrag`) for an idealized allocator, and write an imperative `fibonacci` procedure using these abstractions.

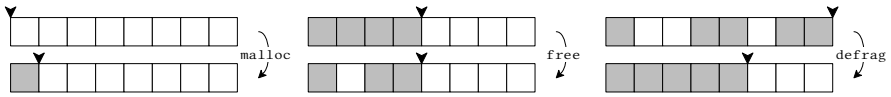

Figure 13.5: Memory Operations

Figure 13.5 illustrates how each of these functions works. A bump allocator splits an array of memory cells into two regions: a used portion and a free portion. The frontier between these two regions is tracked as an index into the array. Allocation increments the index, returning the address of a free cell to the user. Once the frontier index reaches the end of memory, defragmentation compacts all active cells and resets the frontier index.

Students write a simple memory allocator following the scheme from Figure 13.5 and then an imperative-style `fibonacci` procedure that uses their allocator. Bugs in the allocator propagate and compute nonsensical results when executing `fibonacci`.

The final task is to construct a trace contract that guarantees a memory cell is returned from `malloc` only if it (1) has never been allocated in the past or (2) has been allocated but was subsequently freed. This contract provides a simulacrum of what sophisticated tools, such as Valgrind, check about real-world low-level programs [130]. Students are expected to write a trace contract, which uses an accumulator that keeps track of the list of currently allocated memory cells, in order to enforce the property.

Figure 13.6 shows a solution to this exercise. The `allocated` list stores the current set of allocated cells. When `malloc` (`free`) is called, the respective accumulating function is invoked with the current set of cells and the newly allocated (freed) cell. If the operation is invalid, `#f` is returned. Otherwise, an updated set of cells is returned and becomes the new accumulator. One subtlety that students encounter is the requirement that reference equality (i.e., `eq?`) must be used for mutable values.

```
(: malloc (-> (AllOf (Maybe Cell)
                     (Record check-malloc allocated))))
(define (malloc) ── implementation ── )

(: free (-> (AllOf Cell (Record check-free allocated)) Any))
(define (free c) ── implementation ── )

(: allocated (List Cell))
(define allocated empty)

(: check-alloc (-> (List Cell) Cell (Maybe (List Cell))))
(define (check-alloc cells c)
  (cond [(false? c) cells]
        [(memq? c cells) #f]
        [else (cons c cells)]))

(: check-free (-> (List Cell) Cell (Maybe (List Cell)))
(define (check-free cells c)
  (if (memq? c cells)
      (filter (λ (d) (not (eq? d c))) cells)
      #f))
```

Figure 13.6: Trace Contract for a Memory Manager

CHANDY–LAMPORT SNAPSHOT ALGORITHM Distributed systems rely on snapshots to capture the current status of all processes in a system. In this assignment, students are asked to implement the Chandy–Lamport distributed snapshot algorithm using a purely functional message-passing interface. The assignment serves as an introduction to concurrent programming for many students and hints at the challenging task of reasoning about and maintaining invariants in concurrent systems.

Consider a network of banks that continuously transfer money among themselves. At any point in time, a bank might want to see a snapshot of how much money is in the network and where it is. Since no individual has a global view of the network, banks must request this information from all others in the network. The challenge is to implement an algorithm that computes a *consistent* snapshot of the network: intuitively, all money is accounted for in the snapshot. This problem is well-known in the field of distributed systems and the Chandy–Lamport algorithm is the standard solution [25].

Students implement Chandy–Lamport using a message-passing library similar to the `big-bang` interface from HtDP [50]. They must check that their implementation always provides consistent snapshots (i.e., no money is lost or gained in the snapshot compared to the start-

ing amount). Doing so is fairly nontrivial—small mistakes in the implementation can yield subtly incorrect behavior that leads to inconsistent snapshots. Debugging by "thinking hard" is, for most students, not productive. Describing invariants of the algorithm, writing property tests on smaller pieces of code, and unit-testing functions, all become useful for achieving a correct implementation.

SKILL TRANSFER Pedagogic languages have advantages in the classroom over languages found in industry, but using them risks tying the skills to the language. Students should come away from the course with confidence that they can apply software-specification techniques regardless of the language or tools they end up using in a job. Skill transfer must be addressed explicitly as students often cannot decouple the medium from the message.

One way to encourage skill transfer is to have assignments that go beyond the course content. Two assignments have students use their knowledge and skills outside LSL. The first is to write two memos: one addressed to technically proficient coworkers that analyzes the strengths and weaknesses of a particular PBT library; the other is addressed to a less technically minded manager explaining why the chosen PBT library should be adopted. Any language and library is acceptable, though most students select popular PBT frameworks in Java or Python. Writing memos reinforces lessons from the course by having students explain the benefits of PBT in their own words. The second assignment has students redo several of their previous homework solutions using their chosen PBT library. This exercise helps connect concepts from class to real-world libraries that often have a higher amount of incidental complexity than LSL.

13.5 STUDENT EXPERIENCES

To help assess the redesign, I interviewed students who took the new class as well as students who took the previous version taught with Lean. The Lean version of the course was well liked, but this sentiment may have more to do with the instructor and course logistics than with the material. The Lean course introduced students to mechanized theorem proving, proceeding from fixed data up through numbers and lists. At the end of the course, the final assignment was a proof of the correctness of insertion sort.

I interviewed ten students, four who took the Lean course, and six who took the specification-based one. Those who took the new course finished it around six months before the interview, whereas students who took the Lean course finished it a year or more before the interview. Interviews followed guidelines from Northeastern's institutional review board and participants answered the questions in Figure 13.7.

1. What was the main thing that you learned from this course? What do you think the course is supposed to teach?
2. What is the connection between logic and computation?
3. Imagine you are working on a system, written in Java, for processing financial transactions at a credit card company. Are there ideas, skills, or techniques you learned in this class that you think would help you do a better job implementing this system?
4. Was there anything in the course that seemed completely irrelevant, or that you could not imagine applying in any other context?
5. Have ideas or techniques from the course been useful in other courses?
6. Are there topics or techniques that have been useful on internships or you think will be useful on a job?

Figure 13.7: Interview Questions

LEAN COURSE INTERVIEWS While these four students represent only a small window into the experience of the hundreds of students who have taken this course, the commonality of their responses is revealing. Although none reported disliking the course, none could imagine how any skills they learned had any bearing on software engineering. For a course ostensibly introducing students to the field of formal methods, with the intention of connecting mathematical reasoning to real software, this is a damning result.

Overall, students considered the purpose of the course was to teach the mechanism of formal proof: specifically, proofs on small examples. Student B said the course was about "learning how to correctly construct proofs ... proofs regarding booleans, and then proofs regarding integers." Similarly, Student A reported doing proofs of "very basic things ... like commutativity or associativity of multiplication" and that insertion sort "was a huge proof for us."

When pressed about the skills they were learning, or how they might use them, students reported bleak results. Student C explained that "even though Lean was fun, I feel like I extracted more educational value from doing the on-paper proofs in algorithms." Student A was not sure if the course "taught any practical applications to prove software." Student D, when describing their perspective on the skills they learned, said: "I like [the course]; I can see why it fits into computer science, but I didn't see why it fits into the software concentration."

The question about building a hypothetical payment processor produced universally negative results, with illuminating responses. Some

students noted that they would reach for skills learned in other classes, e.g., Student A said they would "look towards the software side of solutions ... like JUnit," and Student D explained that "we didn't really go up to big applications of proving algorithms." Interestingly, Student B stated that "if I were a regular software developer making a transaction processor for credit cards, then I probably wouldn't be using [anything from the course], but if I were making the libraries that actually did the math, then I feel like I would." This remark gives weight to the hypothesis that students thought the skills they were learning only applied to properties of mathematical domains.

Our final question related to jobs and internships. Student B answered most concisely to the question of whether any skills from the class were useful to them in their job: "admittedly, no." Student D went into more detail, stating "there may be certain applications [for Lean] ... if you need to really prove exhaustively certain applications, but ... industry does not work like that ... people prove programs work ... largely through testing. I'm working at a finance company now, and there is nothing related to [Lean]." This response demonstrates that Student D understood, in theory, the generalization of the simple properties to larger software systems, but did not find anything concretely useful given the vast distance between the programs shown in class and the programs encountered at work. Student D elaborated that the techniques used in the class were "limited to academia" or to "top engineering organizations working on cutting edge research," and not relevant to an ordinary finance company.

SPECIFICATION-BASED COURSE INTERVIEWS As with the first set of interviews, these six students offer a limited view into the experience of the several hundred that went through the class, and further, the apparent depth of their understanding of the material varied widely. For some, the main takeaway appeared to be the importance of carefully thinking about all cases in code and the value of testing. For others, it was clear that they had learned to recognize the value of extracting logical properties of systems and accepted that property-based testing was a useful way to validate those. All, though, seemed both to have understood that the course was about understanding the precise behavior of code, and all were able to recognize that the skills transferred.

When asked what the course was about, Student 1 responded "formalizing how programs work." Other students shared similar ideas, with Student 3 saying the "main things that I learned were thinking about how to check if a function was working correctly." Student 2 said that they "never really thought about the correctness of programs before taking [the course]." For Student 6, the main takeaway was "the importance of testing your code."

Throughout the interviews, details emerged about what the students got out of the course. Student 4 shared that the course helped them "be

very intentional with [their] programs." For Student 5, the idealized memory allocator assignment made a strong impression. Student 5 said that learning about mutation and aliasing "would definitely be useful for pretty much any job" and that hearing about C now conjures a "mental image of how `malloc` and `free` work."

Most students were able to connect skills learned in the class to the hypothetical scenario involving credit card transactions. Student 2 mentioned checking "if money is credited into your account that amount cannot be negative," and Student 1 mentioned making sure "people can't enter zero for their credit card number." Student 6 thought about how such a system would likely have a fraud detection mechanism and could "give the [fraud detection] system a bunch of inputs to test it and make sure it actually catches the fraudulent ones ... or most of the fraudulent ones." Other students provided vague answers. For instance, Student 5 said that in such a system, skills from the class could help with "making sure all the functions that you have do what you want them to do."

Although it was not a direct response to a question, a final interesting statement from Student 6 was that: "for every job, not just CS, but in general, being able to make sure the work you're doing is correct, and does what you actually wanted to do is pretty important." That this student made such an observation, when talking about Logic and Computation, is precisely the goal.

These responses are in sharp contrast to responses from students in the Lean course, where the skills they learned, about the construction of formal proofs, seemed inextricably linked to the small examples through which they were demonstrated. While learning how to construct proofs may be a useful skill, it is not clear that it increases the likelihood that students produce reliable software.

13.6 LESSONS

This chapter has described the implementation and experience teaching a course dedicated to software specification. While some specifics may be particular to Northeastern, there are plenty of broad lessons for others to consider.

Teach software specification explicitly. Software specification and verification are distinct skills. Courses that focus on verification tend to have specifications that are simple and thus the skill of constructing them is not even worth mentioning. Software specification in real life is often challenging and worthy of explicit instruction.

Consider where specification fits into existing curricula. Northeastern has the luxury of an entire course devoted to this topic. At other institutions, small modules on this topic may have to be integrated into an existing course. There are plenty of points where this material makes sense. In a software-engineering course, PBT fits well as a supplemen-

tary testing technique alongside unit tests. In a course on typed functional programming, contracts can complement types and may be used to strengthen type signatures in the absence of full dependent types. Courses on object-oriented programming could benefit from a thorough treatment of data abstraction and its enforcement. Courses on imperative programming could benefit from a thorough treatment of mutation and how to enforce temporal constraints. A software security class might discuss how to formalize the notion of a side channel and detect it at run time. A systems class might discuss correctness properties of a memory allocator and how automated tools can check them. In short, specification is relevant across many domains and deserves to be emphasized across the curriculum.

Use real-world examples and larger programs. The interviews confirm that students came away from the Lean version of the course without seeing its relevance to software construction. This attitude seemed to be primarily due to the kinds of specifications encountered in the course that barely went beyond simple mathematical properties. Without specifications from actual software systems, students see no connection to programming. Assignments that draw from specifications of real-world systems provide a remedy. Even though the programs are simplified, and the programming language is not an industrial-strength one, the specifications can still be thought-provoking.

Provide a gradual path from unit tests to verification. Scaling up specifications demands techniques that scale too. Contracts and property-based testing are a powerful combination that is both reasonably lightweight (hence appropriate for freshmen) and expressive. In a first-year course, using a theorem prover to verify that the Chandy–Lamport algorithm results in a consistent snapshot would be infeasible. As a waypoint between unit tests and full verification, contracts and PBT not only make the transition smoother but are independently useful techniques.

Avoid distractions by using the right tools. Pedagogic tools remove barriers that get in the way of learning skills. Thus, some infrastructure is needed to deliver a reasonably smooth student experience. Many of the assignments in this course would not be possible without dedicated linguistic support.

LSL works due to Northeastern's existing curriculum that uses DrRacket and the teaching languages from HtDP. Similar infrastructure may be created for other educational environments. For example, others have adapted the HtDP curriculum to statically typed languages, such as OCaml, by porting features of the teaching languages as ordinary libraries [7, 145]. Likewise, the functionality of LSL could be packaged as a library or framework in any host language—especially those with metaprogramming facilities. Nearly every language has a library, or often several, inspired by QuickCheck [28]. High-quality libraries implementing software contracts are less common, but plenty of languages have some level of support. For instance, Clojure [76] has

a popular contract library called `clojure.spec`. A lightweight wrapper around these libraries could easily make them suitable for beginning students. While using a fit-for-purpose teaching language enables the highest level of control, many of the benefits can be achieved, with far less instructor effort, using well-designed libraries.

The hope is that these lessons inspire others to contemplate how software specification fits into their undergraduate curricula—including advanced techniques. Even though trace contracts are a fairly recent research development, they are intuitive enough that first-year students can employ them without trouble. Older students, and experienced programmers broadly, should have no problem with even more advanced effectful contracts.

Part V

REFLECTION

There are many papers in the literature that develop effectful contracts on an ad hoc basis. Additionally, there are related systems outside the realm of contracts intended to enforce similar properties. These last two chapters survey some of that related work, show how effectful contracts in this dissertation compare, and point toward the future.

14 COMPARISON TO RELATED WORK

Broadly speaking, related work is in the tradition of contract systems, runtime verification (RV), and type systems. These bodies of research have distinct philosophies regarding how they express and check properties. This chapter provides an overview of the literature and compares it to the effectful contracts presented in the dissertation.

14.1 ALTERNATIVE CONTRACT SYSTEMS

Chapter 4 introduces effect-handler contracts as a unified mechanism. An evaluation of this claim must show that the model and its implementations cover, as much as feasible, existing work.[1]

	Parameter	Trace	Attribute	Effect	Racket
ALLOW CALL	✓				✓
EXCEPTIONS	✓				✓
FRAMING	✓				✓
GHOST STATE		✓	✓		✓
MUST CALL	✓				✓
NON-REENTRANT	✓				✓
PURE					✓
RESTRICTED EFFECT					✓
TERMINATION	✓				✓
UNION CONTRACTS					✓

Table 14.1: Comparison Matrix

Table 14.1 presents an overview of how the mechanisms introduced in the dissertation cover a selection of problems occurring in the literature. Roughly speaking, a ✓ icon indicates that the contract mechanism supports this property.

ALLOW CALL A function may be called only during the dynamic extent of another function.

EXCEPTIONS Only specified exceptions may be raised during a function call. This property is the dynamic analogue to Java's checked exceptions.

1 All the papers surveyed here build systems on top of the low-level constructs. These contributions are orthogonal to, and not subsumed by, effect-handler contracts.

FRAMING Mutations are restricted to specified memory locations.

GHOST STATE Values are associated with a mutable reference, which is then used to check conformance with a protocol.

MUST CALL A function must be called during the dynamic extent of a call to another function.

NON-REENTRANT A function must not call itself recursively.

PURE No computational effects, other than non-termination and error signals, are permitted.

RESTRICTED EFFECT Effects are restricted at a fine-grained level.

TERMINATION A function call must terminate. Specifically, a size-change graph keeps track of changes to the size of arguments.

UNION CONTRACTS Given a set of contracts, the protected value satisfies at least one of the given contracts. Checking the union of flat contracts is easy, but checking the union of higher-order contracts relies on state to keep track of violations and assign blame.

Each of these example properties was distilled from one or more papers in the literature on effectful contracts. The remainder of this section discusses each paper chronologically, comparing it to the effectful contracts presented in the dissertation.

The Java Modeling Language (JML) [24] is a specification language for stating and verifying properties of objects in Java. It encompasses a broad range of features including assertions, class invariant statements, frame conditions, purity constraints, termination constraints, and ghost state declarations—just to name a few. Property checking takes place in one of two modes: static deductive verification (DV) or dynamic runtime-assertion checking (RAC). Some properties, such as termination, can be checked only using DV. JML differs from higher-order contract systems in three major ways. First, properties are described using a restrictive set of "well-defined" terms, a limitation compared to contracts written with ordinary constructs.[2] Second, JML supports only first-order properties. Finally, JML lacks a blame assignment component. Therefore, developers have only stack-trace information to help determine the location of a bug.

Tov and Pucella [143] study interoperability between a language with a substructural type system and one with a plain structural type system Specifically, the boundary form employs a run-time check to ensure that a function argument is *affine*, meaning it can be applied at most once. This check uses a mutable boolean field associated with

2 Although Racket contracts employ specialized notations, such as `->`, these constructs are relatively shallow abbreviations over plain Racket code.

each function value (i.e., ghost state) which indicates whether a function has been applied. As described in Section 6.1, contract-effect handlers can introduce and manipulate ghost state.

For interoperability, a language with a sound gradual type-and-effect system relies on a run-time enforcement mechanism to restrict the effects performed by untyped code. Bañados Schwerter et al. [11] show that the contracts for such a language can be formulated in terms of two operations: `has` (for checking the privileges granted by the current context) and `restrict` (for restricting the privileges of an expression). In the effect-handler language, these primitives can be expressed as main-effect contracts.

Shinnar [131] takes some constructs from JML, in particular framing contracts, and adapts them to Haskell. The implementation uses delimited checkpointing to keep track of state. A delimited checkpoint is a snapshot of memory captured using software transactional memory (STM). Framing contracts can detect and restrict writes to transactional references by comparing memory snapshots. Shinnar proves erasure for a limited model of Haskell with delimited checkpoints, similar to other lines of research [55, 134] that consider erasure for contracts with only a few restricted effects. As seen in Section 6.1, main-effect contracts can express framing constraints.

In some sense, higher-order temporal contracts [44] are closely related to trace contracts. Research on these contracts focuses on two aspects: an operational theory of temporal event sequences and the specification of properties. On the theory side, the work introduces a novel approach to operational semantics that formalizes the meaning of modules as automata that create trees of observable events, similar to game-based denotational semantics. The semantics satisfies a non-interference theorem, meaning that streams of values are kept separate. On the practical side, the work considers specifying properties of event sequences as regular expressions *without* giving programmers access to a data representation of traces. Trace contracts come with more expressive power, yet do not necessarily sacrifice efficiency.

Scholliers et al. [129] develop computational contracts that instantiate aspect-oriented programming for the contract world. At first glance, computational contracts look similar to higher-order temporal contracts. But computational contracts go far beyond any classical contract classification scheme [18, 19], providing unprecedented power and imposing a similarly high cost. A computational contract system empowers programmers to impose arbitrary restrictions on components from the outside and in a post-hoc manner. Thus, computational contracts depart from the idea that contracts are assertions at the boundary between opaque components, instead turning components into glass boxes. Effect-handler contracts can achieve this behavior as long as the excluded function can be modified such that

it is monitorable; if not, this contract is impossible to realize without more invasive techniques.

Moore et al. [104] study authorization contracts for enforcing access control. Specifically, authorization contracts can capture, check, and restore access privileges via an authority environment that records privilege information. The model from Moore et al. [104] is essentially a variant of parameter contracts topped off with a DSL for authorization management. Effectful contracts alone do not implement any of the security aspects of the system. However, authorization contracts could be built on top of parameter contracts.

Nguyễn et al. [112] provide a run-time check for termination by monitoring the size-change property (SCP) of functions dynamically. Any diverging function must exhibit an SCP violation, causing a contract violation. They turn this run-time check into a static one, using existing contract verification techniques [111]. To guarantee termination, they use continuation marks to store size-change information on the stack. Contract-handler contracts can be used to store the same information, as shown in Section 7.1.

While the literature on higher-order contracts tends to mention intersection and union contracts, implementing these in general is a serious challenge. Indeed, Racket rejects `or/c` contracts if the disjuncts are not "first-order distinguishable." Several researchers [62, 83, 147] have studied this problem, and all come to the conclusion that effects are needed. For example, Williams et al. [147] use a mutable blame state to keep track of contract violations. A contract-effect handler can be used to implement this blame state.

14.2 RUNTIME VERIFICATION

Traditional contract systems and RV systems differ along several dimensions. Most importantly, as Meyer [101] observes, contracts are a design tool for the developer; in contrast, RV is a tool for the quality assurance stage of the development process.

SCOPE Contracts are *modular*. A programmer attaches contracts to the interface of a "server" component. When a "client" component imports a server component, it is forced to agree to the contract. Similarly, a client component may impose a contract on imported pieces of functionality to protect itself from a misbehaving service component. In the first case, clients do not need to be adapted to the service contract, and in the second case, service components remain unaware of the client's protective contract. Put differently, it is possible to compile these components in either order, or to link pre-compiled binary objects. This ability, however, is dependent on run-time support for proxy values [134, 146] or a similar mechanism.

RV is *whole program*. A programmer specifies events of interest and properties about event traces. The RV system converts this specification into an executable monitor and weaves interception code into the host program to communicate first-order data about events to a separate monitor process [15]. Monitoring higher-order values is possible with RV, but the encoding uses a complex protocol between the server and the client module; it requires source modification to both components. Implementing the protocol on a modular basis is either impossible, which precludes the binary-linking approach available with contracts, or requires extensions to work [149].

LANGUAGE Contracts are linguistic elements *inside* the language. The programmer uses the same language and the same tools for writing code and contracts. Extending the notation for contracts in a domain-specific manner is useful; the `->` abbreviation for function contracts is the first and simplest example. Racket also treats contracts as first-class, so they can be passed and returned from functions.

RV is extra-linguistic; that is, RV systems exist *outside* the language. Specifications are usually written in a distinct, external logic language and tend to make temporal statements about sequences of first-order data [74]. While this language may contain fragments of host-language code, it is only loosely connected to the host language.

VIOLATIONS As a consequence of the differences along the linguistic axis, contracts and RV differ in two ways concerning the violation of specifications: recovery and error-location information.

When a contract system discovers a violation of an assertion, it raises an exception that includes information about the parties that agreed to the contract and which of them violated it (i.e., blame information). By raising an exception at the very point where a contract violation is discovered, the contract system gives the program a chance to recover with a response targeted to the problem. In a language with resumable exceptions, such as Common Lisp [132], a program may even resume its execution at the place where the violation occurred.

The *precise* error information in violation messages enables the developer to understand the cause of a violation. Lazarek et al. [91] show that this blame information is effective at narrowing the search space during the debugging process. It is also a well-founded concept; Dimoulas et al. [43] provide a framework for proving that blame information points to the component supplying a value that does not meet its specification.

Traditionally, RV systems report violations of specifications with delay and without blame information [138]. The delay is due to the underlying process-communication arrangement between the program and its monitor, posing a challenge to tracking the provenance of val-

ues and for assigning blame. Hence, RV makes it difficult to restart programs unless an additional "diagnosis layer" is supplied [95].

PROPERTIES Contract systems are property *agnostic*. Any predicate, including one that tries to decide a recursively enumerable property, can be used as a contract. As such, contracts are maximally expressive and potentially computationally expensive.

RV research is property *sensitive*. Much of the RV literature focuses on the development of specification languages that can express properties of interest concisely and that can be compiled into efficient monitoring code [95]. Often these languages are variants of temporal logic. Specialized logics can provide hard guarantees about time and space efficiency at the cost of expressive power.

Within the landscape of RV tools, JavaMOP is the best point for comparison. Three key differences stand out. First, trace contracts are intended to be realizable in any programming language. As detailed in Section 10.2, implementing domain-specific contract notations is easiest in languages with macro systems such as Racket, Clojure, Julia, Rust, or Scala. In languages without macro systems, an implementation may require a bit more labor, depending on existing features. Second, a contract-based library is more convenient for programmers to use than a toolchain-based solution such as JavaMOP. A library can be installed directly from a language's package manager and used immediately. Finally, contracts provide additional benefits beyond what JavaMOP delivers, e.g., IDE integration and blame assignment.

LOGIC PLUGINS JavaMOP users can develop their own logic plugins to add new, possibly domain-specific, specification languages. These plugins are essentially ordinary compilers from the surface syntax to Java code. Constructing such a compiler requires the use of traditional compiler-generator tools, such as a parser generator, and a solid understanding of the semantics and behavior of the target language. As a result, developing a new logic or extending an existing one is challenging and not a lightweight exercise.

Domain-specific notations rely on the host language's metaprogramming facilities or established language-implementation patterns. In a language with a macro system, such as Racket, new notations with sophisticated features can be created easily with a combination of declarative grammar specifications and recursive functions. And, if the host allows for declarative DSL specifications, these languages become fairly easy to maintain.

LINGUISTIC INTEGRATION One consequence of JavaMOP's extralinguistic approach is that logical specifications are syntactically isolated from the rest of the program. JavaMOP specifications are typically added as comments to the code or as entirely separate text files.

As a consequence, these artifacts end up as second-class citizens within a software project.

By contrast, contracts make logical specifications an integrated part of the underlying project; they exist *within* the program. As such, all host-language tools continue to work with logical specifications. Notably, DrRacket's existing IDE services seamlessly extend to DSL code.

TRACE MANAGEMENT JavaMOP takes a global perspective on monitoring. System events are captured and stored in a global sequence. This monitoring strategy is central to the run-time verification approach. Since checking globally generated traces requires a slicing technique, JavaMOP's implementation makes slicing independent of the underlying logic to reduce the burden on the developer of a new logic notation [27, 81]. In the end, however, the program-global property approach remains an obstacle to reasoning about code in a compositional manner. Slicing also requires a notion of equality on objects which, if one is not defined already, relies on pointer equality. Depending on pointer equality for objects violates a key principle of object-oriented programming [31].

Contracts, by contrast, often eliminate the need for slicing because the state for a given value is freshly initialized when the value passes through an attachment boundary. Monitoring is local. For instance, the MapIterator property from Section 10.1 can be attached to a specific object and remains quantifier free; the equivalent formula in JavaMOP requires quantifiers and therefore slicing. Localized monitoring also unlocks unique advantages such as blame assignment (if the underlying contract system provides it). Slicing is still occasionally useful in contracts. For example, Section 10.1 shows slicing based on port numbers. A notion of equality is defined for port numbers, so the principles of object-oriented design are maintained.

RUN-TIME ENFORCEMENT As originally published, JavaMOP supported only one run-time enforcement mechanism. The AspectJ [84] weaver would statically inject monitoring code into a system based on the given specification. This architecture requires AspectJ as a compile-time dependency and alters the ordinary build process for a Java project. Now, JavaMOP supports a dynamic enforcement mechanism known as the JavaMOP agent. With this mode, an instrumented JVM supplies the means to record system events during execution. The JavaMOP agent offers some advantages over the static approach but requires a specialized runtime. Both of these mechanisms are somewhat specific to Java and are not readily available in other languages.

Contracts use a completely different implementation approach. While first-order contracts essentially get away with boolean-valued assertions, higher-order contract systems rely on proxy objects to monitor a running program. If the language does not support proxy objects

directly, implementing a reasonable approximation is possible. The key is that proxies provide a local-enforcement mechanism, enabling blame assignment in higher-order settings.

A comparison of trace contracts and RV must address several aspects: (1) whether they can accommodate existing logic notations; (2) what developer-facing features these implementations provide; and (3) how notations support reporting precise specification violations. The literature proposes a wide variety of logic notations for specifications that are suitable to check with trace contracts. Here are six common ones:

PAST-TIME LINEAR TEMPORAL LOGIC (PLTL) is a past-time variant of the more commonly used future-time linear temporal logic. See Section 10.1 for a complete description of PLTL and an example.

REGULAR EXPRESSIONS (RE) are a simple formalism equivalent in expressive power to finite-state machines [87]; they are a well-known way for specifying parts of the behavior of a software system [128].

FINITE STATE MACHINES (NFA, DFA) have a finite number of states and transitions between them. A deterministic finite-state automaton (DFA) denotes the sequences of events that, when run on the state machine, follow a unique path that ends in an accepting state [126]. A non-deterministic finite-state automaton (NFA) denotes the sequences of events that, when run on the state machine, end in an accepting state along *some* path.

QUANTIFIED EVENT AUTOMATA (QEA) resemble DFAs but are augmented with extra features [14]. As the name suggests, the machine supports quantification. Conceptually, a QEA represents a family of automata—one for every possible instantiation of quantified variables. Practical implementations of QEA more efficiently compute transitions via a compact representation of this family of automata.

STREAM LOGIC (SL) describes properties by defining stream equations [34]. A stream is a sequence of events that is accessible with a finite lookback. The dynamic checks for SL specifications compute the value for each stream at the current index, making use of the static dependencies of equations to solve them in the correct order. Irrelevant values, i.e., ones that are beyond the finite lookback, are automatically discarded.

The implementations of these six notations have distinct qualities from the perspective of language creators and users. Four dimensions stand out as particularly relevant: binding positions, definitions, static

semantics, and macro extensibility. Figure 14.1 summarizes how each notation compares along these dimensions.

	PLTL	RE	DFA	NFA	QEA	SL
Binding Rules	✓				✓	✓
Definitions	✓	✓				
Static Semantics	✓					✓
Macro Extensible	✓	✓				

Figure 14.1: Qualitative Assessment of Logic DSLs

BINDING POSITIONS Some DSLs are variable-free, while others come with variable declarations that introduce binding positions and determine the scope of a declaration. Since the `syntax-spec` system comes with a binding-rules mechanism, it allows notation creators to specify which pieces of a formula are variable references, which are binding positions, and the scope of bindings. By declaring the binding structure, implementations can automatically support IDE services, static-semantics passes, and translations into predicates.

Logic notations without quantification (i.e., RE, DFA, and NFA) do not require binding rules. Specifically, the state-machine notations do not use an up-front declaration of states; if they did, binding rules would be required. By contrast, both PLTL and QEA come with variable quantification, and quantified variables are binders. In SL, equations must reference other equations or primitive event streams. The binding rules for SL guarantee that there are no free variables referencing an undefined stream.

SEPARATE DEFINITIONS For some logic notations, formulas can be defined piecemeal via the binding-space mechanism. Developers can build up libraries of reusable formulas that others can then compose into specifications of modules, classes, and other components. For such compositional notations, definition forms are a highly useful mechanism. While the state-machine notations from the literature do not seem compositional, both RE and PLTL greatly benefit from the introduction of definition forms.

STATIC SEMANTICS Some specification notations impose criteria that determine whether well-formed formulas are valid. If an implementation can enforce such static-semantics constraints, it is superior to plain frameworks or functional libraries because it informs developers at compile time about basic mistakes, instead of delaying the discovery of specification mistakes until run time. Programmers can thus avoid unexpected run-time errors and dynamic debugging due to invalid specifications.

While Racket's `syntax-spec` library automatically guarantees that logical specifications are well-formed and closed with respect to any binding rules, the static semantics of DSLs must be written as an additional check. Of the investigated notations, PLTL and SL must satisfy such criteria. Section 10.2 covers the monitorability criteria for PLTL. For SL, the DSL implementation statically checks that the dependency graph of the stream equations has no closed walk with a total weight of zero, where the weights are determined by indices into streams. This check statically generates the dependency graph from the stream equations via local macro expansion, even when references to streams are embedded inside Racket sub-expressions.

MACRO EXTENSIBILITY When logical formulas are compositional, programmers benefit from a macro-extensible language. The RE and PLTL notations make heavy use of this feature because many useful operators can be derived from a small set of primitive operators. By expanding the surface syntax to a small core language, the implementation can be organized like an ordinary compiler (as in functional languages such as Haskell, ML, or Scheme). Equally important, macro extensibility enables users to create syntactic abstractions when functional abstractions do not suffice to hide repeated patterns in formulas.

There is a practical limit to how small the core set of operators should be. In RE, for example, complement and intersection do not add expressive power to the logic in the sense of being able to describe more languages. However, these operators are not straightforward macro translations, and are therefore treated as primitive. Operators with straightforward translations, such as the positive repetition operator R^+, are implemented as macros.

One claimed benefit of contracts is that error messages come with blame information identifying the source of the specification violation. Blame assignment in higher-order settings, and even more so for temporal properties, is subtle. In general, blame assignment points to the components that contribute to a violation of a specification.

To understand whether blame assignment from the underlying contract system works well when used with domain-specific notations, it is necessary to adapt examples from the literature and to test violations on sample code snippets. The Racket implementation should produce blame assignments that point to bugs.

Figure 14.2 list some sample properties from the RV literature, which logics they were implemented in, and how many lines it took to write each formula. All explain properties of the Java API, which are stated informally, but may or may not be checked via the assertion system. For each property, the trace-contract implementation caught all violations in sample programs with the expected blame assignment.

Property	Logic (SLOC)	Description
MapIterator	PLTL (7) RE (1)	An iterator constructed from a collection, which is itself constructed from a map, is invalidated once the underlying map is mutated.
HasNext	QEA (6) RE (1)	A call to the `has-next` method must precede every call to the `next` method.
HashCode	QEA (5) SL (7)	An object's `hash-code` must not change so long as it is being used as a key in a map.
ClosedReader	SL (8)	A call to the `read` method of a reader must not happen after the underlying stream has been closed.
SetFromMap	SL (6)	Creating a set from a map renders the latter unusable.
RemoveOnce	RE (1)	An iterator's `remove` method can be called only once per invocation of the `next` method.
StreamClose	RE (1)	A stream may be used until it is closed or the underlying TCP listener is closed.
PortExclusivity	QEA (5)	Listeners can be created for a port, so long as there is no active listener already associated with that port.

Figure 14.2: Sample Properties

14.3 TYPE SYSTEMS

Researchers often try to move from dynamically checked contracts to statically checked types, because discovering general mistakes during compile time seems advantageous compared to discovering specific mistakes at run time, perhaps even after a program has been deployed.

The work of Strom and Yemini [135] on typestate systems, recently resumed in various forms [80, 125, 148], directly addresses simple but common affinity restrictions in APIs. For example, typestate systems can check constraints such as "method `m` may be called at most once" and even "method `m` must be called before method `n`." These constraints are restricted to regular properties, i.e., those that can be expressed using a finite-state machine.

Session types [79] are a closely related idea. Recently this field has experienced rapid growth. Roughly speaking, session types for objects come with the same expressive power as typestate [65].

Effect systems are also capable, in a limited way, of constraining the order in which effects can be performed. Ordinary effect systems do not consider the order of effects, but sequential effect systems [90, 139] can. Further extensions can verify some temporal logic formulas [70].

No existing static technique can express all the trace-contract examples. By combining traces with plain code, a programmer can formulate arbitrary predicates and check value-dependent constraints on traces. Trace predicates can look for specific values or use specific values to express a constraint, which is impossible with these type systems. Dependent session types [141] may be able to do better, but are still limited to statically decidable properties. Trace contracts, by monitoring programs at run time, are able to take advantage of the precision that run-time checking offers. A combination of session types and contracts [20] can refine the content of messages passed between parties, but the structure of the protocol remains fixed. This approach also does not naturally extend to contracts on higher-order values.

15 CONCLUDING REMARKS

The evidence in support of my thesis, that effectful contracts can be used as a principled foundation to build expressive high-level specification languages, is not without caveats. These limitations deserve consideration. Naturally, they also suggest future work.

15.1 LIMITATIONS

SINGLE-LANGUAGE IMPLEMENTATIONS A significant threat to validity is that the implementations were all carried out in Racket, a language with unique features. Combining an advanced contract system with sophisticated metaprogramming capabilities—Racket made the implementations described in this dissertation relatively easy to create. The core implementation code for all the libraries combined does not exceed a few thousand lines. While Racket's facilities were a boon for the research, it does call into question how applicable this work is to conventional languages. Section 8.4 addresses this question with words but not actions. A thorough validation would require an independent engineering effort to realize effectful contracts in a more conventional language.

BENCHMARK LIMITATIONS Dovetailing with the linguistic limitation, the performance measurements from Chapter 12 are with respect to the Racket implementation only. Aside from the usual concerns about benchmark selection, the number of benchmarks available is small. In contrast, the RV community has successfully tested hundreds of properties in Java's standard library and frequently report overheads far lower than effectful contracts [93]. Whether this discrepancy is due to the underlying runtime (Racket vs. Java) or the architecture (proxies vs. weaving) or optimization effort (little vs. significant) is unclear. Optimistically, I anticipate that effectful contracts are not inherently slower than RV techniques, but I have no data to support that hunch.

LIMITED USER EXPERIENCES So far, there are only a handful of people who have tried the effectful-contract libraries presented in this dissertation. Certainly the experiences from Chapter 13 show promising results, but the students did not have enough time or background knowledge to write truly challenging trace contracts. A proper evaluation of the design will require experienced software developers using the libraries to get real work done.

15.2 FUTURE WORK

Effectful contracts have been studied, on an ad hoc basis, for many years. The work in this dissertation puts effectful contracts on firmer theoretical ground. There is, however, opportunity for follow-up work.

EFFECTFUL CONTRACTS FOR RACKET PROPER As mentioned in Chapter 7, parameter contracts are available in Racket's contract library. Trace contracts and attribute contracts, along with the many auxiliary constructs needed to make them useful, are available only as experimental third-party libraries. There are some obstacles to merging these contributions into Racket proper (e.g., eliminating some dependencies) but no fundamental blockers. Getting these libraries into Racket's built-in contract system will make them far more appealing to potential users.

BLAME STRATEGIES Section 8.2 proposes several strategies for trace-contract blame: no suspects, sets of suspects, and lists of suspects. By default, the trace-contract library does not report suspects and simply gives the name of the party that supplied the final value violating the trace predicate. More detailed information, however, may lead to improvements in bug finding. An investigation into the pragmatics [39] of trace-contract blame is necessary to determine whether the additional cost of maintaining extra suspect information can be exploited to improve the debugging process.

LIFTING METATHEORETIC PROPERTIES Proving metatheoretic properties of the effect-handler contract system (e.g., erasure, blame correctness) opens the door to possibly lifting these properties to surface-level constructs that compile to effect-handler contracts. Sections 7.3, 8.5 and 9.3 give macro translations into Effect Racket. Given such a translation and a compiler-correctness theorem, it may be possible to lift the metatheoretic property to the source language via compilation—saving proof effort. If an effectful contract could be expressed using effect-handler contracts, then all liftable properties would immediately be true at the source. Whether such a lifting strategy is possible is an open question.

MORE DOMAIN-SPECIFIC NOTATIONS The idea of combining contracts with domain-specific specification languages goes far beyond the particular instantiation with trace contracts. For example, parameter contracts with an authorization logic [6] yields authorization contracts [104] that can enforce security properties. Attribute contracts combined with an appropriate specification language yields the runtime equivalent of typestate. Indeed, there are plenty of other combinations not yet studied and worthy of exploration.

CONCURRENT CONTRACTS Chapter 11 presents a case study of how trace contracts combine with existing Racket contracts to monitor properties of a concurrent system. As a case study, the example illustrates the compositionality of trace contracts. However, a single example cannot establish that trace contracts are a broadly useful mechanism to write such contracts. The example is limited to one concurrency primitive (channels) and a single specification language. A rigorous demonstration of the value trace contracts bring to concurrent systems would potentially involve other models of concurrency, including the dataspace model [63], and more powerful specification languages. The infrastructure set up in Chapter 10 provides the necessary foundation for such an effort.

STATIC VERIFICATION While run-time checking of contracts is the standard mode of enforcement, contracts can be statically checked too [13, 113, 150]. Static checking is advantageous since it offers a way to catch errors earlier in the development lifecycle. Because they are more expressive than other forms of specification, contract verification requires advanced static analyses. Similar techniques could be applied to verify effectful contracts too, including contracts that use domain-specific notations such as temporal logic. A tool of this sort could combine the usability benefits of contracts with the power of verification via model checking.

Part VI

APPENDICES

A

PROOF OF ERASURE

$\boxed{e \sim e}$ for EFFECT

$$
\begin{aligned}
\texttt{with}^{\mathcal{C}}\ e_h\ e &\sim \tilde{e} \\
\texttt{mon}_j^{k,l}\ e_c\ e &\sim \tilde{e} \\
\texttt{mark}_j^{k,l}\ (\nu\ \mathcal{M}\ f)\ e &\sim \tilde{e} \\
&\vdots \\
\lambda x.f\ x &\sim \tilde{f} \\
\lambda x.\texttt{let}\ x_j = x\ \texttt{in}\ &\sim \tilde{f} \\
\texttt{let}\ x_k = x\ \texttt{in}& \\
f\ x_k&
\end{aligned}
$$

$\boxed{E^{\mathcal{C}} \sim E^{\mathcal{C}}}$ for EFFECT

$$
\begin{aligned}
\texttt{with}^{\mathcal{C}}\ e_h\ E^{\mathcal{C}} &\sim \widetilde{E^{\mathcal{C}}} \\
\texttt{mon}_j^{k,l}\ e_c\ E^{\mathcal{C}} &\sim \widetilde{E^{\mathcal{C}}} \\
\texttt{mark}_j^{k,l}\ (\nu\ \mathcal{M}\ f)\ E^{\mathcal{C}} &\sim \widetilde{E^{\mathcal{C}}} \\
&\vdots
\end{aligned}
$$

Figure A.1: Expression and Evaluation Context Simulation Relation

Theorem (Erasure). If $\text{eval}(e) = b$ then $\text{eval}(\mathcal{E}[\![e]\!]) = b$.

Proof. The proof proceeds by a simulation argument. See Figure A.1 for the simulation relation on expressions and evaluation contexts. By convention, a metavariable with a tilde is in simulation with its plain counterpart. Let $\tilde{e} = \mathcal{E}[\![e]\!]$. By Lemma A.1, $e \sim \tilde{e}$. It suffices to show that $\tilde{e} \longmapsto^* b$. By induction on $e \longmapsto^* b$.

Case $e = b$**.**

Booleans are preserved by the simulation so $\tilde{e} = b$.

Case $e \longmapsto^+ b$**.**

By Lemma A.2, there exists e' such that $e \longmapsto^+ e' \longmapsto^* b$ and $\tilde{e} \longmapsto^* \widetilde{e'}$. The inductive hypothesis yields that $\widetilde{e'} \longmapsto^* b$. Thus, $\tilde{e} \longmapsto^* \widetilde{e'} \longmapsto^* b$ and therefore $\tilde{e} \longmapsto^* b$. □

Lemma A.1 (Erasure Inclusion). For all $e \in \text{Expr}$, $e \sim \mathcal{E}[\![e]\!]$.

Proof. By induction on e. □

Lemma A.2 (Simulation). If $e \longmapsto^+ v$ then for all $\tilde{e}$ such that $e \sim \tilde{e}$, there exists e', $\widetilde{e'}$ such that $e \longmapsto^+ e' \longmapsto^* v$ and $\tilde{e} \longmapsto^* \widetilde{e'}$.

Proof. Since e reduces to a value, not an error, that means $e = E[e_r] \longmapsto E[e_c]$ for some evaluation context E and expressions e_r, e_c.

Suppose $E \notin \text{Ctx}^{\mathcal{M}}$ or equivalently $E = E^{\mathcal{C}}$. Assume too that $E^{\mathcal{C}}[e_r] \sim \widetilde{e_j}$. By Lemma A.3, $E^{\mathcal{C}}[e_c] \sim \widetilde{e_j}$ as needed. Otherwise, take $E = E^{\mathcal{M}}$. By cases on $E^{\mathcal{M}}[e_r] \longmapsto E^{\mathcal{M}}[e_c]$.

Case $E^{\mathcal{M}}[\texttt{if}\ v\ e_t\ e_f] \longmapsto E^{\mathcal{M}}[e_t], v \neq \texttt{ff}$.

By Lemma A.7, $E^{\mathcal{M}}[e_r] \sim \widetilde{E^{\mathcal{M}}}[\texttt{if}\ \tilde{v}\ \widetilde{e_t}\ \widetilde{e_f}]$ for $e_t \sim \widetilde{e_t}$. Because $\sim$ preserves non-false values, $\widetilde{E^{\mathcal{M}}}[\texttt{if}\ \tilde{v}\ \widetilde{e_t}\ \widetilde{e_f}] \longmapsto \widetilde{E^{\mathcal{M}}}[\widetilde{e_t}]$. By Lemma A.8, $E^{\mathcal{M}}[e_t] \sim \widetilde{E^{\mathcal{M}}}[\widetilde{e_t}]$. The remaining cases in this proof use Lemma A.7 and Lemma A.8 similarly.

Case $E^{\mathcal{M}}[(\lambda x.e_b)\ v] \longmapsto E^{\mathcal{M}}[e_b[v/x]]$.

If $\widetilde{e_r} = (\lambda x.\widetilde{e_b})\ \tilde{v}$, then the result follows by Lemma A.9. If $e_r = (\lambda x.f\ x)\ v$ and $\widetilde{e_r} = \tilde{f}\ \tilde{v}$, then the result follows letting $e' = E^{\mathcal{M}}[f\ v]$ and $\widetilde{e'} = \widetilde{E^{\mathcal{M}}}[\tilde{f}\ \tilde{v}]$. If

$$\begin{aligned} e_r = (\lambda x.&\texttt{let}\ x_j = x\ \texttt{in} \\ &\texttt{let}\ x_k = x\ \texttt{in} \\ &f\ x_k)\ v \end{aligned}$$

and $\widetilde{e_r} = \tilde{f}\ \tilde{v}$, then the result also follows letting $e' = E^{\mathcal{M}}[f\ v]$ and $\widetilde{e'} = \widetilde{E^{\mathcal{M}}}[\tilde{f}\ \tilde{v}]$ because $e \longmapsto^+ e'$ (in three steps).

Case $E^{\mathcal{M}}[\texttt{with}^{\mathcal{M}}\ v_h\ v] \longmapsto E^{\mathcal{M}}[v]$.

Let $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\texttt{with}^{\mathcal{M}}\ \tilde{v_h}\ \tilde{v}]$. Then $\tilde{e} \longmapsto \widetilde{E^{\mathcal{M}}}[\tilde{v}]$ as needed.

Case $E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ v_h\ v] \longmapsto E^{\mathcal{M}}[v]$.

Let $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\tilde{v}]$. No step is needed.

Case $E^{\mathcal{M}}[\texttt{with}^{\mathcal{M}}\ v_h\ E_k^{\mathcal{M}}[\texttt{do}\ v]] \longmapsto E^{\mathcal{M}}[v_h\ v_d\ (\lambda x.\texttt{with}^{\mathcal{M}}\ v_h\ E_k^{\mathcal{M}}[e_c])]$.

By Lemma A.4 and Lemma A.5.

Case $E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ \langle v_d, v_h\rangle\ E_k^{\mathcal{C}}[\texttt{do}\ v]] \longmapsto E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ v_h\ E_k^{\mathcal{C}}[v_c]]$.

By Lemma A.3.

Case $E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ f\ E_k^{\mathcal{C}}[\texttt{do}\ v]] \longmapsto E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ (f\ v)\ E_k^{\mathcal{C}}[\texttt{do}\ v]]$.

Let $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\widetilde{e_b}]$ for $e_b = E_k^{\mathcal{C}}[\texttt{do}\ v]$. Because $E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ (f\ v)\ E_k^{\mathcal{C}}[\texttt{do}\ v]] \sim \widetilde{E^{\mathcal{M}}}[\widetilde{e_b}]$, that implies $E^{\mathcal{M}}[\texttt{with}^{\mathcal{C}}\ (f\ v)\ E_k^{\mathcal{C}}[\texttt{do}\ v]] \sim \tilde{e}$.

Case $E^{\mathcal{M}}[\texttt{mon}_j^{k,l}\ \texttt{tt}\ v] \longmapsto E^{\mathcal{M}}[v]$.

Let $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\tilde{v}]$ for $v \sim \tilde{v}$. Since $E^{\mathcal{M}}[v] \sim \widetilde{E^{\mathcal{M}}}[\tilde{v}]$ that implies $E^{\mathcal{M}}[v] \sim \tilde{e}$.

Case $E^{\mathcal{M}}[\texttt{mon}_j^{k,l}\ \texttt{ff}\ v] \longmapsto E^{\mathcal{M}}[\texttt{err}_j^k]$.

Contradiction since $\texttt{err}_j^k$ does not reduce to a value.

Case $E^{\mathcal{M}}[\texttt{mon}_j^{k,l}\ f\ v] \longmapsto E^{\mathcal{M}}[\texttt{mon}_j^{k,l}\ (f\ v)\ v]$.

Let $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\tilde{v}]$ for $v \sim \tilde{v}$. Because $E^{\mathcal{M}}[\texttt{mon}_j^{k,l}\ (f\ v)\ v] \sim \widetilde{E^{\mathcal{M}}}[\tilde{v}]$ that implies $E^{\mathcal{M}}[\texttt{mon}_j^{k,l}\ (f\ v)\ v] \sim \tilde{e}$.

Case $E^{\mathcal{M}}[\mathtt{mon}_j^{k,l}\,(v_d \Rightarrow v_c)\;f] \longmapsto E^{\mathcal{M}}[\lambda x.\,\_\!\_\,]$**.**

This step produces an expression that is in simulation with e:

$$\begin{aligned}
E^{\mathcal{M}}[\lambda x.\,_\!_\,] &= E^{\mathcal{M}}[\lambda x.\mathtt{let}\ x_j = \mathtt{mon}_j^{l,j}\,v_d\ x\ \mathtt{in}\\
&\qquad\qquad \mathtt{let}\ x_k = \mathtt{mon}_j^{l,k}\,v_d\ x\ \mathtt{in}\\
&\qquad\qquad \mathtt{mon}_j^{k,l}\,(v_c\ x_j)\ (f\ x_k)]\\
&\sim \widetilde{E^{\mathcal{M}}}[\lambda x.\mathtt{let}\ x_j = x\ \mathtt{in}\\
&\qquad\qquad \mathtt{let}\ x_k = x\ \mathtt{in}\\
&\qquad\qquad \tilde{f}\ x_k]\\
&\sim \widetilde{E^{\mathcal{M}}}[\tilde{f}]\\
&= \tilde{e}
\end{aligned}$$

Case $E^{\mathcal{M}}[\mathtt{mon}_j^{k,l}\,(v_d\ \mathcal{M}\ g)\ f] \longmapsto E^{\mathcal{M}}[\lambda x.\mathtt{mark}_j^{k,l}\,(v_d\ \mathcal{M}\ g)\ (f\ x)]$**.**

Thus, $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\tilde{f}]$ and $e' = E^{\mathcal{M}}[\lambda x.\mathtt{mark}_j^{k,l}\,(v_d\ \mathcal{M}\ g)\ (f\ x)] \sim \widetilde{E^{\mathcal{M}}}[\lambda x.\tilde{f}\ x] \sim \widetilde{E^{\mathcal{M}}}[\tilde{f}]$.

Case $E^{\mathcal{M}}[\mathtt{mark}_j^{k,l}\,((v_d\ \mathcal{M}\ g)\ \mathcal{M}\ v)\ ] \longmapsto E^{\mathcal{M}}[v]$**.**

Thus, $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\tilde{v}]$, and $e' = E^{\mathcal{M}}[v] \sim \widetilde{E^{\mathcal{M}}}[\tilde{v}]$.

Case $E^{\mathcal{M}}[\mathtt{mon}_j^{k,l}\,(\mathcal{C}\ g)\ f] \longmapsto E^{\mathcal{M}}[\lambda x.\mathtt{with}^{\mathcal{C}}\ g\ (f\ x)]$**.**

Thus, $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\tilde{f}]$, and $e' = E^{\mathcal{M}}[\lambda x.\mathtt{with}^{\mathcal{C}}\ g\ (f\ x)] \sim E^{\mathcal{M}}[\lambda x.\tilde{f}\ x] \sim E^{\mathcal{M}}[\tilde{f}]$.

Otherwise.

The remaining cases are similar to one of the above. □

Lemma A.3 (Contract Irrelevance). If $E^{\mathcal{C}}[e_s] \sim \tilde{e}$ then $E^{\mathcal{C}}[e_t] \sim \tilde{e}$.

Proof. Intuitively, any expression can be replaced inside a contract-checking context because it is erased by the simulation. There are only two situations that can occur during reduction:

Case $E^{\mathcal{C}} = E^{\mathcal{M}}[\mathtt{mon}_j^{k,l}\ E\ e_b]$**.**

Therefore, $E^{\mathcal{C}}[e_s] = E^{\mathcal{M}}[\mathtt{mon}_j^{k,l}\ E[e_s]\ e_b] \sim \widetilde{E^{\mathcal{M}}}[\widetilde{e_b}] = \tilde{e}$. For the same reason, $E^{\mathcal{C}}[e_t] \sim \tilde{e}$.

Case $E^{\mathcal{C}} = E^{\mathcal{M}}[\mathtt{with}^{\mathcal{C}}\ E\ e_b]$**.**

Similar to the above. □

Lemma A.4 (Up Empty). If $E \sim \widetilde{E}$ then $\uparrow_n \widetilde{E}[\mathtt{do}\ v] = (\mathtt{let}\ y_n = v\ \mathtt{in}\ \square)$.

Proof. By induction on $E \sim \widetilde{E}$. □

Lemma A.5 (Down Empty). If $E \sim \widetilde{E}$ then $\downarrow_n \widetilde{E} = z_n$.

Proof. By induction on $E \sim \widetilde{E}$. □

Lemma A.6 (Unhandled Preservation). If $E^{\mathcal{M}}$ unhandled then it holds that $\widetilde{E^{\mathcal{M}}}$ unhandled.

Proof. By induction on $E^{\mathcal{M}} \sim \widetilde{E^{\mathcal{M}}}$. □

Lemma A.7 (Simulation Decomposition). If $e \sim \tilde{e}$ and $e = E^{\mathcal{M}}[e_s]$ for $e_s \notin \mathsf{Val}$, then exists $\widetilde{E^{\mathcal{M}}}$ and $\widetilde{e_s}$ such that $\tilde{e} = \widetilde{E^{\mathcal{M}}}[\widetilde{e_s}]$ where $E^{\mathcal{M}} \sim \widetilde{E^{\mathcal{M}}}$ and $e_s \sim \widetilde{e_s}$.

Proof. By induction on $e \sim \tilde{e}$. □

Lemma A.8 (Simulation Composition). If $E^{\mathcal{M}} \sim \widetilde{E^{\mathcal{M}}}$ and $e \sim \tilde{e}$, then $E^{\mathcal{M}}[e] \sim \widetilde{E^{\mathcal{M}}}[\tilde{e}]$.

Proof. By induction on $E^{\mathcal{M}} \sim \widetilde{E^{\mathcal{M}}}$. □

Lemma A.9 (Substitution). If $e \sim \tilde{e}$ and $v \sim \tilde{v}$ then $e[v/x] \sim \tilde{e}[\tilde{v}/x]$.

Proof. By induction on $e \sim \tilde{e}$. □

B

DRAWING LIBRARY PROPERTIES

1. A call to `get-data-from-file` must return `#f` unless the bitmap is created with `save-data-from-file` and the image is loaded successfully.
2. The `load-file` method of `bitmap%` cannot be called with bitmaps created by `make-platform-bitmap`, `make-screen-bitmap`, or the normal constructor `make-bitmap`, in `canvas%`.
3. The `get-text-extent`, `get-char-height`, and `get-char-width` methods can be called before a bitmap is installed. All other methods must be called after a bitmap is installed.
4. The method `set-argb-pixels` cannot be called if the given bitmap is produced by `make-screen-bitmap` or `make-bitmap` in `canvas%`.
5. A bitmap can be installed into at most one bitmap DC and only when it is not used by a control (as a label), a `pen%`, or a `brush%`.
6. A brush cannot be modified while installed into a DC.
7. A brush cannot be modified if it is obtained from a `brush-list%`.
8. A color cannot be modified if it is created by passing a string to `make-object` or by retrieving a color from the color database.
9. The methods `start-doc`, `start-page`, `end-page`, and `end-doc` from `dc<%>` must be called in the correct order.
10. Some methods of `dc-path%` extend an open sub-path, some close an open sub-path, and some add closed sub-paths to an existing path. These paths must all be kept consistent, e.g., if a method can only extend an open sub-path, then it cannot be called on an object where no sub-path is open.
11. A pen cannot be modified if it is obtained from a `pen-list%`.
12. A pen cannot be modified while installed into a DC.
13. If `as-eps` is set in a `post-script-dc%` object, then only one page can be created.
14. The `is-empty?` method of `region%` can only be called when associated with a DC.
15. There are no restrictions on the sequence of methods `start-doc`, `start-page`, `end-page`, and `end-doc` for `record-dc%`.

BIBLIOGRAPHY

[1] D. Ahman and A. Bauer, "Runners in Action," in *European Symposium on Programming* (*ESOP*), 2020. DOI: 10.1007/978-3-030-44914-8_2.

[2] L. Andersen, "Adding Visual and Interactive Syntax to Textual Programs," Ph.D. dissertation, Northeastern University, 2022.

[3] L. Andersen, V. St-Amour, J. Vitek, and M. Felleisen, "Feature-Specific Profiling," *Transactions on Programming Languages and Systems* (*TOPLAS*), 2018. DOI: 10.1145/3275519.

[4] L. Andersen, M. Ballantyne, and M. Felleisen, "Adding Interactive Visual Syntax to Textual Code," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2020. DOI: 10.1145/3428290.

[5] L. Andersen, M. Ballantyne, C. Moy, M. Felleisen, and S. Chang, "Mixing Visual and Textual Code," *Journal of Functional Programming* (*JFP*), 2026.

[6] O. Arden, J. Liu, and A. C. Myers, "Flow-Limited Authorization," in *Computer Security Foundations* (*CSF*), 2015. DOI: 10.1109/csf.2015.42.

[7] K. Asai, "OCaml Blockly," *Journal of Functional Programming* (*JFP*), 2025. DOI: 10.1017/s0956796825000073.

[8] J. M. Ashley and R. K. Dybvig, "An Efficient Implementation of Multiple Return Values in Scheme," in *LISP and Functional Programming* (*LFP*), 1994. DOI: 10.1145/182590.156784.

[9] H. G. Baker, "Equal Rights for Functional Objects or, The More Things Change, The More They Are The Same," *ACM SIGPLAN OOPS Messenger*, 1993. DOI: 10.1145/165593.165596.

[10] M. Ballantyne, "Building Domain-Specific Languages with Multi-Language Macros," Ph.D. dissertation, Northeastern University, 2025.

[11] F. Bañados Schwerter, R. Garcia, and É. Tanter, "A Theory of Gradual Effect Systems," in *International Conference on Functional Programming* (*ICFP*), 2014. DOI: 10.1145/2692915.2628149.

[12] H. P. Barendregt, *The Lambda Calculus*. North-Holland Publishing Co., 1981.

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier For Object-Oriented Programs," in *Formal Methods for Components and Objects (FMCO)*, 2006. DOI: 10.1007/11804192_17.

[14] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, "Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors," in *Formal Methods (FM)*, 2012. DOI: 10.1007/978-3-642-32759-9_9.

[15] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to Runtime Verification," in *Lectures on Runtime Verification*, Springer, 2018. DOI: 10.1007/978-3-319-75632-5_1.

[16] D. Basin, F. Klaedtke, and E. Zălinescu, "The MonPoly Monitoring Tool," in *Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, 2017. DOI: 10.29007/89hs.

[17] L. E. Bassham et al., "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," National Institute of Standards and Technology, Tech. Rep., 2010, https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final.

[18] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Contract Aware Components, 10 Years After," in *International Workshop on Component and Service Interoperability (WCSI)*, 2010. DOI: 10.4204/EPTCS.37.1.

[19] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," *Computer*, 1999. DOI: 10.1109/2.774917.

[20] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida, "A Theory of Design-by-Contract for Distributed Multiparty Interactions," in *International Conference on Concurrency Theory*, 2010. DOI: 10.1007/978-3-642-15375-4_12.

[21] D. Brumley and D. Boneh, "Remote Timing Attacks are Practical," *Computer Networks*, 2005. DOI: 10.1016/j.comnet.2005.01.010.

[22] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Foundations of Computer Science (FOCS)*, 2001. DOI: 10.1109/SFCS.2001.959888.

[23] R. Cartwright and M. Felleisen, "Extensible Denotational Language Specifications," in *Theoretical Aspects of Computer Software (TACS)*, 1994. DOI: 10.1007/3-540-57887-0_99.

[24] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *Formal Methods for Components and Objects*, 2006. DOI: 10.1007/11804192_16.

[25] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *Transactions on Computer Systems*, 1985. DOI: 10.1145/214451.214456.

[26] S. Chang, E. Barzilay, J. Clements, and M. Felleisen, "From Stack Traces to Lazy Rewriting Sequences," in *Implementation and Application of Functional Languages* (*IFL*), 2011. DOI: 10.1007/978-3-642-34407-7_7.

[27] F. Chen and G. Roşu, "MOP: An Efficient and Generic Runtime Verification Framework," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2007. DOI: 10.1145/1297027.1297069.

[28] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *International Conference on Functional Programming* (*ICFP*), 2000. DOI: 10.1145/351240.351266.

[29] J. Clements and M. Felleisen, "A Tail-Recursive Machine with Stack Inspection," in *Transactions on Programming Languages and Systems* (*TOPLAS*), 2004. DOI: 10.1145/1034774.1034778.

[30] J. Clements, M. Flatt, and M. Felleisen, "Modeling an Algebraic Stepper," in *European Symposium on Programming* (*ESOP*), 2001. DOI: 10.1007/3-540-45309-1_21.

[31] W. R. Cook, "On Understanding Data Abstraction, Revisited," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2009. DOI: 10.1145/1640089.1640133.

[32] M. Crestani and M. Sperber, "Experience Report: Growing Programming Languages For Beginning Students," in *International Conference on Functional Programming* (*ICFP*), 2010. DOI: 10.1145/1932681.1863576.

[33] R. Culpepper and M. Felleisen, "Fortifying Macros," in *International Conference on Functional Programming* (*ICFP*), 2010. DOI: 10.1145/1863543.1863577.

[34] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. Sipma, S. Mehrotra, and Z. Manna, "LOLA: Runtime Monitoring of Synchronous Systems," in *Temporal Representation and Reasoning* (*TIME*), 2005. DOI: 10.1109/TIME.2005.26.

[35] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The Lean Theorem Prover (System Description)," in *Conference on Automated Deduction (CADE)*, 2015. DOI: 10.1007/978-3-319-21401-6_26.

[36] M. Degen, P. Thiemann, and S. Wehr, "True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness)," in *Arbeitstagung Programmiersprachen (ATPS)*, 2009.

[37] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore, "ACL2s: "The ACL2 Sedan"," in *User Interfaces for Theorem Provers (UITP)*, 2007. DOI: 10.1016/j.entcs.2006.09.018.

[38] C. Dimoulas and M. Felleisen, "On Contract Satisfaction in a Higher-Order World," *Transactions on Programming Languages and Systems (TOPLAS)*, 2011. DOI: 10.1145/2039346.2039348.

[39] C. Dimoulas and M. Felleisen, "The Rational Programmer: Investigating Programming Language Pragmatics," *Communications of the ACM (CACM)*, 2025. DOI: 10.1145/3708981.

[40] C. Dimoulas, R. B. Findler, and M. Felleisen, "Option Contracts," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2013. DOI: 10.1145/2509136.2509548.

[41] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen, "Correct Blame for Contracts: No More Scapegoating," in *Principles of Programming Languages (POPL)*, 2011. DOI: 10.1145/1926385.1926410.

[42] C. Dimoulas, M. S. New, R. B. Findler, and M. Felleisen, "Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl)," in *International Conference on Functional Programming (ICFP)*, 2016. DOI: 10.1145/2951913.2951930.

[43] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen, "Complete Monitors for Behavioral Contracts," in *European Symposium on Programming (ESOP)*, 2012. DOI: 10.1007/978-3-642-28869-2_11.

[44] T. Disney, C. Flanagan, and J. McCarthy, "Temporal Higher-Order Contracts," in *International Conference on Functional Programming (ICFP)*, 2011. DOI: 10.1145/2034773.2034800.

[45] C. Eastlund, D. Vaillancourt, and M. Felleisen, "ACL2 for Freshmen: First Experiences," in *ACL2 Workshop*, 2007.

[46] M. Felleisen, "The Theory and Practice of First-Class Prompts," in *Principles of Programming Languages (POPL)*, 1988. DOI: 10.1145/73560.73576.

[47] M. Felleisen, "On the Expressive Power of Programming Languages," *Science of Computer Programming*, 1991. DOI: 10.1016/0167-6423(91)90036-W.

[48] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[49] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, "The Structure and Interpretation of the Computer Science Curriculum," *Journal of Functional Programming* (*JFP*), 2004. DOI: 10.1017/s0956796804005076.

[50] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, "A Functional I/O System or, Fun for Freshman Kids," in *International Conference on Functional Programming* (*ICFP*), 2009. DOI: 10.1145/1596550.1596561.

[51] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs*. MIT Press, 2018.

[52] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A Programmable Programming Language," *Communications of the ACM* (*CACM*), 2018. DOI: 10.1145/3127323.

[53] R. B. Findler and M. Blume, "Contracts as Pairs of Projections," in *Functional and Logic Programming* (*FLP*), 2006. DOI: 10.1007/11737414_16.

[54] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, "DrScheme: A Programming Environment for Scheme," *Journal of Functional Programming* (*JFP*), 2002. DOI: 10.1017/S0956796801004208.

[55] R. B. Findler and M. Felleisen, "Contract Soundness for Object-Oriented Languages," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2001. DOI: 10.1145/504311.504283.

[56] R. B. Findler and M. Felleisen, "Contracts for Higher-Order Functions," in *International Conference on Functional Programming* (*ICFP*), 2002. DOI: 10.1145/581478.581484.

[57] M. Flatt and R. K. Dybvig, "Compiler and Runtime Support for Continuation Marks," in *Programming Language Design and Implementation* (*PLDI*), 2020. DOI: 10.1145/3385412.3385981.

[58] M. Flatt, R. B. Findler, and PLT, *The Racket Guide*, https://docs.racket-lang.org/guide, 2025.

[59] M. Flatt and PLT, "Reference: Racket," PLT Design Inc., Tech. Rep. PLT-TR-2010-1, 2010, https://racket-lang.org/tr1/.

[60] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen, "Adding Delimited and Composable Control to a Production Programming Environment," in *International Conference on Functional Programming* (*ICFP*), 2007. DOI: 10.1145/1291151.1291178.

[61] M. Flatt et al., "Rhombus: A New Spin on Macros without All the Parentheses," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2023. DOI: 10.1145/3622818.

[62] T. Freund, Y. Hamdaoui, and A. Spiwack, "Union and Intersection Contracts Are Hard, Actually," in *Dynamic Languages Symposium* (*DLS*), 2021. DOI: 10.1145/3486602.3486767.

[63] T. Garnock-Jones and M. Felleisen, "Coordinated Concurrent Programming In Syndicate," in *European Symposium on Programming* (*ESOP*), 2016. DOI: 10.1007/978-3-662-49498-1_13.

[64] M. Gasbichler and M. Sperber, "Integrating User-Level Threads with Processes in Scsh," *Higher-Order and Symbolic Computation*, 2005. DOI: 10.1007/s10990-005-4879-2.

[65] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira, "Modular Session Types for Distributed Object-Oriented Programming," in *Principles of Programming Languages* (*POPL*), 2010. DOI: 10.1145/1706299.1706335.

[66] J. Gibbons and N. Wu, "Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl)," in *International Conference on Functional Programming* (*ICFP*), 2014. DOI: 10.1145/2628136.2628138.

[67] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-Based Testing In Practice," in *International Conference on Software Engineering* (*ICSE*), 2024. DOI: 10.1145/3597503.3639581.

[68] H. Goldstein, B. C. Pierce, and A. Head, "Tyche: In Situ Analysis Of Random Testing Effectiveness," in *User Interface Software and Technology* (*UIST*), 2023. DOI: 10.1145/3586182.3615788.

[69] Google, *MediaPlayer*, https://developer.android.com/reference/android/media/MediaPlayer, 2025.

[70] C. S. Gordon, "A Generic Approach to Flow-Sensitive Polymorphic Effects," in *European Conference on Object-Oriented Programming* (*ECOOP*), 2017. DOI: 10.4230/LIPIcs.ECOOP.2017.13.

[71] B. Greenman, A. Takikawa, M. S. New, D. Feltey, R. B. Findler, J. Vitek, and M. Felleisen, "How to Evaluate the Performance of Gradual Typing Systems," *Journal of Functional Programming* (*JFP*), 2019. DOI: 10.1017/S0956796818000217.

[72] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi, "Relationally-Parametric Polymorphic Contracts," in *Dynamic Languages Symposium* (*DLS*), 2007. DOI: 10.1145/1297081.1297089.

[73] A. Harsányi, *Using `current-process-memory` with `'cumulative` Argument*, https://groups.google.com/g/racket-users/c/xqOY8uevGzE/m/mBtHeq2jAwAJ, 2020.

[74] K. Havelund, G. Reger, D. Thoma, and E. Zălinescu, "Monitoring Events that Carry Data," in *Lectures on Runtime Verification*, Springer, 2018. DOI: 10.1007/978-3-319-75632-5_3.

[75] G. Hendershott, *Racket Mode*, https://racket-mode.com, 2025.

[76] R. Hickey, "A History of Clojure," in *History of Programming Languages (HOPL)*, 2020. DOI: 10.1145/3386321.

[77] D. Hillerström and S. Lindley, "Shallow Effect Handlers," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2018. DOI: 10.1007/978-3-030-02768-1_22.

[78] R. Hinze, J. Jeuring, and A. Löh, "Typed Contracts for Functional Programming," in *Functional and Logic Programming (FLP)*, 2006. DOI: 10.1007/11737414_15.

[79] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language Primitives and Type Discipline for Structured Communication-Based Programming," in *European Symposium on Programming (ESOP)*, 1998. DOI: 10.1007/BFb0053567.

[80] C. Jaspan and J. Aldrich, "Checking Framework Interactions with Relationships," in *European Conference on Object-Oriented Programming (ECOOP)*, 2009. DOI: 10.1007/978-3-642-03013-0_3.

[81] D. Jin, P. O. Meredith, D. Griffith, and G. Rosu, "Garbage Collection for Monitoring Parametric Properties," in *Programming Language Design and Implementation (PLDI)*, 2011. DOI: 10.1145/1993498.1993547.

[82] S. Juhošová, A. Zaidman, and J. Cockx, "Pinpointing the Learning Obstacles of an Interactive Theorem Prover," in *International Conference on Program Comprehension (ICPC)*, 2025. DOI: 10.1109/ICPC66645.2025.00024.

[83] M. Keil and P. Thiemann, "Blame Assignment for Higher-Order Contracts with Intersection and Union," in *International Conference on Functional Programming (ICFP)*, 2015. DOI: 10.1145/2784731.2784737.

[84] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *European Conference on Object-Oriented Programming (ECOOP)*, 2001. DOI: 10.1007/3-540-45337-7_18.

[85] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *European Conference on Object-Oriented Programming*, 1997. DOI: 10.1007/BFb0053381.

[86] G. A. Kildall, "A Unified Approach To Global Program Optimization," in *Principles of Programming Languages* (*POPL*), 1973. DOI: 10.1145/512927.512945.

[87] S. Kleene, "Representation of Events in Nerve Nets and Finite Automata," in *Automata Studies*, Princeton University Press, 1956. DOI: 10.1515/9781400882618-002.

[88] C. Klein, M. Flatt, and R. B. Findler, "Random Testing For Higher-Order, Stateful Programs," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2010. DOI: 10.1145/1869459.1869505.

[89] D. E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison-Wesley Publishing Company, 1969.

[90] E. Koskinen and T. Terauchi, "Local Temporal Reasoning," in *Logic in Computer Science* (*LICS*), 2014. DOI: 10.1145/2603088.2603138.

[91] L. Lazarek, A. King, S. Sundar, R. B. Findler, and C. Dimoulas, "Does Blame Shifting Work?" In *Principles of Programming Languages* (*POPL*), 2020. DOI: 10.1145/3371133.

[92] C. S. Lee, N. D. Jones, and A. M. Ben-Amram, "The Size-Change Principle for Program Termination," in *Principles of Programming Languages* (*POPL*), 2001. DOI: 10.1145/360204.360210.

[93] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications," in *Automated Software Engineering* (*ASE*), 2016. DOI: 10.1145/2970276.2970356.

[94] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM* (*CACM*), 2009. DOI: 10.1145/1538788.1538814.

[95] M. Leucker and C. Schallhart, "A Brief Account of Runtime Verification," *The Journal of Logic and Algebraic Programming*, 2009. DOI: 10.1016/j.jlap.2008.08.004.

[96] O. Lichtenstein, A. Pnueli, and L. Zuck, "The Glory of the Past," in *Logics of Programs*, 1985. DOI: 10.1007/3-540-15648-8_16.

[97] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," in *Technical Symposium on Computer Science Education* (*SIGCSE*), 2011. DOI: 10.1145/1953163.1953308.

[98] J. Matthews and A. Ahmed, "Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!" In *European Symposium on Programming* (*ESOP*), 2008. DOI: 10.1007/978-3-540-78739-6_2.

[99] M. McIlroy, "Mass Produced Software Components," in *NATO Software Engineering Conference*, 1968.

[100] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1988.

[101] B. Meyer, "Applying 'Design by Contract'," *Computer*, 1992. DOI: 10.1109/2.161279.

[102] J. C. Mitchell and G. D. Plotkin, "Abstract Types Have Existential Type," *Transactions on Programming Languages and Systems (TOPLAS)*, 1988. DOI: 10.1145/44501.45065.

[103] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, 1965. DOI: 10.1109/JPROC.1998.658762.

[104] S. Moore, C. Dimoulas, R. B. Findler, M. Flatt, and S. Chong, "Extensible Access Control with Authorization Contracts," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016. DOI: 10.1145/2983990.2984021.

[105] J. H. Morris, "Protection in Programming Languages," *Communications of the ACM (CACM)*, 1973. DOI: 10.1145/361932.361937.

[106] C. Moy, "Faster, Simpler Red-Black Trees," in *Trends in Functional Programming (TFP)*, 2023. DOI: 10.1007/978-3-031-38938-2_3.

[107] C. Moy, C. Dimoulas, and M. Felleisen, "Effectful Software Contracts," in *Principles of Programming Languages (POPL)*, 2024. DOI: 10.1145/3632930.

[108] C. Moy and M. Felleisen, "Trace Contracts," *Journal of Functional Programming*, 2023. DOI: 10.1017/S0956796823000096.

[109] C. Moy, R. Jung, and M. Felleisen, "Contract Systems Need Domain-Specific Notations," in *European Conference on Object-Oriented Programming (ECOOP)*, 2025. DOI: 10.4230/LIPIcs.ECOOP.2025.42.

[110] C. Moy and D. Patterson, "Teaching Software Specification (Experience Report)," in *International Conference on Functional Programming (ICFP)*, 2025. DOI: 10.1145/3747533.

[111] P. C. Nguyễn, T. Gilray, S. Tobin-Hochstadt, and D. Van Horn, "Soft Contract Verification for Higher-Order Stateful Programs," in *Principles of Programming Languages (POPL)*, 2018. DOI: 10.1145/3158139.

[112] P. C. Nguyễn, T. Gilray, S. Tobin-Hochstadt, and D. Van Horn, "Size-Change Termination as a Contract," in *Programming Language Design and Implementation (PLDI)*, 2019. DOI: 10.1145/3325984.

[113] P. C. Nguyễn, S. Tobin-Hochstadt, and D. Van Horn, "Higher Order Symbolic Execution for Contract Verification and Refutation," *Journal of Functional Programming* (*JFP*), 2017. DOI: 10.1017/S0956796816000216.

[114] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer Verlag, 2005.

[115] C. Okasaki, "Red-Black Trees In A Functional Setting," *Journal of Functional Programming* (*JFP*), 1999. DOI: 10.1017/s0956796899003494.

[116] Oracle, *Iterator*(*Java SE 23 & JDK 23*), https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Iterator.html, 2024.

[117] Oracle, *Map* (*Java SE 23 & JDK 23*), https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Map.html, 2024.

[118] Z. Owens, "Contract Monitoring as an Effect," in *Higher-Order Programming with Effects* (*HOPE*), 2012.

[119] R. Page, C. Eastlund, and M. Felleisen, "Functional Programming and Theorem Proving for Undergraduates: A Progress Report," in *Functional and Declarative Programming in Education* (*FDPE*), 2008. DOI: 10.1145/1411260.1411264.

[120] D. L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM* (*CACM*), 1972. DOI: 10.1145/355602.361309.

[121] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM* (*CACM*), 1972. DOI: 10.1145/361598.361623.

[122] G. Plotkin, "Call-by-Name, Call-by-Value and the λ-Calculus," *Theoretical Computer Science*, 1975. DOI: 10.1016/0304-3975(75)90017-1.

[123] G. Plotkin and M. Pretnar, "Handlers of Algebraic Effects," in *European Symposium on Programming* (*ESOP*), 2009. DOI: 10.1007/978-3-642-00590-9_7.

[124] M. Pretnar, "An Introduction to Algebraic Effects and Handlers," in *Mathematical Foundations of Programming Semantics* (*MFPS*), 2015. DOI: 10.1016/j.entcs.2015.12.003.

[125] R. Pucella and J. A. Tov, "Haskell Session Types with (Almost) No Class," in *Haskell Symposium*, 2008. DOI: 10.1145/1411286.1411290.

[126] M. O. Rabin and D. Scott, "Finite Automata and their Decision Problems," *IBM Journal of Research and Development*, 1959. DOI: 10.1147/rd.32.0114.

[127] R. L. Rivest, A. Shamir, and A. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," in *Communications of the ACM* (*CACM*), 1978. DOI: 10.1145/359340.359342.

[128] U. Sammapun and O. Sokolsky, "Regular Expressions for Run-Time Verification," in *Automated Technology for Verification and Analysis* (*ATVA*), 2003.

[129] C. Scholliers, É. Tanter, and W. De Meuter, "Computational Contracts," *Science of Computer Programming*, 2015. DOI: 10.1016/j.scico.2013.09.005.

[130] J. Seward and N. Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision," in *USENIX Annual Technical Conference* (*ATC*), 2005.

[131] A. E. Shinnar, "Safe and Effective Contracts," Ph.D. dissertation, Harvard University, 2011.

[132] G. L. Steele, *Common Lisp the Language*. Digital Press, 1990.

[133] T. S. Strickland, C. Dimoulas, A. Takikawa, and M. Felleisen, "Contracts For First-Class Classes," *Transactions on Programming Languages and Systems* (*TOPLAS*), 2013. DOI: 10.1145/2518189.

[134] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt, "Chaperones and Impersonators: Run-Time Support for Reasonable Interposition," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2012. DOI: 10.1145/2384616.2384685.

[135] R. E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *Transactions on Software Engineering* (*TSE*), 1986. DOI: 10.1109/TSE.1986.6312929.

[136] J. Swaine, B. Fetscher, V. St-Amour, R. B. Findler, and M. Flatt, "Seeing the Futures: Profiling Shared-Memory Parallel Racket," in *Functional High-Performance Computing* (*FHPC*), 2012. DOI: 10.1145/2364474.2364485.

[137] J. Swaine, K. Tew, P. A. Dinda, R. B. Findler, and M. Flatt, "Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2010. DOI: 10.1145/1869459.1869507.

[138] C. Swords, "A Unified Characterization of Runtime Verification Systems as Patterns of Communication," Ph.D. dissertation, Indiana University, 2019.

[139] R. Tate, "The Sequential Semantics of Producer Effect Systems," in *Principles of Programming Languages* (*POPL*), 2013. DOI: 10.1145/2429069.2429074.

[140] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as Libraries," in *Programming Language Design and Implementation* (*PLDI*), 2011. DOI: 10.1145/1993316.1993514.

[141] B. Toninho, L. Caires, and F. Pfenning, "Dependent Session Types via Intuitionistic Linear Type Theory," in *Principles and Practice of Declarative Programming* (*PPDP*), 2011. DOI: 10.1145/2003476.2003499.

[142] N. Toronto and A. Harsányi, *Plot: Graph Plotting*, https://docs.racket-lang.org/plot/index.html, 2011.

[143] J. A. Tov and R. Pucella, "Stateful Contracts for Affine Types," in *European Symposium on Programming* (*ESOP*), 2010. DOI: 10.1007/978-3-642-11957-6_29.

[144] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, 1937.

[145] C. Uehara and K. Asai, "Cross Validation of the Universe Teachpack of Racket in OCaml," in *Trends in Functional Programming in Education* (*TFPIE*), 2015.

[146] T. Van Cutsem and M. S. Miller, "Proxies: Design Principles for Robust Object-oriented Intercession APIs," in *Dynamic Languages Symposium* (*DLS*), 2010. DOI: 10.1145/1869631.1869638.

[147] J. Williams, J. G. Morris, and P. Wadler, "The Root Cause of Blame: Contracts for Intersection and Union Types," in *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), 2018. DOI: 10.1145/3276504.

[148] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich, "Gradual Typestate," in *European Conference on Object-Oriented Programming* (*ECOOP*), 2011. DOI: 10.1007/978-3-642-22655-7_22.

[149] C. Xiang, Z. Qi, and W. Binder, "Flexible and Extensible Runtime Verification for Java," *International Journal of Software Engineering and Knowledge Engineering*, 2015. DOI: 10.1142/S0218194015400343.

[150] D. N. Xu, S. Peyton Jones, and K. Claessen, "Static Contract Checking for Haskell," in *Principles of Programming Languages* (*POPL*), 2009. DOI: 10.1145/1480881.1480889.