A UNIFIED CHARACTERIZATION OF RUNTIME VERIFICATION SYSTEMS AS PATTERNS OF COMMUNICATION

Cameron Swords

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics, Computing, and Engineering
Indiana University
February 2019

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Amr Sabry, Ph.D.

Lawrence S. Moss, Ph.D.

Jeremy Siek, Ph.D.

Sam Tobin-Hochstadt, Ph.D.

03/01/2018

Acknowledgements

Writing a dissertation is, in a visceral sense, not a one-person task. My quest to discover and write about the ideas in this thesis was, at times, arduous, encouraging, jovial, and dismal, and numerous people helped me along the way, in nearly every way imaginable.

It started, I suppose, when my wife Rebecca Swords applied to Indiana University. We both planned to pursue higher degrees, and she graduated ahead of me and began attending IU. While there, she took Dan Friedman's course on programming languages, and began studying programming languages as a field. When I visited her at IU that fall, I saw two lectures: in the first, Dan Friedman and Will Byrd magicked together a type inferencer in miniKanren in no more than 20 lines; in the second, Sam Tobin-Hochstadt, then a post-doctoral researcher at Northeastern University visiting IU for the week, presented Typed Racket. Between the two, programming language research piqued my interest.

When I followed Rebecca to IU in the fall of 2011, I immersed myself in the PL research there, signing up for Dan's programming language course and attending the weekly PL Wonks talks[1], where I got to know much of the IU PL cohort (past and current). At the end of the semester, Dan recruited me as his new lead teaching assistant, a position I likely would have held for significantly longer if Amr Sabry hadn't offered me a research assistantship the next semester to explore software contract verification[2]. As is evidenced by this document, that work ultimately yielded this dissertation.

Amr allowed me to start small, encouraging me to read papers and learn the field of software contracts while explaining everything he could, pushing me toward the driving narrative of "contracts as effects" until it petered out, leaving us with the fundamental question this dissertation aims to answer: what is contract verification, and what, exactly, is its role in program execution? That was, in a real sense, the turning point in our work: the very moment we decided contracts *must* be separate evaluators, it became clear that their patterns of interaction *were* their differences, leading to our current, process-based approach to modeling runtime verification. Even so, the ideas presented in this work are the result of years of design, refinement, and exploration, all guided by Amr's strict insistence on precise semantics, expressive formalisms, and coffee breaks. Beyond my

[1]I also decided these meetings would benefit from cookies, so that first semester I made a batch every Thursday night (or sometimes Friday, during the day) to bring. It still tickles me that this small contribution has become a long-standing tradition.

[2]As Jason Hemann may tell you, however, I stayed on as an informal member of Dan's staff under the title "*wartime consigliere*" for a good while after Spring 2012.

research topic, Amr has also been an excellent adviser and teacher, explaining, discussing, and offering a whole cornucopia of interesting ideas and advice ranging from algebraic effects to quantum computation, but also everything from academic politicking to educational theory. I am fortunate that I came to work for Amr all those years ago, and I am not certain how things would have turned out if not for that good bit of luck. All things considered, I still hope to see his master plan for taking over the (programming) world come to fruition.

Beyond Amr, I had an excellent committee for this work, including Sam Tobin-Hochstadt, a clever man who seems to have read every programming language paper ever written, and whose brilliant insights and discussions in our weekly meetings guided immense refinements to the ideas presented here (and a goodly number of jokes); Jeremy Siek, an IU professor-née-alumnus who has an excellent knack for poking at the finer details of semantics, ensuring their correctness; and Larry Moss, whose passion for mathematics and logic has enriched my own.

In addition to my committee, I worked with a number of additional co-authors at IU, including Dan Friedman, who taught me a great deal about how to indent Scheme (amongst other things); Jason Hemann, a fellow Texas expatriate who shared my sorrow for Indiana's lack of breakfast tacos, and who always seemed willing to convince me I knew what I was doing, even when it was clear he was the one who did; and, last but certainly not least, Mike Vitousek, a good friend and excellent collaborator who I worked with to produce many results, including an academic publication, a series of beverages requiring ABV calculations, a handful of tabletop games, and, in one particularly strange series of situations, a metaphysical discussion of the totemic *Raccoon*.

Aside from my published co-authors, countless other people collaborated by being friends, coworkers, confidants, and more. It seems fairest to list them roughly chronologically:

- Lindsey Kuper and Alex Rudnick, who always shared unending generosity, intensive discussions, and kind enthusiasm;
- Aaron Todd, and all the C we wrote together in Operating Systems;
- Cassandra Sparks, who provided thought-provoking challenges and *progress & preservation*;
- Will Byrd, a jolly good fellow who contributed, amongst other things, the numerous, late-night conversations about how to succeed in graduate school that set my initial trajectory;
- Chris Frisz, who taught me about the *important business* of graduate school;
- Claire Alvis, who insisted that I should be held to a higher standard, not a lower one, as her student *and* friend;

- Kyle Carter, an exemplary friend and fellow student at the start of my dissertation, and, through some strange set of events, also the end—*Halleloo!*;
- Andy Keep, a wizard and drinking buddy of the highest caliber;
- Jaime Guerrero, an excellent friend and likely the suavest gentleman I'll ever know;
- Spenser Bauman, whose shared love of quoting Futurama and over-enthusiasm for nerd humor leads me to say: *wiggy wam wam wazzle!*—also, thanks for the Thin Mints;
- Ethan Swords, my brother, compatriot, and closest friend, whom I was fortunate enough to share some of my graduate school experience with;
- Andre Kuhlenschmidt, a fellow compiler-hacker whose seemingly-endless quest for knowledge is encouraging and awing at the same time (and his wife, Laura, for putting up with us);
- Chris Wailes, who seems convinced enough that the world should bend to his will that it just may;
- Tori Lewis, who helps the rest of us remember sanity by straying from it;
- and Andrew Kent, who makes grad school look far easier than anyone should be able to.

This is, of course, not an exhaustive list: others include the additional members of our long-running tabletop RPG crew, which variously included, at different times, Peter Fogg, Matt Heimerdinger, Rajan Walia, Vikraman Choudhury, and Sarah Spall; the many IU undergraduates who became friends (and subsequently graduated), including Tim Zakian, Michael DeWitt, and Joshua Cox; IU students, alumni, and associated, including Eric Holk, Ambrose Bonnaire-Sergeant, Michael Vollmer, Marcela Poffald, David Christiansen, Caner Derici, Praveen Narayanan, Ryan Scott, Buddhika Chamith, and Aaron Hsu, all excellent people who I wish I knew better than I do; Aaron Turon, Alex Crichton, Nick Cameron, Niko Matsakis, and all the other amazing developers working on Rust at Mozilla Research; the many, many reviewers that rejected my work with insightful feedback, ultimately leading to the refinements and corrections that produced the semantics in Chapters 4–6; my parents, for raising me; and many others who each have helped by discussing ideas, providing advice and insights, and, in many cases, just sharing good times.

Finally, I must thank my ultimate collaborator: my wife, Rebecca Swords. At this point, she has discussed more of my ideas and encouraged more of my research (and also potentially proof-read more of my papers) than anyone else in my life. This dissertation might have been written without some subset of the people named above, but I can say with certainty that it would not have been possible without her.

Cameron Swords

A UNIFIED CHARACTERIZATION OF RUNTIME VERIFICATION SYSTEMS AS PATTERNS OF COMMUNICATION

Runtime verification, as a field, provides tools to describe how programs should behave during execution, allowing programmers to inspect and enforce properties about their code at runtime. This field has resulted in a wide variety of approaches to inspecting and ensuring correct behavior, from instrumenting individual values in order to verify global program behavior to writing ubiquitous predicates that are checked over the course of execution. Although each verification approach has its own merits, each has also historically required ground-up development of an entire framework to support such verification.

In this work, we start with software contracts as a basic unit of runtime verification, exploring the myriad of approaches to enforcing them—ranging from the straightforward pre-condition and post-condition verification of Eiffel to lazy, optional, and parallel enforcement strategies—and present a unified approach to understanding them, while also opening the door to as-yet-undiscovered strategies. By observing that contracts are fundamentally about communication between a program and a monitor, we reframe contract checking as communication between concurrent processes. This brings out the underlying relations between widely-studied verification strategies, including strict and lazy enforcement as well as concurrent approaches, including new contracts and strategies. We introduce a concurrent core calculus (with proofs of type safety), show how we may encode each of these contract verification strategies in it, and demonstrate a proof (via simulation) of correctness for one such encoding.

After demonstrating this unified framework for contract verification strategies, we extend our verification framework with meta-strategy operators—strategy-level operations that take one or more strategies (plus additional arguments) as input and produce new verification behaviors—and use these extended behavioral constructs to optimize contract enforcement, reason about global program behavior, and even perform runtime instrumentation, ultimately developing multiple runtime verification behaviors using our communication-based view of interaction.

Finally, we introduce an extensible, Clojure-based implementation of our framework, demonstrating how our approach fits into a modern programming language by recreating our contract verification and general runtime verification examples.

Contents

List of Figures

List of Theorems

CHAPTER 1

Introduction

Runtime verification is an execution-based approach to program correctness in which verification code runs alongside a program at *execution time* to inspect the program, ensuring it adheres to a given specification or description of behavior. It allows programmers to express and verify programmatic properties in terms of runtime operations and correct values flowing through the program, forgoing the need for compile-time proofs of behavior.

To illustrate this trade-off, consider ensuring that a program *always* checks that an iterator has additional values before retrieving a value. To verify this property at compile time, we must either keep track of each iterator's state at the type level (e.g., by using a type parameterized by the state of a given iterator as iterator (hasNext) and iterator (unknown), raising a type error if the program retrieves a value in the latter case) or extend the compiler itself to reason about correct iterator usage. Alternatively, we can machine the runtime itself to programmatically enforce correct iterator use by attaching an additional Boolean flag to each, setting it to true when we ask if the iterator has another value and false when we retrieve said value, throwing an error if we retrieve a value when the iterator has the false flag. This runtime-based approach allows us to programmatically ensure each individual run is correct without encoding correctness as a compile-time artifact.

Moreover, in languages without rich type systems (e.g., Python, Ruby, Racket, and Clojure), runtime-based verification can stand in for compile-time types, ensuring each function behaves correctly. The functional programming community, in particular, have adopted the notion of runtime property verification in the form of *software contracts* to ensure program correctness. In this work, we will start with software contracts in their many forms, expanding and generalizing software contract verification and, ultimately, building a general runtime verification and inspection system.

Contracts. Originally presented as part of the Eiffel language [66], software contracts describe a mechanism for specifying local program correctness at loop, function, and module boundaries, allowing programmers to create complete, runtime-checked program specifications from individual components. This piecewise construction of specifications suggests that software contracts are an ideal foundation for building a framework for runtime verification, and thus we use it as our starting point.

Findler and Felleisen [36] brought contracts to higher-order, functional programming (and, in particular, the Racket programming language [39]), introducing and defining a semantic core for contract verification: in their simplest form, programmers form contracts from writing predicates (i.e., functions that return Boolean values) that verify properties on values flowing through the program. When a predicate returns a false value, the program terminates and reports the incorrect value to the programmer.

Programming, however, is more complex: programs utilize higher-order functions, effectful operations, massive data structures, lazy evaluation mechanisms, and more, and contract systems must address this added complexity. For higher-order functions, we must delay contract enforcement [36]; effectful operations may affect values that were previously checked; massive data structures may be prohibitively expensive to inspect; and over-evaluating input may change program behavior.

In addition to these specific solutions, researchers have proposed myriad contract enforcement strategies in response to these rising complexities, including lazy monitors [15, 22], concurrent monitoring systems [26], optional enforcement [29], and probabilistic contract verification [50, 68]. Each of these solutions solve some, but not every, programmer concern in contract verification. Moreover, these systems often appear as isolated, incompatible alternatives, fixing the evaluator interaction pattern in the host language, providing a *single* enforcement strategy to the programmer.

Despite appearances, however, this cornucopia of verification systems share a common core: checking that a given program fragment satisfies a contract requires executing some verification code, and this execution is fundamentally distinct from the execution the original program fragment. If we separate these two code fragments, we can view the verification portion of the program as a separate process, allowing each program to proceed independently and synchronize at specific points. This separation leads to a critical insight: if we preserve this distinction, these variations on verification correspond to variations on evaluator interactions, and we may directly encode each into a unified system, allowing programmers to vary monitor behavior on a *per-contract* basis, choosing the appropriate behavior for each contract and, ultimately, extending contract behavior to encompass other runtime verification systems. Moreover, this separation allows us to model other runtime verification systems, such as ensuring a program adheres to behavioral models (such as ensuring a programmer invokes `hasNext` on an iterator before retrieving the next element) and function profiling, in the same framework.

1.1. Software Contracts

Software contracts are pervasive in modern functional programming, from Racket's ubiquitous usage to Liquid Haskell and Typed Racket's adoption of refinement types. In their usual presentation, contract systems define a core calculus and extend it with a particular flavor of software contract verification, typically including the following forms:

$$\begin{array}{rcl} E & := & x \mid V \mid E\ E \mid \textsf{if}\ E\ \textsf{then}\ E\ \textsf{else}\ E \mid E\ \text{binop}\ E \\ & \mid & (E, E) \mid \textsf{fst}\ E \mid \textsf{snd}\ E \mid \textsf{raise} \mid \textbf{mon}\ E\ E \\ \\ V & := & \lambda\, x.\ E \mid n \mid \textsf{true} \mid \textsf{false} \mid (V, V) \mid \textit{string} \end{array}$$

Most of these operations are standard: we have variables, values, application, branching, infix binary operations, pairs, and errors[1]. The monitor form, included as the last form of E, performs runtime monitor installation. These monitors are most typically defined via *contract combinators*, specialized constructors that accept functions or other contracts as input and produce contracts. The simplest combinator is pred/c, the predicate (or flat) contract combinator whose input is a predicate and whose output is a contract that enforces it. We can use this to ensure a value is a natural number:

$$\textsf{nat/c} \ := \ \textsf{pred/c}\ (\lambda\, x.\ x \geq 0) \tag{1}$$

To evaluate programs in this calculus, we also need dynamic semantics to describe how **mon** should proceed[2]. We will build this piecemeal, matching on the contract by combinator:

$$\textbf{mon}\ (\textsf{pred/c}\ F)\ V \rightarrow \textsf{if}\ F\ V\ \textsf{then}\ V\ \textsf{else raise} \tag{2}$$

When we monitor a predicate contract, we check if the predicate holds for the provided value. If it does, we return that value. If it does not, we raise an error. For example, we can check nat/c:

[1]For now, we treat errors as 'unit'-raised errors. Later we will extend them to provide more information about the error that occurred.
[2]We elide the other semantics and refer the interested reader to Types and Programming Languages for their implementation [73].

Example 1.1 (Using a predicate contract).

$$\begin{array}{ll} & \textbf{mon}\ \mathsf{nat/c}\ 5 \\ \to & \mathsf{if}\ (\lambda\ x.\ x \geq 0)\ 5\ \mathsf{then}\ 5\ \mathsf{else\ raise} \\ \to & \mathsf{if}\ 5 \geq 0\ \mathsf{then}\ 5\ \mathsf{else\ raise} \\ \to & \mathsf{if\ true\ then}\ 5\ \mathsf{else\ raise} \\ \to & 5 \end{array}$$

We can also define combinators over structures, such as pair/c E E for pair contracts:

$$\textbf{mon}\ (\mathsf{pair/c}\ C_1\ C_2)\ V \to (\textbf{mon}\ C_1\ (\mathsf{fst}\ V), \textbf{mon}\ C_2\ (\mathsf{snd}\ V)) \tag{3}$$

This contract combinator takes two *sub-contracts* and applies the first to the first element of the pair and the second to the second element of the pair. For example, we might check if we have a pair of natural numbers:

Example 1.2 (Using a pair contract).

$$\begin{array}{ll} & \mathsf{fst}\ (\textbf{mon}\ (\mathsf{pair/c}\ \mathsf{nat/c}\ \mathsf{nat/c})\ (5,6)) \\ \to & \mathsf{fst}\ (\textbf{mon}\ \mathsf{nat/c}\ (\mathsf{fst}\ (5,6))\ ,\ \textbf{mon}\ \mathsf{nat/c}\ (\mathsf{snd}\ (5,6))) \\ \to^* & \mathsf{fst}\ (\textbf{mon}\ \mathsf{nat/c}\ 5\qquad\qquad ,\ \textbf{mon}\ \mathsf{nat/c}\ (\mathsf{snd}\ (5,6))) \\ \to^* & \mathsf{fst}\ (5,\ \textbf{mon}\ \mathsf{nat/c}\ 6) \\ \to^* & \mathsf{fst}\ (5,6) \\ \to^* & 5 \end{array}$$

The last combinator we will introduce for now is fun/c, the *function contract combinator*:

$$\textbf{mon}\ (\mathsf{fun/c}\ C_1\ C_2)\ F \to \lambda\ x.\ \textbf{mon}\ C_2\ (F\ (\textbf{mon}\ C_1\ x)) \tag{4}$$

This contract ensures that each input to the function satisfies the specification given by C_1 (called the *pre-condition*) and each function output satisfies the specification given by C_2 (called the *post-condition*). Unlike the previous combinators, this monitoring results in a pair of *suspended monitors*: unlike predicate and pair contracts, a function contract does not have all of the information necessary to verify its input and output. Rather than attempting to statically verify that every path through the program satisfies these contracts, we check each input and output that occurs over the course of the program, verifying each input invocation site and output result. To demonstrate this mechanism, consider ensuring a procedure works over natural numbers (i.e., takes natural numbers as inputs and produces natural numbers as outputs):

Example 1.3 (Using a function contract).

$$\begin{array}{ll} & (\textbf{mon}\ (\mathsf{fun/c}\ \mathsf{nat/c}\ \mathsf{nat/c})\ (\lambda\ n.\ 1))\ 5 \\ \rightarrow & (\lambda\ x.\ \textbf{mon}\ \mathsf{nat/c}\ ((\lambda\ n.\ 1)\ (\textbf{mon}\ \mathsf{nat/c}\ x)))\ 5 \\ \rightarrow & \textbf{mon}\ \mathsf{nat/c}\ ((\lambda\ n.\ 1)\ (\textbf{mon}\ \mathsf{nat/c}\ 5)) \\ \rightarrow^* & \textbf{mon}\ \mathsf{nat/c}\ ((\lambda\ n.\ 1)\ 5) \\ \rightarrow^* & \textbf{mon}\ \mathsf{nat/c}\ 1 \\ \rightarrow^* & 1 \end{array}$$

1.2. Variations on Contract Systems

As discussed above, there are myriad variations on the eager-style contract verification system we present above, each of which appear in the software contract verification literature as modifications to the calculus in the previous section. In this section, we introduce three such variations, briefly describing their verification behavior and contrasting it with the eager verification mechanism above (and postponing further details until Chapter 2). We focus on these strategies in particular due to their broad utility, straight-forward explanations, and proliferation in the literature.

Semi-Eager Verification. The software contract system presented in §1.1 exhibits potentially surprising behavior: Example 1.2 and Example 1.3 both perform contract checks for values that do not impact in the final program result. In Example 1.2, the second element of the pair, 6, is never used, and, similarly, the function input 5 is never used in Example 1.3. If these values had *not* been natural numbers, however, they would each have raised errors in the program. In programs where over-evaluation might produce divergence or performance degradation, this can make contracts prohibitively expensive (such as traversing a binary tree to inspect each element during element insertion) and even impossible to check (such as verifying that a stream contains only of natural numbers).

Hinze et al. [52] propose a lazier monitoring mechanism to address these issues: unlike the software contract system in §1.1, this system only checks contracts *for values used in the final program result.* To demonstrate this behavior, consider the following examples:

Example 1.4 (Semi-eager contract enforcement).

$$\mathsf{fst}\ (\textbf{mon}\ (\mathsf{pair/c}\ \mathsf{nat/c}\ \mathsf{nat/c})\ (5, \text{-}1)) \rightarrow^* 5 \tag{5}$$

$$(\textbf{mon}\ (\mathsf{fun/c}\ \mathsf{nat/c}\ \mathsf{nat/c})\ (\lambda\ n.\ 1))\ \text{-}1 \rightarrow^* 1 \tag{6}$$

Because -1 is not used to determine the final program result in either of these examples, the programs terminate without error.

Promise-Based Verification. A second variation of software contract enforcement utilizes computational promise behavior: any time a monitor enforces a contract, the programmer receives a promise to the contract result, and they can retrieve it (or not) as they see fit. For example, a user might initialize a contract, perform a secondary computation, and then verify the contract result.

Example 1.5 (Promise contract verification).

$$\begin{array}{lll}
 & \textit{User Process} & \textit{Monitoring Process} \\
 & \mathsf{let}\ x = \mathbf{mon}\ \mathsf{nat/c}\ \text{-1} & \\
 & \quad k = \mathsf{gen\text{-}new\text{-}encryption\text{-}key} & \\
 & \mathsf{in}\ \mathsf{encrypt}\ k\ (\mathsf{force}\ x) & \\
\rightarrow & \mathsf{let}\ x = \langle \mathsf{box} \rangle & \textit{update-box}\ (\mathsf{if}\ (\lambda\ x.\ x \geq 0)\ \text{-1}\ \mathsf{then}\ \text{-1}\ \mathsf{else}\ \textit{error}) \\
 & \quad k = \mathsf{gen\text{-}new\text{-}encryption\text{-}key} & \\
 & \mathsf{in}\ \mathsf{encrypt}\ k\ (\mathsf{force}\ x) & \\
\rightarrow^{*} & \mathsf{let}\ x = \langle \mathsf{box} \rangle & \textit{update-box error} \\
 & \quad k = \textit{encryption-key} & \\
 & \mathsf{in}\ \mathsf{encrypt}\ k\ (\mathsf{force}\ x) & \\
\rightarrow^{*} & \mathsf{encrypt}\ \textit{encryption-key}\ (\mathsf{force}\ \langle \mathsf{box} \rangle) &
\end{array}$$

This allows programmers to utilize multi-processor architectures to provide performance improvements over the interleaving monitoring system presented in §1.1. More importantly, it describes a variation of *how* to verify contracts: we may check them in a separate process while the program continues. It also allows programmers to control *when* and *if* they get the contract result, where unforced promises will not produce errors in the user process.

Concurrent Verification. A third variation, originally proposed by Dimoulas et al. [26] as *Future Contracts*, performs contract verification in parallel for predicate contracts, synchronizing with the program at "effectful" points (such as input/output events) to report any violations. For example, consider a program that checks a predicate contract and synchronizes with the monitor at print operations:

Example 1.6 (Concurrent contract verification).

$$\begin{array}{llcl}
 & \textit{User Process} & & \textit{Monitoring Process} \\
 & \mathsf{let}\ x = \mathbf{mon}\ \mathsf{nat/c}\ \text{-}1\ \mathsf{in}\ \mathsf{display}\ (\mathsf{add}_1\ x) & & \\
\rightarrow & \mathsf{let}\ x = \text{-}1\ \mathsf{in}\ \mathsf{display}\ (\mathsf{add}_1\ x) & & \mathsf{if}\ (\lambda\ x.\ x \geq 0)\ \text{-}1\ \mathsf{then}\ \textit{done}\ \mathsf{else}\ \textit{report error} \\
\rightarrow & \mathsf{display}\ (\mathsf{add}_1\ \text{-}1) & & \mathsf{if}\ \text{-}1 \geq 0\ \mathsf{then}\ \textit{done}\ \mathsf{else}\ \textit{report error} \\
\rightarrow & \mathsf{display}\ 0 & & \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ \textit{done}\ \mathsf{else}\ \textit{report error} \\
\rightarrow^* & \mathsf{seq}\ \mathsf{sync}\ (\mathsf{print}\ 0) & \Longleftrightarrow & \textit{raise error} \\
\rightarrow^* & \mathsf{raise} & &
\end{array}$$

The initial program continues its computation until synchronization, and then the error occurs. In general, the monitoring process manages a queue of pending contracts, reporting any violations during synchronization. This allows programmers to utilize parallelism without explicitly managing verification results.

Other Variations. These are not the only variations; for example we can also verify contracts in each of the following ways:

- lazy contract verification, which only enforces contracts as the user evaluator completely explores each monitored term;
- probabilistic contract verification, which probabilistically ensures contracts adhere to their pre- and post-conditions for random inputs;
- and best-effort checking, which only reports a contract result if the contract terminates before the program completes.

As with the verification strategies we describe above, these variants provide programmers with flexibility and utility to address specific situations.

1.3. Separating Contract Monitors into their Own Evaluators

In both promise-based and concurrent verification, we extract the contract monitor into a separate process. It turns out, however, that we may model even the naïve contract system presented in §1.1 with interacting evaluators, wherein the contract monitor is a separate, concurrent process that interacts with the user program to report the verification result. To illustrate this behavior, consider evaluating the expression

$$5\ +\ \mathbf{mon}\ \mathsf{nat/c}\ (1\ +\ 2) \tag{7}$$

via the rules in §1.1. As before, we evaluate the monitored expression $(1\ +\ 2)$, then the monitor **mon** nat/c 3, and finally the result $5\ +\ 3$, yielding 8. This suggests that monitoring proceeds as a separate evaluator, performing contract verification in isolation from the user program.

Dimoulas et al. [26] first observe this separation in order to take advantage of computational parallelism for increased performance and Disney et al. [31] later use a similar approach to ensure contract non-interference. More than that, however, separating contract verification into a different evaluator allows us to discuss the fundamental nature of software contract verification, allowing us to inspect *how* and *when* each contract verification strategy interacts with the user program. For example, eager contracts (presented in §1.1) postpone the user program while evaluating the monitored expression, resuming the initial computation with the verification result; semi-eager contracts do not suspend the user program until the user program requires the contract result; and promise-based contracts synchronize with the user program during specified events; and promise-based contracts synchronize when the user program demands them. Ultimately, this separation allows us to explore, compare, and *combine* contract verification strategies in a unified way.

1.4. Unified Contract System

Each of the software contract verification systems presented in §1.1 and §1.2 have immense utility, and a programmer may require each verification technique over the course of a program. For example, a programmer may wish to check that a tree is a binary-search tree in each of the following ways:

- via an O(n) traversal (e.g., after initial construction);
- by locally checking the parts the program uses (e.g., to preserve asymptotics during a BST-insertion routine);
- via an O(n) promise-based system (e.g., when the program is performing other operations in the main process);
- and via an O(n) concurrent traversal (e.g., to ensure we read a *sorted* tree from a file while the program continues).

Programmers *should not* settle for a system that does not support each of these strategies; they should be able to use any strategy, and move freely between them between.

In this thesis, we present a single, unified framework that provides each of these verification strategies (and more), allowing programmers to select the best strategy (or *combination* of strategies) on a per-contract basis. Furthermore, we demonstrate that this unification and the principled interactions of combined strategies follow from the fundamental insight presented in the previous section: each software contract monitoring strategy may be directly expressed in a unified framework as a *description of internal verifier behavior* (i.e., how the verifier proceeds) and a *pattern of communication* (i.e., how the verifier interacts with the user program).

1.5. General Runtime Verification & Verification Meta-strategies

In addition to allowing programmers to select individual verification strategies, we also develop general runtime verification (such as ensuring a program adheres to a finite-state machine of behavior) using *meta-strategies*, or strategy operators that take and return new strategies. In general, a meta-strategy may augment a basic strategy's behavior, giving programmers additional flexibility and specialization. These meta-strategies may be manually implemented by the programmer on a per-contract basis, but providing them as part of the unified framework eases developer overhead, and allows us to reason about them in a framework context.

These meta-strategies include the following (among others):

- comm, which takes a strategy and a channel. After enforcing the contract, this strategy will communicate the result across the channel. The goal is to suppose, e.g., lazily verifying tree fullness checking without manually constructing the side-channel system presented by Swords et al. [80].
- memo, which takes a strategy and a hash map. We store the verification result in the given map, and, if the expression is already in the hash map, the verifier skips enforcement, using the previous result. This allows programmers to avoid unnecessary contract recomputation over the course of execution.
- random, which takes a strategy and a number $0 \leq e \leq 1$. During contract enforcement, we probabilistically check the contract, using e as the check rate (and thus skipping some enforcements).
- and transition, which allows programmers to specify state transitions and ensure the program follows them [30].

As we will demonstrate, we can use these meta-strategies in a variety of ways, including augmenting contract performance, expanded contract verification, and refocusing the underlying verification framework to support generic runtime verification, including ensuring instruction execution order (e.g., by maintaining a state-machine for behavior) and function-level time profiling.

1.6. Thesis Statement

With the above background, we can state the thesis of this dissertation:

> Runtime verification systems may be expressed as a collection of separate, concurrent processes that interact with the user program, and variations on verification systems may be encoded as variations on patterns of communication to provide programmers with general, practical runtime verification tools.

My dissertation will defend this thesis as follows:

- *runtime verification systems may be expressed as separate, concurrent processes that systematically interact with the host (or user) program*: we demonstrate how the standard higher-order contract systems may be directly modeled as program interactions (Chapter 3) and formally define a straightforward, CSP (Concurrent Synchronous Process) calculus λ^{π}_{cs} for encoding such interactions (Chapter 4).
- *variations on verification systems may be expressed as variations of interactions:* we will demonstrate how we may express variations on runtime verification from across the literature [22, 26, 27, 30, 33, 80] in λ^{π}_{cs} (Chapter 5). We will also extend this approach to additional verification techniques, including state systems and other meta-strategies (Chapter 6).
- *general, practical runtime verification tools:* we will defend the generality of this approach by presenting an implementation built in Clojure that utilizes the JVM parallelization facilities (Chapter 7).

1.7. Previously Published Work

The material in this dissertation stems from research done jointly with multiple collaborators, some of which appears in the following previously-published papers, both directly (e.g., the initial encoding model presented by Swords et al. [80]) and indirectly (e.g., the algebraic effect system presented by Kiselyov et al. [60] and the secondary blame-tracking system presented by Vitousek et al. [83]).

- Cameron Swords. Strategy-Based Contract Monitors, Monitoring Meta-strategies, and Runtime Instrumentation. *Draft*, 2017.
- Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. An Extended Account of Contract Monitoring as Patterns of Communication. To Appear, *Journal of Functional Programming*, 2018.
- Michael M. Vitousek, Cameron Swords, Jeremy G. Siek. Big Types in Little Runtime: Open World Soundness and Collaborative Blame for Gradual Type Systems. In *Symposium on Principles of Programming Languages*, 2017.
- Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. Expressing Contract Monitors as Patterns of Communication. In *International Conference on Functional Programming*, 2015.
- Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible Effects. In *Haskell Symposium*, 2013.

CHAPTER 2

Software Contracts and Variations Therein

SYNOPSIS

In this chapter, we introduce and discuss the technical background that informs the rest of the dissertation. We begin with software contracts and contract verification (§2.1), and proceed by defining and discussing the behavior of multiple variations on contract verification, including eager (§2.2), semi-eager (§2.3), promise-based (§2.4), and concurrent (§2.5).

Runtime systems often live outside of the system they are inspecting, using different languages and systems to monitor a specific program's behavior. Software contracts forgo this need, allowing programmers to write their specifications in the same language as the underlying program, expressing and verifying properties about their program within the same context and runtime system as the program itself.

At their core, software contracts are procedures written in the host language that verify program behavior by inspecting values flowing through the program, ensuring these values adhere to specific properties (such as being a natural number, a list of a certain length, or a binary-search tree). To *verify* a contract, we execute the contract's associated predicates(s) on the monitored values, and, after execution, the contract assertion either returns the input (potentially modified to contain more monitors) or raises an error. In aggregate, these contracts verify that the underlying user program behaves as intended.

The Essence of Contract Monitors. To begin, consider a function that simulates rolling an x-sided die:

$$\lambda\ x.\ (\textsf{rand-nat}\ x) + 1 \tag{8}$$

The procedure uses rand-nat, which expects some natural number greater than 1 and returns a number between 0 and that value (exclusive), and adds 1 to the result. It would be ideal to ensure that x is greater than zero before passing it to rand-nat, and, using pred/c from Chapter 1, we can write a contract to verify this property:

$$\lambda\ x.\ \textsf{rand-nat}\ (\textsf{mon}\ (\textsf{pred/c}\ (\lambda\ n.\ n \geq 1))\ x) + 1 \tag{9}$$

In this example, the rand-nat function takes some argument x and verifies the contract

$$\textsf{pred/c}\ (\lambda\ n.\ n \geq 1)$$

on the value. As before, the pred/c form takes a predicate and yields a predicate contract, which returns the monitored value if the predicate returns true and raises an error if it returns false. This contract, then, will ensure that x is at least 1. But this leaves lingering questions about verification:

- Should we treat contracts as specifications to verify, ensuring values adhere to these contracts regardless of their usage in the program (e.g., if rand-nat works without error for a negative number, should we still report an error)?
- Should the user program wait while contract verification occurs (e.g., should we ensure x has the correct bounds before the underlying, user program may proceed)?
- Should the user program have its own role in contract verification (i.e., asking for the contract result in addition to performing its computation)?

As we saw in the introduction, these questions do not have concrete "yes" or "no" answers, but represent gradient of possible verification strategies. Moreover, these answers may change as programs increase in complexity; for example, consider checking that each element of a pair is a natural number before generating a random number based on the first element:

$$\textsf{nat/c} \triangleq \textsf{pred/c}\ (\lambda\ n.\ n \geq 0) \tag{10}$$

$$\textsf{rand-nat}\ (\textsf{fst}\ (\textsf{mon}\ \textsf{nat/c}\ 5, \textsf{mon}\ \textsf{nat/c}\ \text{-}1)) \tag{11}$$

Should Eqn. 11 raise an error (because -1 is *not* a natural number)? While the invalid value is immediately discarded, and thus will not cause the program to be incorrect, it is also possible that ensuring the contract (and signaling the subsequent error) will reveal additional problems to the programmer.

It is not sufficient to select a single set of answers to these questions. Adding software contracts to a program fundamentally changes that program's behavior. While it is possible to only write idempotent contracts that do not modify their input or change the program's meaning as part of verification [22], this restricted usage eschews the fundamental utility of contracts as language-level procedures [34] that can interact with the full language runtime, and programmers

$$
\begin{array}{rcl}
E & := & x \mid V \mid E\ E \mid \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E \mid E\ \mathit{binop}\ E \mid (E, E) \mid \mathsf{fst}\ E \mid \mathsf{snd}\ E \\
 & \mid & \mathsf{error} \mid \mathbf{mon}\ E\ E \mid \mathsf{pred/c}\ E \mid \mathsf{pair/c}\ E\ E \\
V & := & \lambda\, x.\, E \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid (V, V) \mid \mathsf{pred/c}\ V \mid \mathsf{pair/c}\ V\ V
\end{array}
$$

Figure 2.1. A strategy-agnostic syntax for software contract.

must be free to utilize this additional power how and when they see fit. In this context, however, when contracts are free to change a program's behavior with over-evaluation, divergence, side effects, and even concurrency, programmers *must* have the ability to vary *how* and *when* to verify these contracts.

2.1. Variations on Verification

Contract verification is typically expressed as part of a fixed verification system, extending a core calculus with a single, fixed contract verification mechanism describing *how* and *when* to verify contracts [22, 25, 26, 36, 52]. In this thesis, we refer to these verification mechanisms as monitoring strategies:

Definition 2.1 (Monitoring Strategy). *Given a contract and some term to monitor it on, a **monitoring strategy** describes how and when to verify the contract on the term.*

Our goal is to introduce a unified verification system which supports multiple contract monitoring strategies and, to this end, we start by introducing a number of such strategies, laying the groundwork for our later abstraction. We work from the single, strategy-agnostic syntax previously presented in Chapter 1, reproducing it in Figure 2.1. This calculus follows the core calculi presented by Degen et al. [22] and Dimoulas and Felleisen [25], which extend a standard, call-by-value λ-calculus [1] (variables x; values V ranging over lambda abstractions, numbers, pairs, errors[1], and more) with contract combinators and verification forms:

- **mon** E_1 E_2, which installs a runtime monitor to verify the contract E_1 on the expression E_2 such that the monitor either returns the monitored value or raises an error;
- pred/c E, the *predicate contract combinator*, where E must be a predicate function;
- pair/c E_1 E_2, the *pair contract combinator*, where E_1 and E_2 are subcontracts verified on the first and second elements of the monitored pair (respectively).

[1]Our errors currently raise "empty" errors. We will later extend these errors to include additional *blame* information to help programmers debug contract violations.

These language-level combinators forms allow us to define new contracts, such as $\mathsf{nat/c}$ from Eqn. 10. Using $\mathsf{nat/c}$ and the pair contract combinator, we may also define a contract that will verify its input is a pair of natural numbers:

$$\mathsf{nat\text{-}pair/c} := \mathsf{pair/c}\ \mathsf{nat/c}\ \mathsf{nat/c} \tag{12}$$

Then, we may once again ask what it means to evaluate the following:

$$\mathbf{mon}\ \mathsf{nat/c}\ 5 \rightarrow^* \ ? \tag{13}$$

$$\mathsf{rand\text{-}nat}\ (\mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat\text{-}pair/c}\ (5,\text{-}1))) \rightarrow^* \ ? \tag{14}$$

To determine the answer, we must select some semantic definition of contract verification, and, in doing so, we will also (perhaps unintentionally) answer the questions about the nature of verification we raised above. Over the rest of the chapter, we will introduce a number of **mon** semantic reductions to explore four contract verification strategies.

2.2. Eager Verification

Eager contract verification, first presented by Meyer [66], brought into the functional world by Findler and Felleisen [36], and repeatedly refined [22, 23, 27, 35], treats contracts as *fully-verified* specifications, evaluating the monitored term as necessary while the user program waits for the verification result (even if the user program would not normally evaluate it). In its purest form, we can define eager contract verification as:

$$\mathbf{mon}\ (\mathsf{pred/c}\ V_c)\ V \rightarrow \mathsf{if}\ V_c\ V\ \mathsf{then}\ V\ \mathsf{else}\ \mathsf{error} \tag{15}$$

$$\mathbf{mon}\ (\mathsf{pair/c}\ V_{c1}\ V_{c2}\ )\ (V_1, V_2) \rightarrow (\mathbf{mon}\ V_{c1}\ V_1, \mathbf{mon}\ V_{c2}\ V_2) \tag{16}$$

In the case of predicate contracts, we take the input term, reduce it to a value, and then determine if the predicate holds; for pair contracts, we destruct the (evaluated) pair, proceeding via recursion. To demonstrate this behavior, consider the following trace monitoring "$\mathsf{nat/c}$" on the term "2 + 3" before passing the result to a procedure:

$$\begin{array}{ll}
 & (\lambda\ x.\ 10)\ (\mathbf{mon}\ \mathsf{nat/c}\ (2\ +\ 3)) \\
\rightarrow & (\lambda\ x.\ 10)\ (\mathbf{mon}\ \mathsf{nat/c}\ 5) \\
\rightarrow & (\lambda\ x.\ 10)\ (\mathsf{if}\ (\lambda\ n.\ n\ \geq\ 0)\ 5\ \mathsf{then}\ 5\ \mathsf{else}\ \mathsf{error}) \\
\rightarrow^* & (\lambda\ x.\ 10)\ (\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 5\ \mathsf{else}\ \mathsf{error}) \\
\rightarrow & (\lambda\ x.\ 10)\ 5 \\
\rightarrow & 10
\end{array}$$

When the evaluator encounters the monitoring form (line 3), the user program (attempting to apply "$\lambda\ x.\ 10$" to "$2\ +\ 3$") suspends, waiting while the contract system verifies the contract. After contract verification, the monitoring reduction yields control back to the user portion of the program with the monitored value (in this case, "5"), which is immediately discarded by the application (line 6). If 5 had not passed the contract, however, the monitor would subvert the user program control to raise an error.

This contract verification strategy treats contracts as concrete specifications to *fully* verify over a program's execution, regardless of the program's execution trace (e.g., how it uses the contracted terms). This means that even unused values will raise contract violation errors. For example, consider reducing Eqn. 11 under an eager monitoring strategy:

$$\begin{array}{ll} & \mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat\text{-}pair/c}\ (5, \text{-}1)) \\ \rightarrow^* & \mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat/c}\ 5, \mathbf{mon}\ \mathsf{nat/c}\ \text{-}1) \\ \rightarrow^* & \mathsf{fst}\ (5, \mathbf{mon}\ \mathsf{nat/c}\ \text{-}1) \\ \rightarrow^* & \mathsf{fst}\ (5, \mathsf{error}) \\ \rightarrow^* & \mathsf{error} \end{array}$$

When we verify a pair contract with an eager monitoring strategy, we immediately monitor each subcontract on the appropriate sub-component of the pair, interrupting the user program to completely verify the subcontracts. In this case, "-1" is *not* a natural number, and thus the entire program terminates with an error even though the second element of the pair is not part of the final program result.

Eager verification signals errors for unused values, which comes with a number of potential drawbacks:

(1) Treating contracts as fully-verified specifications inhibits verifying properties on infinite structures. For example, ensuring that each element of an infinite stream is a natural number will cause the monitor will diverge.

(2) Suspending the user program while the monitor proceeds can be computationally prohibitive. For example, consider

$$\mathbf{mon}\ \mathsf{prime/c}\ (237^{63} + 567) \tag{17}$$

The user program waits while the contract monitor performs a primality check, potentially bottlenecking the underlying application. Similar situations may occur when, e.g., ensuring each element of a hash map adheres to a specific property. If eager verification is the only monitoring strategy available, programmers may find it too expensive to verify rich properties of their programs.

(3) Interrupting the user evaluator and over-evaluating inputs may produce different program behavior than the original, unmonitored program: contracts are effects, and may not always preserve the underlying program's meaning [72]. For example, consider the following predicate (monitoring a function value):

$$\mathsf{pred/c}\ (\lambda\ f.\ (f\ 5) = 0)$$

If the function diverges on input 5, but otherwise behaves correctly over the course of the program, then monitoring this contract will cause divergence in a program that may have otherwise terminated.

These drawbacks stem from the fundamental assumption of eager contract verification: if we accept contracts as concrete specifications that *must* be completely verified as the user program encounters them, we ensure that each contract is totally monitored [27] at the cost of potential verifying (and potentially evaluation) more that necessary.

2.3. Semi-Eager Verification

The over-evaluation of eager verification suggests an immediate alternative: we may maintain the user program suspension during verification, but postpone contract verification until the program demands the monitor's result. Hinze et al. [52] originally present this *semi-eager* verification strategy in order to address these problems with eager monitors; Degen et al. [22] later define semi-eager verification (including blame information) in the context of lazy programming languages; and Findler et al. [37] use a similar less-eager verification mechanism for contract verification.

In order to provide semi-eager contract verification, the monitoring reduction must "box up" the contract and the monitored expression, suspending contract verification until the user evaluator demands the value. We encode this mechanism as follows, representing contract-expression pairs as $\langle contract \mid expression \rangle$:

$$\mathbf{mon}\ V_1\ V_2 \rightarrow \langle V_1 \mid V_2 \rangle \tag{18}$$

When the user evaluator demands the value (e.g., the boxed expression occurs in evaluation position), the evaluator suspends the user program and verifies the contract, resuming the program with the monitored value if the contract holds or raising an error if it does not. We can achieve this result by specializing the evaluator to perform contract verification when the program requires the boxed expression (e.g., $\langle V_1 \mid V_2 \rangle\ V_3$ will need to unbox the operator and perform verification). This evaluation mechanism is reminiscent of call-by-name evaluation, where some subset $\mathcal{D}$ of evaluation contexts $\mathcal{D}$ are "forcing" contexts [6] that trigger contract verification:

$$\mathcal{D}[\langle \mathsf{pred/c}\ V_c \mid V \rangle] \rightarrow \mathcal{D}[\mathsf{if}\ V_c\ V\ \mathsf{then}\ V\ \mathsf{else}\ \mathsf{error}] \tag{19}$$

$$\mathcal{D}[\langle \mathsf{pair/c}\ V_{c1}\ V_{c2}\ \mid\ V \rangle] \rightarrow \mathcal{D}[(\mathbf{mon}\ V_{c1}\ (\mathsf{fst}\ V), \mathbf{mon}\ V_{c2}\ (\mathsf{snd}\ V))] \tag{20}$$

To illustrate semi-eager verification, consider evaluating the pair example from Eqn. 11 (without the rand-nat operator):

$$\begin{array}{ll}
 & \mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat\text{-}pair/c}\ (5, \text{-}1)) \\
\rightarrow^* & \mathsf{fst}\ \langle \mathsf{nat\text{-}pair/c} \mid (5, \text{-}1) \rangle \\
\rightarrow^* & \mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat/c}\ 5, \mathbf{mon}\ \mathsf{nat/c}\ \text{-}1) \\
\rightarrow^* & \mathsf{fst}\ (\langle \mathsf{nat/c} \mid 5 \rangle, \langle \mathsf{nat/c} \mid \text{-}1 \rangle) \\
\rightarrow & \langle \mathsf{nat/c} \mid 5 \rangle \\
\rightarrow & \mathsf{if}\ (\lambda\ n.\ n\ \geq\ 0)\ 5\ \mathsf{then}\ 5\ \mathsf{else}\ \mathsf{error} \\
\rightarrow^* & 5
\end{array}$$

This example illustrates the primary behavioral difference between eager and semi-eager verification: in semi-eager verification, contracts are no longer fully-verified specifications that we *must* ensure, and the contract system will only verify contracts as the user program uses them. This monitoring variant allows programmers to trust that any value used in the program result adheres to its contract(s) without risking over-evaluation. It checks precisely those values we access in a stream or hash-map in, allowing us to preserve program behavior *and* localized contract verification.

As with eager verification, semi-eager verification has its own drawbacks:

(1) Semi-eager verification is not *faithful* to the contract specification [22]: only those values the program uses will have their contracts verified and, as a result, we cannot trust our contracts as full specifications. This behavior may lead to unobserved program errors (such as the example above).

(2) Semi-eager verification is not *idempotent* [22]: depending on how we implement contract verification, it is possible that these two terms may have different behaviors:

$$\begin{array}{l}
\mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat\text{-}pair/c}\ (5, \text{-}1)) \\
\mathsf{fst}\ (\mathbf{mon}\ \mathsf{nat\text{-}pair/c}\ (\mathbf{mon}\ \mathsf{nat\text{-}pair/c}\ (5, \text{-}1)))
\end{array}$$

If monitoring the outer contract demands the contracted values $\langle \textsf{nat/c} \mid 5\rangle$ and $\langle \textsf{nat/c} \mid \text{-}1\rangle$, then the second expression will raise an error, while the first will return 5. This strange behavior may mislead programmers using a semi-eager contract system[2].

(3) Semi-eager verification can lead to a "verification anti-pattern": in semi-eager verification, programmers may find themselves evaluating terms expressly for contract verification. This anti-pattern manifests in our previous example by taking the second element of the pair in order to ensure contract verification, even if we will not otherwise require it.

(4) This verification technique still suspends the user evaluator while verification proceeds; it just does so at a finer-grained level.

As we can see, semi-eager contract verification is not a catch-all solution: while we may use semi-eager monitors to check a different set of contracts than eager monitors, we have fewer guarantees and a similar level of user evaluator interruption across the program.

An Aside on Function Contracts. Function contracts, created with the function contract combinator fun/c, perform contract verification on a function, ensuring its input adheres to the given *pre-condition* and its output adheres to the given *post-condition*. These contracts present a unique problem when compared to other structural contracts: unlike pairs or larger structures, it is, in general, impossible to ensure that a procedure behaves correctly for every input, as they typically work over infinite input spaces. To avoid this problem, Findler and Felleisen [36] propose an alternative approach to function contracts wherein a function contract yields a *new* function, wrapping the monitored function in pre- and post-condition checks (where V_{c1} is the pre-condition, V_{c2} is the post-condition, and V is the monitored function):

$$\textbf{mon}\ (\textsf{fun/c}\ V_{c1}\ V_{c2}\ )\ V \to \lambda\ x.\ \textbf{mon}\ V_{c2}\ (V\ (\textbf{mon}\ V_{c1}\ x)) \tag{21}$$

For example, we can verify that a function takes and returns only natural numbers:

$$\textbf{mon}\ (\textsf{fun/c nat/c nat/c})\ (\lambda\ n.\ n\ +\ 5) \to \lambda\ x.\ \textbf{mon}\ \textsf{nat/c}\ ((\lambda\ n.\ n\ +\ 5)\ (\textbf{mon}\ \textsf{nat/c}\ x)) \tag{22}$$

This definition of fun/c is "natively" semi-eager: the implicit delaying nature of λ suggests we should check only those values that flow in and out of the function. This is not, however, the only solution; other alternatives to verification include:

- Probabilistic verification [33, 50], wherein the contract system generates sample inputs and verifies that the monitored procedure adheres to its pre- and post-conditions for these inputs.

[2]Owens [72] identifies a similar problem with pairs of functions in eager verification.

- Static contract verification [33, 69, 86], which performs static analysis to ensure that the procedure adheres its contract. This approach also allows systems to erase contracts that will provably hold.

For the moment, we forgo these alternatives for our semantic presentation so that function contracts resemble their structural counterparts, taking some structure (e.g., a function) as input and returning that same structure with contracts nested in it as the verification result, but we will revisit these alternatives in later chapters.

2.4. Promise-Based Verification

In order to address the burden of evaluator interruption, Dimoulas et al. [26] introduce the notion of *future contracts*, presenting a concurrency model with a user and a monitoring process, wherein the user process communicates contracts and expressions to the monitoring process and the monitoring process concurrently performs contract verification. We may replicate this multi-process approach by replacing the "boxing" and "unboxing" of semi-eager verification with *promises* [41] and inter-process communication, written $\{\!\!\{ e \}\!\!\}$[3]:

$$\textbf{mon}\ (\mathsf{pred/c}\ V_c)\ V \rightarrow \mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{if}\ V_c\ V\ \mathsf{then}\ \mathsf{write}\ \iota\ V\ \mathsf{else}\ \mathsf{write}\ \iota\ \mathsf{error}))\ \{\!\!\{\mathsf{read}\ \iota\}\!\!\} \tag{23}$$

$$\textbf{mon}\ (\mathsf{pair/c}\ V_{c1}\ V_{c2}\ )\ (V_1, V_2) \rightarrow \mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{write}\ \iota\ (\textbf{mon}\ V_{c1}\ V_1, \textbf{mon}\ V_{c2}\ V_2)))\ \{\!\!\{\mathsf{read}\ \iota\}\!\!\} \tag{24}$$

For predicate contracts, we create a new process (via spawn) that evaluates the contract and either writes the original value or an error across some channel ι. For pair contracts, we decompose the contract and monitor each subcontract, writing the final result across some channel ι.

We have taken a number of liberties in this encoding of promise-based verification, assuming a process-creation operation spawn, channels ι with channel communication, "promise boxes" $\{\!\!\{ e \}\!\!\}$ that we may later force with the force operation, and our ability to write errors directly across channels. We take these liberties for presentation purposes, and address each of them in our semantic model in Chapter 5.

As for usage, consider a program that ensures some value is a natural number, generates a new encryption key, and then uses that key to encrypt the natural number. If generating a new encryption

[3]Here, e is the expression to retrieve the computational result when forcing the promise

key is computationally intensive, it may be worthwhile to verify this contract concurrently:

	User Process		*Monitoring Process*
	$\textbf{let}\ msg = \textbf{mon}\ \mathsf{nat/c}\ \text{-}1$ $\textbf{in}\ \ encrypt\ (gen\text{-}new\text{-}key)\ msg$		
$\rightarrow^*$	$\textbf{let}\ msg = \{\!\{\mathsf{read}\ \iota\}\!\}$ $\textbf{in}\ \ encrypt\ (gen\text{-}new\text{-}key)\ msg$		$\mathsf{if}\ \ \ (\lambda\ n.\ n\ \geq\ 0)\ \text{-}1$ $\mathsf{then}\ \mathsf{write}\ \iota\ \text{-}1$ $\mathsf{else}\ \ \mathsf{write}\ \iota\ \mathsf{error}$
$\rightarrow^*$	$encrypt\ (gen\text{-}new\text{-}key)\ \{\!\{\mathsf{read}\ \iota\}\!\}$	$\rightarrow^*$	$\mathsf{write}\ \iota\ \mathsf{error}$
$\rightarrow^*$	$encrypt\ V\ (\mathsf{force}\ \{\!\{\mathsf{read}\ \iota\}\!\})$		
$\rightarrow^*$	$encrypt\ V\ (\mathsf{read}\ \iota)$		
$\rightarrow^*$	error		

The user evaluator communicates the contract and expression to a concurrent process before resuming its computation, and later retrieves the concurrent monitoring process's verification result. Unlike semi-eager verification, promise-based monitors do not interrupt the user evaluator during verification, allowing the user process to proceed concurrently with contract monitor. When the user program requires the contract result, the evaluator forces the promise, retrieving it[4]. This approach reveals a potential optimization in multi-processor settings: the user evaluator may proceed in parallel with the monitoring evaluator, allowing the user evaluator to spend less time awaiting monitoring results and potentially yielding program speed-up.

As with our other variations, this decision not to view contracts as concrete specifications and remove user program suspensions has its own potential issues:

(1) Promise-based verification falls victim to some of the previous concerns of semi-eager verification (including lack of idempotence and verification anti-patterns).
(2) The user evaluator may end up "wasting cycles" on speculative computation, performing operations that the program must discard after a contract signals an error, or, worse, that the program must roll back (e.g., in the case of effects such as file output).
(3) The cost of communication and promises may dominate program performance.
(4) Effectful contracts (e.g., a contract that maintains internal state) no longer have a guaranteed execution order, and may yield unexpected and unpredictable results. For example, consider

[4]In their original presentation, *future contracts* synchronized whenever the user program performed effectful operations, requiring machinery wrapping each effect.

a monitor that updates a reference before ensuring its input is a natural number:

$$\mathsf{pred/c}\ (\lambda\ x.\ \mathsf{seq}\ (\mathsf{update\text{-}ref}\ \mathsf{ref}\ (!\mathsf{ref}\ +\ x))\ (x\ \geq\ 0)) \tag{25}$$

If the user program relies on ref, the programmer may be unable to predict the program's outcome. Worse, if update-ref is not *atomic* [48], this behavior may lead to different results depending on process scheduling.

As with eager and semi-eager verification before it, we can see that promise-based verification is also not a perfect-fit solution: although it addresses some of our problems with both eager and semi-eager verification, a promise-based verification strategy introduces its own set of programmer concerns.

2.5. Concurrent Verification

In an attempt to remove some of the semantic complexity of the previous verification approaches, it is worthwhile to consider *asynchronous* concurrent verification, wherein the contract verification system forgoes explicitly reporting the result to the user evaluator. Instead, the monitoring process verifies the contract concurrently, either terminating silently if the contract holds or raising an error if it does not[5]:

$$\mathsf{mon}\ (\mathsf{pred/c}\ V_c)\ V \rightarrow \mathsf{seq}\ (\mathsf{spawn}(\mathsf{if}\ V_c\ V\ \mathsf{then}\ V\ \mathsf{else}\ \mathsf{error}))\ V \tag{26}$$

As with promise-based monitoring, we create a new process to monitor the contract. Unlike promise-based monitoring, however, we return the original value to the user evaluator without creating a promise, allowing both computations to proceed without further synchronization.

Consider the following usage:

	User Process	*Monitoring Process*
	$\mathsf{let}\ x = \mathsf{mon}\ \mathsf{nat/c}\ \text{-1}\ \mathsf{in}\ (2+4)+x$	
$\rightarrow^*$	$\mathsf{let}\ x = \text{-1}\ \mathsf{in}\ (2+4)+x$	$\mathsf{if}\ (\lambda\ n.\ n\ \geq\ 0)\ \text{-1}\ \mathsf{then}\ \mathsf{unit}\ \mathsf{else}\ \mathsf{error}$
$\rightarrow$	$(2+4)+\text{-1}$	$\mathsf{if}\ \text{-1}\ \geq\ 0\ \mathsf{then}\ \mathsf{unit}\ \mathsf{else}\ \mathsf{error}$
$\rightarrow$	$6+\text{-1}$	$\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ \mathsf{unit}\ \mathsf{else}\ \mathsf{error}$
$\rightarrow$	5	error

[5]We elide pair-based contract verification for concurrent monitors for simplicity. Such a definition would follow the promise-based definition in Eqn. 24.

This program may *either* produce “5” *or* raise an error. Which result should we expect? More specifically, if the user evaluator does not explicitly ask for the contract result, should the language runtime await it anyway? Either answer is valid, and each has further implications.

If we do not wait, we get a concurrent, "best-effort" checking system. In the above example, we may either receive 5 or `error` as the final answer, dependent upon process scheduling. While ultimately weaker than the previous verification techniques, this best-effort approach may be ideal for monitoring expensive properties during a program, such as a contract that performs a probabilistic primality test in an infinite loop, randomly checking the divisibility of its input until the user program terminates [18, 32].

This concurrent verification strategy, too, has its own pitfalls:

(1) Concurrent verification may be too weak: a best-effort approach to verification may inhibit programmers from reliably ensuring program properties, (e.g., a probabilistic primality check contract may not find a counter-example even if the number is not prime).
(2) Scheduler-dependent results may not detect some contract violations, or may only detect them in some program executions.
(3) As with promise-based contracts, contracts with side effects that modify global state be altogether unpredictable.

Alternatively, we may instrument the language runtime to wait for these monitoring processes to run to completion. This “finally-concurrent” verification recovers the guarantees lost in concurrent verification and ensures that every contract we monitor is fully verified, addressing issues 1 and 2 above at the cost of waiting for these monitors. Moreover, finally-concurrent contract verification may also take advantage of parallel performance without the need for later synchronization. Even so, it has its own issues:

(1) Finally-concurrent verification yields a similar flavor of over-evaluation as eager monitoring: verifying a contract on a stream will never terminate, and as a result, it is possible to create concurrent contract monitors that never terminate.
(2) Effectful contracts cause the same problems as they do in concurrent and promise-based verification.

2.6. Verification in Review

We have seen just five of the myriad verification strategies which make up the contract verification design space. We have chosen these five because of their frequency in the literature, their apparent utility, and their direct encodings. However, these are only a small portion of the far-reaching mechanisms for contract verification, and we explore and encode additional verification approaches further in §5.7.

Researches have explored this verification design space, and two, in particular, make strong arguments about the relative utility of different verification strategies:

- Findler et al. [37] identify a number of contracts where semi-eager verification is critical to program performance, and argue that it must be a programmer option, going so far as to introduce it as a secondary, ad-hoc mechanism alongside Racket's fixed, eager verification.
- Degen et al. [22] declare that "faithfulness is better than laziness" for lazy languages, advocating that, in a fixed-strategy setting, concrete contract verification is more valuable than preserving the underlying program's performance or behavior.

This presentation, following our work in Swords et al. [80], takes a different stance: programmers should have mechanisms to choose *how* and *when* contract verification occurs on a per-contract basis, selecting the most appropriate strategy in each case. As we will see over the course of this dissertation, separating verification into a separate, interacting process allows us to provide multiple verification strategies in a unified system, ultimately constructing a framework which allows programmers to construct and extend generic contract verification to include generalized runtime verification mechanisms.

CHAPTER 3

Uniting Contract Verification Strategies in a Unified Framework

SYNOPSIS

Each of the contract verification strategies we have seen has its own strengths and weaknesses. While other authors advocate for singular verification mechanisms or providing secondary verification mechanisms through non-standard interfaces, our work in Swords et al. [80] proposes another alternative: allowing programmers to choose their verification strategy on a contract-by-contract basis to address different needs over the course of a program. In this section, we explore this approach to verification, introducing a strategy-parameterized monitoring form (§3.1), demonstrating how we may reuse contracts to provide wildly different behaviors by varying their verification strategies (§3.2), and explaining how verification strategies may each be uniformly encoded as variations of the same structure (§3.3).

In traditional contract systems, language designers provide a single, fixed verification behavior as a permanent fixture of the evaluation semantics for a given language, and, as a result, the system gives the programmer little choice in how contract verification proceeds. Such fixtures have two immediate effects:

- **Users cannot choose how and when to verify contracts.** They must rely on the existing system's behavior, even when its behavior is not ideal. For example, consider inspecting that a tree is a binary-search tree (e.g., children to the left of a node have smaller values and children to the right have larger values). Fully verifying this contract before a binary-search tree insertion routine will impact the asymptotic behavior of a program. If, however, the programmer has the option to use a semi-eager contract in this situation, they may check only those nodes the insertion explores, locally verifying the property while preserving asymptotics.
- **Contract verification is an unknown quantity.** If the verification strategy is a fixed, opaque entity to the programmer, they may find it difficult or outright impossible to predict

the impact and behavior of contracts across their program, making it difficult to reason about contracts with dependent values and side effects.

To address these problems, we follow Swords et al. [80], abstracting away from the traditional, "one-strategy" verification mechanisms by exposing contract verification strategies as an additional parameter to the verification form, ultimately allowing programmers to choose their verification strategy on a contract-by-contract basis to address different needs over the course of a program.

To demonstrate the immediate utility of such an approach, consider verifying that a tree is a binary-search tree in each of the following ways in the same program:

- via eager monitoring after initial construction, as a post-condition to ensure that, e.g., the *list-to-tree* procedure produces the correct structure;
- via semi-eager monitoring, to verify that each element explored during a binary tree lookup satisfies the contract while preserving asymptotics;
- via promise-based monitoring, to inspect the tree while the program performs additional, unrelated operations;
- and via concurrent monitoring, to ensure that a tree read from a file has the correct structure.

Each of these tree verification situations occurs naturally over the course of a program, and yet standard contract systems support only one of these behaviors. A multi-strategy system, however, allows us the flexibility to write each of these monitors across the same program, suiting the needs of each situation.

Through the rest of this chapter, we will look at a programming mechanism that allows us to provide this behavior, examine examples using this mechanism, and discuss the underlying principles that allow us to implement it.

3.1. Multi-Strategy Monitoring

To facilitate multi-strategy monitoring, we must provide a general mechanism to allow programmers to select which strategy to use for each verification site. To this end, we introduce a revised **mon** form:

$$E := \ldots \;\mid\; \textbf{mon}\ E\ E\ E\ E$$

As before, the **mon** operation takes a contract and an expression to monitor. In addition to these arguments, this revised **mon** form takes two additional arguments:

- It takes a *strategy* argument—a value indicating how the contract monitor should proceed with verification:

$$V \ := \ \ldots \ | \ S \qquad\qquad S \ := \ \textsf{eager} \ | \ \textsf{semi} \ | \ \textsf{promise} \ | \ \textsf{concurrent} \ | \ \textsf{fconc}$$

  These strategies are first-class values, allowing programmers to use, retain, and pass monitoring behaviors as parameters over the course of a program.
- It also takes a *blame* argument, indicating the parties responsible for the contract verification in the event of a violation. We assume the standard blame disciplines described in the literature [27, 36], using a three-tuple allowing us to blame the contracted value, context, and contract dependent upon when and how the contract violation occurs. We use "B" as a stand-in value for this blame tuple in our examples.

3.2. Small Examples of Multi-Strategy Monitors

We may now use our revised **mon** form in conjunction with these explicit strategy forms to produce each monitoring behavior. For example, we can consider the same contract under **eager** and **semi** verification, demonstrating that unused values will not signal violations using the **semi** strategy:

$$\begin{array}{l|l}
\textsf{let } x = \textsf{mon nat/c eager } 5 \ B \textsf{ in } 3 \ \to^* \ 3 & \textsf{let } x = \textsf{mon nat/c eager } \text{-1} \ B \textsf{ in } 3 \ \to^* \ \textbf{raise} \\
\textsf{let } x = \textsf{mon nat/c semi } 5 \ B \textsf{ in } 3 \ \to^* \ 3 & \textsf{let } x = \textsf{mon nat/c semi } \text{-1} \ B \textsf{ in } 3 \ \to^* \ 3
\end{array}$$

Similarly, we can use **promise** to verify a contract concurrently while the computation proceeds, replicating our promise-based verification behavior from Chapter 2:

$$\begin{array}{ll|ll}
 & \textit{User Process} & & \textit{Monitoring Process} \\
 & \textbf{let } msg = \textsf{mon nat/c promise } \text{-1} \ B & & \\
 & \textbf{in } \ encrypt \ (gen\text{-}new\text{-}key) \ msg & & \\
\to^* & \textbf{let } msg = \{\textsf{read } \iota\} & & \textsf{if } \ (\lambda \ n. \ n \ \geq \ 0) \ \text{-1} \\
 & \textbf{in } \ encrypt \ (gen\text{-}new\text{-}key) \ msg & & \textsf{then write } \iota \ \text{-1} \\
 & & & \textsf{else } \ \textsf{write } \iota \ (\textsf{error } B) \\
\to^* & encrypt \ (gen\text{-}new\text{-}key) \ \{\textsf{read } \iota\} & \to^* & \textsf{write } \iota \ (\textsf{error } B) \\
\to^* & encrypt \ V \ (\textsf{read } \iota) & & \\
\to^* & \textsf{error } B & &
\end{array}$$

The **concurrent** and **fconc** strategies also proceed as described in the previous chapter, allowing the programmer to indicate that verification should proceed using concurrent or finally-concurrent verification, respectively:

$$\begin{array}{llll} \mathsf{let}\ x = \mathbf{mon}\ \mathsf{nat/c}\ \mathbf{concurrent} & \text{-1}\ B\ \mathsf{in}\ (2+4)+x & \rightarrow^* & 5\ \textit{or}\ \mathbf{raise} \\ \mathsf{let}\ x = \mathbf{mon}\ \mathsf{nat/c}\ \mathbf{fconc} & \text{-1}\ B\ \mathsf{in}\ (2+4)+x & \rightarrow^* & \mathbf{raise} \end{array}$$

Exposing verification strategies in this manner allows programmers to *precisely* choose *how* and *when* to verify each software contract in a unified framework, flexibly moving from one monitoring strategy to another at each verification site.

3.2.1. *Binary-Search Trees with Multi-Strategy Monitors*

In our unified framework, strategies are *first-class values*, meaning that they can flow through the program like any other value. This allows programmers to write contracts (and contract combinators) that take verification strategies as arguments, allowing programmers to reuse the same contract to produce a variety of verification behaviors.

To demonstrate the utility of this approach, consider a "bst/c" contract over binary trees, parameterized by a strategy indicating how to recursively verify the contract at each node (i.e., the contract applied to each child node). Such a parameterized contract allows us to write each of the binary-search tree contracts described in the introduction:

- "**mon** (bst/c **eager**) **eager** *tree B*" eagerly verifies that a tree is a binary-search tree;
- "**mon** (bst/c **semi**) **semi** *tree B*" returns a tree that will monitor that each subtree is correctly-ordered as the program explores it;
- "**mon** (bst/c **promise**) **promise** *tree B*" creates a cascading chain of monitoring processes for each node, and exploring the tree synchronizes with the appropriate processes at each level;
- "**mon** (bst/c **concurrent**) **concurrent** *tree B*" concurrently verifies that the tree is a binary-search tree using a similar set of cascading processes, but as a "best-effort" check;
- and "**mon** (bst/c **fconc**) **fconc** *tree B*" proceeds similar to the **concurrent** contract, forcing the user program to wait for the verifier to complete before terminating.

These are not, however, the only possibilities in our framework: the strategy argument to "**mon**" describes how the *top-level* monitor should behave while the strategy argument to "bst/c" separately describes the recursive verification, allowing us to freely modify the exterior verification strategy independently of the "bst/c" argument to produce diverse verification structures. For example, we may create a single, promise-based verifier that eagerly verifies each subcontract as:

$$\mathbf{mon}\ (\mathsf{bst/c}\ \mathbf{eager})\ \mathbf{promise}\ \mathit{tree}\ B$$

The resultant verifier will return a promise to the user process, proceeding with **eager** verification in a separate process and allowing the programmer to retrieve the eager verification result at a later synchronization point.

As we will see in later chapters, this intermixing of verification strategies allows us to express complex verification constructs, verifying rich properties when and how we wish.

3.3. A Unifying Semantics for Contract Verification

Thus far, we have presented our strategy reduction semantics as specialized forms, each designed from the ground up to provide unique verification behavior. As a result, it appears that, in order to develop a unified verification system, we need to build bespoke implementations to adhere to each individual verification behavior, developing unique semantic structures for each. Such an approach, however, soon becomes unwieldy and over-specialized, forcing language developers to carefully craft each combinator-strategy interaction.

This perception, however, is incorrect.

3.3.1. *Separating Evaluators*

Although it may be possible to encode each combinator-strategy as a unique entity, each strategy is, fundamentally, a small variation on the same theme: the monitors executes some verification code, and this execution interacts with the user program at one or more points over the course of verification in order to receive the monitored term and (potentially) report the verification result. Viewed through this lens, we see that each strategy variation describes *how* and *when* this monitoring evaluator interacts with the user program, and how the user program proceeds in the context of these interactions.

To illustrate this concept, consider the expression

$$5 + \textbf{mon}\ \mathsf{nat/c}\ \textbf{eager}\ (1 + 2)\ B$$

To evaluate the expression, we perform the following steps:

(1) The user program will evaluate “$(1 + 2)$”, yielding “3.”
(2) The monitoring expression “$\textbf{mon}\ \mathsf{nat/c}\ \textbf{eager}\ 3\ B$” will suspend the user program while the monitoring evaluator ensures that “3” is a natural number.

(3) The monitoring expression yields "3" and the user evaluator resumes, evaluating "$5 + 3$" to produce "8" as the final result.

We present this derivation in the top half of Figure 3.1, where we have indicated the user portions of the evaluation in $\boxed{\text{red}}$ and the monitoring portions in $\boxed{\text{blue}}$.

This value flow between these "evaluator" regions is reminiscent of the ownership model described by Dimoulas et al. [27], where we account for the contract monitor *itself* as a potential value owner. This separation of concerns leads to our key insight:

> *The contract verification portion* of a program is fundamentally separate from *the user portion,* and we may isolate each individual contract monitor as an individual process that interacts with the user evaluator [26, 31, 80].

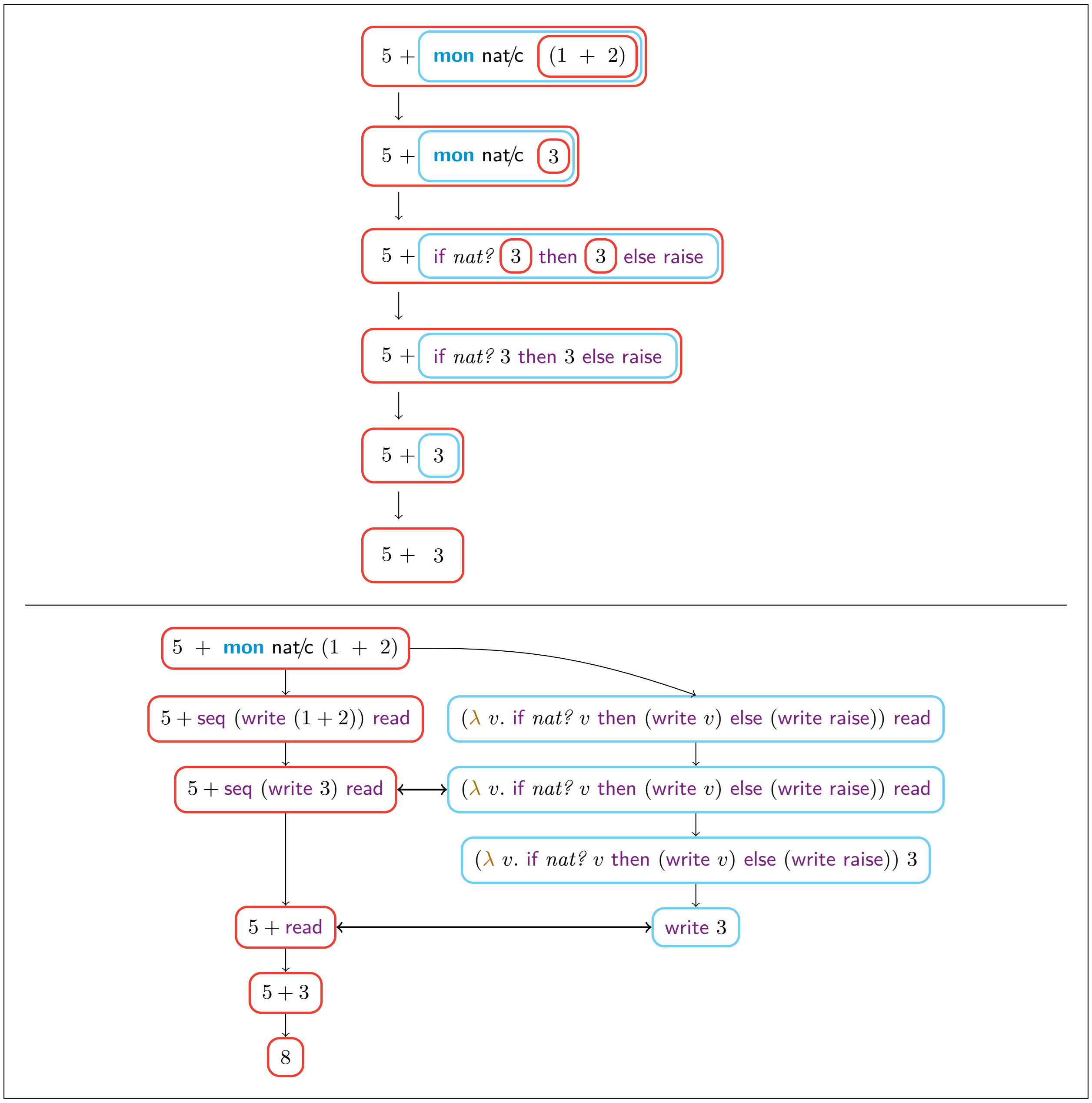
If we apply this approach to our example above, explicitly separating the two evaluators, we arrive at the derivation in the bottom half of Figure 3.1, where the user and monitoring processes *explicitly interact* across a communication channel to compute the final program result. This revised derivation reveals the fundamental nature of contract monitoring: a contract monitor is a *separate evaluator* communicating with the user program.

Extracting and isolating contract monitors in this way yields an immense benefit: in order to model the contract monitor as a separate process, we must *explicitly* encode each communication point between the user program and the monitor, precisely describing the two evaluators' interactions. This insight is critical because, as we will see in the Chapter 5, we may encode each of these verification strategies by *varying this pattern of communication*, allowing us to uniformly implement multiple, interacting verification strategies in a unified framework. Without this separation, any such encoding and interactions would be ad-hoc, forcing us to consider and specifically handle each one.

This approach to complete monitor separation also applies to structural contracts, such as function contracts. For example, consider verifying that a function's input and output are each natural numbers, using eager contract verification (wherein our "**mon**" form and "fun/c" contract combinator take strategy parameters to describe verification behaviors, and B is an opaque blame value):

$$(\textbf{mon}\ (\mathsf{fun/c}\ \mathsf{eager}\ \mathsf{nat/c}\ \mathsf{eager}\ \mathsf{nat/c})\ \mathsf{eager}\ (\lambda\ x.\ x\ -\ 10)\ B)\ 5 \Rightarrow \mathsf{raise}\ B \tag{27}$$

As we evaluate this expression, we see a series of reductions that are expressly *verification* reductions; that is, reductions that have no impact on the program except to verify its contracts.



5 + mon nat/c (1 + 2)
5 + mon nat/c 3
5 + if nat? 3 then 3 else raise
5 + if nat? 3 then 3 else raise
5 + 3
5 + 3
5 + mon nat/c (1 + 2)
5 + seq (write (1 + 2)) read
(λ v. if nat? v then (write v) else (write raise)) read
5 + seq (write 3) read
(λ v. if nat? v then (write v) else (write raise)) read
(λ v. if nat? v then (write v) else (write raise)) 3
5 + read
write 3
5 + 3
8

Figure 3.1. Separating an eager, flat contract into a pattern of communication. (We take *nat?* to mean $\lambda\, x.\, x \geq 0$ to simplify our presentation.) The first image depicts a single evaluator indicating the different evaluation components of a software contract system, where we color the user components in $\boxed{\text{red}}$ and the software contract system component in $\boxed{\text{blue}}$. The second image depicts the same verification with explicit interactions between the user program and a separate, monitoring evaluator.

Consider the trace given in the top half of Figure 3.2, where we highlight these reductions in *blue*[1]: each highlighted point in the evaluation, the contract system interrupts the user portion of the program in order to verify a contract, returning control to the user program only after completing verification. In this example, this interaction occurs as follows:

- First, the function monitor creates a contracted version of the function.
- Next, the monitoring expression "**mon** **eager** nat/c 5 (*invert* B)" ensures "5" is a natural number (and modified blame to appropriately indicate the function's input, and not the function itself, is responsible in the case of the contract violation), suspending the user program while checking this contract.
- After this contract verification, the user program proceeds with "$((\lambda\ x.\ x\ -\ 10)\ 5)$."
- Finally, the post-condition monitor attempts to verify that "-5" is a natural number, raising an error when the evaluator detects the contract violation.

If we explicitly model these evaluators, separating out the monitoring portion of the program from the user portion, we arrive at the derivation in the bottom half of Figure 3.2, where we can see the evaluator interactions and suspension in each case. As in our last example, we have multiple evaluators each verifying its own contract.

3.3.2. *Multi-Strategy Verification as Patterns of Communication*

Separating the monitoring evaluators from the user program exposes contract verification as a *pattern of communication* between processes, and this communication-based interpretation for contract monitors may be extended to other verification strategies. For example, consider semi-eagerly verifying that "-1" is a natural number in two programs, where the first does not use the verification result:

$$\begin{aligned} &\mathsf{let}\ x = \mathbf{mon}\ \mathbf{semi}\ \mathsf{nat/c}\ \text{-}1\ B\ \mathsf{in}\ 10 \\ \Rightarrow &\mathsf{let}\ x = \mathsf{delay}\ (\ldots\ \textit{check contract}\ \ldots)\ \mathsf{in}\ 10 \\ \Rightarrow &10 \end{aligned} \tag{28}$$

[1]We make the assumption that contracts are *applied* to the subject term and blame information to proceed with verification. As we will see later, this assumption leads to direct combinator definitions.

$$\begin{aligned}
&(\textsf{mon}\ (\textsf{fun/c}\ \textsf{eager}\ \textsf{nat/c}\ \textsf{eager}\ \textsf{nat/c})\ \textsf{eager}\ (\lambda\ x.\ x\ -\ 10)\ B\ 5\\
\Rightarrow\ &((\textsf{fun/c}\ \textsf{nat/c}\ \textsf{eager}\ \textsf{nat/c}\ \textsf{eager})\ (\lambda\ x.\ x\ -\ 10)\ B)\ 5\\
\Rightarrow\ &(\lambda\ x.\ \textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ x\ (\textit{invert}\ B)))\ B)\ 5\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ 5\ (\textit{invert}\ B)))\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{nat/c}\ 5\ \bar{B}))\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{if}\ \textit{nat?}\ 5\ \textsf{then}\ 5\ \textsf{else}\ \textsf{raise}\ \bar{B}))\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ 5)\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ (5\ -\ 10)\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ \text{-}5\ B\\
\Rightarrow\ &\textsf{nat/c}\ \text{-}5\ B\\
\Rightarrow\ &\textsf{if}\ \textit{nat?}\ \text{-}5\ \textsf{then}\ \text{-}5\ \textsf{else}\ \textsf{raise}\ B\\
\Rightarrow\ &\textsf{raise}\ B
\end{aligned}$$

$$\begin{aligned}
&(\textsf{mon}\ (\textsf{fun/c}\ \textsf{nat/c}\ \textsf{eager}\ \textsf{nat/c}\ \textsf{eager})\ \textsf{eager}\ (\lambda\ x.\ x\ -\ 10)\ B\ 5\\
\Rightarrow\ &(\textsf{read}\ \iota_0)\ 5\\
\mid\ \ &\textsf{write}\ \iota_0\ (\textsf{mon}\ (\textsf{fun/c}\ \textsf{nat/c}\ \textsf{eager}\ \textsf{nat/c}\ \textsf{eager})\ \textsf{eager}\ (\lambda\ x.\ x\ -\ 10)\ B)\\
\Rightarrow\ &(\lambda\ x.\ \textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ x\ (\textit{invert}\ B)))\ B)\ 5\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ 5\ (\textit{invert}\ B)))\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{read}\ \iota_1))\ B\ \mid\ \textsf{write}\ \iota_1\ (\textsf{nat/c}\ 5\ \bar{B})\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ (\textsf{read}\ \iota_1))\ B\ \mid\ \textsf{write}\ \iota_1\ (\textsf{if}\ \textit{nat?}\ 5\ \textsf{then}\ 5\ \textsf{else}\ \textsf{raise}\ \bar{B})\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ ((\lambda\ x.\ x\ -\ 10)\ 5)\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ (5\ -\ 10)\ B\\
\Rightarrow\ &\textsf{mon}\ \textsf{nat/c}\ \textsf{eager}\ \text{-}5\ B\\
\Rightarrow\ &(\textsf{read}\ \iota_2)\qquad\qquad\qquad\qquad\qquad\mid\ \textsf{write}\ \iota_2\ (\textsf{nat/c}\ \text{-}5\ B)\\
\Rightarrow\ &(\textsf{read}\ \iota_2)\qquad\qquad\qquad\qquad\qquad\mid\ \textsf{write}\ \iota_2\ (\textsf{if}\ \textit{nat?}\ \text{-}5\ \textsf{then}\ \text{-}5\ \textsf{else}\ \textsf{raise}\ B)\\
\Rightarrow\ &\textsf{raise}\ B
\end{aligned}$$

Figure 3.2. Separating an eager, function contract into a pattern of communication. We take *invert* to indicate blame inversion [36], use $\bar{B}$ to indicate the inverted blame value, and directly encode communication channels ι (defined in Chapter 4) for communication. The first trace depicts a single evaluator, with the verification reductions highlighted in *blue*. The second trace depicts verification with separate evaluators, where we use the " | " symbol to delineate evaluators.

$$\begin{aligned}
&\mathsf{let}\ x = \mathbf{mon}\ \mathsf{semi}\ \mathsf{nat/c}\ \text{-1}\ B\ \mathsf{in}\ x\ +\ 10 \\
\Rightarrow\ &\mathsf{let}\ x = \mathsf{delay}\ (\ldots\ \textit{check contract}\ \ldots)\ \mathsf{in}\ x\ +\ 10 \\
\Rightarrow\ &(\mathsf{read}\ \iota) + 10\ \mid\ \mathsf{write}\ \iota\ (\mathsf{nat/c}\ \text{-1}\ b) \\
\Rightarrow^*&(\mathsf{raise}\ B) + 10 \\
\Rightarrow\ &\mathsf{raise}\ B
\end{aligned} \tag{29}$$

We have taken delay here to serve as a delaying operation to allow us to avoid evaluating the semi-eager contract until its use. As described in the previous chapters, semi-eager verification ensures we do not verify contracts on unused values, and thus, in the first example, reducing "let" eliminates this delayed contract and the contract is never verified.

Notice how semi-eager verification compares to our previous, eager examples: in semi-eager verification, we delay creating the verification process until the delayed expression occurs in a forcing position. Once it does, however, we create the appropriate monitoring evaluator and perform verification while the user program awaits the result. Each strategy we have seen so far follows this style of variation: the monitoring form subverts the user program, establishing an appropriate pattern of communication to produce the desired verifier interactions.

This collection of insights about the core nature of contract verification situates us to begin construction of a multi-strategy contract verification system built using interacting processes to produce a wide range of strategy behaviors. To this end, we must define a language that supports these features.

CHAPTER 4

λ^{π}_{cs}: A Language for Implementing Runtime Verification as Patterns of Communication

SYNOPSIS

Thus far, we have explored contract verification strategies and introduced the notion of separating verification into additional processes that interact with the user program. To introduce formal semantics for this approach, we need a calculus with facilities to serve as a starting point for these encodings. In this chapter, we introduce λ^{π}_{cs}, a variant of Concurrent ML with additional facilities that will serve as the foundation of our strategy encodings (§4.2) and prove type safety for the core language (§4.3).

The previous chapter hinted at the tools we need for encoding contract verification as patterns of communication in a unified system: we require multiple processes, inter-process communication, errors with propagation, and delayed expressions with a forcing mechanism.

In this chapter, we introduce λ^{π}_{cs}, a variant of Concurrent ML [55, 74, 75] with multiple processes and a concurrent evaluation relation that will serve as a basis for our contract (and general runtime) verification framework. We present the static portion (sans types) of this language in Figure 4.1 and the dynamic portion in Figure 4.2; we define the type system in §4.3 at end of the chapter.

4.1. The Basics of the λ^{π}_{cs} Calculus

Our λ^{π}_{cs} calculus is, in general, unremarkable: as with most modern languages, our calculus supports process creation and communication events, raising and catching errors, and delaying and forcing individual terms in the usual way. These are common features in most modern languages, and, as a result, the contract framework we present in the next chapter is a highly-portable implementation.

At its core, λ^{π}_{cs} includes a term language e with values v, a term reduction relation "$\rightarrow$", a context-based reduction relation "$\longmapsto$" that works with evaluation contexts D and E (where $D \subseteq E$), and a concurrent reduction relation "$\Rightarrow$". The concurrent evaluation relation extends "$\longmapsto$" to a

Syntax

$$
\begin{array}{llll}
\textsc{Exprs} & e & := & x \mid v \mid e\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & \mid & e\ \mathit{binop}\ e \mid \mathit{unop}\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \\
 & & \mid & \mathsf{case}\ (e;\ x \rhd e;\ x \rhd e) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \\
 & & \mid & \mathsf{force}\ e \mid \mathsf{raise}\ e \mid \mathsf{catch}\ e\ e \\
 & & \mid & \mathsf{spawn}\ e \mid \mathsf{spawn}_f\ e \mid \mathsf{chan} \mid \mathsf{read}\ e \mid \mathsf{write}\ e\ e \\
 & & \mid & \mathbf{mon}\ e\ e\ e\ e \\
 & & & \\
\textsc{Values} & v & := & \lambda\ x.\ e \mid \iota \mid \mathsf{delay}\ e \mid (v, v) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \\
 & & \mid & n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid B \mid s \\
\textsc{Strategies} & s & := & \mathbf{eager} \mid \mathbf{semi} \mid \mathbf{promise} \mid \cdots \\
 & & & \\
\textsc{D-Contexts} & D & := & \square \mid D\ e \mid v\ D \mid \mathsf{if}\ D\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & \mid & D\ \mathit{binop}\ e \mid v\ \mathit{binop}\ D \mid \mathit{unop}\ D \\
 & & \mid & (D, e) \mid (v, D) \mid \mathsf{fst}\ D \mid \mathsf{snd}\ D \\
 & & \mid & \mathsf{case}\ (D;\ x \rhd e;\ x \rhd e) \mid \mathsf{inl}\ D \mid \mathsf{inr}\ D \\
 & & \mid & \mathsf{force}\ D \mid \mathsf{raise}\ D \mid \mathsf{catch}\ D\ e \\
 & & \mid & \mathsf{chan} \mid \mathsf{read}\ D \mid \mathsf{write}\ D\ e \mid \mathsf{write}\ v\ D \\
 & & \mid & \mathbf{mon}\ D\ e\ e\ e \mid \mathbf{mon}\ v\ D\ e\ e \mid \mathbf{mon}\ v\ v\ e\ D \\
 & & & \\
\mathcal{E}\textsc{-Contexts} & \mathcal{E} & := & D \mid D[\mathcal{E}] \mid \mathsf{catch}\ v\ \mathcal{E} \\
 & & & \\
\textsc{ProcId} & \pi & \in & \mathbb{N} \\
\textsc{Process} & \mathit{proc} & = & \langle e_i \rangle^{\pi_i} \\
\textsc{Proc. Set} & P & \in & \mathit{Fin}(\mathit{proc}) \\
 & & & \\
\textsc{Proc. Decomp.} & P + \langle e \rangle^{\pi} & \equiv & P \cup \{\langle e \rangle^{\pi}\} \\
 & & & \\
\textsc{Proc. Config.} & K, T, P & & K \in \mathit{Fin}(\text{channel names}) \\
 & & & T \in \mathit{Fin}(\text{process ids})
\end{array}
$$

Figure 4.1. Syntax definitions for λ^{π}_{cs}.

finite set of processes (i.e., terms with associated process identification numbers) and introduces reductions to handle channel creation, process creation, and process communication.

4.2. Language Features

In this section, we present λ^{π}_{cs} in detail, discussing the extended features in detail in preparation for defining our monitoring framework.

4.2.1. *Term Language Features*

The term language e contains a number of standard operations [73]. We define these operations in the usual way, introducing a term reduction relation "$\rightarrow$" to perform subject reductions and a context-level reduction relation "$\longmapsto$" to lift these subject reductions to the evaluation contexts D

Dynamic Semantics

$\boxed{e \to e'}$

$(\lambda\ x.\ e)\ v$	$\to$	$e[v/x]$	
$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2$	$\to$	e_1	
$\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2$	$\to$	e_2	
$v_1\ binop\ v_2$	$\to$	v	*where* $\delta(binop, v_1, v_2) = v$
$unop\ v'$	$\to$	v	*where* $\delta(unop, v') = v$
$\mathsf{case}\ (\mathsf{inl}\ v;\ x_1 \rhd e_1;\ x_2 \rhd e_2)$	$\to$	$e_1[v/x_1]$	
$\mathsf{case}\ (\mathsf{inr}\ v;\ x_1 \rhd e_1;\ x_2 \rhd e_2)$	$\to$	$e_2[v/x_2]$	
$\mathsf{force}\ (\mathsf{delay}\ e)$	$\to$	e	
$\mathsf{force}\ v$	$\to$	v	*where* $v \neq \mathsf{delay}\ e$
$\mathsf{catch}\ v_1\ v_2$	$\to$	v_2	
$\mathsf{catch}\ v_1\ (\mathsf{raise}\ v_2)$	$\to$	$v_1\ v_2$	

$\boxed{e \longmapsto e'}$

$$\frac{e \to e'}{\mathcal{E}[e] \longmapsto \mathcal{E}[e']} \qquad \frac{}{\mathcal{E}[D[\mathsf{raise}\ v_2]] \longmapsto \mathcal{E}[\mathsf{raise}\ v_2]}$$

$\boxed{e_1 \overset{\iota}{\frown\!\!\smile} e_2 \text{ with } (e_1', e_2')}$

$$\frac{}{\mathsf{write}\ \iota\ v \overset{\iota}{\frown\!\!\smile} \mathsf{read}\ \iota \text{ with } (\mathsf{unit}, v)} \qquad \frac{e_1 \overset{\iota}{\frown\!\!\smile} e_2 \text{ with } (e_1', e_2')}{e_2 \overset{\iota}{\frown\!\!\smile} e_1 \text{ with } (e_2', e_1')}$$

$\boxed{\textit{Notation}}$

$$P + \langle e \rangle^{\pi} \approx P \uplus \langle e \rangle^{\pi}$$

$\boxed{K, T, P \Rightarrow K', T, P'}$

$$\frac{e \longmapsto e'}{K, T, P + \langle e \rangle^{\pi} \Rightarrow K, T, P + \langle e' \rangle^{\pi}}$$

$$\frac{\pi' \notin \mathrm{dom}(P)}{K, T, P + \langle \mathcal{E}[\mathsf{spawn}\ e\ ] \rangle^{\pi} \Rightarrow K, T, P + \langle \mathcal{E}[\mathsf{unit}] \rangle^{\pi} + \langle e \rangle^{\pi'}}$$

$$\frac{\pi' \notin \mathrm{dom}(P)}{K, T, P + \langle \mathcal{E}[\mathsf{spawn}_f\ e\ ] \rangle^{\pi} \Rightarrow K,\ T \cup \{\pi'\},\ P + \langle \mathcal{E}[\mathsf{unit}] \rangle^{\pi} + \langle e \rangle^{\pi'}}$$

$$\frac{\iota \notin K}{K, T, P + \langle \mathcal{E}[\mathsf{chan}] \rangle^{\pi} \Rightarrow K \cup \{\iota\}, T, P + \langle \mathcal{E}[\iota] \rangle^{\pi}}$$

$$\frac{e_1 \overset{\iota}{\frown\!\!\smile} e_2 \text{ with } (e_1', e_2') \quad \iota \in K}{K, T, P + \langle \mathcal{E}_1[e_1] \rangle^{\pi_1} + \langle \mathcal{E}_2[e_2] \rangle^{\pi_2} \Rightarrow K, T, P + \langle \mathcal{E}_1[e_1'] \rangle^{\pi_1} + \langle \mathcal{E}_2[e_2'] \rangle^{\pi_2}}$$

Figure 4.2. Dynamic semantics for λ_{cs}^{π}.

and E. These reduction relations also include operators to raise and catch errors and to delay and force expressions, which we now explain in depth as each plays a critical role in encoding verification strategies in the next chapter.

Raising and Catching Errors. In order to construct contracts that report errors, we require error-signalling facilities. To this end, we introduce "raise" and "catch" in λ_{cs}^{π}, which allow us to signal errors (via "raise") and catch and handle them (via "catch"). To simplify our type system, we specialize these operators to blame values B, which are opaque tuples of blame information in the style of Dimoulas et al. [27]. Raising an error escapes the surrounding program context, excluding handlers (i.e., D contexts):

$$((\lambda\ x.\ x\ +\ 10)(\mathsf{raise}\ B\ +\ 100)) \rightarrow^{*} \mathsf{raise}\ B$$

As expected, the "catch" operation uses a handler expression to process errors, yielding values instead.

$$\begin{array}{ll} & \mathsf{catch}\ (\lambda\ x.\ 5)\ (100\ +\ (\mathsf{if}\ \mathit{even?}\ 5\ \mathsf{then}\ 5\ \mathsf{else\ raise}\ B)) \\ \longmapsto & \mathsf{catch}\ (\lambda\ x.\ 5)\ (\mathsf{raise}\ B) \\ \longmapsto & (\lambda\ x.\ 5)\ B \\ \longmapsto & 5 \end{array}$$

In this example, the error in the alternative branch of the "if" expression propagates through the addition expression before arriving at the catch form. The catch form then passes he blame value B as an argument to the handler "$\lambda\ x.\ 5$," which discards the argument and yields "5" as the expression result.

Conversely, the evaluator discards the handler form if the term inside the catch expression terminates with a value:

$$\begin{array}{ll} & \mathsf{catch}\ (\lambda\ x.\ 5)\ (\mathsf{if}\ \mathit{even?}\ 4\ \mathsf{then}\ 4\ \mathsf{else\ raise}\ B) \\ \longmapsto & \mathsf{catch}\ (\lambda\ x.\ 5)\ 4 \\ \longmapsto & 4 \end{array}$$

In this example, the expression inside the handler completes with the value 4 and the handler yields this as the final result.

In the next chapter, we will use "raise" and "catch" to signal and propagate contract errors.

Delaying and Forcing Expressions. The delay and force operations allow us to create thunk-like objects [53], forcing their evaluation at a later point. Individual delayed expressions do not reduce outside of force operations:

$$\mathsf{delay}\ (10\ +\ 5) \not\longmapsto$$

As expected, the forcing form extracts and evaluates the delayed expression:

$$\begin{array}{ll} & \mathsf{let}\ x = \mathsf{delay}\ (10\ +\ 5)\ \mathsf{in}\ (1\ +\ 2)\ +\ \mathsf{force}\ x \\ \longmapsto & (1\ +\ 2)\ +\ \mathsf{force}\ (\mathsf{delay}\ (10\ +\ 5)) \\ \longmapsto & 3\ +\ \mathsf{force}\ (\mathsf{delay}\ (10\ +\ 5)) \\ \longmapsto & 3\ +\ (10\ +\ 5) \\ \longmapsto^{*} & 18 \end{array}$$

In this example, we create a delayed expression "x" and later force it, trigger the evaluation of "$(10\ +\ 5)$" in the body of the let expression.

Finally, observe that delayed expressions inhibit raising errors:

$$\mathsf{catch}\ (\lambda\ x.\ 5)\ (\mathsf{delay}\ (\mathsf{raise}\ B)) \longmapsto \mathsf{delay}\ (\mathsf{raise}\ B)$$

In the next chapter, we will use "delay" and "force" to postpone contract verification in the case of semi-eager and promise-based verification strategies.

4.2.2. *Process-Level Operations*

The final feature of λ^{π}_{cs} that we introduce is the process component, which lifts individual expressions into a multi-process collection, providing rules for process creation, communication, and individual process reduction. We define processes in λ^{π}_{cs} via process identification numbers π such that $\langle e\rangle^{\pi}$ is a process. Next, we define *process configurations* K, T, P as a three-tuple:

- K is a set of channel names used in the configuration
- T is a *termination set* of process identification numbers, indicating the processes that need to complete for a configuration to be considered an *answer configuration*, defined as:

  Definition 4.1 (Answer Configuration). *A process configuration K, T, P is an answer configuration if, for each $\pi \in T$, $\langle e\rangle^{\pi} \in P$ and $e \not\longmapsto$.*

  We will use this definition of answer configuration in Chapter 5 to determine when finally-concurrent verifiers are complete.
- P is the set of processes $\langle e\rangle^{\pi}$ in the configuration.

When convenient, we will elide K and T from our traces and write $P + \langle e\rangle^{\pi}$ to mean $P \uplus \langle e\rangle^{\pi}$ to simplify presentation.

Process Reduction. As mentioned above, "$\Rightarrow$" extends "$\longmapsto$" to process configurations K, T, P such that any valid "$\longmapsto$" reduction may be performed by an individual process:

$$K, T, \{\langle \mathsf{catch}\ f\ (10\ +\ \mathsf{raise}\ B)\rangle^{\pi}\} \ \Rightarrow\ K, T, \{\langle \mathsf{catch}\ f\ (\mathsf{raise}\ B)\rangle^{\pi}\} \ \Rightarrow\ K, T, \{\langle f\ B\rangle^{\pi}\}$$

Furthermore, "$\Rightarrow$" is non-deterministic, and thus each of the following reductions are valid in λ^{π}_{cs}:

$$K, T, \{\langle 10\ +\ 20\rangle^{\pi_0}, \langle 20\ +\ 30\rangle^{\pi_1}\} \ \Rightarrow\ K, T, \{\langle 30\rangle^{\pi_0}, \langle 20\ +\ 30\rangle^{\pi_1}\} \ \Rightarrow\ K, T, \{\langle 30\rangle^{\pi_0}, \langle 50\rangle^{\pi_1}\}$$
$$K, T, \{\langle 10\ +\ 20\rangle^{\pi_0}, \langle 20\ +\ 30\rangle^{\pi_1}\} \ \Rightarrow\ K, T, \{\langle 10\ +\ 20\rangle^{\pi_0}, \langle 50\rangle^{\pi_1}\} \ \Rightarrow\ K, T, \{\langle 30\rangle^{\pi_0}, \langle 50\rangle^{\pi_1}\}$$

Each of these reductions is valid in λ^{π}_{cs}, and we will use this behavior later to provide "best-effort" checking.

Finally, observe that processes continue to exist in the configuration after evaluation: if a process terminates in some value v, it will remain in the process set without further reduction:

$$K, T, \{\langle \mathsf{unit}\rangle^{\pi}, \langle 2\ +\ 3\rangle^{\pi'}\} \ \Rightarrow K, T, \{\langle \mathsf{unit}\rangle^{\pi}, \langle 5\rangle^{\pi'}\} \ \not\Rightarrow$$

This process behavior allows us to use monotonicity arguments in proving type safety in §4.3.

Process Creation. We add new processes to a configuration with the "spawn" operation[1]:

$$K, T, \{\langle \mathsf{seq}\ (\mathsf{spawn}\ e\ )\ e_2\rangle^{\pi}\} \ \Rightarrow\ K, T, \{\langle \mathsf{seq}\ \mathsf{unit}\ e_2\rangle^{\pi}, \langle e\rangle^{\pi'}\}$$

To create a new process, we allocate a unique process identification number and add the process to the process configuration. If, however, we create the process spawn_f, we add its process identification number to the termination set T, ensuring the process will run to completion before considering the configuration an *answer configuration*:

$$\begin{array}{rll} & K, \{\pi_0\}, \{\langle \mathsf{seq}\ (\mathsf{spawn}\ (10\ +\ 5)\ )\ 2\rangle^{\pi_0}\} & \\ \Rightarrow & K, \{\pi_0\}, \{\langle \mathsf{seq}\ \mathsf{unit}\ 2\rangle^{\pi_0}, \langle 10\ +\ 5\rangle^{\pi_1}\} & \\ \Rightarrow & K, \{\pi_0\}, \{\langle 2\rangle^{\pi_0}, \langle 10\ +\ 5\rangle^{\pi_1}\} & \textit{Answer Configuration} \end{array} \qquad (30)$$

$$\begin{array}{rll} & K, \{\pi_0\}, \{\langle \mathsf{seq}\ (\mathsf{spawn}_f\ (10\ +\ 5)\ )\ 2\rangle^{\pi_0}\} & \\ \Rightarrow & K, \{\pi_0, \pi_1\}, \{\langle \mathsf{seq}\ \mathsf{unit}\ 2\rangle^{\pi_0}, \langle 10\ +\ 5\rangle^{\pi_1}\} & \\ \Rightarrow & K, \{\pi_0, \pi_1\}, \{\langle 2\rangle^{\pi_0}, \langle 10\ +\ 5\rangle^{\pi_1}\} & \\ \Rightarrow & K, \{\pi_0, \pi_1\}, \{\langle 2\rangle^{\pi_0}, \langle 15\rangle^{\pi_1}\} & \textit{Answer Configuration} \end{array} \qquad (31)$$

In the first example, "spawn" creates π_1, which does not add its process identification number to T, and so we consider the configuration complete when the process π_0 finishes. In the second example, however, "spawn_f"' adds π_1 to T, and thus process π_1 runs to completion before the configuration

[1]We use "seq" to mean "$\lambda\ x\ y.\ y$" and "$\mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2$" to mean "$(\lambda\ x.\ e_2)\ e_1$"; we also use their extended, multi-argument forms in the usual way.

is complete. This difference will play a role in defining our concurrent and finally-concurrent verification strategies in the next chapter.

Process Communication. Processes may communicate across channels ι via the "read" and "write" operators. We use "chan" to create a new channel, add it to K, and continues the program with it:

$$\begin{array}{ll} & K, T, \{\langle \mathsf{let}\ i = \mathsf{chan}\ \mathsf{in}\ \mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{write}\ i\ 10)\ )\ (\mathsf{read}\ i)\rangle^{\pi}\} \\ \Rightarrow & K \uplus \{\iota\}, T, \{\langle \mathsf{let}\ i = \iota\ \mathsf{in}\ \mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{write}\ i\ 10)\ )\ (\mathsf{read}\ i)\rangle^{\pi}\} \\ \Rightarrow & K \uplus \{\iota\}, T, \{\langle \mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{write}\ \iota\ 10)\ )\ (\mathsf{read}\ \iota)\rangle^{\pi}\} \end{array}$$

We define inter-process communication in terms of *matched events* in Figure 4.2:

$$e_1 \overset{\iota}{\frown\!\!\!\smile} e_2 \text{ with } (e_1', e_2')$$

The matched event relation ensures that processes only communicate values, and that communication proceeds regardless of process order in P (since P is an unordered set).

When two processes have matched events, they may communicate as a reduction, wherein the writer continues with unit and the reader continues with the written value:

$$\begin{array}{ll} & K \uplus \{\iota\}, T, \{\langle \mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{write}\ \iota\ 10)\ )\ (\mathsf{read}\ \iota)\rangle^{\pi}\} \\ \Rightarrow^{*} & K \uplus \{\iota\}, T, \{\langle \mathsf{read}\ \iota\rangle^{\pi}, \langle \mathsf{write}\ \iota\ 10\rangle^{\pi'}\} \\ \Rightarrow & K \uplus \{\iota\}, T, \{\langle 10\rangle^{\pi}, \langle \mathsf{unit}\rangle^{\pi'}\} \end{array}$$

Here, $\mathsf{write}\ \iota\ 10$ matches with $\mathsf{read}\ \iota$ at line two as:

$$\mathsf{write}\ \iota\ 10 \overset{\iota}{\frown\!\!\!\smile} \mathsf{read}\ \iota \text{ with } (\mathsf{unit}, 10)$$

After the reduction, the writing process continues with unit and the reading process continues with 10.

4.3. Types & Type Safety

Before we continue with defining contract verification in λ^{π}_{cs}, we pause to prove type safety for this language core. Our proof follows Reppy [74]—we present a type system for the term language, define "well-formed" process configurations based on well-formed terms, and use process configuration "evaluation traces" to define type safety for a process collection. We have also formalized much of this chapter in Coq[2].

[2]`https://www.github.com/cgswords/dissertation`

4.3.1. *Typing Rules*

We begin with a term-level type system, presented in Figure 4.3, which follows standard conventions [73]: we provide direct types for our built-in features, including base types, standard type constructors, and blame types. Our type judgments include two environments:

- The Γ environment associates types with variables, and we use it in the standard way to ensure arguments and bindings are correctly-typed.
- The Δ environment associates types with channels across an entire process collection to ensure well-typed communication.

Our type judgments are generally standard, but include the following anomalies:

- We include errors of the form raise e, typing them at any type (to allow them to occur anywhere in an expression).
- We include a `blame` type for opaque blame values B, which are akin to ML's `exn` types [67] insofar as we only use blame to carry error information when an expression raises a contract violation.
- We include a `strat` type as the type of verification strategies.
- The delay construct is unusually typed: to ease the return type of **mon**, we type a delayed expression at its internal type; otherwise, we would need to parameterize **mon**'s return type based on the input strategy, which would induce tracking verification strategies at the type level and employing type-level meta-functions to determine the correct type structure in each case, adding immense user complexity. Since real-world software contract systems occur predominantly in dynamically-typed languages [36, 37, 81], this simplification seems a practical allowance. This allowance, however, means that we need to consider a class of well-typed but irreducible, or *unforced*, terms, such as "(delay (λ x. e)) e'." We formalize this situation as a relation in Figure 4.4.

To type process configurations, we first define *well-formed* configurations:

Definition 4.2 (Well-Formed Configuration). *A process configuration K, T, P is well-formed if*

- *for each process $\langle e \rangle^{\pi}$,*
  - *e contains no free variables,*
  - *there is no $e' \not\equiv e$ such that $\langle e' \rangle^{\pi} \in P$, i.e., each process in P is uniquely numbered,*
- *the set of terminating process ids $T \subseteq dom(P)$;*

Type Definitions Judgments

$$
\begin{array}{lcll}
\textsc{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{blame} \mid () \mid \tau \to \tau \mid \tau + \tau \mid \tau \times \tau \\
 & & \mid & \texttt{chan}\ \tau \mid \texttt{strat} \\
\textsc{Typ Envs.} & \Gamma & ::= & \cdot \mid (x, \tau), \Gamma \\
\textsc{Chan Envs.} & \Delta & ::= & \cdot \mid (\iota, \tau), \Gamma \\
\textsc{Proc Types.} & \rho & ::= & \cdot \mid (\pi, \tau), \rho
\end{array}
$$

Term Typing Judgments

$\boxed{\Gamma; \Delta \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma; \Delta \vdash x : \tau} \qquad \frac{\Gamma; \Delta \vdash e_1 : \tau_1 \to \tau \quad \Gamma; \Delta \vdash e_2 : \tau_1}{\Gamma; \Delta \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \texttt{bool} \quad \Gamma; \Delta \vdash e_2 : \tau \quad \Gamma; \Delta \vdash e_3 : \tau}{\Gamma; \Delta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau \to \texttt{blame} \to \tau \quad \Gamma; \Delta \vdash e_2 : \texttt{strat} \quad \Gamma; \Delta \vdash e_3 : \tau \quad \Gamma; \Delta \vdash e_4 : \texttt{blame}}{\Gamma; \Delta \vdash \mathbf{mon}\ e_1\ e_2\ e_3\ e_4 : \tau}$$

$$\frac{\begin{array}{c}\delta_\tau(binop) = \tau_1 \to \tau_2 \to \tau \\ \Gamma; \Delta \vdash e_1 : \tau_1 \quad \Gamma; \Delta \vdash e_2 : \tau_2\end{array}}{\Gamma; \Delta \vdash e_1\ binop\ e_2 : \tau} \qquad \frac{\delta_\tau(unop) = \tau_1 \to \tau \quad \Gamma; \Delta \vdash e : \tau_1}{\Gamma; \Delta \vdash unop\ e : \tau}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \quad \Gamma; \Delta \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma; \Delta \vdash e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma; \Delta \vdash e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash \mathsf{snd}\ e : \tau_2}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1 + \tau_2 \quad (x_1, \tau_1), \Gamma; \Delta \vdash e_1 : \tau \quad (x_2, \tau_2), \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta \vdash \mathsf{case}\ (e;\ x_1 \rhd e_1;\ x_2 \rhd e_2) : \tau}$$

$$\frac{\Gamma; \Delta \vdash e : \tau_1}{\Gamma; \Delta \vdash \mathsf{inl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma; \Delta \vdash e : \tau_2}{\Gamma; \Delta \vdash \mathsf{inr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathsf{force}\ e : \tau} \qquad \frac{\Gamma; \Delta \vdash e : \texttt{blame}}{\Gamma; \Delta \vdash \mathsf{raise}\ e : \tau} \qquad \frac{\Gamma; \Delta \vdash e_1 : \texttt{blame} \to \tau \quad \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta \vdash \mathsf{catch}\ e_1\ e_2 : \tau}$$

$$\frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathsf{spawn}_f\ e\ : ()} \qquad \frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathsf{spawn}\ e\ : ()} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{chan} : \texttt{chan}\ \tau}$$

$$\frac{\Gamma; \Delta \vdash e : \texttt{chan}\ \tau}{\Gamma; \Delta \vdash \mathsf{read}\ e : \tau} \qquad \frac{\Gamma; \Delta \vdash e_1 : \texttt{chan}\ \tau \quad \Gamma; \Delta \vdash e_2 : \tau}{\Gamma; \Delta \vdash \mathsf{write}\ e_1\ e_2 : ()}$$

$$\frac{(x, \tau_1), \Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \lambda\ x : \tau_1.\ e : \tau_1 \to \tau} \qquad \frac{\Delta(\iota) = \tau}{\Gamma; \Delta \vdash \iota : \texttt{chan}\ \tau} \qquad \frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathsf{delay}\ e : \tau} \qquad \frac{}{\Gamma; \Delta \vdash n : \texttt{int}}$$

$$\frac{}{\Gamma; \Delta \vdash \mathsf{true} : \texttt{bool}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{false} : \texttt{bool}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{unit} : ()} \qquad \frac{}{\Gamma; \Delta \vdash b : \texttt{blame}}$$

Figure 4.3. Term-Language Typing Judgments for λ^{π}_{cs}.

Unforced Terms

$$\boxed{\mathit{unforced}(e)}$$

$$\frac{}{\mathit{unforced}((\mathsf{delay}\ e)\ v)} \qquad \frac{}{\mathit{unforced}(\mathsf{if}\ (\mathsf{delay}\ e)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)}$$

$$\frac{}{\mathit{unforced}((\mathsf{delay}\ e)\ \mathit{binop}\ e')} \qquad \frac{}{\mathit{unforced}(v\ \mathit{binop}\ (\mathsf{delay}\ e))}$$

$$\frac{}{\mathit{unforced}(\mathit{unop}\ (\mathsf{delay}\ e))} \qquad \frac{}{\mathit{unforced}(\mathsf{fst}\ (\mathsf{delay}\ e))} \qquad \frac{}{\mathit{unforced}(\mathsf{snd}\ (\mathsf{delay}\ e))}$$

$$\frac{}{\mathit{unforced}(\mathsf{case}\ ((\mathsf{delay}\ e);\ x_1 \rhd e_1;\ x_2 \rhd e_2))} \qquad \frac{}{\mathit{unforced}(\mathsf{raise}\ (\mathsf{delay}\ e))}$$

$$\frac{}{\mathit{unforced}(\mathbf{mon}\ (\mathsf{delay}\ e)\ e_2\ e_3\ e_4)} \qquad \frac{}{\mathit{unforced}(\mathbf{mon}\ v\ (\mathsf{delay}\ e)\ e_3\ e_4)}$$

$$\frac{}{\mathit{unforced}(\mathsf{read}\ (\mathsf{delay}\ e))} \qquad \frac{}{\mathit{unforced}(\mathsf{write}\ (\mathsf{delay}\ e)\ e')}$$

$$\frac{\mathit{unforced}(e)}{\mathit{unforced}(\mathcal{E}[\mathrm{e}])}$$

Figure 4.4. Unforced terms due to misplaced delay expressions for λ^{π}_{cs}.

- *and the set of free channels $FC(P) \subseteq K$.*

The idea is that a process configuration is well-formed if each individual process contains a well-formed expression, each process has a unique process identification number π, T contains only process identification numbers for processes in P, and the associated channel list K contains all the channels that occur in P.

We type processes with a map ρ from process ids π to associated types τ in a channel environment Δ as:

Definition 4.3 (Process Typing). *A well-formed configuration K, T, P has type ρ under channel environment Δ, written*

$$\Delta \vdash K, T, P : \rho$$

if:

- $K \subseteq \mathit{dom}(\Delta)$
- $\mathit{dom}(P) \subseteq \mathit{dom}(\rho)$
- *for each* $\langle e \rangle^{\pi} \in P$, $\cdot, \Delta \vdash e : \rho(\pi)$

4.3.2. *Proving Type Safety for λ_{cs}^{π}*

With our type system in place, we now prove type safety for λ_{cs}^{π}. To prove type safety, we perform the following steps:

(1) We define *well-formed* reductions for process configurations in terms of *traces* and *computations*.
(2) We show term-level type safety
(3) Finally, we extend term-level type safety to the configuration level via these traces and computations.

To start, observe that well-formedness is closed under reduction:

LEMMA 4.1 (Well-Formed Step). *If K, T, P is well-formed under some ρ, Δ and $K, T, P \Rightarrow K', T', P'$, then there exist some ρ', Δ' such that $\Delta \subseteq \Delta'$, $\rho \subseteq \rho'$, and $\Delta' \vdash K', T', P' : \rho'$ (i.e., K', T', P' is well-formed).*

PROOF. By inversion on $\Rightarrow$. □

COROLLARY 4.1 (Well-Formed Multistep). *Lemma 4.1 extends for $\Rightarrow^*$*

PROOF. By induction on the length of the evaluation sequence. □

Traces & Computations. To prove process-level type safety, we introduce the notion of traces [74] to deal with the non-deterministic nature of '$\Rightarrow$' in λ_{cs}^{π}. We use π_0 to indicate the process identifier of the initial process in any given computation, and define traces as: We define traces as:

DEFINITION 4.4 (Trace). *A trace $\mathcal{T}$ is a (possibly infinite) sequence of well-formed configurations*

$$\mathcal{T} = \langle K_0, T_0, P_0; K_1, T_1, P_1; \cdots \rangle$$

such that $K_i, T_i, P_i \Rightarrow K_{i+1}, T_{i+1}, P_{i+1}$.

By Corollary 4.1, if K_0, T_0, P_0 is well-formed, then any sequence of evaluation steps starting with K_0, T_0, P_0 is a trace. Next, we define the *possible* states of a process with respect to a configuration as:

DEFINITION 4.5 (Process States). *Let P be a well-formed process set and let $\langle e \rangle^{\pi} \in P$. The state of π in P is either zombie, unforced, blocked, or ready, depending on the form of e:*

- *if $e \in v$, then it is a zombie;*
- *if $unforced(e)$, then it is unforced;*
- *if $e = E[e_0]$ and $e_0 =$ read ι or $e_0 =$ write ι v and there does not exist some $\langle E'[e'] \rangle^{\pi'} \in P$ with $e_0 \overset{\iota}{\smile} e_1$, then π is blocked in P;*
- *otherwise, π is ready in P.*

We define the set of ready processes as $Ready(P)$.

Next, we define that terminal configurations P as:

Definition 4.6 (Terminal Configuration). *a configuration P is a terminal configuration if*

$$Ready(P) = \emptyset$$

We can see that any terminal configuration with only blocked processes is *deadlocked* in the usual sense [20].

Now, we define *computations* which quantify over non-deterministic reductions:

Definition 4.7 (Computation). *A computation is a maximal trace that is either infinite or is finite and ends in a terminal configuration. If e is an expression, then we define the computations of e to be:*

$$Comp(e) = \{\mathcal{T} \mid \mathcal{T} \textit{ is a trace with } K_0 = \emptyset, T_0 = \{\pi_0\}, P_0 = \{\langle e \rangle^{\pi_0}\}\}$$

Next, we define the set of trace processes, which describe each process created over the course of a computation:

Definition 4.8 (Trace Processes).

$$Procs(\mathcal{T}) = \{\pi \mid \exists K_i, T_i, P_i \in \mathcal{T} \textit{ with } \pi \in dom(P_i)\}$$

Finally, our non-deterministic reduction semantics forces us to define notions of convergence and divergence *relative to the computation of an expression.*

Definition 4.9 (Convergence and Divergence). *A process $\pi \in Procs(\mathcal{T})$:*

- *converges to a value v in $\mathcal{T}$, written $\pi \Downarrow_{\mathcal{T}} v$, if $K, T, P + \langle v \rangle^\pi \in \mathcal{T}$;*
- *is unforced in T if it evaluates to some expression e in $\mathcal{T}$, written $\pi \Downarrow_{\mathcal{T}} e$, if $K, T, P + \langle e \rangle^\pi \in \mathcal{T}$ and $unforced(e)$;*

- *converges to an error* raise B *in* $\mathcal{T}$, *written* $\pi \Downarrow_{\mathcal{T}}$ raise B, *if* $K, T, P + \langle \text{raise } B\rangle^{\pi} \in \mathcal{T}$.
- *diverges in* $\mathcal{T}$, *written* $\pi \Uparrow_{\mathcal{T}}$, *if for every* $K, T, P \in \mathcal{T}$, *with* $\pi \in dom(P)$, π *is ready or blocked.*

Our divergence includes deadlocked processes, processes that enter infinite loops, and process which the configuration does not evaluate enough to terminate. (We use this latter case to describe "best-effort" contracts.)

Type Safety. Finally, we define type safety for term languages via progress and preservation [85], abstracted over Δ:

Lemma 4.2 (Term Progress). *If there exists* Δ *such that* $\cdot, \Delta \vdash e : \tau$, *then either:*

- $\exists e', e \longmapsto e'$
- $e \in v$
- $e =$ raise B, *for some* B
- *unforced*(e)
- *or* e *is a process reduction.*

Proof. Straightforward, by induction on e. We have formalized this proof in Coq. □

Lemma 4.3 (Term Preservation). *If there exists* Δ *and* e' *such that* $\cdot, \Delta \vdash e : \tau$ *and* $e \longmapsto e'$, *then* $\cdot, \Delta' \vdash e' : \tau$.

Proof. Straightforward, by induction on $\longmapsto$ (and therein, $\rightarrow$). We have formalized this proof in Coq. □

Next, we define preservation for process collections:

Lemma 4.4 (Concurrent Type Preservation). *If a configuration is well-formed with* $K, T, P \Rightarrow K', T', P'$, *and there exists* Δ *such that* $\Delta \vdash K, T, P : \rho$, *then there is some channel typing* Δ' *and process typing* ρ' *such that:*

- $\Delta \subseteq \Delta'$
- $\rho \subseteq \rho'$
- $\Delta' \vdash K', T', P' : \rho'$
- $\Delta' \vdash K, T, P : \rho'$

Proof (Sketch). The fourth property follows from the first three; the others proceed by induction on $K, T, P \Rightarrow K', T', P'$. $\square$

Now, toward defining preservation for process configurations, we show that we can classify any given expression:

Lemma 4.5 (Uniform Evaluation). *If e is an expression with some trace $\mathcal{T} \in Comp(e)$ where $\pi \in Procs(\mathcal{T})$, then either $\pi \Uparrow_{\mathcal{T}}$, $\pi \Downarrow_{\mathcal{T}} v$, $\pi \Downarrow_{\mathcal{T}}$ raise v, $\pi \Downarrow_{\mathcal{T}} e'$ with unforced(e'), or $P_i(\pi)$ is stuck for some $K_i, T_i, P_i \in \mathcal{T}$.*

Proof (Sketch). This follows immediately from the definitions. $\square$

Our next lemma states that any stuck, but not *unforced* term, is untypeable:

Lemma 4.6 (Untypeability of Wrong Configurations). *If $P(\pi)$ is irreducible and not unforced(e) in a well-formed configuration K, T, P, then there are not some Δ, ρ such that $\cdot, \Delta \vdash P(\pi) : \rho(\pi)$. In other words, K, T, P is untypeable.*

Proof (Sketch). We assume, toward a contradiction, that there are some Δ and ρ that correctly type $P(\pi)$. Then $P(\pi) = E[e']$ for some E and e', and it suffices to show that e' is untypeable, which is a contradiction. Our proof proceeds by induction on the possible structures of e', demonstrating that each redex is reducible if it is typeable, a contradiction since it this redex is stuck, and thus the redex must not be typeable. $\square$

Next, we show syntactic soundness by first demonstrating uniform evaluation, e.g., that every program is in one of four states.

Theorem 4.1 (Syntactic Soundness). *Let e be an expression with $\cdot, \cdot \vdash e : \tau$. Then for any $\mathcal{T} \in Comp(e)$, $\pi \in Procs(\mathcal{T})$, with K_i, T_i, P_i the first occurrence of π in $\mathcal{T}$, there exist Δ, ρ such that*

- $\Delta \vdash K_i, T_i, P_i : \rho$
- $\rho(\pi_0) = \tau$
- *and one of the following holds:*
  - $\pi \Uparrow_{\mathcal{T}}$
  - $\pi \Downarrow_{\mathcal{T}} v$ *such that there exists* $\Delta', \Delta \subseteq \Delta'$ *and* $\cdot, \Delta' \vdash v : \rho(\pi)$

– $\pi \Downarrow_{\mathcal{T}}$ *raise* B

– $\pi \Downarrow_{\mathcal{T}} e'$, *unforced*$(e')$ *such that there exists* $\Delta', \Delta \subseteq \Delta'$ *and* $\cdot, \Delta' \vdash e' : \rho(\pi)$

Proof. The existence of Δ and ρ follow from Lemma 4.4. By Lemma 4.5 (the uniform evaluation lemma), we know that then either $\pi \Uparrow_{\mathcal{T}}$, $\pi \Downarrow_{\mathcal{T}} v$, $\pi \Downarrow_{\mathcal{T}}$ raise v, $\pi \Downarrow_{\mathcal{T}} e'$ with *unforced*(e'), or $P_i(\pi)$ is stuck for some $K_j, T_j, P_j \in \mathcal{T}$.

Assume toward a contradiction that π is stuck in K_j, T_j, P_j. By Lemma 4.1, K_j, T_j, P_j is well-formed, and, by Lemma 4.6, it must be untypeable. But, since the configuration $\emptyset, \{\pi_0\}, \langle e\rangle^{\pi_0}$ is typeable, by Lemma 4.4, there is a Δ', ρ' such that $\Delta' \vdash K_j, P_j, T_j : \rho'$, a contradiction. Thus π cannot be stuck.

Otherwise, $\pi \Uparrow_{\mathcal{T}}$, $\pi \Downarrow_{\mathcal{T}} v$, $\pi \Downarrow_{\mathcal{T}}$ raise v, or $\pi \Downarrow_{\mathcal{T}} e'$ with *unforced*(e').

If $\pi \Uparrow_{\mathcal{T}}$ or $\pi \Downarrow_{\mathcal{T}}$ raise B, we are done (in the former case due to divergence and in the latter because raise B is well-typed at any type).

If $\pi \Downarrow_{\mathcal{T}} v$, then let $K_j, T_j, P_j \in \mathcal{T}$ such that $P_j(\pi) = v$. By Lemma 4.9, there is some Δ' and ρ' such that $\Delta \subseteq \Delta'$ and $\rho \subseteq \rho'$ such that $\Delta' \vdash P_j : \rho'$. Since $\rho \subseteq \rho'$, $\rho(\pi) = \rho'(\pi)$, and so $\cdot, \Delta' \vdash v : \rho(\pi)$.

If $\pi \Downarrow_{\mathcal{T}} e'$, *unforced*$(e')$, then let $K_j, T_j, P_j \in \mathcal{T}$ such that $P_j(\pi) = e'$. By Lemma 4.9, there is some Δ' and ρ' such that $\Delta \subseteq \Delta'$ and $\rho \subseteq \rho'$ such that $\Delta' \vdash P_j : \rho'$. Since $\rho \subseteq \rho'$, $\rho(\pi) = \rho'(\pi)$, and so $\cdot, \Delta' \vdash v : \rho(\pi)$. □

Before stating soundness, we define evaluation of a process identification in a trace as

$$\begin{array}{lll} eval_{\mathcal{T}}(\pi) = & v & \text{if } P(\pi) \Downarrow_{\mathcal{T}} v \\ eval_{\mathcal{T}}(\pi) = & e' & \text{if } P(\pi) \Downarrow_{\mathcal{T}} e', \text{ with } unforced(e') \\ eval_{\mathcal{T}}(\pi) = & \text{WRONG} & \text{if } P(\pi) \Downarrow_{\mathcal{T}} e, \text{ with } e \text{ stuck.} \end{array}$$

Finally, we state soundness:

Theorem 4.2 (Soundness). *If e is a program with $\cdot, \cdot \vdash e : \tau$, then for any computation $\mathcal{T} \in Comp(E)$ and any process ID $\pi \in Procs(\mathcal{T})$:*

(a) *If $eval_T(\pi) = v$ and K_i, T_i, P_i is the first occurrence of π in $\mathcal{T}$, then for any Δ, ρ such that $\Delta \vdash K_i, T_i, P_i : \rho$ and $\rho(\pi_0) = \tau$, then there exists $\Delta', \Delta \subseteq \Delta'$ such that $\cdot, \Delta' \vdash v : \rho(\pi)$*

(b) *If $eval_T(\pi) = e'$ with unforced(e') and K_i, T_i, P_i is the first occurrence of π in $\mathcal{T}$, then for any Δ, ρ such that $\Delta \vdash K_i, T_i, P_i : \rho$ and $\rho(\pi_0) = \tau$, then there exists $\Delta', \Delta \subseteq \Delta'$ such that $\cdot, \Delta' \vdash e' : \rho(\pi)$*

(c) $eval_{\mathcal{T}}(\pi) \neq \text{WRONG}$

SKETCH. The proof follows directly from Theorem 4.1 and the definition of *eval*. $\square$

Summary. We have now introduced λ^{π}_{cs} and proven type safety. In the next chapter, we turn our attention to encoding contract verification strategies as patterns of communication in a unified framework build on top of this λ^{π}_{cs} calculus.

CHAPTER 5

Contracts as Patterns of Communication

SYNOPSIS

With our λ^{π}_{cs} calculus in place, we may use its concurrency facilities to define contract verification as patterns of communication. In this chapter, we define a series of contract combinators (§5.1); encode the **eager**, **semi**, **promise**, **concurrent**, and **fconc** strategies in λ^{π}_{cs} (§5.2-5.6); discuss additional verification strategies in this context (§5.7); and use these definitions to explore advanced contract verification techniques (§5.8).

In the last chapter, we introduced λ^{π}_{cs}, a calculus with concurrency and communication facilities based on Concurrent ML [55, 74, 75]. Using this language, we now define a contract verification framework to provide multiple monitoring strategies in a single, unified system, expressing each in terms of the λ^{π}_{cs} primitives. Before we establish these definitions, however, we pause to define contract combinators in λ^{π}_{cs} in order to define contracts that characterize these distinct verification behaviors.

5.1. Contract Combinators in λ^{π}_{cs}

We begin with defining the pred/c, pair/c, and fun/c contract combinators. For our presentation in λ^{π}_{cs}, we diverge from the expression-form combinator structures we used in Chapter 2: our aim is to make a general, *flexible* verification framework, which is at odds with fixing the shape and structure of contract combinators as part of the core syntax. To avoid this problem, we now introduce an alternative, spartan definition of software contracts:

DEFINITION 5.1 (Software Contract in λ^{π}_{cs}). *A contract c is a procedure with type signature $\tau \rightarrow$ `blame` $\rightarrow \tau$ that takes a value v (at type τ) and a set of blame information B, that, when applied, either yields a new expression v' (also at type τ) or raises an error.*

This definition is intentionally permissive: in order to ensure that each verification strategy is contract-agnostic, we treat contracts as "black box" verifiers that we apply directly to the monitored term.

Using this definition of software contracts, we can define our combinators as higher-order functions that produce "contract-shaped" procedures. To start, we define pred/c, the predicate contract combinator, as:

$$\mathsf{pred/c} \triangleq \lambda\, pred.\ \lambda\, val\ blame.\ \mathsf{if}\ pred\ val\ \mathsf{then}\ val\ \mathsf{else}\ \mathsf{raise}\ blame \tag{32}$$

The pred/c contract combinator take a predicate and produces a contract-shaped procedure that verifies that the predicate holds for its input. As in Chapter 2, the contract either yields the original value (if the predicate holds) or raises the appropriate error (if it does not).

Our pair/c combinator in λ^{π}_{cs} follows this same definition structure, including additional parameters to allow contract definitions to specify which strategy to use for each of their subcontracts:

$$\begin{aligned}\mathsf{pair/c} \triangleq\ & \lambda\, con_1\ strat_1\ con_2\ strat_2.\\ & \quad \lambda\, pair\ blame.(\mathbf{mon}\ con_1\ strat_1\ (\mathsf{fst}\ pair)\ blame),\\ & \qquad\qquad\qquad \mathbf{mon}\ con_2\ strat_2\ (\mathsf{snd}\ pair)\ blame)\end{aligned} \tag{33}$$

When a monitor verifies a pair/c contract, the pair-contract procedure asserts con_1 on the first element of the pair using $strat_1$ and con_2 on the second element of the pair using $strat_2$. As we will see in the next sections, these additional arguments provide programmers with precise control over the verification strategy for each subcontract.

Our fun/c combinator mirrors our pair/c definition, accepting pre- and post-condition subcontracts and their associated strategies:

$$\begin{aligned}\mathsf{fun/c} \triangleq\ & \lambda\, con_1\ strat_1\ con_2\ strat_2.\\ & \quad \lambda\, func\ blame.\\ & \qquad \lambda\, x.\ \mathbf{mon}\ con_2\ strat_2\ (func\ (\mathbf{mon}\ con_1\ strat_1\ x\ (\mathit{invert}\ blame)))\ blame\end{aligned} \tag{34}$$

This definition adopts the same function verification tactic as in Chapter 2: enforcing the resultant contract on a procedure produces a new, contracted variant of the procedure to "stand in" for the original. When applied, this contracted variant checks the first contract, or *pre-condition*, on each input, and checks the second contract, or *post-condition*, on each of *func*'s results [36, 79]. In this revised definition, however, we verify the pre-condition con_1 and post-condition con_2 using their associated strategies $strat_1$ and $strat_2$ (respectively). The *invert* operator performs blame inversion in the usual way [27, 36], ensuring the procedure's context will be blamed if the function input violates its contract.

With these combinators in place, we turn our attention to the contract monitoring strategies described in Chapter 2, providing semantic definitions for each in terms of λ_{cs}^{π}.

5.2. Eager Contract Verification—*Interrupting the User Evaluator*

We begin with eager contract verification, where each contract is completely verified at assertion time. Following Chapter 3, we model this verification as two, interacting evaluators: the standard, "user" evaluator and a secondary, "monitoring" evaluator which performs the contract verification and reports the result.

To model eager verification in this multi-process context as a *pattern of communication*, we utilize the communication structure outlined in Chapter 3, wherein the initiating process:

(E_{U1}) creates a new communication channel ι (via chan);
(E_{U2}) spawns a monitoring process that will receive the monitored term, evaluate the contract, and communicate the result via ι;
(E_{U3}) provides the (evaluated) subject value to the monitoring process across ι;
(E_{U4}) and retrieves the result from the monitoring process across ι and handles them (as explained below).

Dually, the monitoring process:

(E_{M1}) receives the subject value v across ι;
(E_{M2}) runs contract c on the value with the provided blame information;
(E_{M3}) examines the verification result, injecting values to the right and, similarly, injecting contract errors to the left (via catch);
(E_{M4}) and writes the injected value across ι to the user process.

We present this interaction in Figure 5.1, with the monitoring process colored blue. These two evaluators synchronize at (E_{U3}, E_{M1}), to communicate the subject value to the monitoring process, and again at (E_{U4}, E_{M4}), to communicate the verification result. Because read is *blocking*, the user process will wait at (E_{U4}) until the monitor completes, replicating the evaluator-interrupting behavior we describe in §2.2. Recall that the **mon** language form accepts, as input, the contract to verify, the strategy describing how verification should proceed, the expression to check it on,

$$\begin{tikzcd}
\mathsf{mon}\ \mathsf{nat/c}\ \mathsf{eager}\ (2\ +\ 3)\ B \arrow[d] \arrow[drr, bend left] & & \\
\mathsf{seq}\ (\mathsf{write}\ \iota\ (2\ +\ 3))\ (\mathit{conres}\ (\mathsf{read}\ \iota)) \arrow[d] & & \\
\mathsf{seq}\ (\mathsf{write}\ \iota\ 5)\ (\mathit{conres}\ (\mathsf{read}\ \iota)) \arrow[rr, leftrightarrow] \arrow[ddd] & & \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B))) \arrow[d] \\
& & \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ 5\ B))) \arrow[d] \\
& & \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ 5)) \arrow[d] \\
\mathit{conres}\ (\mathsf{read}\ \iota) \arrow[rr, leftrightarrow] \arrow[d] & & \mathsf{write}\ \iota\ (\mathsf{inr}\ 5) \\
\mathit{conres}\ (\mathsf{inr}\ 5) \arrow[d] & & \\
5 & &
\end{tikzcd}$$

Figure 5.1. Checking nat/c with **eager**.

and blame information. Note, however, that we do not evaluate the monitored term as part of "**mon**": our evaluation contexts presented in Figure 4.1 only evaluate the contract, strategy, and blame expressions, allowing each strategy to individually control when to evaluate the term. Using our **mon** form, we now encode the **eager** verification strategy in λ^{π}_{cs}, following the interactions described above:

Definition 5.2 (Eager Verification as a Pattern of Communication).

$$\begin{aligned}
\textbf{mon}\ con\ \textbf{eager}\ exp\ B\ \rightarrow\ & \mathit{let}\ i\ =\ \mathit{chan} \\
& \mathit{in}\ \mathit{seq} \\
& \quad (\mathit{spawn}\ (\mathit{write}\ i\ (\mathit{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (con\ (\mathit{read}\ i)\ B))))\ ) \\
& \quad (\mathit{write}\ i\ exp) \\
& \quad (\mathit{conres}\ (\mathit{read}\ i))
\end{aligned}$$

As indicated by (E_{M4}), we use the *conres* helper to interpret the monitor result in the initiating process; we define this helper procedure as follows:

Definition 5.3 (The *conres* helper procedure).

$$\mathit{conres} \triangleq \lambda\ x.\ \mathit{case}\ (x;\ y \rhd \mathit{raise}\ y;\ z \rhd z)$$

If the verifier left-injects the value (indicating a contract violation), we re-raise the error in the process expecting the contract result, and, similarly, if the verifier right-injects it (indicating that **mon** did not raise an error), we return it to the process[1].

5.2.1. *Predicate Monitoring with Eager Verification*

To demonstrate verification structure in action, consider checking the predicate contract nat/c (again defined as "pred/c (λ x. $x \ \geq\ 0$)"):

$$
\begin{array}{ll}
 & \{\langle 1 + \textbf{mon}\ \mathsf{nat/c}\ \mathsf{eager}\ e\ B\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle 1 + (\mathsf{seq}\ (\mathsf{spawn}\ (\mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B))))\ ) \\
 & \qquad\qquad (\mathsf{write}\ \iota\ e) \\
 & \qquad\qquad (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
\Rightarrow^* & \{\langle 1 + (\mathsf{seq}\ (\mathsf{write}\ \iota\ e)\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
 & ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 1 + (\mathsf{seq}\ (\mathsf{write}\ \iota\ v)\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
 & ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 1 + (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
 & ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ v\ B)))\rangle^{\pi_1}\}
\end{array}
$$

In this example, the user process (π_0) creates a verification process (π_1) at line 2, evaluates the monitored term in the user process, sends this value to the monitoring process (line 4), and, finally, performs a blocking read to retrieve the verification result. When v = 5, this verification computation ensures that 5 is a natural number, right-injects the result, and writes it across the communication, ultimately yielding 6 as the user program result:

$$
\begin{array}{lll}
\Rightarrow^* & \{\langle 1 + (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, & \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ 5\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 1 + (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, & \langle \mathsf{write}\ \iota\ (\mathsf{inr}\ 5)\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 1 + (\mathit{conres}\ (\mathsf{inr}\ 5))\rangle^{\pi_0} & ,\langle \mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 6\rangle^{\pi_0} & ,\langle \mathsf{unit}\rangle^{\pi_1}\}
\end{array}
$$

Similarly, when v = -1, this computation detects the contract violation, left-injects the blame information (signaling a contract error), and reports this left-injected blame value to *conres* in the

[1]If, for some reason, a monitor causes a secondary error to occur, such as by violating a different contract as part of its verification, the **mon**/*conres* mechanism also ensures this error is properly propagated to the initiating evaluator. Our pair contract example later in this section utilizes this behavior to propagate a subcontract violation.

$$
\begin{aligned}
& \{\langle \mathsf{fst}\ (\mathbf{mon}\ \mathbf{nat\text{-}pair/c}^{eager}\ \mathbf{eager}\ (5, \text{-}1)\ B)\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
& ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathbf{nat\text{-}pair/c}^{eager}\ (5, \text{-}1)\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
& ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ (\mathsf{fst}\ (5, \text{-}1))\ B\ \ ))\rangle^{\pi_1} \\
& \qquad\qquad\qquad\qquad\qquad\qquad ,\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ (\mathsf{snd}\ (5, \text{-}1))\ B) \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
& ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (5, \mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ (\mathsf{snd}\ (5, \text{-}1))\ B)))\rangle^{\pi_1} \\
& ,\langle \mathsf{unit}\rangle^{\pi_2}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
& ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (5, \mathit{conres}\ (\mathsf{read}\ \iota_2))))\rangle^{\pi_1} \\
& ,\langle \mathsf{unit}\rangle^{\pi_2}\ ,\ \langle \mathsf{write}\ \iota_2\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ \text{-}1\ B)))\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
& ,\langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (5, \mathit{conres}\ (\mathsf{inl}\ B))))\rangle^{\pi_1} \\
& ,\langle \mathsf{unit}\rangle^{\pi_2}\ ,\ \langle \mathsf{unit}\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (5, \mathsf{raise}\ B)))\rangle^{\pi_1}, \langle \mathsf{unit}\rangle^{\pi_2},\ \langle \mathsf{unit}\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{inl}\ B)\rangle^{\pi_1} \qquad\qquad\qquad\qquad\qquad , \langle \mathsf{unit}\rangle^{\pi_2},\ \langle \mathsf{unit}\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle \mathsf{fst}\ (\mathit{conres}\ (\mathsf{inl}\ B))\rangle^{\pi_0}\ , \langle \mathsf{unit}\rangle^{\pi_1} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , \langle \mathsf{unit}\rangle^{\pi_2},\ \langle \mathsf{unit}\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle \mathsf{raise}\ B\rangle^{\pi_0} \qquad\qquad\qquad\quad , \langle \mathsf{unit}\rangle^{\pi_1} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , \langle \mathsf{unit}\rangle^{\pi_2},\ \langle \mathsf{unit}\rangle^{\pi_3}\}
\end{aligned}
$$

Figure 5.2. Enforcing $\mathbf{nat\text{-}pair/c}^{eager}$ on the pair $(5, \text{-}1)$.

initiating process, raising an error in π_0:

$$
\begin{aligned}
\Rightarrow^* \quad & \{\langle 1 + (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ \text{-}1\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* \quad & \{\langle 1 + (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{raise}\ B))\rangle^{\pi_1}\} \\
\Rightarrow^* \quad & \{\langle 1 + (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{inl}\ B)\rangle^{\pi_1} \\
\Rightarrow^* \quad & \{\langle 1 + (\mathit{conres}\ (\mathsf{inl}\ B))\rangle^{\pi_0}\ , \langle \mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^* \quad & \{\langle \mathsf{raise}\ B\rangle^{\pi_0} \qquad\qquad\qquad , \langle \mathsf{unit}\rangle^{\pi_1}\}
\end{aligned}
$$

In each case, the user process performs steps (E_{U1})–(E_{U4}), suspending its computation and awaiting the verification result, while the monitoring process performs (E_{M1})–(E_{M4}), communicating the contract result to the user process. Taken together as a pattern of communication, this interaction models our eager contract verification behavior.

5.2.2. *Pair and Function Contracts with Eager Verification*

Recall that eager monitors may "over-evaluate" their input, detecting and signaling contract violations for unused values. In faithfully recreating eager monitoring, we have preserved this property.

For example, we may define and verify an eager variation of nat-pair/c from Chapter 2 using our revised pair/c operation as:

$$\textbf{nat-pair/c}^{eager} \triangleq \textsf{pair/c nat/c eager nat/c eager} \tag{35}$$

Now, we consider evaluating

$$\textsf{fst}\ (\textbf{mon}\ \textbf{nat-pair/c}^{eager}\ \textbf{eager}\ (5, \text{-}1)\ B)$$

We give a trace in Figure 5.2. This interaction results in an error in the user process (π_0) because -1 is not a natural number, and the monitor reports this verification result, ultimately re-raising the error in the user process via *conres*. Notice that there are *three* monitoring evaluators in our trace: the monitoring evaluator checking **nat-pair/c**^{eager} (π_1), the evaluator checking nat/c on the first element of the pair (π_2), and the evaluator checking nat/c on the second element of the pair (π_3). This separation and interaction mirrors our previous description of eager checking: at *each level*, the initiating evaluator writes a value across a channel and awaits the monitoring evaluator's result. Using the raise and catch infrastructure at each monitoring process allows us to propagate blame errors to the initiating process.

Function contracts proceed similarly, but without the nesting monitors. Using our revised fun/c combinator, we may define a function contract as:

$$\textbf{nat-fun/c}^{eager} \triangleq \textsf{fun/c nat/c eager nat/c eager} \tag{36}$$

For function contracts, the user portion of the program performs the function application between checking the pre- and post-condition, explicitly evaluating the application in the *user process* before verifying the post-condition contract. To illustrate this behavior, consider monitoring **nat-fun/c**^{eager} on "$\lambda\ x.\ 1$", which follows the trace given in Figure 5.3. As evaluation proceeds, the term

$$((\lambda\ x.\ 1)\ (\textbf{mon}\ \textsf{nat/c}\ \textbf{eager}\ 5\ (\textit{invert}\ B)))$$

occurs in the user process, triggering the pre-condition check while the post-condition check awaits the function result. After verifying the pre-condition, the *user evaluator* performs the actual function application "$(\lambda\ x.\ 1)\ 5$".

5.2.3. *Embedding Findler and Felleisen [36] into λ^{π}_{cs}*

Our goal in this work is to construct a single, unified framework for contract semantics, a sort of "assembly language" target for recreating, understanding, and comparing these semantics and, more

$$
\begin{array}{l}
\quad \{\langle(\mathbf{mon}\ \mathbf{nat\text{-}fun/c}^{eager}\ \mathbf{eager}\ (\lambda\ x.\ 1)\ B)\ 5\rangle^{\pi_0}\} \\
\Rightarrow^* \{\langle(\lambda\ x.\ \mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ (\lambda\ x.\ 1)\ (\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ x\ (\mathit{invert}\ B))\ B)\ 5\rangle^{\pi_0} \\
\quad ,\langle\mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^* \{\langle\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ (\lambda\ x.\ 1)\ (\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ 5\ (\mathit{invert}\ B))\ B\rangle^{\pi_0}\} \\
\Rightarrow^* \{\langle\mathsf{seq}\ (\mathsf{write}\ \iota\ ((\lambda\ x.\ 1)\ (\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{eager}\ 5\ (\mathit{invert}\ B))))\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
\quad ,\langle\mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi_2}\} \\
\Rightarrow^* \{\langle\mathsf{seq}\ (\mathsf{write}\ \iota\ ((\lambda\ x.\ 1)\ (\mathit{conres}\ (\mathsf{read}\ \iota'))))\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
\quad ,\langle\mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi_2} \\
\quad ,\langle\mathsf{write}\ \iota'\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ 5\ (\mathit{invert}\ B))))\rangle^{\pi_3}\} \\
\Rightarrow^* \{\langle\mathsf{seq}\ (\mathsf{write}\ \iota\ ((\lambda\ x.\ 1)\ 5))\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
\quad ,\langle\mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi_2}\} \\
\Rightarrow^* \{\langle\mathsf{seq}\ (\mathsf{write}\ \iota\ 1)\ (\mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0} \\
\quad ,\langle\mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi_2}\} \\
\Rightarrow^* \{\langle 1\rangle^{\pi_0}\quad ,\quad \langle\mathsf{unit}\rangle^{\pi_1}\quad ,\quad \langle\mathsf{unit}\rangle^{\pi_2}\quad ,\quad \langle\mathsf{unit}\rangle^{\pi_3}\}
\end{array}
$$

Figure 5.3. Enforcing **nat-fun/c**^{eager} on the function $(\lambda\ x.\ x)$ with input 5, eliding process of the form $\langle\mathsf{unit}\rangle^{\pi}$ except in the last step.

generally, runtime verification. To this end, we take a moment to perform a sort of sanity check by proving that eager runtime verification in λ^{π}_{cs} simulates the runtime verification mechanisms of the λ^{CON} calculus presented by Findler and Felleisen [36] (given in Figure A.1), up to alpha-equivalence and unit elimination. (The below is only a sketch of the proof; we give the full proof in Appendix A.)

First, we simplify the language λ^{CON} as follows: we remove list and fixpoint operations (since neither are relevant to the discussion) and, more importantly, their outer val rec form, defined as follows (wherein Findler and Felleisen [36] also define evaluation contexts such that the recursive bindings are evaluated before the form's body):

$$
\begin{array}{rcl}
p & = & d \cdots c \\
d & = & \mathsf{val\ rec}\ x : c = c
\end{array}
$$

Our simulation works via three translation relations from λ^{CON} to λ^{π}_{cs}, defined as "$\rightsquigarrow$", "$\twoheadrightarrow_e$", and "$\twoheadrightarrow_v$" in Appendix A as Figure A.2 and Figure A.3. This translation relies on one additional modification to λ^{CON}: Findler and Felleisen [36] use their language's if operation to perform predicate contract verification, and we must be able to identify when such an expression is a contract verification expression (as opposed to a conditional expression in the user program portion) so that we can extract it into a separate process. In order to distinguish between conditionals that are part of contract verification (e.g., if *contract value* then *value* else *error*) from other if expressions, we "recolor" the if expressions to indicate their origin:

- We color each if expression that originates in the user program with $\circ$, indicating it is part of the user program.
- We modify the reduction relation "$\rightarrow$" in λ^{CON} to produce $\mathsf{if}_\bullet$ forms for contract verification:

$$C[V^{\mathsf{contract}(V_2),p,n}] \longrightarrow C[\mathsf{if}_\bullet\ V_2\ (V)\ \mathsf{then}\ V\ \mathsf{else}\ \mathsf{blame}(p)]$$

Evaluation for both $\mathsf{if}_\circ\ c\ \mathsf{then}\ c\ \mathsf{else}\ c$ and $\mathsf{if}_\bullet\ c\ \mathsf{then}\ c\ \mathsf{else}\ c$ otherwise proceed as $\mathsf{if}\ c\ \mathsf{then}\ c\ \mathsf{else}\ c$ in λ^{CON}, and now our translation can determine which if expressions are part of contract enforcement in order to correctly translate them into λ^{π}_{cs}.

Using this recoloring, we define the "$\rightsquigarrow$" operator to relate a term c with an expression e, a set of new channels K, and a set of processes P, where the translated expression will fill process π_0, using "$\twoheadrightarrow_e$" and "$\twoheadrightarrow_v$" as helper relations to translate terms and values in λ^{CON} into equivalent term-level expressions in λ^{π}_{cs} respectively.

Using this translation, we state the main embedding theorem:

Lemma 5.1 (Embedding Reduction). *If $c \in \lambda^{CON}$ such that $\cdot \vdash c : t$ (that is, c is well-typed), $c \longrightarrow c'$, and $c \rightsquigarrow (K,\ P,\ e)$ and $c' \rightsquigarrow (K',\ P',\ e')$, then $K, \{\pi_0\}, \{\langle e\rangle^{\pi_0}\} + P \Rightarrow^* K'',\ \{\pi_0\},\ P''$ such that $K'',\ \{\pi_0\},\ P'' =_{\alpha,\mathsf{unit}} K', \{\pi_0\}, \{\langle e'\rangle^{\pi_0}\} + P'$.*

Proof. (sketch). The proof proceeds by induction on $\longrightarrow$, such that, if $c \longrightarrow c'$, then its translation e will reduce to e' in zero or more steps using the $\Rightarrow$ reduction. The crux of the translation hinges on differentiating between contract-checking conditional terms and other conditional branching operations, using our recolored if expressions to ensure we correctly translate contract verification forms as contract monitoring structures. □

Next, we state the embedding theorem as:

Theorem 5.1 (Embedding Correctness). *If $c \in \lambda^{CON}$ such that $\cdot \vdash c : t$, $c \longrightarrow^* V$, $c \rightsquigarrow (K,\ P,\ e)$, and $V \twoheadrightarrow_v v$, then $K, \{\pi_0\}, \{\langle e\rangle^{\pi_0}\} + P \longmapsto^* K', \{\pi_0\}, \langle v\rangle^{\pi_0} + P'$.*

Proof. First, no translation will produce a spawn_f form, and thus T remains constant. Then the proof proceeds by induction on the length of $\longrightarrow^*$ and Lemma 5.1. □

This proof demonstrates that our approach to **eager** monitoring faithfully recreates the original presentation and, more generally, that defining contracts as patterns of communication preserves the behavior of previous models while exposing their internal workings at a finer granularity. Additional simulation proofs will be more complex but follow this same approach: syntactic munging plus a little process management. We provide the full proof in Appendix A.

5.2.4. *The Drawbacks of Eager Verification*

As in §2.2, the examples presented in this section illustrate the over-eager nature of eager verification: while the user program did not inspect the second element of the pair "(5,-1)", and the function "$\lambda\ x.\ 1$" did not use its argument, the monitoring process still verified that each was a natural number and signalled an error. Using our revised encoding of eager verification as individual processes, we can now see that the fundamental problem is *preemption*: eager evaluation explicitly suspends the initiating evaluator while verifying the contract, only resuming the initiating evaluator once verification is complete, tying the overall computation performance to the contract system's performance. If the user evaluator *did not* wait for the monitor result, however, the overall performance would be decoupled from the contract system, helping to alleviate this situation. In our λ^{π}_{cs} framework, we can provide this alternative variation on verification by merely varying the *pattern of communication* between these evaluators.

5.3. Semi-Eager Contract Verification—*Postponing Contract Verification*

Our next monitoring strategy is semi-eager contract verification, indicated with the **semi** strategy parameter. Recall that, in semi-eager verification, the monitor must suspend verification until the user evaluator demands the result. In Chapter 2, we described this mechanism as "boxing up" the contract and value; to replicate this behavior in λ^{π}_{cs}, we use the delay and force operators to suspend and later resume verification.

Using this model, semi-eager verification performs a single action at assertion time:

(S_{U1}) creates a delayed expression d that, when forced, will perform verification, and returns it to the user.

When a process later forces this delayed expression, the forcing process (which is *not necessarily* the initiating process) proceeds with "eager-style" verification, wherein the forcing process:

(S_{U2}) creates a new communication channel ι;

(S_{U3}) spawns a monitoring process to evaluate the contract and communicate the result via ι;

(S_{U4}) provides the (evaluated) subject value to the monitoring process across ι;

(S_{U5}) and retrieves the result across ι and handles them (via *conres*).

As with eager verification, the monitoring process:

(S_{M1}) receives the subject value v across ι;

(S_{M2}) runs contract c on the value with the provided blame information;

(S_{M3}) injects the result appropriately;

(S_{M4}) and writes the injected value across ι to the user process.

This pattern of interaction is almost identical to **eager** verification, synchronizing at process states (S_{U4}, S_{M1}) and (S_{U5}, S_{M4}): the only difference is that the verifier captures this entire computation in a delayed expression (at S_{U1}), giving the initiating evaluator freedom to either invoke or ignore the verification as necessary. We present this interaction in Figure 5.4, and we extend our definition of **mon** with this encoding:

Definition 5.4 (Semi-Eager Verification as a Pattern of Communication).

```
mon con semi exp B  →  delay
                        (let i = chan
                         in seq
                            (spawn (write i (catch inl (inr (con (read i) B)))) )
                            (write i exp)
                            (conres (read i)))
```

This implementation directly corresponds to the **eager** implementation in Definition 5.2, delaying the entire verification expression (via a delay highlighted in yellow to show its addition) in order to “package up” the verification computation until an evaluator forces it.

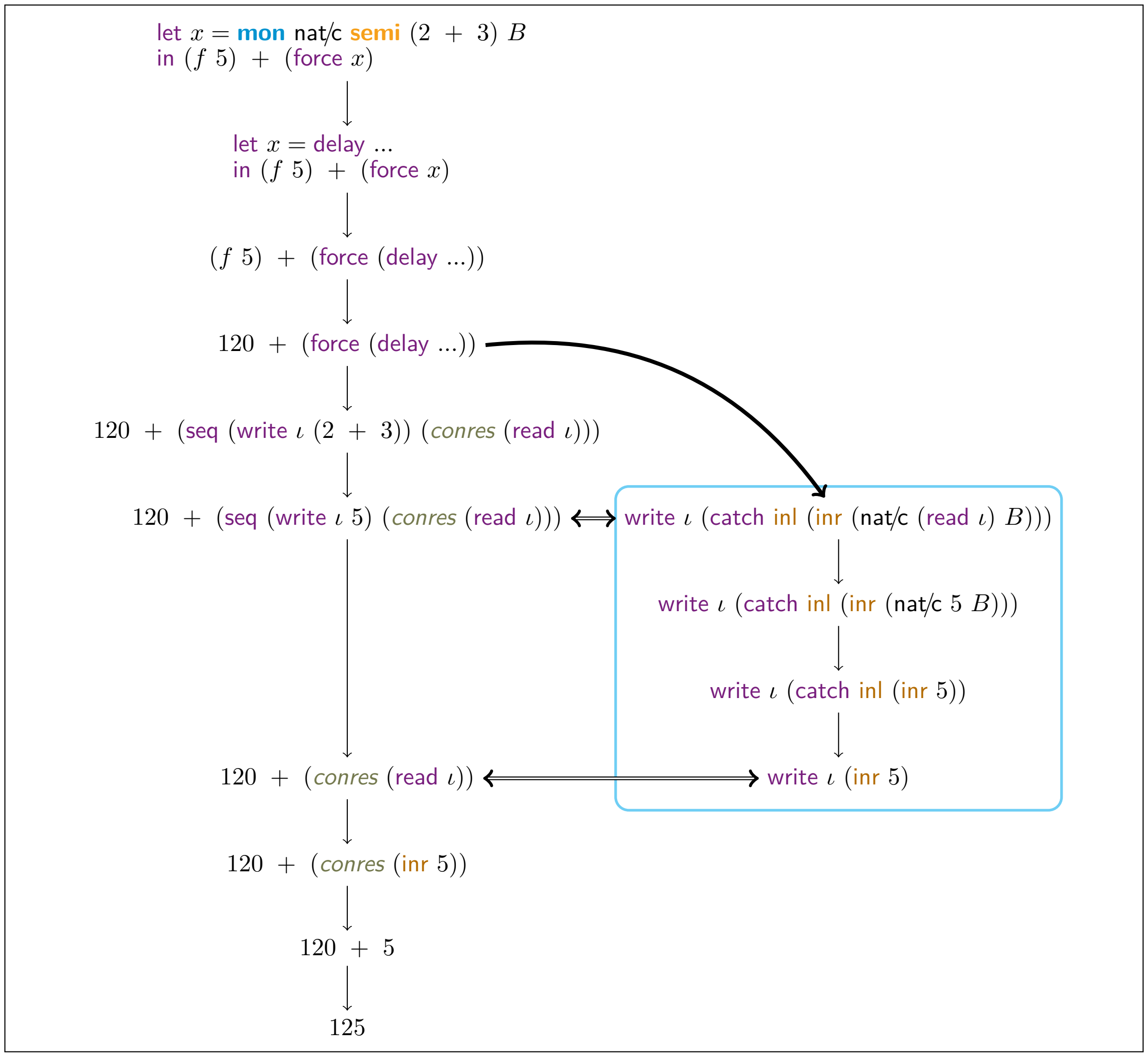

let x = mon nat/c semi (2 + 3) B
in (f 5) + (force x)
let x = delay ...
in (f 5) + (force x)
(f 5) + (force (delay ...))
120 + (force (delay ...))
120 + (seq (write ι (2 + 3)) (conres (read ι)))
120 + (seq (write ι 5) (conres (read ι)))
write ι (catch inl (inr (nat/c (read ι) B)))
write ι (catch inl (inr (nat/c 5 B)))
write ι (catch inl (inr 5))
120 + (conres (read ι))
write ι (inr 5)
120 + (conres (inr 5))
120 + 5
125

Figure 5.4. Checking nat/c with **semi**. The monitoring process is not created until the initiating evaluator forces the **mon** result. We indicate the monitoring process with $\boxed{\text{blue}}$.

5.3.1. *Predicate Monitoring with Semi-Eager Verification*

We may now use the **semi** strategy keyword to perform semi-eager predicate contract verification, such as verifying that 5 is a natural number:

$$
\begin{aligned}
& \{\langle \mathsf{force}\ (\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{semi}\ 5\ B)\rangle^{\pi_0}\} \\
\Rightarrow^* \quad & \{\langle \mathsf{force}\ (\mathsf{delay}\ (\mathsf{let}\ i = \mathsf{chan}\ \mathsf{in}\ (\mathsf{seq}\ (\mathsf{spawn}\ \ldots\ )\ (\mathsf{write}\ i\ 5)\ (\mathit{conres}\ (\mathsf{read}\ i)))))\rangle^{\pi_0}\} \\
\Rightarrow^* \quad & \{\langle \mathsf{let}\ i = \mathsf{chan}\ \mathsf{in}\ (\mathsf{seq}\ (\mathsf{spawn}\ \ldots\ )\ (\mathsf{write}\ i\ 5)\ (\mathit{conres}\ (\mathsf{read}\ i)))\rangle^{\pi_0}\} \\
\Rightarrow^* \quad & \{\langle \mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ 5\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* \quad & \{\langle 5\rangle^{\pi_0}, \langle \mathsf{unit}\rangle^{\pi_1}\}
\end{aligned}
$$

Observe the force wrapping the verification expression: if we had not included it, the program would yield the delay verification expression as the final result:

$$
\begin{array}{rl}
& \{\langle \textbf{mon}\ \mathsf{nat/c}\ \textbf{semi}\ 5\ B\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle \mathsf{delay}\ (\mathsf{let}\ i = \mathsf{chan}\ \mathsf{in}\ (\mathsf{seq}\ (\mathsf{spawn}\ \ldots\ )\ (\mathsf{write}\ i\ 5)\ (\mathit{conres}\ (\mathsf{read}\ i))))\rangle^{\pi_0}\}
\end{array}
$$

The program Figure 5.4 proceeds similarly: the verification creates a delayed cell, and the property goes unchecked until the cell is forced[2].

5.3.2. *Pair and Function Contracts with Semi-Eager Verification*

For structural contracts, this system of delaying and forcing contracts gives programmers immense control over *which parts* of the structure to verify. For example, a semi-eager pair contract that ensures each element is a natural number may eschew checking unused parts of the pair:[3]

$$
\begin{array}{rl}
& \{\langle \mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ (\textbf{mon}\ \textbf{nat-pair/c}^{semi}\ \textbf{semi}\ (5,\text{-}1)\ B)))\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle \mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ (\mathsf{delay}\ \ldots\text{“check } \textbf{nat-pair/c}^{semi} \text{ on } (5,\text{-}1)\text{”}\ldots\ ))))\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle \mathsf{force}\ (\mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ \iota)))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{inr}\ (\mathsf{delay}\ \ldots, \mathsf{delay}\ \ldots))\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle \mathsf{force}\ (\mathsf{fst}\ (\mathsf{delay}\ \ldots\text{“check } \mathsf{nat/c} \text{ on } 5\text{”}\ldots\ ,\ \mathsf{delay}\ \ldots\text{“check } \mathsf{nat/c} \text{ on -1”}\ldots))\rangle^{\pi_0}, \ldots\} \\
\Rightarrow^* & \{\langle \mathsf{force}\ (\mathsf{delay}\ \ldots\text{check } \mathsf{nat/c} \text{ on } 5\ldots)\rangle^{\pi_0}, \langle \mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 5\rangle^{\pi_0}, \langle \mathsf{unit}\rangle^{\pi_1}, \langle \mathsf{unit}\rangle^{\pi_2}\}
\end{array}
$$

In this example, we assert **nat-pair/c**^{semi} on a pair, yielding a delayed reference that, when forced, verifies **nat-pair/c**^{semi} on $(5, \text{-}1)$. This results in a *pair* of delayed cells that, when forced, will each verify the appropriate subcontract on the appropriate subcomponent of the pair, i.e., the first will monitor nat/c on 5 and the second will monitor nat/c on -1. The last part of this trace retrieves the first element and forces it, verifying 5 is a natural number, and returns the monitor result. Since we never force the second element, we never attempt to verify that -1 is a natural number, and thus the program terminates without signaling a contract violation.

In general, the program only needs to force values it intends to use, and, as a result, the program verifies only those values required for its final result.

[2]This potential for unforced delay cells is why we check delay e at the type of e: otherwise, not only would each delay flow through the program's type system, but the overall type of **mon** would *depend on the strategy.* This strategy-dependent type approach gets particularly problematic in the context of higher-order function contracts, where we must determine the input function's type based on the strategies used in the function contract's subcomponents.
[3]The **nat-pair/c**^{semi} contract is similar to **nat-pair/c**^{eager}, replacing **eager** with **semi**.

5.3.3. *On the Use of Delay and Force*

In **semi** monitoring (and **promise**, below), we use delay to encode unevaluated expressions as values that the user must explicitly force. We take this as a necessary evil to facilitate our discussion: some verification strategies require fine-grained delaying and forcing behavior in call-by-value calculi to correctly recreate less-eager evaluation mechanisms. We elect to do this with explicit delay and force operations in our presentation in order to clarify the nature of the evaluator interactions for these strategies, requiring the user programs to propagate force throughout their expressions in order to better clarify the nature of these evaluator interactions. We tolerate this intrusion to better examine and explain the nature of these synchronizing monitors so that the working semanticist may directly compare how and when the user evaluator interacts with them.

In a programming language intended for everyday use, however, this forcing mechanism will clutter the program and inconvenience the programmer. As such, we suggest that, when adding such a feature to a programming language implementation, the implementer should conceal the delayed expressions from the user, opting for implicit forcing at evaluation sites (using some transparent structure such as *chaperones* [79] for managing delayed structures) to remove this syntactic complexity. We include such a variant of λ^{π}_{cs}, λ^{π}_{cd}, that reflects this style of implicit forcing in Appendix B.

5.3.4. *The Drawbacks of Semi-Eager Verification*

As with its presentation in §2.3, this semi-eager enforcement style may still couple the user and monitor evaluators, suspending the user program during each verification step: when forcing the first subcomponent of the pair in our previous example, the user program must wait for the verification result before proceeding. Our next three strategies leverage the concurrent facilities in λ^{π}_{cs} to address this problem.

5.4. Promise-based Contract Verification—*Concurrent Checking with Synchronization*

Our next contract verification strategy is promise-based verification, indicated with the **promise** parameter. In this verification approach, monitor expressions return "promises" to the initiating evaluator while verification proceeds concurrently. As with the box-driven description in Chapter 2, we utilize delay (and read's blocking nature) to provide promise-like behavior in verification results: when the initiating process forces the promise, it reads the contract result from appropriate channel, blocking until the concurrent verification is complete (in the case that the verification has previously finished, the initiating process receives the result immediately).

Our implementation of promises follows a pattern of communication similar to our previous verification strategies, with a critical difference: after creating the verification process, the initiating process constructs a promise that will retrieve the verification result when forced. To accomplish this, the initiating process:

(P_{U1}) creates a new communication channel ι;
(P_{U2}) spawns a monitoring process that will evaluate the contract;
(P_{U3}) provides the (evaluated) subject value to the monitoring process across ι;
(P_{U4}) and, finally, returns delay (*conres* (read ι)) as our "promise."

Dually, the monitoring process:

(P_{M1}) receives the subject value v across ι;
(P_{M2}) runs contract c on the value;
(P_{M3}) injects the result appropriately;
(P_{M4}) and writes the injected value across ι to the user process.

This interaction contains two evaluator synchronization points: first at (P_{U3}, P_{M1}), and, later, at (P_{M4}), when a process (not necessarily the initiating one) forces the delayed expression created in (P_{U4}). The forced expression performs a blocking read across ι, receiving the contract result via *conres*. We present this interaction in Figure 5.5, and we extend **mon** to support **promise** as:

Definition 5.5 (Promise-Based Verification as a Pattern of Communication).

```
mon con promise exp B  →  let i = chan
                          in seq
                             (spawn (write i (catch inl (inr (con (read i) B)))) )
                             (write i exp)
                             ( delay (conres (read i))))
```

As with **semi**, this implementation directly corresponds to the **eager** implementation in Definition 5.2, aside from the addition of delay to delay reading the result. This should be unsurprising: the variation between eager and promise-based contract monitoring is precisely *when* the initiating evaluator receives the answer, allowing programmers to perform secondary computations while verification continues concurrently.

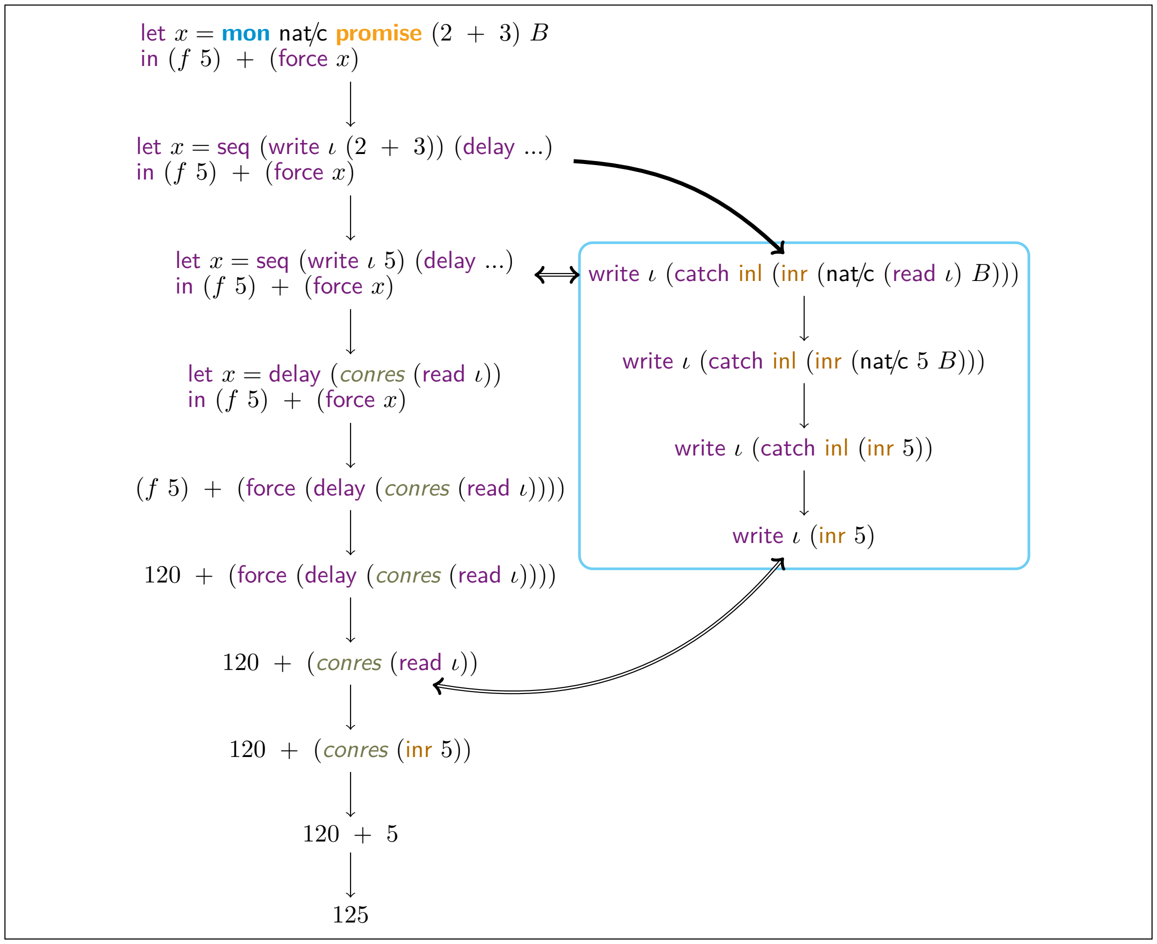

let x = mon nat/c promise $(2 + 3)$ B
in $(f\ 5) + $ (force x)
let x = seq (write ι $(2 + 3)$) (delay ...)
in $(f\ 5) + $ (force x)
let x = seq (write ι 5) (delay ...)
in $(f\ 5) + $ (force x)
write ι (catch inl (inr (nat/c (read ι) B)))
write ι (catch inl (inr (nat/c 5 B)))
write ι (catch inl (inr 5))
write ι (inr 5)
let x = delay (*conres* (read ι))
in $(f\ 5) + $ (force x)
$(f\ 5) + $ (force (delay (*conres* (read ι))))
$120 + $ (force (delay (*conres* (read ι))))
$120 + $ (*conres* (read ι))
$120 + $ (*conres* (inr 5))
$120 + 5$
125

Figure 5.5. Checking nat/c with **promise**. The monitoring process does not communicate its final result until the initiating evaluator forces the **mon** result. The expression $(f\ 5)$ is a stand-in for additional computation in the initiation process before synchronization. We indicate the monitoring process with blue.

5.4.1. *Verifying Contracts with Promise-Based Verification*

Verifying a predicate contract with a promise-based structural contract creates a separate process to perform contract verification, yielding a promise in the initiating process that retrieves the verification result when forced.

$$\{\langle \mathsf{force}\ (\mathbf{mon}\ \mathsf{nat/c}\ \mathbf{promise}\ 5\ B)\rangle^{\pi_0}\}$$

$$\Rightarrow^* \quad \{\langle \mathsf{force}\ (\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i)))\rangle^{\pi_0}, \langle \mathsf{write}\ i\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathit{con}\ 5\ B)))\rangle^{\pi_1}\}$$

$$\Rightarrow^* \quad \{\langle \mathit{conres}\ (\mathsf{read}\ \iota))\rangle^{\pi_0}, \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ 5\ B)))\rangle^{\pi_1}\}$$

$$\Rightarrow^* \quad \{\langle 5\rangle^{\pi_0}, \langle \mathsf{unit}\rangle^{\pi_1}\}$$

If we had not included force, the program would have yielded a delay as the final result. Unlike our previous **semi** strategy, however, promise-based monitoring introduces an opportunity for parallel performance: the **promise** strategy allows contract verification to happen *while* the initiating process continues. For example, we can begin verifying a contract while performing additional computations, such as in Figure 5.5, and the following trace:

$$\begin{aligned}\text{let } x &= \textbf{mon}\ \mathsf{nat/c}\ \textbf{promise}\ 5\ B\\ y &= \mathit{factorial}\ 50\\ \text{in }&(\text{force } x) + y\end{aligned}$$

If the **promise**-verified contract proceeds in parallel, and the verification is complete by the time the program reaches the body of the let structure, then the user program experiences minimal contract overhead.

5.4.2. *Pair and Function Contracts with Promise-Based Verification*

Structural contracts follow directly from the interactions we have previously presented, allowing us to begin verification on subcomponents before retrieving them. For example, $\textbf{nat-pair/c}^{prom}$ will create three concurrent processes that work together to ensure that a pair contains two natural numbers, returning each verification results as the user requests it.

We provide such a trace in Figure 5.6, wherein each subcomponent contract creates a new promise-evaluation process as subcontract enforcement proceeds, resulting in three contract evaluators: the evaluator that performs the outer pair/c contract and one for each of the two nat/c subcontracts. Unlike the evaluator interaction pattern in **eager** verification, however, each of these evaluators proceeds independently: the pair/c process returns a *pair* of new "promises" to the initiating evaluator *while* the two nat/c subcontracts proceed with verification. Moreover, even though we create these promises in a cascading manner, the synchronizing evaluator (here π_0) retrieves each result directly from the appropriate evaluator instead of receiving the complete result from the initial verification evaluator. Additionally, since the second element of the pair is never retrieved, the error is never reported to π_0, and thus the program does not raise a top-level error.

Function contracts proceed similarly, evaluation proceeds in the function's body while pre-condition verification proceeds concurrently, allowing us to perform additional operations as part of the function's body before retrieving and using the argument.

$$
\begin{array}{ll}
 & \{\langle(\lambda\ x.\ \mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ x)))\ (\mathbf{mon}\ \mathbf{nat\text{-}pair/c}^{prom}\ \mathbf{promise}\ (5,\text{-}1)\ B)\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle(\lambda\ x.\ \mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ x)))\ (\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i)))\rangle^{\pi_0} \\
 & ,\langle\mathsf{write}\ i\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathbf{nat\text{-}pair/c}^{prom}\ (5,\text{-}1)\ B)))\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle(\lambda\ x.\ \mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ x)))\ (\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i)))\rangle^{\pi_0} \\
 & ,\langle\mathsf{write}\ i\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ ((\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_1))),(\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_2))))))\rangle^{\pi_1} \\
 & ,\langle\mathsf{write}\ i_1\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ 5\ B)))\rangle^{\pi_2} \\
 & ,\langle\mathsf{write}\ i_2\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ \text{-}1\ B)))\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle\mathsf{force}\ (\mathsf{fst}\ (\mathit{conres}\ (\mathsf{read}\ i)))\rangle^{\pi_0} \\
 & ,\langle\mathsf{write}\ i\ (\mathsf{inr}\ ((\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_1))),(\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_2)))))\rangle^{\pi_1} \\
 & ,\langle\mathsf{write}\ i_1\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ 5))\rangle^{\pi_2} \\
 & ,\langle\mathsf{write}\ i_2\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{nat/c}\ \text{-}1\ B)))\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle\mathsf{force}\ (\mathsf{fst}\ ((\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_1))),(\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_2)))))\rangle^{\pi_0} \\
 & ,\langle\mathsf{unit}\rangle^{\pi_1} \\
 & ,\langle\mathsf{write}\ i_1\ 5\rangle^{\pi_2} \\
 & ,\langle\mathsf{write}\ i_2\ (\mathsf{catch}\ \mathsf{inl}\ \mathsf{raise}\ B)\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle\mathsf{force}\ (\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ i_1)))\rangle^{\pi_0} \\
 & ,\langle\mathsf{unit}\rangle^{\pi_1} \\
 & ,\langle\mathsf{write}\ i_1\ 5\rangle^{\pi_2} \\
 & ,\langle\mathsf{write}\ i_2\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{raise}\ B))\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle\mathit{conres}\ (\mathsf{read}\ i_1)\rangle^{\pi_0},\langle\mathsf{unit}\rangle^{\pi_1},\langle\mathsf{write}\ i_1\ 5\rangle^{\pi_2},\langle\mathsf{write}\ i_2\ (\mathsf{inl}\ B)\rangle^{\pi_3}\} \\
\Rightarrow^* & \{\langle 5\rangle^{\pi_0},\langle\mathsf{unit}\rangle^{\pi_1},\langle\mathsf{write}\ i_1\ 5\rangle^{\pi_2},\langle\mathsf{write}\ i_2\ (\mathsf{inl}\ B)\rangle^{\pi_3}\}
\end{array}
$$

Figure 5.6. Enforcing **nat-pair/c**^{prom} on the pair (5, -1) using **promise**.

5.4.3. *Semi-Eager and Promise-Based Verification*

Consider the connection between semi-eager and promise-based monitoring: in each case, the initiating process returns the delayed expression that, when forced, yields the verification result. As a result, these verification strategies have observationally equivalent behavior in the initiating process in terms of when errors occur:

$$
\begin{array}{ll}
 & \{\langle\mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ (\mathbf{mon}\ \mathbf{nat\text{-}pair/c}^{semi}\ \mathbf{semi}\ (5,\text{-}1)\ B)))\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle\mathsf{force}\ (\mathsf{delay}\ (\mathsf{chan}(\lambda\ i.\ \mathsf{seq}\ \ldots\ )))\rangle^{\pi_0},\langle\mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 5\rangle^{\pi_0},\langle\mathsf{unit}\rangle^{\pi_1},\langle\mathsf{unit}\rangle^{\pi_2}\}
\end{array}
$$

$$
\begin{array}{ll}
 & \{\langle\mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ (\mathbf{mon}\ \mathbf{nat\text{-}pair/c}^{prom}\ \mathbf{promise}\ (5,\text{-}1)\ B)))\rangle^{\pi_0}\} \\
\Rightarrow^* & \{\langle\mathsf{force}\ (\mathsf{delay}\ (\mathit{conres}\ (\mathsf{read}\ \iota)))\rangle^{\pi_0},\langle\mathsf{unit}\rangle^{\pi_1}\},\langle(\mathsf{write}\ \iota\ \mathsf{inl}\ 5)\rangle^{\pi_1}\} \\
\Rightarrow^* & \{\langle 5\rangle^{\pi_0},\langle\mathsf{unit}\rangle^{\pi_1},\langle\mathsf{unit}\rangle^{\pi_2}\}
\end{array}
$$

In both **semi** and **promise** verification, the initiating process has precise control over how to retrieve the contract result, using the same mechanism in both places; this equivalence, however, assumes that the monitored term and contract are both pure; if either is not, **semi** and **promise** would no longer be interchangeable.

More formally:

THEOREM 5.2 (Semi-Eager and Promise-Based Local Observational Equivalence). *For some context E, contract e_1, monitored term e_2, and blame label set B such that e_1 and e_2 are both effect-free (i.e., they do not create or communicate with additional processes, including contract verification) and $E[\textbf{mon}\ e_1\ \textit{\textbf{semi}}\ e_2\ B]$ and $E[\textbf{mon}\ e_1\ \textit{\textbf{promise}}\ e_2\ B]$ are well-formed (i.e., well-typed) expressions, either:*

- *there is some K_1, P_1 and K_2, P_2 such that $K, T, P + \langle E[\textbf{mon}\ e_1\ \textit{\textbf{semi}}\ e_2\ B]\rangle^\pi \Rightarrow^* K_1, T, P_1 + \langle v\rangle^\pi$ and $K, T, P + \langle E[\textbf{mon}\ e_1\ \textit{\textbf{promise}}\ e_2\ B]\rangle^\pi \Rightarrow^* K_2, T, P_2 + \langle v\rangle^\pi$;*
- *there is some K_1, P_1 and K_2, P_2 such that $K, T, P + \langle E[\textbf{mon}\ e_1\ \textit{\textbf{semi}}\ e_2\ B]\rangle^\pi \Rightarrow^* K_1, T, P_1 + \langle \textit{raise}\ B\rangle^\pi$ and $K, T, P + \langle E[\textbf{mon}\ e_1\ \textit{\textbf{promise}}\ e_2\ B]\rangle^\pi \Rightarrow^* K_2, T, P_2 + \langle \textit{raise}\ B\rangle^\pi$;*
- *or both diverge.*

PROOF (SKETCH). First, we observe that each of **semi** and **promise** will create a delayed expression in process π. If E does not force this delay, the program will complete identically in either case.

If, however, E forces this delay, we must inspect the outcome of the contract enforcement. First, note that, since e_1 and e_2 are effect-free, well-formed expressions, there are some v_1 and v_2 such that $e_1 \longmapsto^* v_1$ and $e_2 \longmapsto^* v_2$ (following our type soundness theorem presented in Chapter 4). In either case, the monitoring evaluator will then evaluate $v_1\ v_2\ B$, which has three possible outcomes:

- $v_1\ v_2\ B \rightarrow^* v$, some value;
- $v_1\ v_2\ B \rightarrow^* \mathsf{raise}\ B$, an error;
- or $v_1\ v_2\ B$ diverges.

In the first two cases, process π receives the resultant value as (or $\mathsf{inl}\ B$) under either strategy, yielding $K + K', T, P + P' + \langle E[\mathsf{conres}\ (\mathsf{inr}\ v)]\rangle^\pi$ (or $E[\mathsf{conres}\ (\mathsf{inl}\ B)]$), where K' and P' will be identical in the case of flat contracts, but may differ if v_1 is a structural contract that initiates additional checks. Finally, in the third case, if the process diverges, the forcing location in E will block forever, causing the configuration to diverge. □

This suggests that promise-based verification has the most utility when either (a) both the contract and monitored term are pure, or (b) the side effects are independent of the computation's outcome and each other.

5.4.4. *The Drawbacks of Promise-Based Verification*

Since semi-eager and promise-based verification are (typically) observationally equivalent, promise-based verification suffers some of the same issues as discussed in the previous section, including under-evaluation and being continuously driven by the user evaluator. In addition, promise-based verification may cause problems with effectful contracts by performing the effectful portions "out of order," leading to difficult-to-debug problems. Next, we will explore concurrent contracts, which avoid the problems of later synchronization by eschewing the need to retrieve contract results.

5.5. Concurrent Contract Verification—*Complete Evaluator Decoupling*

In our next strategy, **concurrent**, the monitoring evaluator proceeds concurrently without reporting its result, terminating in a value or error. In either case, the initiating evaluator is free to continue without later synchronization. Modeled as patterns of communication, the initiating process:

(C_{U1}) creates a new communication channel ι;
(C_{U2}) spawns a monitoring process that will evaluate the contract;
(C_{U3}) provides the (evaluated) subject value to the monitoring process across ι;
(C_{U4}) and continues with the evaluated subject value.

Dually, the monitoring process:

(C_{M1}) receives the subject value v across ι;
(C_{M2}) and runs the contract c on the value with the provided blame information.

These two evaluators synchronize once, at (C_{U3}, C_{M1}), to communicate the subject value. We present this interaction in Figure 5.7 (with the monitoring process indicated with blue), where the new, monitoring evaluator proceeds without further interaction. We extend **mon** with this encoding of **concurrent** as:

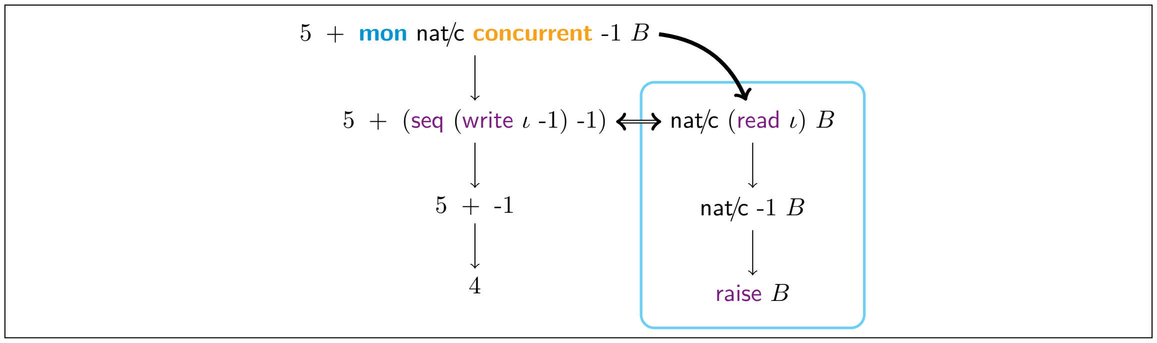


Figure 5.7. Checking nat/c with **concurrent**. The monitoring process continues concurrently while the initiating process computes the result.

Definition 5.6 (Concurrent Verification as a Pattern of Communication).

$$\begin{aligned} \textbf{mon}\ con\ \textbf{concurrent}\ exp\ B \ \rightarrow\ & \mathit{let}\ i\ =\ \mathit{chan} \\ & \mathit{in}\ \mathit{seq}\ (\mathit{spawn}\ (con\ (\mathit{read}\ i)\ B)\ ) \\ & \qquad (\mathit{let}\ x = exp\ \mathit{in}\ \mathit{seq}\ (\mathit{write}\ i\ x)\ x) \end{aligned} \tag{37}$$

This implementation starts a monitoring verifier via spawn, communicates the monitored term to the verifier, and immediately continues with the user program. By using spawn, **concurrent** contracts represent contracts that *may not finish*: using our previous definition of *answer configurations* (Definition 4.1), we see that **concurrent** contracts represent contracts that *may not* finish; if the contract is still running when π_0 terminates, the configuration terminates, discarding the verifier. As a result, this “best-effort” verification method allows us to view contracts as “soft” specifications.

5.5.1. *Predicate Monitoring with Concurrent Verification*

With our definition of the **concurrent** verification strategy in place, we can use it to verify an expression is a natural number:

$$\{\langle 5 + (\textbf{mon}\ \mathsf{nat/c}\ \textbf{concurrent}\ 3\ B\rangle^{\pi 0}\}\ \Rightarrow^{*} \{\langle 8\rangle^{\pi 0}, \langle 3\rangle^{\pi 1}\} \tag{38}$$

$$\begin{aligned} \{\langle 5 + (\textbf{mon}\ \mathsf{nat/c}\ \textbf{concurrent}\ \text{-}1\ B\rangle^{\pi 0}\}\ \Rightarrow^{*} & \{\langle 4\rangle^{\pi 0}, \langle ...\rangle^{\pi 1}\} \qquad \mathit{or} \\ & \{\langle ...\rangle^{\pi 0}, \langle \mathsf{raise}\ B\rangle^{\pi 1}\}\ \mathit{or} \\ & \{\langle 4\rangle^{\pi 0}, \langle \mathsf{raise}\ B\rangle^{\pi 1}\} \end{aligned} \tag{39}$$

We use *“or”* in the second example because our “⇒” relation is non-deterministic: without imposing a scheduling order for processes, the monitor may or may not complete before the initiating evaluator, hence the description of **concurrent** as “best-effort” checking.

5.5.2. *Function Contracts with Concurrent Verification*

For structural and functional contracts, concurrent verification will yield numerous, unsynchronized processes, potentially reporting errors non-deterministically if there are multiple contract violations. This has the downside of so-called "heisenbug"-style contract violations (wherein different traces may terminate with different errors), but the upside is that programmers may utilize concurrent behavior for weak, long-lived contracts. It is also imaginable that we may report these violations as "warnings" to the programmer instead, indicating problematic values without bringing the program to a halt. For example, we may define a new pair contract as:

$$\mathbf{nat\text{-}pair/c}^{conc} \triangleq \mathsf{pair/c}\ \mathsf{nat/c}\ \mathsf{concurrent}\ \mathsf{nat/c}\ \mathsf{concurrent} \tag{40}$$

Enforcing **nat-pair/c**^{conc} will yield a configuration with four processes, and multiple possible outcomes:

$$\begin{array}{ll}
\{\langle \mathsf{fst}\ (\mathsf{mon}\ \mathbf{nat\text{-}pair/c}^{conc}\ \mathsf{concurrent}\ (5,\text{-}1)\ B\rangle^{\pi 0}\} & \\
\Rightarrow^{*}\{\langle 5\rangle^{\pi 0}, \langle(5,\text{-}1)\rangle^{\pi 1}, \langle 5\rangle^{\pi 2}, \langle \ldots\rangle^{\pi 3}\} & \textit{or} \\
\{\langle \ldots\rangle^{\pi 0}, \langle(5,\text{-}1)\rangle^{\pi 1}, \langle 5\rangle^{\pi 2}, \langle \mathsf{raise}\ B\rangle^{\pi 3}\} & \textit{or} \\
\{\langle \ldots\rangle^{\pi 0}, \langle \ldots\rangle^{\pi 1}, \langle \ldots\rangle^{\pi 2}, \langle \mathsf{raise}\ B\rangle^{\pi 3}\} & \textit{or} \\
\textit{etc...} &
\end{array}$$

Because we do not impose scheduling, and each **mon** form with the **concurrent** strategy spawns a new thread, it is possible for a subcontract to signal a violation before its parent contract is complete.

Similarly, consider this concurrent verification behavior with a function contract:

$$\begin{array}{l}
\{\langle(\mathsf{mon}\ (\mathsf{fun/c}\ \mathsf{nat/c}\ \mathsf{concurrent}\ \mathsf{nat/c}\ \mathsf{concurrent})\ \mathsf{eager}\ (\lambda\ x.\ x\ +\ 1)\ B)\ 5\rangle^{\pi_0}\} \\
\Rightarrow^{*}\{\langle((\lambda\ x.\ \mathsf{mon}\ \mathsf{nat/c}\ \mathsf{concurrent}\ ((\lambda\ x.\ x\ +\ 1)\ (\mathsf{mon}\ \mathsf{nat/c}\ \mathsf{concurrent}\ x\ (\mathit{invert}\ B)))\ B)\ 5\rangle^{\pi_0} \\
\quad ,\ \langle \mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^{*}\{\langle \mathsf{mon}\ \mathsf{nat/c}\ \mathsf{concurrent}\ ((\lambda\ x.\ x\ +\ 1)\ (\mathsf{mon}\ \mathsf{nat/c}\ \mathsf{concurrent}\ 5\ (\mathit{invert}\ B)))\ B\rangle^{\pi_0} \\
\quad ,\ \langle \mathsf{unit}\rangle^{\pi_1}\} \\
\Rightarrow^{*}\{\langle \mathsf{mon}\ \mathsf{nat/c}\ \mathsf{concurrent}\ ((\lambda\ x.\ x\ +\ 1)\ 5)\ B\rangle^{\pi_0}\ ,\ \langle \mathsf{unit}\rangle^{\pi_1}\ ,\ \langle \mathsf{nat/c}\ 5\ (\mathit{invert}\ B))\rangle^{\pi_2}\} \\
\Rightarrow^{*}\{\langle \mathsf{mon}\ \mathsf{nat/c}\ \mathsf{concurrent}\ 6\ B\rangle^{\pi_0}\ ,\ \langle \mathsf{unit}\rangle^{\pi_1}\ ,\ \langle 5\rangle^{\pi_2}\} \\
\Rightarrow^{*}\{\langle 6\rangle^{\pi_0}\ ,\ \langle \mathsf{unit}\rangle^{\pi_1}\ ,\ \langle 5\rangle^{\pi_2}\ ,\ \langle \mathsf{nat/c}\ 6\ B)\rangle^{\pi_3}\}
\end{array}$$

This verification trace has some additional irregularities of note:

(1) We verify the top-level **mon** with **eager**. If we had used **concurrent**, the function contract combinator would have produced the monitored procedure in the monitoring process, and

the user program would have proceeded with "$\lambda\ x.\ x\ +\ 1$". An alternative approach to checking a function contract with **concurrent** could be running a sort of enumerative analysis concurrently, checking the function against inputs for the remainder of the program's run. This approach, however, would require us to perform input-based dispatched to determine if the value is a procedure (via, e.g., Racket's `procedure?` operation [39]), removing our uniform approach to contract verification. Clojure's `core.spec` [50] adopts this style of verification for higher-order function contracts, where higher-order function inputs are randomly checked with sampled values to ensure they conform to the specification. We discuss it further in §5.7.

(2) The user process π_0 proceeds without regard for the pre- or post-condition enforcement. While this may not always be the case, based on scheduling, it further illustrates the "best-effort" nature of concurrent verification.

Overall, this trace proceeds by verifying the pre- and post-condition concurrently, eschewing further interactions with the user process. As a result, this concurrent verification allows the user evaluator to proceed without awaiting the contract result, and, further, avoid additional overhead when retrieving contract results. Ultimately, this concurrent strategy allows user computations to forgo contract results in favor of a type of "succeed or interrupt" mechanism, allowing the program to proceed without regard for valid verification results (and interrupting it in the case that a contract violation is discovered).

5.5.3. *The Drawbacks of Concurrent Verification*

As we discussed in Chapter 2, this verification technique may often be "too weak" for properties that the programmer must rely on and, as such, programmers may prefer to ensure the contract monitor will *eventually* finish, regardless of later synchronization.

5.6. Finally-Concurrent Contract Verification—*Verification Without Synchronization*

In order to provide programmers with "start and forget" verification with stronger guarantees, we introduce **fconc** verification. Similar to **concurrent** verification, **fconc** monitoring processes elides secondary synchronization with the initiating process. Unlike **concurrent**, however, we ensure the monitor completes before considering the configuration "done." To this end, we use the spawn variant spawn_f, which creates a new process and adds its process identification number to a list of final processes T, ensuring any answer configuration includes its full evaluation. This interaction

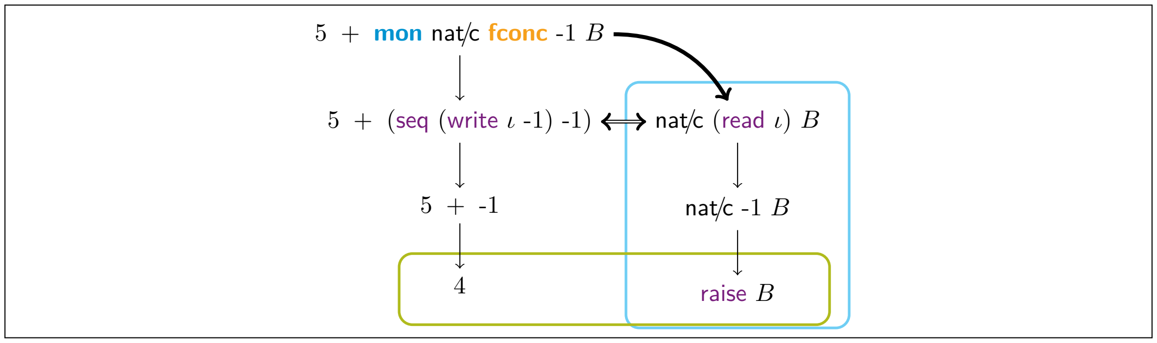


Figure 5.8. Checking nat/c with **fconc**. The initiating and monitoring processes each proceed concurrently, and the $\boxed{\text{green}}$ box indicates an *answer configuration*.

follows the concurrent interaction above, with a singular difference; in **fconc**, the initiating **fconc** process:

(F_{U1}) creates a new communication channel ι;
(F_{U2}) spawns a *final* monitoring process that will evaluate the contract;
(F_{U3}) provides the (evaluated) subject value to the monitoring process across ι;
(F_{U4}) and continues with the evaluated subject value.

Dually, the *final* monitoring process:

(F_{M1}) receives the subject value v across ι;
(F_{M2}) and runs the contract c on the value with the provided blame information.

We present this interaction in Figure 5.8 (with the monitoring process colored $\boxed{\text{blue}}$), which is nearly identical to Figure 5.7, with the addition of the answer configuration in $\boxed{\text{green}}$. The *only* difference between **fconc** and **concurrent** is this notion of process finality, and thus its implementation only exchanges spawn for spawn$_f$:

Definition 5.7 (Finally-Concurrent Verification as a Pattern of Communication).

$$\begin{aligned}\textbf{mon}\ \mathit{con}\ \textbf{fconc}\ \mathit{exp}\ B\ \to\ & \mathit{let}\ i\ =\ \mathit{chan} \\ & \mathit{in}\ \mathit{seq}(\mathit{spawn}_f\ (\mathit{con}\ (\mathit{read}\ i)\ B)\ ) \\ & \qquad (\mathit{let}\ x = \mathit{exp}\ \mathit{in}\ \mathit{seq}\ (\mathit{write}\ i\ x)\ x)\end{aligned} \tag{41}$$

When we use fconc to assert a contract, we may now trust that the contract will run to completion before the program enters an *answer configuration*:

$$\begin{aligned}
&\emptyset, \{\pi_0\}; && \{\langle(\textsf{mon}\ \textsf{nat/c}\ \textsf{fconc}\ \text{-1}\ B) + (\textsf{mon}\ \textsf{nat/c}\ \textsf{fconc}\ 3\ B)\rangle^{\pi_0}\}\\
\Rightarrow^*&\{\iota\}, \{\pi_0, \pi_1, \pi_2\}; && \{\langle \text{-1} + 3\rangle^{\pi_0}, \langle \textsf{nat/c}\ \text{-1}\ B\rangle^{\pi_1}, \langle \textsf{nat/c}\ 3\ B\rangle^{\pi_2}\}\\
\Rightarrow^*&\{\iota\}, \{\pi_0, \pi_1\}; && \{\langle 2\rangle^{\pi_0}\{\langle \textsf{raise}\ B\rangle^{\pi_1}, \langle \textsf{nat/c}\ 3\ B\rangle^{\pi_2}\}\\
\Rightarrow^*&\{\iota\}, \{\pi_0, \pi_1\}; && \{\langle 2\rangle^{\pi_0}\{\langle \textsf{raise}\ B\rangle^{\pi_1}, \langle 3\rangle^{\pi_2}\}
\end{aligned}$$

Even though one contract raised an error, we must still wait for each fconc contract to complete before termination. Aside from this termination behavior, fconc continues exactly as concurrent verification.

This alternative "start and forget" contract verification technique exposes a new avenue for verification: programmers can, e.g., read in a file and speculatively start examining and using the input while being sure that, before the program completes, they will know the data is correct. It may suffer from some of the same problems as we discuss in Chapter 2, including prolonging the program to finalize verification.

5.7. Additional Verification Strategies in λ^{π}_{cs}

In this work, we selected five strategies because of their frequency in the literature, their apparent utility, and their direct encodings. Unsurprisingly, these are not the only variations on contract verification. In this section, we briefly introduce and discuss three additional strategies, sketching their encodings in the λ^{π}_{cs} framework.

5.7.1. *Random Checking.*

Ergün et al. [33] and Dimoulas et al. [29] each describe random testing for program correctness, which has since gained popularity in Clojure's `core.spec` library[50]. These random testing methods verify higher-order function contracts using generative checking (i.e., producing sample inputs to ensure that the function behaves correctly). We can replicate this behavior by introducing a gen strategy that accepts a generator g and ensures the function adheres to its contracts for values provided by this generator:

$$\textsf{mon}\ \mathit{con}\ (\textsf{gen}\ g)\ \mathit{exp}\ B \ \rightarrow \textsf{let}\ i\ =\ \textsf{chan}\\ \textsf{in seq}\ (\textsf{spawn}\ (\textsf{write}\ i\ (\textsf{catch inl}\ (\textsf{let}\ f = (\mathit{con}\ (\textsf{read}\ i)\ B)\\ \textsf{in}\ (\textsf{inr}\ (\textsf{seq}\ (\textsf{check-fn}\ f\ g)\ f))))) \\ (\textsf{write}\ i\ \mathit{exp}) \\ (\mathit{conres}\ (\textsf{read}\ i)) \tag{42}$$

If f is a function, the check-fn procedure (elided) will use the provided generator to randomly test f before returning it; otherwise, following `core.spec`, this strategy behaves as **eager** verification.

5.7.2. *Future Contracts.*

Dimoulas et al. [27] introduce concurrent contracts via *future contracts* (discussed in Chapter 2), which we have used as a basis for our **promise** contract verification. Recall that, in their original presentation, future contracts sent terms and their contracts, as messages, to a secondary evaluator for verification and retrieving results during effectful operations. It is straightforward to imagine replicating this strategy in λ^{π}_{cs}, sending contract-expression pairs to a global, concurrent verification process and extending effectful operations with synchronization operations.

5.7.3. *Lazy Contracts.*

First described by Chitil et al. [17] as *assertions* (without blame mechanisms), Degen et al. [22] later formalized *lazy* contract verification as allowing the user evaluator to "drive" the contract evaluator: the idea is that contract verification should only inspect those terms and values that the user program evaluates, and, as a result the verifiers "block" on unevaluated terms, awaiting their usage in the user program to continue with verification. For example, verifying a predicate contract on a pair will suspend verification until the user evaluator evaluates subcomponent of the pair; if the user program never does, the monitor will never verify the contract. Degen et al. [24] present an implementation of this system for Haskell, using individual call-by-need reference cells register callbacks for contract monitors to driving the contract verification mechanism as these reference cells are evaluated.

This model of monitoring, when translated into λ^{π}_{cs}, demonstrates its intrusive interaction with the main evaluator: to facilitate this user-driven monitoring, we must construct a layer of indirection for both evaluators such that the user evaluator's forcing an expression drives the lazy monitor. To do this, we must recursively parse the input expression e, breaking it out into a structure for the

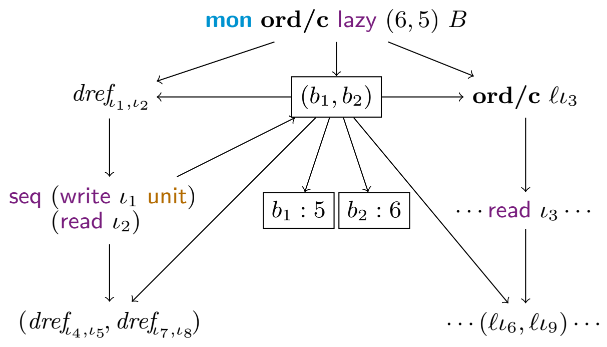


Figure 5.9. Lazy contract communication.

user process to evaluate and a second, mirrored structure for the monitoring evaluator to wait on. We provide a sample sketch of this approach on a pair contract in Figure 5.9, using so-called "lazy references" ℓi that produce values as the user programs evaluates the delayed references.

Between this massive intrusion (and structural meta-operation in the form of building lazy references), we can see that lazy evaluation is not particularly practical, and, furthermore, Degen et al. [24] observe that such lazy verification violates basic blame consistency. Taken together, this suggests that lazy contract verification ultimately has only questionable utility. That said, this is not a shortcoming of λ^{π}_{cs}: Degen et al. [22] hint at the intrusive nature, and, by encoding it in λ^{π}_{cs}, we have exposed the precise mechanisms such verification requires.

5.8. Mixing Strategies with Contracts in λ^{π}_{cs}

Beyond choosing which strategy to use for each contract, programmers may also freely intermix strategies in λ^{π}_{cs}, yielding flexibility and utility beyond traditional contract systems. In this section, we summarize our strategies so far, giving their implementations together in Figure 5.10, and then showcase this additional advantage of λ^{π}_{cs} by providing three examples: dependent function contracts, a flexible binary-search tree, and a lazy tree fullness contract.

5.8.1. *Dependent Function Contracts*

Our first advanced contract in λ^{π}_{cs} is fun/dc, a *dependent* variant of fun/c [36]. The idea is that the post-condition contract takes the *function's argument* as input before producing a contract, allowing the post-condition contract to verify the function output in terms of the input. For example, we

```
mon con eager exp B  →      let i  =  chan
                            in seq
                                (spawn (write i (catch inl (inr (con (read i) B)))))
                                (write i exp)
                                (conres (read i))

mon con semi exp B  →       delay
                             (let i  =  chan
                              in seq
                                  (spawn (write i (catch inl (inr (con (read i) B)))))
                                  (write i exp)
                                  (conres (read i)))

mon con promise exp B  →    let i  =  chan
                            in seq
                                (spawn (write i (catch inl (inr (con (read i) B)))))
                                (write i exp)
                                (delay (conres (read i))))

mon con concurrent exp B → let i  =  chan
                            in seq (spawn (con (read i) B))
                                   (let x = exp in seq (write i x) x)

mon con fconc exp B  →      let i  =  chan
                            in seq (spawn_f (con (read i) B))
                                   (let x = exp in seq (write i x) x)
```

Figure 5.10. The aggregate contract verification strategies presented in Chapter 5.

can verify that a square root procedure behaves correctly as:

$$\text{sqrt/dc} := \lambda\ s_1\ s_2\ s_3.\ \text{fun/dc nat/c}\ s_1\ (\lambda\ n.\ \text{pred/c}\ (\lambda\ m.\ (m * m) = (\text{force}\ n)))\ s_2\ s_3$$

This contract allow us to ensure that a function's result is exactly the square root of its input[4]. Note that we have added a "preemptive" force in the post-condition to ensure that we will force the contract before use, addressing those cases where the verification strategy used for the post-condition input results in a delayed expression.

[4]This procedure should technically use some ϵ to ensure the result is within bounds to deal with floating-point errors, but we elide this complexity for presentation.

We define this fun/dc combinator as:

$$\begin{aligned}\mathsf{fun/dc} \triangleq\ & \lambda\ con_1\ strat_1\ fcon_2\ strat_2\ strat_p. \\ & \quad \lambda\ f\ b. \\ & \quad\quad \lambda\ x.\ \mathsf{let}\ input := \mathbf{mon}\ con_1\ strat_1\ x\ (\mathit{invert}\ b) \\ & \quad\quad\quad\quad postInput := \mathbf{mon}\ con_1\ strat_p\ x\ (\mathit{indy}\ b) \\ & \quad\quad\quad \mathsf{in}\ \ \mathbf{mon}\ (fcon_2\ postInput)\ strat_2\ (f\ input)\ b\end{aligned} \tag{43}$$

This definition differs from our definition of fun/c in Eqn. 34 as follows:

- In this definition we utilize the *indy* blame mechanism presented by Dimoulas et al. [27], wherein we verify the pre-condition monitor on the function input twice: first using standard pre-condition blame (as *invert blame*), and a second time with *indy*-style contract blame (as "*indy blame*"), which will blame the post-condition contract if it misuses the input value, e.g., misapplying a procedure argument. Otherwise, as Dimoulas et al. [27] observe, the post-condition may misuse the function input in a way that results in the blame labels indicating the function itself is responsible for the error, resulting in incorrect blame assignment.
- Our λ^{π}_{cs} definition of fun/dc introduces a new point of flexibility with dependent contracts: a secondary possibility for control with the second verification site. We may use a *different strategy* when checking the input for the post-condition than for the main function.

To further demonstrate this second point, consider our choice of strategy in sqrt/dc: when using the **eager** strategy, we recover behavior akin to the behavior described by Dimoulas et al. [27]:

$$(\mathbf{mon}\ (\mathsf{sqrt/dc}\ \mathbf{eager}\ \mathbf{eager}\ \mathbf{eager})\ \mathbf{eager}\ sqrt\ B)\ 25$$

This is not, however, the only option: since λ^{π}_{cs} allows us to freely intermix strategies, we can explore, for example, ensuring the pre-condition holds on the post-condition's input via **promise**:

$$(\mathbf{mon}\ (\mathsf{sqrt/dc}\ \mathbf{eager}\ \mathbf{eager}\ \mathbf{promise})\ \mathbf{eager}\ sqrt\ B)\ 25$$

When we apply the resultant contracted procedure to 25, the *postInput* check runs concurrently, freeing up the user process to continue computing sqrt/dc *while* the secondary process concurrently verifies the post-condition's input contract. Consider, further, using semi-eager verification for each sub-contract:

$$(\mathbf{mon}\ (\mathsf{sqrt/dc}\ \mathbf{semi}\ \mathbf{semi}\ \mathbf{semi})\ \mathbf{eager}\ sqrt\ B)\ 25$$

In this example, the program forces the *postInput* variable *after* *sqrt* has completed. In larger programs, where *sqrt* itself may throw an error, this can cut down on potential computation repetition before the error. And these are only *three* examples!

This straightforward extension to a standard contract combinator begins to illustrate the immense user control in multi-strategy verification: we can control complex contracts with secondary, hidden behavior, precisely specifying how they interact with the user program at each verification step. Our next two examples will further reinforce this additional flexibility.

5.8.2. *A Flexible Binary-Search Tree Contract*

Next, we revisit our strategy-parameterized binary tree contract from Chapter 3. While it is possible to construct this contract using only the primitives in λ^{π}_{cs}, we take some liberties here for simplicity of presentation, namely:

- We assume a tree data structure in the form of leaf nodes and internal nodes "node $val\ t_{left}\ t_{right}$", where we may directly match on trees with these constructors as[5]:

$$\mathsf{case}\ (t;\ \mathsf{leaf} \rhd e_1; \mathsf{node}\ val\ t_{left}\ t_{right} \rhd e_2)$$

- We also assume a fixpoint operator μ as

$$(\mu f\ x.e)\ v \to e[\mu f\ x.e/f][v/x]$$

Recall from Chapter 3 that our binary-search tree contract should act as follows:

- **mon eager** (bst/c **eager**) *tree B* will eagerly ensure its input is a binary-search tree;
- **mon semi** (bst/c **semi**) *tree B* will return a tree that will verify each node is correctly-ordered as the program explores it;
- **mon promise** (bst/c **promise**) *tree B* will create a cascading chain of monitoring processes for each node, and a program exploring the tree will synchronize with the appropriate processes at each level;
- and **mon concurrent** (bst/c **concurrent**) *tree B* will concurrently enforce that the tree is a binary-search tree using a similar set of cascading processes.
- and **mon fconc** (bst/c **fconc**) *tree B* will act the same as the **concurrent** contract, but force the program to wait for verification to complete before terminating.

[5]We use these to avoid defining binary trees as a series of sum types.

Ideally, we would like to define bst/c in terms of a tree contract combinator tree/c. We can define this structural contract combinator recursively as:

$$\begin{aligned}
\mathsf{tree/c} :=\ & \mu\ con\ c_{leaf}\ s_{leaf}\ c_{node}\ s_{node}\ s_{rec}. \\
& \quad \lambda\ tree\ b. \\
& \quad\quad \mathsf{case}_{tree}\ (tree; \\
& \quad\quad\quad \mathsf{leaf} \qquad\quad \triangleright \mathbf{mon}\ c_{leaf}\ s_{leaf}\ \mathsf{leaf}\ b; \\
& \quad\quad\quad \mathsf{node}\ v\ tl\ tr\ \triangleright \mathsf{let}\ c_{rec} = con\ c_{leaf}\ s_{leaf}\ c_{node}\ s_{node}\ s_{rec} \\
& \quad\quad\quad\quad \mathsf{in\ node}\ (\mathbf{mon}\ c_{node}\ s_{node}\ v\ b) \\
& \quad\quad\quad\quad\quad (\mathbf{mon}\ c_{rec}\ s_{rec}\ tl\ b) \\
& \quad\quad\quad\quad\quad (\mathbf{mon}\ c_{rec}\ s_{rec}\ tr\ b))
\end{aligned} \tag{44}$$

This combinator takes five arguments:

(1) c_{leaf} is a contract for *leaf* nodes (which is generally uninteresting, because leaves are unit values in this representation);
(2) s_{leaf} is the strategy describing how to enforce c_{leaf};
(3) c_{node} is a contract for internal tree values;
(4) s_{node} is the strategy describing how to enforce c_{node};
(5) and s_{rec} is the strategy describing how to recursively enforce the resultant tree contract on each node's sub-trees.

To demonstrate its usage, we may use this combinator to define a contract to verify each element in a given tree is nat/c:

$$\mathsf{any/c} := \mathsf{pred/c}\ (\lambda\ x.\ \mathsf{true}) \tag{45}$$

$$\mathbf{nat\text{-}tree/c}^{eager} := \mathsf{tree/c}\ \mathsf{any/c}\ \mathbf{eager}\ \mathsf{nat/c}\ \mathbf{eager}\ \mathbf{eager} \tag{46}$$

The resultant contract works as follows:

- Each (nullary) leaf value is essentially ignored via any/c, the contract that always holds.
- At each internal node, we eagerly verify the node's value is a natural number and, further, eagerly traverse the left and right subtrees.

Using this tree/c combinator, we can also define a generalized version of this contract that takes two strategy arguments representing the enforcement strategies for each interim value and the recursive strategy (respectively):

$$\textbf{nat-tree/c}^{s} := \lambda\ s_1\ s_2.\ \mathsf{tree/c}\ \mathsf{any/c}\ s_1\ \mathsf{nat/c}\ s_1\ s_2 \tag{47}$$

We can redefine **nat-tree/c**^{eager} in terms of this combinator as:

$$\textbf{nat-tree/c}^{eager} := \textbf{nat-tree/c}^{s}\ \textbf{eager eager} \tag{48}$$

Unfortunately, we cannot define bst/c in terms of tree/c: in order to ensure that a tree is a binary-search tree, we must check that each value in the left sub-tree is less than (or equal to) its parent node's value and, similarly, each value in the right sub-tree is greater (or equal to) than its parent node's value. This value flow requires a *dependent* tree contract, similar to dependent contracts, wherein each sub-contract receives the current node's value as an input before assertion. We can define such a combinator as:

```
tree/dc := λ $c_{leaf}$ $s_{leaf}$ $c_{node}$ $s_{node}$ $c_{left}$ $c_{right}$ $s_{rec}$.          (49)
              λ $tree$ $b$.
                case$_{tree}$ ($tree$;
                    leaf          ▷mon $c_{leaf}$ $s_{leaf}$ leaf $b$;
                    node $v$ $tl$ $tr$  ▷node (mon $c_{node}$ $s_{node}$ $v$ $b$)
                                     (mon ($c_{left}$ $v$) $s_{rec}$ $tl$ $b$)
                                     (mon ($c_{right}$ $v$) $s_{rec}$ $tr$ $b$))
```

This combinator takes seven arguments[6]

(1) c_{leaf} is a contract for leaf nodes ;

(2) s_{leaf} is the strategy describing how to enforce c_{leaf};

(3) c_{node} is a contract for internal tree values;

(4) s_{node} is the strategy describing how to enforce c_{node};

(5) c_{left} and c_{right} are two procedures that expect a node value as input and yield the appropriate contracts for the left and right sub-trees (respectively);

(6) and s_{rec} is the strategy describing how to recursively enforce the resultant contract on the left and right sub-trees.

[6]Other variants of this contract include one that uses a single strategy at every contract, and one that uses the same strategy for the node's value and sub-trees. These alternatives, however, trade expressiveness for programmatic ease.

We can use this dependent contract combinator to define $\mathsf{bst/c}$ directly[7]:

$$\begin{array}{lll}\mathsf{bst/c} := \lambda\ s.\ \mu\ \mathsf{bst/c}\ s\ lo\ hi. & & \\ \quad \mathsf{tree/dc}\ \mathsf{any/c} & & \textbf{eager} \\ \quad\quad (\mathsf{pred/c}\ (\lambda\ x.\ (lo \leq x)\mathit{and}(x \leq hi))) & & \textbf{eager} \\ \quad\quad (\lambda\ v.\ \mathsf{bst/c}\ s\ lo\ v)\ (\lambda\ v.\ \mathsf{bst/c}\ s\ v\ hi)\ s & & \end{array} \tag{50}$$

This contract verifies that each internal value in the tree is within the correct numeric bounds, propagating the node values downward. As $\mathsf{bst/c}$ **eager**, this contract must traverse the entire tree to enforce this constraint, requiring $O(n)$ time (whereas an insertion algorithm would require $O(\log n)$ time in a sorted tree). We can weaken this guarantee, however, and use $\mathsf{bst/c}$ **semi** to enforce the invariant on exactly the nodes we visit during the program, recovering $O(\log n)$ complexity for insertion. Furthermore, we can completely decouple the evaluation via $\mathsf{bst/c}$ **promise**, starting the entire assertion in concurrent processes and only synchronizing with (and waiting on) those nodes required by the program; perform best-effort verification with $\mathsf{bst/c}$ **concurrent**; use **fconc** for finally-concurrent, start-and-forget verification; and check $\mathsf{bst/c}$ **eager** under **promise** as "**mon** ($\mathsf{bst/c}$ **promise**) **eager** *tree B*", constructing a promise that will concurrently enforce the entire contract. Each of these variations follows from the same definition of $\mathsf{bst/c}$.

Even further, we may define a secondary version of $\mathsf{bst/c}$ that takes an additional value-enforcement parameter, allowing us to control exactly when to verify each *node value* in addition to the general recursive verification scheme:

$$\begin{array}{lll}\mathsf{bst/c}^s := \lambda\ s_1\ s_2.\ \mu\ \mathsf{bst/c}\ s\ lo\ hi. & & \\ \quad \mathsf{tree/dc}\ \mathsf{any/c} & & \textbf{eager} \\ \quad\quad (\mathsf{pred/c}\ (\lambda\ x.\ (lo \leq x)\mathit{and}(x \leq hi))) & & s_2 \\ \quad\quad (\lambda\ v.\ \mathsf{bst/c}\ s_1\ lo\ v)\ (\lambda\ v.\ \mathsf{bst/c}\ s_1\ v\ hi) s_1 & & \end{array} \tag{51}$$

5.8.3. *A Lazy Tree Fullness Contract.*

Our final example tackles the problem of lazily ensuring that a binary tree is full (that is, each node's sub-trees have the same height). Findler et al. [37] identify such checks as particularly difficult in the context of verification, as verifying this property for a given node require the verifier to fully inspect the node's children. This style of *upward value propagation* is presented in Figure 5.11, and this value flow will not allow lazy structural contracts to incrementally verify this property.

[7]We could also ensure the tree's values are natural numbers by adding that requirement to the value contract conjunction.

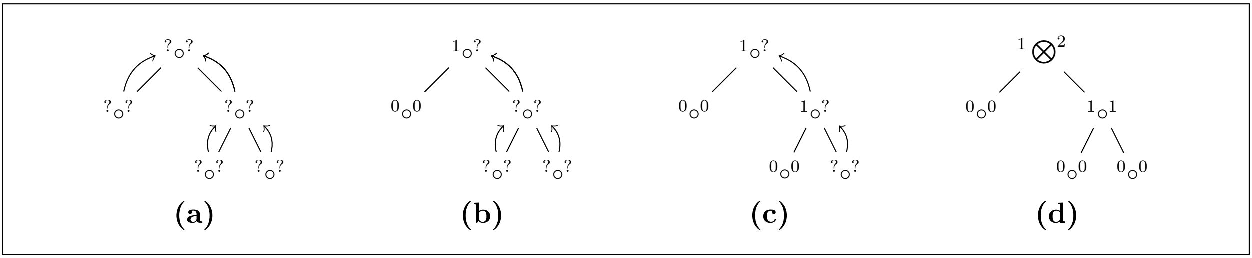


Figure 5.11. Evolution of contract checking during tree traversal for a full binary tree using asynchronous callbacks.

To further explain this problem, consider using the predicate full? to define a predicate contract:

```
full? := let f = μ full tree .                                                    (52)
                 case_tree (tree;
                       leaf        ▷0;
                       node v tl tr ▷let hl = full tl
                                         hr = full tr
                                     in if (hr = hl) and (hl ≥ 0) and (hr ≥ 0)
                                        then 1 + hl
                                        else -1
         in λ t. 0 ≤ (f t)
```

$$\text{full/c} := \text{pred/c full?} \tag{53}$$

In general, we must traverse an entire tree to know if it is full: each node must first inspect its children, using the recursive results to determine if it is full before propagating this information upward to ensure the property for the entire structure. This style of value propagation through monitored structures generally inhibits semi-eager enforcement: if we enforce full/c using **semi** on a tree t, the monitor will traverse the *entire* tree when forced.

In λ^{π}_{cs}, however, we have an alternative solution: we can create custom value flows for contracts, allowing us to use a series of interacting processes that communicate via secondary communication channels to verify this property. To construct this mechanism, we use the **concurrent** strategy, creating processes that serve as “callbacks” to propagate the information upward.

While complex in concept, the only additional facility we require is a choice-based reading operation in order to communicate with multiple subcontracts at once. This choice operator is a straightforward addition to λ^{π}_{cs}: we extend the "matches" relation from Figure 4.2 to support choice as:

$$\frac{e_1 \overset{\iota}{\frown\!\!\!\smile} e_2 \text{ with } (e_1', e_2')}{e_1 \overset{\iota}{\frown\!\!\!\smile} \mathsf{choice}\ e_2\ e_3 \text{ with } (e_1', e_2')} \qquad \frac{e_1 \overset{\iota}{\frown\!\!\!\smile} e_3 \text{ with } (e_1', e_3')}{e_1 \overset{\iota}{\frown\!\!\!\smile} \mathsf{choice}\ e_2\ e_3 \text{ with } (e_1', e_3')}$$

The [SYNCHRONIZE] operation in Figure 4.2, in conjunction with this extended definition, allows us to perform choice-based communication via choice, i.e., to retrieve a result from one of two channels:

$$\begin{aligned} &\{\langle \mathsf{choice}\ \iota_1\ \iota_2\rangle^{\pi_0}, \langle \mathsf{write}\ \iota_1\ 1\rangle^{\pi_1}, \langle \mathsf{write}\ \iota_2\ 2\rangle^{\pi_2}\} \\ \Rightarrow &\{\langle 1\rangle^{\pi_0}, \langle \mathsf{unit}\rangle^{\pi_1}, \langle \mathsf{write}\ \iota_2\ 2\rangle^{\pi_2}\} \\ \mathit{or}\ &\{\langle 2\rangle^{\pi_0}, \langle \mathsf{write}\ \iota_1\ 1\rangle^{\pi_1}, \langle \mathsf{unit}\rangle^{\pi_2}\} \end{aligned}$$

With this choice operator in place, we may now express a lazy fullness contract using "callback"-style communication:

```
full/fc := μ full i .                                                    (54)
           let i_l = chan
               i_r = chan
           in tree/dc (pred/c (λ _. seq (write i 0) true))              eager
                      (pred/c (λ _. let h1 = choice (read i_l) (read i_r)
                                        h2 = choice (read i_l) (read i_r)
                                    in if h1 = h2
                                       then (seq (write i (1 + h1)) true)
                                       else false))
                      concurrent
                      (λ _. full i_l)
                      (λ _. full i_r)
                      semi))
```

We parameterize each contract invocation by a communication channel i, indicating where to write the current nodes height. At leaf nodes, the contract writes 0 to i and succeeds. At internal nodes, we pass two fresh channels, i_l and i_r to the left and right sub-trees respectively. Next we assert ($\mathit{full}\ i_l$) and ($\mathit{full}\ i_r$) on the appropriate sub-trees, utilizing the dependent contract to delay these invocations until usage time (to prevent divergence). We verify each of these subcontracts with **semi** postponing verification verified until the initiating process demands them. Finally, the **concurrent**-verified node value contract retrieves its sub-tree heights across i_l and i_r. If these two heights are equal, the contract writes the appropriate height across i and succeeds (triggering its parent's fullness test); if not, the contract signals a violation.

This communication pattern allows each contract to propagate values via side-channel communication, working together to lazily establish global properties about programs. Such a contract is only possible after fully separating of the monitor evaluator from the user evaluator and exposing communication tools to contract writers that facilitate contract verification via custom-crafted traversal of monitored structures.

Effectful Tree Fullness. Unsurprisingly, this is not the only solution to lazily ensuring tree fullness. Utilizing effectful contracts, we can imagine a dependent contract that keeps track of its recursion depth and, in each leaf node, reports this depth to a secondary process, which will raise a contract violation if it ever receives two disagreeing depths (indicating the tree is not full). This style of effectful contract verification allows programmers to verify global properties with less overhead, and, as we will see in Chapter 6, it may be directly encoded as a *contract verification meta-strategy*, modifying the underlying strategy's behavior to provide this additional runtime verification mechanism.

CHAPTER 6

Beyond Contracts: Verification Meta-strategies

SYNOPSIS

Strategies, on their own, do not form a complete calculus; they are, in a sense, the basic "values" of a rich *strategy calculus* that describes the nature of runtime verification. To further illustrate this idea, we now introduce operators of this calculus in the form of *meta-strategies*, which are strategy-level operators that take one or more strategies (plus additional arguments) as input and produce new verification behaviors. In this chapter, we define a number of such meta-strategies, including **with**, **comm**, **random**, **memo**, and **transition** (§6.1-6.4); sketch additional points in this meta-strategy design space (§6.5); and, finally, demonstrate how these meta-strategies pave the way for reasoning about contract interactions with other effects (§6.6).

Contract verification strategies each describe how to monitor contracts in terms of how and when they interact with the user evaluator. In this chapter, we introduce and explore the notion of *meta-strategies*, or strategy-modifying strategies, that augment this contract verification behavior to extend λ^{π}_{cs} to general runtime verification techniques (including, e.g., spot-checking and state-machine verification).

At their core, meta-strategies take and produce new strategies, supplementing or altering the sub-strategy behavior. For example, the **with** meta-strategy takes an effectful procedure as an argument and applies this procedure to the monitored value before returning the monitored value to the initiating process. Combining this meta-strategy and the effectful *println* operation, we can write a function contract that verifies the function's input is a natural number *and* prints that input to the console:

$$addWithPrint \triangleq \textbf{mon}\ \textbf{eager}\ (\mathsf{fun/c}\ (\textbf{with}\ \mathit{println}\ \textbf{eager})\ \mathsf{nat/c}\ \textbf{eager}\ \mathsf{nat/c})\ (\lambda\ x.\ x\ +\ 10)\ B$$

$$\{\langle addWithPrint\ 10\rangle^{\pi}\}\ \Rightarrow^{*}\ \{\langle 20\rangle^{\pi}, \cdots\} \qquad \texttt{console: 10}$$

The **with** meta-strategy also serves as a key component in general runtime instrumentation. For example, we can use it to store each of a procedure's inputs in a list by using **with** and an operator that adds its argument to a reference cell[1]:

$$
\begin{array}{lll}
\mathsf{let} & \mathit{args} & = \mathit{ref\ emptyList} \\
 & \mathit{logger} & = \lambda\ x.\ \mathit{args}\ :=\ (\mathit{cons}\ x\ (!\mathit{args})) \\
 & \mathsf{log/c} & = \mathsf{fun/c}\ (\mathbf{with}\ \mathit{logger}\ \mathbf{eager})\ \mathsf{nat/c}\ \mathbf{eager}\ \mathsf{nat/c} \\
 & \mathit{incrWithLog} & = \mathbf{mon}\ \mathbf{eager}\ \mathsf{log/c}\ (\lambda\ x.\ x\ +\ 1)\ B \\
\mathsf{in} & \mathsf{seq} & \mathit{incrWithLog}\ 10 \\
 & & \mathit{incrWithLog}\ 5 \\
 & & \mathit{incrWithLog}\ 0 \\
 & & !\mathit{args} \\
\Rightarrow^{*} & [0,\ 5,\ 10] &
\end{array}
$$

In this example, we define *logger*, which adds its argument to the list in the *args* reference cell, and provide this *logger* as the secondary procedure to **with** in the pre-condition strategy for log/c. As the program proceeds, each invocation of *incrWithLog* stores the function argument in *args*, resulting in the collective list of arguments as output.

In the rest of this chapter, we will introduce a number of additional verification meta-strategies and use them to develop additional, generalized runtime inspection and verification tools.

6.1. The With Meta-strategy—*Performing Additional Operations*

We now begin our formal definitions of meta-strategies by defining **with**, which takes a sub-strategy and a procedure and, similar to a dependent contract [36], applies the procedure to the contracted result. Unlike a dependent contract, however, the meta-strategy applies the function and discards the result (instead of using the result as a new contact). Enforcement under the **with** meta-strategy proceeds as:

(M_{W1}) the meta-strategy checks its contract using the sub-strategy s;

(M_{W2}) passes the contracted result as an argument to the provided procedure f, discarding the application's result;

(M_{W3}) and, finally, returns the contracted result to the user process.

[1]Throughout this chapter, we use additional, minor extensions to λ^{π}_{cs}, including references, which we create with *ref initialValue*, dereference with *!reference*, and update with *reference* $:=$ *newValue*; and lists, wherein we create new, empty lists with *emptyList*, add elements to the front as *cons newValue list*, and print as $[elem_1,\ elem_2,\ \cdots]$. These operators follow the usual semantics as presented by Pierce [73].

We implement this meta-strategy behavior in λ^{π}_{cs} as[2]:

Definition 6.1 (The **with** meta-strategy).

$$\begin{aligned}\textbf{mon}\ (\textbf{\textit{with}}\ f\ \ s)\ con\ exp\ B\ \ \rightarrow\ & \textit{let}\ x = \textbf{mon}\ s\ con\ exp\ B\\ & \textit{in}\ \textit{seq}\ (f\ \ (\textbf{mon}\ s\ con\ exp\ (\textit{indy}\ B)))\\ & \qquad x\end{aligned}$$

Our blame in this definition is similarly informed by indy-style blame [27] in dependent function monitoring (§5.8.1): we use the *indy* operation to establish blame, ensuring any contract violation that occurs while computing f blames *the contract itself* for over-exploring the value and causing the violation. This is, in some sense, an over-simplification, however: Dimoulas et al. [27] define *indy* verification to introduce contracts themselves as separate, accountable parties in blame tuples, and it is conceivable that we should extend this approach to account for each meta-strategy as a separate blame entity. This extension is, however, straight-forward following the ownership model presented by Dimoulas et al. [27], and so we leave it as future work.

In addition to our previous use-cases, this meta-strategy provides immense utility for combining contracts with additional runtime verification and inspection. To illustrate this utility, consider the following example, wherein we use **with** to implement general function time profiling as follows (using *curTime* to get the current system time):

$$tagTimer \triangleq \lambda\ timeRef.\ \lambda\ tag.\ timeRef\ :=\ cons\ ([tag,\ curTime])\ (!timeRef)$$

$$\begin{aligned}slowFact \triangleq\ & \mu\ slowFact\ n.\\ & \mathsf{if}\ n\ =\ 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{seq}\ (sleep\ 10)\ (n\ *\ (slowFact\ (n\ -\ 1)))\end{aligned}$$

$$\begin{aligned}\mathsf{let}\ \ & timerRef\ =\ \ ref\ emptyList\\ & timer\ =\ \ tagTimer\ timerRef\\ & sfTimed\ =\ \ \textbf{mon}\ \ \textbf{eager}\\ & \qquad\qquad (\mathsf{fun/c}\ \ (\textbf{with}\ (\lambda\ x.\ timer\ \text{``pre"})\ \textbf{eager})\ \ \ \mathsf{nat/c}\\ & \qquad\qquad\qquad\quad (\textbf{with}\ (\lambda\ x.\ timer\ \text{``post"})\ \textbf{eager})\ \ \ \mathsf{nat/c})\\ & \qquad\qquad slowFact\\ \mathsf{in}\ \ & \mathsf{seq}\ sfTimed\ 10\\ & \quad !timerRef\end{aligned}$$

$$\Rightarrow^{*}\ \ [[\text{``post"},\ 1493529262817],\ [\text{``pre"},\ 1493529262254]]$$

[2]It is conceivable that we should use additional processes for implementation meta-strategies in order to follow our previous patterns-of-communication approach. It is, in this context, straight-forward to abstract meta-strategies into their own processes, and so we eschew the abstraction to clarify presentation.

This example uses **with** to instrument each pre-condition and post-condition to store a tagged timestamp in the *timerRef* global reference, allowing the programmer to track how much time their program spends in that particular procedure (including procedures it calls). At the end of the program, we can get an entire list of these calls and, e.g., use this information to compute the average function runtime or its runtime as a percentage of the whole program.

This style of strategy and meta-strategy dispatch gives programmers an extensible, uniform interface to inspect and verify general runtime performance of programs. Furthermore, this interface allows programmers to perform this instrumentation *in tandem* with contract verification: in the above example, contracts on *slowFact* ensure that each input and output is a natural number *in addition* to recording timing information.

6.1.1. *The Communication Meta-strategy—Communicating Contract Results*

Our next meta-strategy is **comm**, a special-case variation of the **with** meta-strategy that provides inter-process communication with the goal of simplifying the implementation of side-channel contracts [80]. The idea is to communicate the result of the **with** procedure across some communication channel ι, allowing programmers to preserve the result of the meta-strategy computation *and* continue with the original program. This "communication" meta-strategy takes a procedure f and a communication channel ι as input and, when used, provides the monitored term to the procedure f and write the computation's result across channel ι before returning the contracted value.

Similar to the **with** meta-strategy, **comm** enforcement proceeds as:

(M_{C1}) the meta-strategy checks its contract using the sub-strategy s;
(M_{C2}) passes contracted result as an argument to the provided procedure f;
(M_{C3}) writes the application's result across ι;
(M_{C3}) and, finally, returns the contracted result to the user process.

We implement this meta-strategy in λ^{π}_{cs} as:

Definition 6.2 (The **comm** meta-strategy).

$$\begin{array}{lll} \textsf{mon}\ (\textsf{comm}\ f\ \iota\ s)\ con\ exp\ B & \rightarrow & \textsf{let}\ x = \textsf{mon}\ s\ con\ exp\ B \\ & & \textsf{in}\ \textsf{seq}\ (\textsf{write}\ \iota\ (f\ (\textsf{mon}\ s\ con\ exp\ (\textsf{indy}\ B)))) \\ & & \quad\quad x \end{array}$$

We may now use **comm** to simplify our implementation of full/fc from Chapter 5, replacing the ad-hoc communication in the leaf case with **comm**:

```
full/fc := μfull i .                                                              (55)
        let i_l = chan
            i_r = chan
        in tree/dc any/c                                                 (comm (λ _. 0) i eager)
               (pred/c (λ _. let h1 = choice (read i_l) (read i_r)
                                 h1 = choice (read i_l) (read i_r)
                             in if h1  =  h2
                                then (seq (write i (1 + h1)) true)
                                else false))
               concurrent
               (λ _. full i_l)
               (λ _. full i_r)
               semi))
```

Observe, however, that we cannot use this meta-strategy in the node value case for two reasons:

- First, the communication meta-strategy will block indefinitely until a process reads from ι, an unfortunate, but not surprising, result of write's blocking nature in λ^{π}_{cs}. If, however, we were to introduce some form of non-blocking write (or simply spawn the "write ι f val" expression in a new process), we could implement some revised **comm-nb** meta-strategy with this non-blocking form and use it to perform concurrent side-channel communication.
- The node-value contract relies on the communication result, and decoupling this value flow would require further communication structures.

Even so, the **comm** meta-strategy has secondary utility: this style of meta-strategy also allows us to interact with effects modeled as *sessions*, including state and logging effects implemented as separate processes that provides the effects. Orchard and Yoshida [70] explore the relation between session-based processes and effects, demonstrating how processes may be used to replicate effectful behavior. In Section 6.6, we adopt this approach to implement a refined fullness contract, tracking the depth of each leaf node and ensuring they match (a sufficient condition for tree fullness). This close interaction between contracts and other suggests that we may also quantify contracts in such a framework, but this remains as future work.

6.2. The Random Meta-strategy—*Probabilistic Contract Enforcement*

Our next meta-strategy, **random**, explores the idea of *random enforcement*. Ergün et al. [33] and Dimoulas et al. [29] each describe notions of random testing for program correctness, wherein we may forgo contract verification at each assertion site (e.g., only occasionally ensuring that a function behaves correctly). In our meta-strategy implementation, we take some "checking rate" and verify the given contract at that rate, skipping other verification sites. To provide this behavior, the **random** meta-strategy proceeds as:

(M_{R1}) the meta-strategy generates a random floating-point number n between 0 and 1;
(M_{R2}) if n is less than the checking rate, performs contract verification, returning the result;
(M_{W3}) otherwise, the monitor yields original expression to the initiating evaluator.

We implement this meta-strategy in λ^{π}_{cs} as follows, where the *random* operator returns a random value between 0 and 1:

Definition 6.3 (The **random** meta-strategy).

$$\begin{aligned}\textsf{mon}\ (\textit{random}\ rate\ s)\ con\ exp\ B\ \rightarrow\ & \textit{if}\ rate > (\textit{random})\\ & \textit{then}\ \textsf{mon}\ s\ con\ exp\ B\\ & \textit{else}\ exp\end{aligned}$$

This meta-strategy provides an alternative mechanism for decreasing contract verification overhead in favor of a spot-checking verification, eschewing some number of checks to recover performance while providing a margin of verification. For example, consider a binary-search tree insertion operator *bstInsert*, which takes a value and tree and inserts the value into the tree using a binary-search tree insertion algorithm. In Chapter 5, we discussed **semi** as a mechanism to recover asymptotic behavior by delaying each check. Using **random**, we can introduce another alternative to eliminate some overhead by probabilistically verifying bst/c on *bstInsert* one-tenth of the time:

$$\begin{aligned}\textit{bstInsert}^{r}\ \triangleq\ \textsf{mon}\ \ & \textbf{eager}\\ & (\textsf{fun/c}\ \ (\textbf{random}\ 0.1\ \textbf{eager})\ (\textsf{bst/c}\ (\textbf{random}\ 0.1\ \textbf{eager}))\\ & \qquad\qquad (\textsf{fun/c}\ \textbf{eager}\ \textsf{nat/c}\ \textbf{eager}\ (\textsf{bst/c}\ \textbf{eager})))\\ & \textit{bstInsert}\end{aligned}$$

The contract's pre-condition enforces each bst/c via "**random** 0.1 **eager**", ensuring verification only proceeds (eagerly) one-tenth of the time. This will notably reduce the contract overhead of insertion while still spot-checking that *bstInsert* exhibits correct behavior over the course of the program. Moreover, we may still mix and match strategies, allowing us to, for example, use **fconc** to randomly, concurrently ensure the pre- and post-conditions:

```
bstInsert^r ≜ mon eager
                  (fun/c   (random 0.1 fconc) (bst/c (random 0.1 fconc))
                           (fun/c eager nat/c (random 0.1 fconc) (bst/c (random 0.1 fconc))))
                  bstInsert
```

Using **fconc** as the sub-strategy and argument to bst/c, λ^{π}_{cs} will probabilistically check each level of the pre- and post-condition contracts in another process while the main program continues, stripping away contract overhead in favor of spot-checked correctness.

6.3. The Memoization Meta-strategy—*Caching Contract Results*

In most modern contract systems, contract verification is pervasive to the point of impacting program performance [17, 36, 37]. In this section, we introduce another alternative: when contracts are *idempotent* (i.e., they will always return the same result for a given input), we may cache these results, avoiding reverification in the future.

To implement this meta-strategy, we follow the standard approach to memoization in dynamic programming solutions [19], establishing a hash map that contains contract-value pairs as keys and verification results as values[3]. To this end, we will need the following operations:

- *contains?* check if a map contains a key, returning true or false accordingly;
- *lookup* returns the entry's value in a map;
- and *update-map* adds a new entry and value to a map.

Using these operations, the **memo** meta-strategy:

(M_{M1}) looks up the contract/expression pair in the provided map

(M_{R2}) if the map contains the pair, it returns the result and dispatches appropriately;

(M_{W3}) otherwise, it proceeds with contract evaluation, modifying the contract to store the enforcement result in the memoization map.

The contract modification (M_{W3}) is a subtle requirement: we cannot be sure the contract holds until the underlying verification proceeds, and less-eager structures (such as **semi**) need a mechanism to update the map after performing verification.

[3]We actually only need to retain the contract-value pairs, as negative results signal errors. We use a hash-map interface for uniformity with other memoization approaches.

We implement this meta-strategy as:

Definition 6.4 (The memo meta-strategy).

```
mon (memo map s) con exp B  →
        if contains? map (con, exp)
        then if lookup map (con, exp)
                then exp
                else raise B
        else let con′ = λ x b. case (catch inl (inr (con x b)));
                                              y ▷ seq (update-map map (con, exp) false) (raise y);
                                              z ▷ seq (update-map map (con, exp) true) z)
            in mon s con′ exp B
```

As stated above, we use contract-expression pairs (as (con, exp)) with the associated memoization map. In addition, we introduce the revised contract definition as con': when the contract result is not part of the map, we wrap the underlying contract check in a variation of the conres handler that stores the verification result before reporting the result to the program (by either returning the value or re-raising the violation, as necessary).

As with **random**, **memo** is particularly effective for reducing performance overhead for contracted functions. For example, the following code will check each input and output to *fibonacci* exactly once, memoizing the results[4]:

```
fib_m ≜ let map    = emptyMap
            fibCon = fun/c (memo map eager) nat/c (memo map eager) nat/c
        in  mon eager fibCon fibonacci B
```

Repeated invocations of fib_m will store the pre- and post-condition verification results, trading contract verification overhead for storage space.

6.4. The Transition Meta-Strategy—*Ensuring State Machines with Verification*

Our final verification meta-strategy deals with verifying state-based program behavior, similar to the *protocols* described by Dimoulas et al. [30], wherein a series of contracts collaborate to ensure a program is proceeding according to some state machine. The idea is that, instead of tracking or maintaining an individual state, we can use collaborative contracts to model a program's correct behavior as a series of state transitions, ensuring the program performs a specific set of operations

[4]We use the subscript m to indicate a monitored variation.

in a specific order, e.g., verifying that a program does not request an iterator's next value without first checking it has one [54].

To specify these transition rules, we introduce a *pair* of meta-strategies:

- The **transition** meta-strategy takes the strategy state (i.e., a memory reference to the current state), a "from" state indicating the transition source, a "to-state" list of transition targets (allowing us to model non-deterministic state transition machines), and a sub-strategy to enforce the underlying contract:

  $$\textbf{transition}\ \textit{state}\ \textit{from}\ [\textit{to-state}]\ \textit{subStrat}$$

  We model "to-state" as a list in order to support non-deterministic machines, where the list represents *each* of the valid next states. In addition, we assume that the current state value is stored as such a list, classifying a valid transition as any transition whose "from" argument is an element of this list. If the state list does not include this "from" argument, the meta-strategy signals an error.
- The **transition-as** meta-strategy takes the strategy state and a procedure *transition-f* parameterized by the current strategy state, the contracted value, and blame information:

  $$\textbf{transition-as}\ \textit{state}\ \textit{op}\ \textit{strat}$$

  If the transition function raises an error, the meta-strategy signals an error; otherwise, we update the state with the function result and return the verification result to the user program. Similar to dependent contracts, this meta-strategy allows the transition function to examine the *contract input*, parameterizing its transition by the values currently flowing through the program.

We define these two strategies as follows, using *elem?* to check if an individual element is in a list to ensure correct transitions, raising errors when appropriate:

DEFINITION 6.5 (The state-transition **transition** and **transition-as** meta-strategies).

$$\begin{array}{lll}\textbf{mon}\ (\textbf{\textit{transition}}\ \textit{state}\ \textit{from}\ \textit{toList}\ \textit{strat})\ \textit{con}\ \textit{exp}\ b & \rightarrow & \textit{if}\ \textit{elem?}\ (!\textit{state})\ \textit{from} \\ & & \textit{then}\ \textit{seq}\ \textit{state}\ :=\ \textit{toList} \\ & & \qquad\qquad \textbf{mon}\ \textit{strat}\ \textit{con}\ \textit{exp}\ b \\ & & \textit{else}\ \textit{raise}\ b\end{array}$$

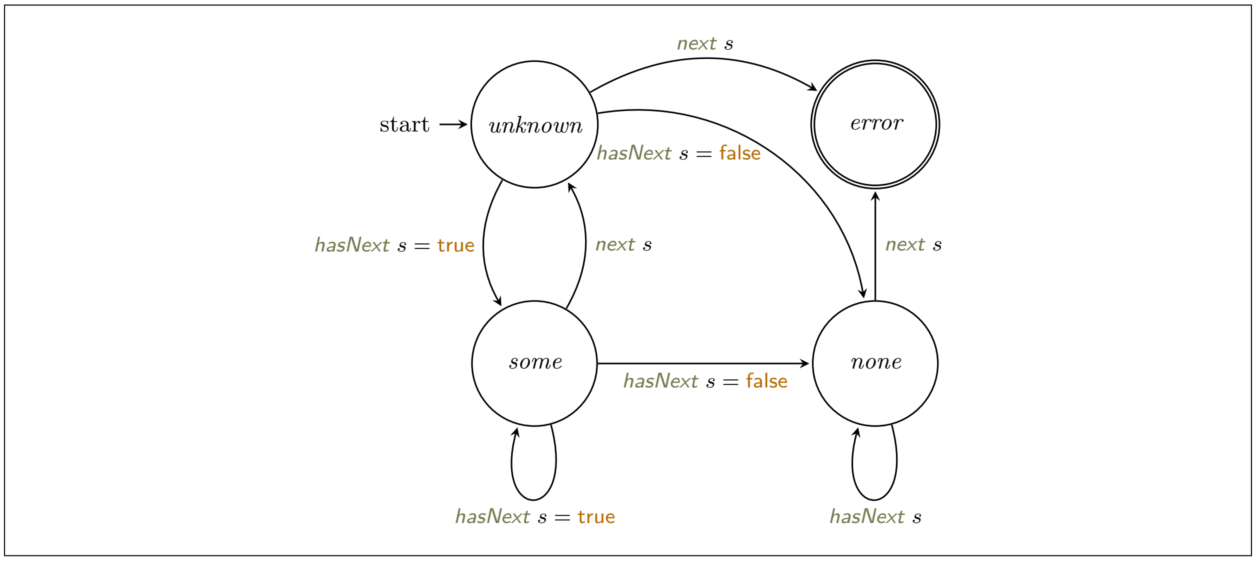


Figure 6.1. A state machine ensuring a program checks an iterator has another element before attempting to retrieve it.

$$\begin{array}{lll}\textsf{mon}\ (\textit{transition-as}\ state\ op\ strat)\ con\ exp\ b & \rightarrow & \textsf{let}\ rinput = \textsf{mon}\ strat\ con\ exp\ (\textsf{indy}\ b) \\ & & \textsf{in seq}\ state\ := \ \textsf{catch}\ (\lambda\ b'.\ \textsf{raise}\ b') \\ & & \qquad\qquad\qquad (op\ (!state)\ rinput\ b) \\ & & \qquad \textsf{mon}\ strat\ con\ exp\ b\end{array}$$

In **transition-as**, we once again use indy-style blame to ensure that the contract, and not the user program, is correctly blamed for the error in the transition function.

We also provide *makeContractState* to create a contract state:

$$makeContractState \triangleq \lambda\ startState.\ \textsf{ref}\ [startState]$$

We can now use this transition-style meta-strategy to ensure that, for example, a program uses a list iterator correctly, *always* calling hasNext on the iterator (and receiving true) before invoking next to retrieve the next iterator value. (We provide the state machine describing this property in Figure 6.1.)

In order to model this behavior in λ^{π}_{cs} using our transition-based meta-strategies, we treat state machine errors as contract errors and write monitors that model and enforce this behavior for

$\mathsf{hasNext}$ and next, implementing the $\mathsf{next/c}$ and $\mathsf{hasNext/c}$ contracts as:

$$\begin{aligned}
iterState &\triangleq makeContractState\ \text{“}\mathsf{unknown}\text{"} \\
hasNextTrans &\triangleq \lambda\ curState\ conRes\ b.\ \mathsf{if}\ (\mathsf{force}\ conRes)\ \mathsf{then}\ [\text{“}\mathsf{some}\text{"}]\ \mathsf{else}\ [\text{“}\mathsf{none}\text{"}] \\
\mathsf{hasNext/c} &\triangleq \mathsf{fun/c}\ \mathbf{eager}\ \mathsf{iter/c}\ (\mathbf{transition\text{-}as}\ iterState\ hasNextTrans\ \mathbf{eager})\ \mathsf{any/c} \\
\mathsf{next/c} &\triangleq \mathsf{fun/c}\ \mathbf{eager}\ \mathsf{iter/c}\ (\mathbf{transition}\ iterState\ \text{“}\mathsf{some}\text{"}\ [\text{“}\mathsf{unknown}\text{"}]\ \mathbf{eager})\ \mathsf{any/c}
\end{aligned}$$

In this encoding, we use strings to represent each state[5] as follows:

- For $\mathsf{hasNext}$, the state transition depends on the procedure's result, and thus we must use **transition-as** with the user-defined *hasNextTrans* to determine the correct transition state (i.e., “some" if $\mathsf{hasNext}$ returns true and “none" if $\mathsf{hasNext}$ returns false).
- We model next's behavior using **transition**, where the *only* valid transition is from “some" to “unknown"; using next on an iterator in any other state will produce a runtime violation in accordance with our implementation of **transition**.

Using these definitions, we can now define next_m and $\mathsf{hasNext}_m$, monitored variants of next and $\mathsf{hasNext}$ that use these contracts:

$$\mathsf{next}_m \triangleq \mathbf{mon}\ \mathbf{eager}\ \mathsf{next/c}\ \mathsf{next}\ B$$

$$\mathsf{hasNext}_m \triangleq \mathbf{mon}\ \mathbf{eager}\ \mathsf{hasNext/c}\ \mathsf{hasNext}\ B$$

These monitored procedures ensure each iterator uses $\mathsf{hasNext}_m$ before next_m:

$$\begin{array}{l}
\mathsf{let}\ iter = \mathsf{toIterator}\ [1] \\
\mathsf{in}\ \mathsf{if}\ (\mathsf{hasNext}_m\ iter)\ \mathsf{then}\ (\mathsf{next}_m\ iter)\ \mathsf{else}\ \mathsf{unit} \\
\Rightarrow^* 1 \\
\\
\mathsf{let}\ iter = \mathsf{toIterator}\ [1] \\
\mathsf{in}\ \mathsf{next}_m\ iter \\
\Rightarrow^* \mathsf{raise}\ B
\end{array}$$

In the first example, the program terminates as expected. In the second, however, the program invokes next_m on the iterator before invoking $\mathsf{hasNext}_m$, an invalid state transition, and the transition-enforcement mechanism raises an error.

[5]Recall that $\mathsf{any/c}$ is the permissive contract that allows for any result.

6.5. Additional Contract Meta-strategies

As with verification strategies, this collection of meta-strategies is only an initial exposure of the design space, intended as a starting point for further design and discussion. In this section, we introduce three additional meta-strategies, sketching their encodings in the λ^{π}_{cs} framework.

6.5.1. *Two-Way Communication.*

Our **comm** strategy provides one-way communication with a process, but advanced verification structures frequently require two-way interaction. We can provide this two-way operation as a meta-strategy with call/return style communication. This **comm/ret** meta-strategy proceeds as **comm**, taking a procedure and a channel, writing the procedure's result across the channel. In addition, the **comm/ret** strategy reads a verification flag from the channel, allowing the process to signal a verification error to the monitoring form. We define this strategy as:

Definition 6.6 (The **comm/ret** meta-strategy).

$$\begin{array}{lll} \textbf{mon}\ (\textit{comm/ret}\ f\ i\ s)\ con\ exp\ B & \rightarrow & \textit{let}\ x = \textbf{mon}\ s\ con\ exp\ B \\ & & \quad \textit{seq}(\textit{write}\ \iota\ (f\ (\textbf{mon}\ s\ con\ exp\ (\textit{indy}\ B)))) \\ & & \textit{in} \quad \textit{if}\ (\textit{read}\ \iota)\ \textit{then}\ \mathsf{unit}\ \textit{else}\ \textit{raise}\ B \\ & & \qquad x \end{array}$$

As we will see in the Section 6.6, this strategy allows us to use communication with a secondary process to verify global program properties, e.g., ensuring that a tree is full by using our secondary process to track the depth of leaf node, signaling an error if any two nodes occur at different depths.

6.5.2. *Optional Verification.*

Dimoulas et al. [29] define *option contracts*, wherein modules may optionally "exercise" (verify) or "waive" (discard) contract verification, avoiding unnecessary contract verification to recover performance. It is possible to replicate this type behavior in λ^{π}_{cs} as a sort of **option** meta-strategy, utilizing our delaying structures in λ^{π}_{cs} to encode contracts via a procedure that, when forced, accepts "exercise" and "waive" as inputs, performing the appropriate operation in each case:

Definition 6.7 (The **option** meta-strategy).

$$
\begin{aligned}
\textsf{mon}\ (\textsf{option}\ strat)\ con\ exp\ B \ \rightarrow\ \textsf{delay}\ (\lambda\ op.\ & \textsf{if}\ op\ ==\ \text{“exercise"} \\
& \textsf{then}\ \textsf{mon}\ con\ strat\ exp\ b \\
& \textsf{else if}\ op\ ==\ \text{“waive"} \\
& \quad \textsf{then}\ exp \\
& \quad \textsf{else raise}\ B)
\end{aligned}
$$

We may additionally define *exercise* and *waive* to force the value and provide the appropriate input, allowing us to exercise or waive any **option**-verified contracts.

$$\textsf{exercise}\ :=\ \lambda\ x.\ (\textsf{force}\ x)\ \text{“exercise"}$$

$$\textsf{waive}\ :=\ \lambda\ x.\ (\textsf{force}\ x)\ \text{“waive"}$$

While this meta-strategy replicates the basic style of option contracts, their original presentation also provides a blame characterization in which waived contracts may blame the waiving party if a violation is later discovered. Our calculus does not support the rich blame encoding infrastructure used by Dimoulas et al. [29], however, and incorporating this style of blame-carrying calculus with λ_{cs}^{π} remains a future work.

6.5.3. *Security Enforcement Contracts.*

Moore et al. [68] introduce contracts to model authorization and access control, which provide a domain-specific language for writing security-centric contracts, ensuring specific procedures cannot call specific operations (e.g., reading a file's contents). To recreate this behavior, we can introduce a **security** meta-strategy that supports the custom syntactic forms described by Moore et al. [68]. This system will also require us to further modify λ_{cs}^{π} to support the dynamically-bound parameter scope system this contract model requires, which we speculate we may encode via process interactions in our λ_{cs}^{π} framework. Such an encoding remains as future work in the meta-strategy design space.

6.6. Contracts as Generalized Runtime Effects

As discussed earlier in this chapter, Orchard and Yoshida [70] explore the dual nature of algebraic effects and session-style processes, demonstrating that we may directly encode algebraic effects in terms of processes. This encoding relies on the notion that algebraic effects may be modeled as separate, interacting processes, e.g., a recursive process may serve as a “state” effect by accepting *get* and *put* operations across a channel, responding appropriately and recurring on itself with the current state. In the context of λ^{π}_{cs}, a contract communicating with such an "effect" can utilize these operations to interact with the effect for verification purposes, allowing us to encode any general runtime verification effect expressible as an algebraic effect and/or session as part of the λ^{π}_{cs} semantic framework.

To illustrate the utility of this approach, we spend the rest of this section developing the effect-based tree fullness contract previously described in Chapter 5, using the **comm/ret** meta-strategy and a secondary, state management process to develop an effect-based semi-eager tree fullness contract. The tree contract will keep track of the current tree depth, reporting it to the *state effect process* at each leaf node, and the state effect process will ensure that each leaf node occurs at the same depth (ensuring the tree is full), recurring with the depth value.

We begin with defining the state effect process as a procedure that takes a communication channel i. This procedure uses a recursive fixpoint to maintain its *state* as an argument between invocations, initializing it with -1 and modifying it based on its interactions (e.g., tracking the depth of the tree's leaf nodes):

```
stateProc ≜ λ i.
        let s = μ stateProc state.
                let action = read i
                in  if (fst action == "end")
                    then unit
                    else if ((fst action) == "depth") and state < 0
                    then seq write i true
                             stateProc (snd action)
                    else if (fst action) == "depth"
                    then if (snd action) == state
                         then seq write i true
                                  stateProc (snd action)
                         else seq write i false
                                  stateProc (snd action)
                    else stateProc state
        in s -1
```

We structure this state effect as a session-style recursive procedure, where each loop of the process reads an *action* across the communication channel i and then performs the appropriate behavior:

- If the i input is a pair whose first element is "end", the manager terminates with unit.

- If the i input is a pair whose first element is "depth", we compare the second element of the pair with the recursive state:
  - If the recursive state is less than zero, indicating the state has not seen a correct depth yet, we write true across i (indicating the contract holds) and recur with the second element.
  - If the two states match (indicating the leaf nodes are at the same depth, and thus this part of the tree is also full), we write true across i (indicating the contract holds) and recur with the depth value.
  - If the state does not match the new input, we write false across i (indicating that **comm/ret** should signal a contract violation) and recur with the depth value.
- If the message does not match one of the two forms above, we discard it, recurring appropriately.

Using this definition, we can now write a fullness contract that interacts with this state effect process to semi-eagerly verify tree fullness, using **comm/ret** at the leaf nodes to report the depth of each leaf node to the state effect:

```
full/pc := λ i.                                                                    (56)
          let full = μfull depth.
                  tree/dc any/c                (comm/ret i (λ _. ("depth", d)) eager)
                          nat/c                semi
                          (λ _. full (d + 1))
                          (λ _. full (d + 1))
                          semi
          in full 0
```

We can verify this contract on a *tree* by allocating a channel, spawning our state process, and proceeding with verification:

```
let i = chan
in seq (spawn (stateProc i B) )
       (mon (full/pc i) semi tree B)
```

As the program explores the contracted tree, each node will semi-eagerly propagate its depth downward to its subcontracts, and each leaf node will report its depth to the state process. If any two leaves occur at different depths, indicating the tree is *not full*, and the program explores both leaves, the **comm/ret** form will receive false and raise an error at the exploration site.

While this state procedure resembles the timer implementation in §6.1, this approach illustrates a more general result: our λ^{π}_{cs} framework allows programmers to write effectful contracts that forgo ad-hoc effect interactions in favor of these principled effect interaction mechanisms. Moreover, this allows contracts to interact with *any* algebraic effect, subsuming the majority of programmatic effects [10, 12, 56, 64, 70]. The multi- and meta-strategy approach in λ^{π}_{cs} provides a generalized interaction system between these types of effects and contract monitors.

CHAPTER 7

Pisces: An Implementation with Advanced Examples

SYNOPSIS

Thus far, we have used λ^{π}_{cs} to define a runtime verification framework with contract primitives, introducing verification strategies and meta-strategies to provide myriad verification (and inspection) behaviors. Recall, however, our claim in Chapter 4 that the λ^{π}_{cs} features we used to build our verification framework are standard to many modern languages. To realize this claim, we now introduce PISCES, an extensible verification library for the Clojure programming language that follows our formal semantics. In this chapter, we introduce basic Clojure syntax (§7.1), present PISCES incrementally as a series of Clojure definitions (§7.2), and use PISCES to recreate examples from our previous chapters (§7.4).

In Chapter 4, we described the only four mechanisms required to encode our multi-strategy verification system in an existing language: error propagation (including raising and catching errors), forcing and delaying expressions, processes with communication, and, finally, some mechanism to delay arguments before evaluation.

In this chapter, we demonstrate this in the Clojure programming language [49], using its implementation of these features to define PISCES, an extensible runtime verification library that provides many of the same facilities as our λ^{π}_{cs}-based verification framework. First, however, we take a moment to introduce and discuss the Clojure language, explaining its basic syntactic forms, in order to explain our PISCES implementation.

7.1. Basic Clojure Operations

Clojure is a higher-order, functional language in the Lisp family [49, 65] that natively targets the Java Virtual Machine (JVM) [62]. Its syntax introduces minor syntactic variations on classic Lisp expressions, but overall follows the standard structure of Lisp programs. This section provides a brief overview of these syntactic operations, and we refer the interested reader to the Clojure documentation for more details [49].

7.1.1. *Basic Syntax*

As with other Lisp dialects, Clojure's basic unit of code is the *function*, which we may define as:

```
(defn add1
  [x]
  (+ x 1))
```

In this example, we use the defn form to indicate we are creating a new function definition, providing the name add1, a list of arguments (in this case, the single argument x), and the function body "(+ x 1)" (which uses Clojure's prefix-style syntax to increment the function's input by 1).

Clojure also provides a let form, similar in behavior to our let form in λ^{π}_{cs}, as:

```
> (let [x 1
        y 2]
    (+ x y))
=> 3
```

In this example, x is bound to 1 and y is bound to 2, producing 3 as the final result of the let body "(+ x y)". Note that, unlike our own let form, Clojure does not symbolically delineate bindings in its let binding form.

Beyond these basics, Clojure maintains many features of Lisp, including higher-order functions. For example, we may define app−to−5, which applies its input argument (as a procedure) to 5:

```
(defn app-to-5
  [f]
  (f 5))

> (app-to-5 add1)
=> 6
```

In this example, we pass add1 to app−to−5, which then applies add1 to 5, yielding 6. In addition to higher-order procedures, Clojure also supports anonymous functions via fn:

```
> (app-to-5 (fn [x] (+ x 2)))
=> 7
```

Here, we provide the function "(fn [x] (+ x 2))" to app−to−5, yielding 7 as the program result.

7.1.2. *Advances Features of Clojure*

In addition to the basic forms above, Clojure provides a number of operations we will use over the course of developing Pisces, including defining and using record structures, raising errors, Clojure's delay and @ operators, and syntactic macros (to avoid over-evaluation). As we will see in the next section, we may elide the need for error propagation and process creation to simplify our Pisces implementation.

Records in Clojure. Clojure allows programmers to define record structures via defrecord. For example, we can define a record that represents a point in 2D space:

```
(defrecord Point [x y])
```

This definition introduces a new constructor Point. that creates a Point record with fields x and y[1]:

```
(def point-a (Point. 2 5))
```

We can retrieve values from a record by affixing : to the name of its field, resulting in record accessors:

```
> (:x point-a)
=> 2

> (:y point-a)
=> 5
```

Further, we can ask if a given value is an instance of a specific record as:

```
> (instance? Point point-a)
true

> (instance? Point 25)
false
```

Finally, we observe that records in Clojure are modeled as Java classes in the JVM, which is relevant to our discussion of errors below.

Errors in Clojure. Since Clojure is based on the JVM, which works with Exception objects, we must create such objects to raise errors. Clojure provides this creation through its record-creation syntax as:

```
(Exception. "test error")
```

Here, "test error" is an error message to report to the user when raising the exception.

We may raise this exception (escaping the surrounding program context), via Clojure's throw operation:

```
> (+ (+ 2 3) (throw (Exception. "test error")))
=> Exception test error <debug info...>
```

[1]Observe that we use def instead of defn here to indicate to Clojure that we are defining a value (not a procedure using its procedure syntax).

In this example, throwing our exception discards the outer addition context, reporting the exception as the program result.

Delaying and Forcing Expressions. Similar to λ^{π}_{cs}, Clojure allows us to delay expressions via a delay operation, producing a delayed value:

```
> (delay (+ 2 3))
=> <delayed ....>
```

In this example, the sub-expression "(+ 2 3)" is stored, unevaluated, as a delayed value. Clojure also provides a short-hand forcing operation @ for evaluating these delayed values:

```
> (let [x (delay (+ 2 3))]
    @x)
=> 5
```

In this example, the @ affixed to the front of the expression acts as a forcing form, evaluating the sub-expression and yielding 5 as a result.

Syntactic Macros. Clojure provides syntactic macros, allowing users to write language forms for rewriting syntax. For example, we may use Clojure's defmacro to write a syntactic form that produces a new, unevaluated expression from its input:

```
(defmacro plus-three
  [x y z]
  '(+ ~x (+ ~y ~z)))
```

In this definition, we use the quote operator "'" to indicate we are creating a new syntactic form and the unquote operator "~" to embed the expressions bound to x, y, and z, yielding the final term:

```
> (plus-three (* 2 3) 3 4)
-> (+ (* 2 3) (+ 3 4))
=> 13
```

As we will see in the next section, this syntax-rewriting mechanism will ultimately allow us to define a **mon** form that delays the monitored term's evaluation in the appropriate way.

7.2. Implementing the Pisces Library

With a basic understanding of Clojure's syntax, we now turn our attention to Pisces, introducing our implementation incrementally: first, we introduce record structures for creating blame, strategies, and meta-strategies; next, we define our monitoring form (e.g., **mon**) in terms of these structures; after that, we use these structures to define a series of contract combinators; and, finally, we define a number of strategies and meta-strategies to use with this core. This piecewise implementation approach ensures PISCES is extensible, allowing programmers to define additional verification strategies and contract combinators for their own use.

7.2.1. *Core* PISCES

We begin with defining the blame, strategy, and meta-strategy record structures and the **mon** syntactic macro, allowing us to define the rest of the PISCES library in terms of these definitions.

Blame. Our blame structure follows our previous λ_{cs}^{π} presentation (which, in turn, follows Dimoulas et al. [27]), using a three-value Blame record to maintain a server (accounting for the monitored value), a client (accounting for the program context), and the contract itself:

```
(defrecord Blame [server client contract])
```

In addition to this record form, we define invert−blame and indy−blame to allow combinators to correctly restructure blame information as necessary:

```
(defn invert-blame
  [blame]
    (Blame. (:client blame) (:server blame) (:contract blame)))

(defn indy-blame
  [blame]
    (Blame. (:client blame) (:contract blame) (:contract blame)))
```

These definitions each create a new Blame. record, using the associated record accessors to rearrange the blame information as appropriate. We also introduce blm as an initial blame value:

```
(def blm (Blame. "server" "client" "contract"))
```

This value allows programmers to test blame locally without manually writing each individual field. Automatically extracting blame information, such as in Racket [36], remains future work.

Strategy and Meta-Strategy Records. In PISCES, we model each strategy and meta-strategy as an instance of a Strategy or Metastrat record, respectively. These records carry verification implementations, ensuring verification behavior is *implicitly* strategy-defined (as part of each record's impl field) , allowing users to define their own verification strategies. We define these records as:

```
(defrecord Strategy [sname impl])

(defrecord Metastrat [sname impl substrat])
```

The Strategy record takes a strategy name and an associated implementation procedure that represents *how* to proceed with verification, and, similarly, the Mestrat takes a strategy name, implementation, and verification *sub-strategy* describing how underlying verification should proceed[2].

We postpone defining individual verification strategies until the next section, first focusing on how **mon** will use these individual strategies in order to guide our implementation.

Defining Monitors. We define the **mon** language form in terms of two sub-procedures, mon-flat and mon-meta, which retrieve and invoke the :impl field from strategies and meta-strategies, respectively. Our mon-flat proceeds as follows:

```
(defn mon-flat
  [contract strat dterm blame]
  (cond
    (instance? Strategy strat)
      ((:impl strat) contract dterm blame)
    :else (throw (Exception. (str "Invalid strategy: " strat "\n")))))
```

In this implementation, mon-flat expects a contract to verify, a strategy strat, monitored value dterm (which, as we will see below, is delayed as part of **mon** to avoid over-evaluation), and blame information. The implementation proceeds by ensuring strat is a strategy before retrieving the implementation (via :impl), and applying it to the other inputs[3].

Our mon-meta procedure follows this definition, accepting the same series of arguments and dispatching based on the input strategy[4]:

```
(defn mon-meta
  [contract strat dterm blame]
  (cond
    (instance? Metastrat strat)
      ((:impl strat) contract (:substrat strat) dterm blame)
    (instance? Strategy strat)
      (mon-flat contract strat dterm blame)
    :else (Exception. (str "Invalid strategy: " strat "\n"))))
```

In the case that strat is a Metastrat, we retrieve the implementation and apply it to the contract, monitored term, verification sub-strategy (which we retrieve with "(:substrat strat)"), and blame information. If strat is a Strategy, however, we dispatch to mon-flat to handle it. This definition allows us to use mon-meta as the single entry point for verification, thus defining **mon** as a macro that rewrites to an invocation of mon-meta as:

[2]It's possible that meta-strategies should manage substrategies as part of their own implementation, but we leave this possibility as future work.
[3]If the strat argument is *not* a Strategy, we throw the appropriate exception.
[4]We use Clojure's cond operator, which takes a series of tests and associated clauses, evaluating the first clause whose associated test returns true.

```
(defmacro mon
  "Check a contract with a specific strategy"
  [contract strat expr blame]
  `(mon-meta ~contract ~strat (delay ~expr) ~blame))
```

In order to ensure contracted expressions are not immediately evaluated, we use Clojure's defmacro to wrap expr in a delay form before passing it into mon-meta. This delayed structure may then be forced by each strategy if and when it becomes necessary, allowing individual verification strategies to avoid over-evaluation.

Finally, we define an **extract** operator, analogous to force in λ_{cs}^{π}, to force delayed expressions (and, as we will see later, computational futures):

```
(defn extract
  [exp]
  (if (or (delay? exp) (future? exp)) @exp exp))
```

Our **extract** operation checks if the input is a delayed term or computation future, forcing either and returning the expression in any other case. This allows us to blindly **extract** any value without Clojure producing an error due to the misapplication of its @ operator.

7.2.2. *Contract Combinators*

With our core definitions in place, we now turn our attention to defining predc, pairc, and func in terms of **mon** and the blame definitions in the previous section, introducing each as a "user-level" procedure on top of this core system. As with λ_{cs}^{π}, we encode contracts as functions that take a monitored value and blame information as input and perform their own verification, maintaining our contract-agnostic verification approach in PISCES.

Predicate Contracts. We start with predc, a predicate contract combinator, taking a predicate as input and returning a verification procedure that will return the input or raise an error (based on the verification result):

```
(defn predc
  [f]
    (fn [x blame]
      (if (f x)
          x
          (throw (Exception.
                   (str "Contract violation: " x
                        " violated " f "\n" "Blame: " blame))))))
```

This definition follows our previous pred/c definition, using Clojure's throw operator to raise the appropriate error when the predicate does not hold.

As expected, we may now use this definition to recreate our previous contracts in Clojure, e.g. anyc and natc:

```
(def anyc (predc (fn [x] true)))

(def natc (predc (fn [x] (>= x 0))))
```

To define anyc, we invoke predc on "(fn [x] true)", a predicate will returns true for any input, and, similarly, we define natc with a predicate that verifies its input is greater than or equal to zero. We will use these predicates in the next section to demonstrate that our verification strategy definitions proceed correctly.

Pair Contracts. Similar to predc, our definition of pairc follows our pair/c definition in Chapter 5, where we use Clojure's list-creation syntax [fst snd] to create a two-element list of monitored results:

```
(defn pairc
    [c1 c2 s]
    (fn [pair blame]
        [(mon c1 s (first pair) blame)
         (mon c2 s (second pair) blame)]))
```

In this presentation, we opt for a single strategy parameter to pairc, deviating from pair/c by accepting and using the same strategy argument s for both subcontract verifications.

Using this combinator, we can now define natpairc, a strategy-parameterized contract that verifies each element in a pair is a natural number:

```
(defn natpairc
  [strat]
  (pairc natc natc strat))
```

Function Contracts. Finally, we define func, a variant of fun/c from Chapter 5. Unlike our previous definition, this implementation deals with a variable number of arguments, allowing the user to write contracts for functions with multiple inputs. For example, using func, we may write a contract for a binary-search tree insertion procedure that takes a tree and insertion value as inputs, semi-eagerly verifying the input tree is a binary-search tree, eagerly verifying the input value is a natural number, and, finally, eagerly verifying its result is a binary-search tree. We would hope to write such a contract as:

```
(def bst-insertc-ses
  (func bstc semi natc eager bstc eager))
```

In order to handle this variable-arity input, we must pre-process the contract inputs, "raveling" together the contract-strategy pairs and grouping them with the appropriate function argument at each invocation. To this end, we begin with defining a specialized con−ravel procedure:

```
(defn con-ravel
  [args ins]
  (if (empty? ins)
      (list)
      (cons (concat (take 2 args) (list (first ins)))
            (con-ravel (drop 2 args) (rest ins)))))
```

This procedure takes a list of args, representing the contracts and their strategies, and a list of ins, representing the contracted procedure's input. It then "ravels" these lists together by retrieving the first two elements of args (a contract and its strategy) and combining it with the first element of ins, proceeding recursively until there are no more procedure arguments (and raising a runtime error if there are an insufficient number of contract-strategy pairs).

Using this definition, we build func as follows:

```
1  (defn func
2    [& scs]
3    (fn [f blame]
4        (fn [& ins]
5          (let [l  (* 2 (count ins))
6                cl (count scs)]
7            (if (not (= (+ 2 l) cl))
8              (throw
9                (Exception. "Invalid number of arguments for contracts")))
10           (let [mon-sets (con-ravel scs ins)
11                 posts    (drop l scs)]
12             (mon (first posts)
13                  (second posts)
14                  (apply f
15                         (map (fn [x]
16                                (mon (first x)
17                                     (second x)
18                                     (second (rest x))
19                                     (invert-blame blame)))
20                              mon-sets))
21                  blame))))))
```

Unlike our previous two combinators, this definition departs from its λ^{π}_{cs} counterpart, and thus we walk through it line-by-line:

- (line 1) we define the func form;
- (line 2) func takes a variable number of arguments as [& scs], which indicates to Clojure to bind *all* of the arguments to scs as a list;
- (line 3) we follow our λ^{π}_{cs} definition of fun/c, accepting a procedure to monitor and its associated blame information;
- (line 4) we produce a new, variable-argument procedure whose inputs are bound, as a list, to ins as the contract result;

- (line 5–9) when applied, we ensure that ins has the appropriate number of arguments to match our contracts, and otherwise signal an error;
- (lines 10–11) we group together the input contracts with their strategies as mon−sets and bind the post-condition with its strategy as posts;
- (lines 12–13) we monitor the post-condition on the procedure result;
- (line 14) we apply the monitored procedure f to the monitored pre-condition arguments
- (lines 15–20) and, as the input to f, we map over mon−sets, monitoring each contract on its appropriate input.

While complicated in construction, this provides an elegant interface for users to define contracts for any function regardless of argument count. For example, we can use func to define a function contract with one input and one output, ensuring each is a natural number, *and* to define a function contract with two inputs and a single output:

```
(defn nat-natc
  (func natc eager natc eager))

(defn natxnat-natc
  (func natc eager natc eager natc eager))
```

Out first contract, nat−natc, will verify the monitored procedure takes and returns natural numbers while our second, natxnat−natc, will verify the monitored procedure takes two natural numbers as arguments and returns a natural number.

7.2.3. *Defining Contracts and Strategies*

With our structural definitions, monitoring forms, and combinators in place, we now define a number of verification strategies for PISCES using the Strategy record structure, developing implementations of skip, eager, semi-eager, promise-based, concurrent, and spot-checking verification. These implementations, given in Figure 7.1, each generally follow our semantic descriptions in Chapter 5, but we omit process creation and communication whenever possible to reduce the computational overhead of verification.

Skip Verification. The skip verification strategy *skips* enforcement, discarding its contract and returning the monitored value. In our implementation, we define skip-check, the skip verification procedure, as a function that applies extract to the delayed term and returns the result. Then, using this procedure, we define a Strategy instance skip that uses skip-check as its implementation. The resultant strategy will forgo all contract checking.

```clojure
; Skip Strategy
(defn skip-check
  [contract dterm blame]
  (extract dterm))

(def skip  (Strategy. "skip" skip-check))

; Eager Strategy
(defn eager-check
  [contract dterm blame]
  (contract (extract dterm) blame))

(def eager (Strategy. "eager" eager-check))

; Semi-Eager Strategy
(defn semi-check
  [contract dterm blame]
  (delay (contract (extract dterm) blame)))

(def semi
  (Strategy. "semi-eager" semi-check))

; Promise-based Verification
(defn prom-check
  [contract dterm blame]
  (future (contract (extract dterm) blame)))

(def prom  (strategy. "promise" prom-check))

; Concurrency-based Verification
(defn conc-check
  [contract dterm blame]
  (let [v (extract dterm)]
    (do (a/go (contract v blame))
        v)))

(def conc (Strategy. "conc" conc-check))

; Spot-checking Verification
(defn spot-check
  [generator]
  (fn [contract dterm blame]
    (let [f (extract dterm)]
      (if (not (fn? f))
          (throw (Exception.  (str f " is not a function"))))
      (let [c (contract f blame)]
        (doall
          (map  (fn [x] (c x))  (gen/sample generator 20)))
        f)))))

(defn spot
  [g]
  (Strategy. "spot" (spot-check g)))
```

Figure 7.1. Monitoring strategy implementations in Clojure.

For example, consider applying natc to -1 under the **skip** strategy:

```
> (+ 5 (mon natc skip -1 blm))
=> 4
```

This strategy allows programmers to disable runtime verification errors (such as in the case of deployed systems), or allow programmers to use meta-strategies with effectful operations while forgoing contract verification itself.

Eager Verification. Our **eager** verification strategy in PISCES follows our definition in Chapter 5, suspending the user evaluator, completely verifying the contract, and continuing with the result:

```
> (+ 5 (mon natc eager -1 blm))
=> Contract Violation: '-1' violated 'natc'
   Blame: ...
```

To implement **eager**, we define the procedure eager-check to perform eager verification, applying the input contract as a procedure to the (evaluated) monitored term and the blame information, allowing it to proceed with verification (eschewing process creation to minimize overhead). Then, as with **skip**, we use eager-check to define **eager** as an instance of the Strategy record.

Semi-eager Verification. As with λ^{π}_{cs}, we introduce semi-eager verification in PISCES as **semi**, allowing programmers to postpone contract verification until the program requires the verification result. To reproduce semi-eager verification in PISCES, we define a semi-check procedure which applies delay to the verification expression "(contract (**extract** dterm) blame)", yielding a delayed expression. When the program forces the delayed expression (via **extract**), the evaluator proceeds with contract verification, returning the verification result to the user portion of the program:

```
> (mon natc semi 5 blm)
=> <delay ... (natc 5) ...>

> (let [x (mon natc semi -1 blm)]
    (+ (fact 5) (extract x)))
=> Contract Violation: '-1' violated 'natc'
   Blame: ...
```

In the first example, we see that the verification expression yields a delayed form. In the second example, invoking **extract** on x proceeds with verification, signaling a contract violation (since -1 is not a natural number).

Promise-based Verification. In order to recreate promise-style verification in PISCES, we borrow Clojure's built-in parallelism operation future, which uses the JVM's existing thread model to perform parallel evaluation of a given term [51]. To this end, our prom−check implementation wraps future around the underlying verification expression, creating a new process to perform evaluation while returning a promise-style future to the user program. As with semi, the user program may retrieve this verification result via the extract operator, signaling any violation errors at that time.

```
> (mon natc prom 5 blm)
=> <future ... (natc 5) ...>

> (let [x (mon natc prom -1 blm)]
    (+ (fact 5) (extract x)))
=> Contract Violation: `-1` violated `natc`
   Blame: ...
```

As with to semi, our first example produces a computational future form and our second example signals a contract violation when extracting x.

Concurrent Verification. Our next verification strategy is conc, which provides concurrent verification behavior in the style of the "best-effort" verification mechanisms from Chapter 5, using Clojure's core.async library to create concurrent verification processes that are discarded when the user program terminates. To define this behavior, we import the core.async library as "a", and use a/go (i.e., go as defined in core.async) to create a new process for verification. Then, to define concurrent verification, we extract the dterm expression, binding it to v, and then proceed to create a new, concurrent process with the appropriate verification expression before returning the evaluated term to the initiating process. We also observe that, in Clojure, the monitoring process will signal a contract violation but will not stop other processes from running:

```
> (let [x (mon natc conc -1 blm)]
    (+ x x))
=> Contract Violation: `-1` violated `natc`
   Blame: ...
   -2
; or => -2 without an error
```

In this example, the program will yield -2 as a result, but may also signal the contract violation to the user. This behavior, then, creates the "soft" verification mechanisms described in Chapter 5.

Spot-Checking Verification. Finally, we implement a spot-checking verification mechanism for functions, utilizing Clojure's *input generators* to verify the function's pre- and post-condition contracts hold for some number of generated inputs [30, 33]. To develop this verification technique in PISCES, we define the verification strategy spot using Clojure's test.check generator library (imported as gen). Then, to encode spot-checking verification, we define spot-check, which takes a generator as input and produces a verifier that, when evaluated, takes 20 elements from the generator and applies the contracted function to each input. If the contract produces a violation, the spot-checker reports it to the user process; otherwise, the verifier returns the original function as the contract result.

Unlike our previous strategies, we do not define a single Strategy record using spot-check. Instead, we define a spot procedure that takes an input generator and creates a new Strategy instance for it. This hints at the extensible nature of PISCES: each invocation of spot develops a bespoke verification strategy for the given generator, allowing users to store and reuse these strategies.

For example, using spot and Clojure's test.check natural number generator as gen/nat, we may define and use a spot-checking strategy to ensure a procedure works over natural numbers:

```
(def spot-check-nat (spot gen/nat))

> (mon nat-natc spot-check-nat (fn [x] (+ x 1)) blm)
=> <... function ...>

> (mon nat-natc spot-check-nat (fn [x] (- x 10)) blm)
=> Contract Violation: output `-8` violated `natc`
```

In these examples, we reuse the bespoke spot-check−nat to spot-check that two procedures take and return natural numbers, detecting a violation in the second case.

Overall, this approach to defining new strategies allows programmers to write customized verification strategies, extending PISCES to fit their program's requirements.

7.2.4. *First-class Strategy Verification.*

As with λ_{cs}^{π}, our PISCES library treats strategies as first-class values and, as such, programmers can use the same multi-strategy verification approaches as we saw in Chapter 5. For example, consider a recursive definition of factorial that extracts its input:

```
(defn fact
  [x]
  (let [n (extract x)]
    (if (zero? n) 1 (* n (fact (- n 1))))))

> (+ (fact (mon natc semi 5 blm)) (fact 5))
=> 240
```

Using func, we can define a strategy-parameterized contract as:

```
(defn natfunc [strat] (func natc strat natc eager))
```

We parameterize this contract by its pre-condition subcontract, dictating how to enforce natc on the contracted function's input, fixing the post-condition strategy. Using this variance, we can choose multiple verification behaviors:

```
(def fact-ee (mon (natfunc eager) eager fact blm))

> (fact-ee 5)
=> 120
```

```
(def fact-se (mon (natfunc semi) eager fact blm))

> (fact-se 5)
=> 120
```

```
(def fact-spot (mon (natfunc eager) spot-check-nat fact blm))

> (fact-spot 5)
=> 120
```

In our first example, we use the eager strategy; in the second, we check the pre-condition via semi-eager verification; in our final example, we once again use spot-check−nat to spot-check that the underlying fact procedure works over natural numbers, using eager to ensure the pre- and post-conditions hold for each random test.

7.3. Meta-strategies in Pisces

The last component of our Pisces implementation focuses on meta-strategies, developing with, comm, random, and memo from Chapter 6 in Pisces. We give each of these definitions in Figure 7.2 in terms of the Metastrat record constructor, once again following the pattern of defining a checking procedure to use for a Metastrat instance.

The with Meta-strategy. Following Chapter 6, the with meta-strategy takes a sub-strategy and a procedure, applies the procedure to the contracted result (using *indy*-style blame), and returns the contracted term.

The comm Meta-strategy. As before, the comm meta-strategy is a special-case variation of the with meta-strategy that provides inter-process communication. After contract verification, the meta-strategy applies f to the verification result, writes this new value across the appropriate channel (using a/put! from core.asnyc), and, finally, returns the contracted value to the initiating process.

The random Meta-strategy. The random meta-strategy provides probabilistic checking as per random from Chapter 6. We implement random in Pisces by generating a random number and testing it against the given rate: if the random number is less than the rate, we proceed with verification using the given sub-strategy; if not, we return the (forced) monitored expression.

```
; With-Operator Verification
(defn with-check
  [fun]
  (fn [contract sub-strat dterm blame]
    (let [val (extract dterm)
          res (mon contract sub-strat val blame)]
      (do (fun (mon contract sub-strat val (indy-blame blame)))
          res))))

(defn with
   [fun strat]
   (Metastrat. "with" (with-check fun) strat))

; Communicating Verification
(defn comm-check
  [channel fun]
  (fn [contract sub-strat dterm blame]
    (let [val (extract dterm)
          res (mon contract sub-strat val blame)
          indyb (indy-blame blame)]
    (do (a/put! channel
                (fun (mon contract sub-strat val indyb)))
        res))))

(defn comm
   [chan fun strat]
   (Metastrat. "comm" (comm-check chan fun) strat))

; Random Verification
(defn random-check
  [rate]
  (fn [contract sub-strat dterm blame]
    (if (< (rand) rate)
        (mon contract sub-strat (extract dterm) blame)
        (extract dterm))))

(defn random
  [rate strat]
  (Metastrat. "rand" (random-check rate) strat))

; Memoizing Verification
(defn memo-check
  [contract sub-strat dterm blame]
    (mon (memoize contract) sub-strat (extract dterm) blame))

(defn memo
  [strat]
  (Metastrat. "memo" memo-check strat))
```

Figure 7.2. Monitoring meta-strategy implementations in Clojure.

As expected, this behavior is sufficient to recreate our binary-search tree insertion example Chapter 6, using random to verify the contract one-tenth of the time:

```
(def bst-ins
  (mon (func (random 0.1 eager) (bstc (random 0.1 eager)) eager natc
                  eager anyc)
        eager
        bst-insert
        blm))
```

In this example, we verify the function's post-condition, bstc eager, via (random 0.1 eager), dictating that verification should occur one-tenth of the time.

The memo Meta-strategy. The memo meta-strategy, analogous to memo in Chapter 6, memoizes contracts using Clojure's built-in memoize operation (which takes a procedure and automatically returns a memoized version). Our implementation applies memoize to the contract before proceeding with verification, allowing us to cache the verification results.

As with random, this is particularly effective for reducing performance overhead for contracted functions. For example, the following code will check each input and output to fact exactly once, memoizing the contract results:

```
(def fact-m
  (mon (func (memo eager) natc
                  (memo eager) natc)
        eager
        fact
        blm))
```

In this example, we memoize the pre- and post-conditions, and, as a result, subsequent invocations of fact−m will forgo re-verifying the same contract on the same input:

```
> (fact-m 5)
=> 120

;; Will not verify 5 or 120 are natural numbers
> (fact-m 5)
=> 120
```

7.4. Case Studies: Advanced Examples using Pisces

So far, we have introduced Pisces, a library that recreates the key runtime verification features of λ_{cs}^{π} in the Clojure programming language. To conclude our presentation of this implementation, we now explore three case studies using Pisces:

(1) we revisit our recursive binary-search tree contract parameterized by its recursive and node-level enforcement, exploring how strategies impact performance (§7.4.1);

(2) we implement our transition-based meta-strategies from Chapter 6 to recreate state machine runtime verification in PISCES (§7.4.2);

(3) and, finally, we use our process-based approach to algebraic effects from Section 6.6 to develop a system for function profiling (§7.4.3).

7.4.1. *Multi-Strategy Monitors*

Our first case study revisits our binary-search tree contract from Section 5.8.2. In that section, we demonstrated that the multi-strategy approach to contract verification allows users to flexibly reuse contracts to yield multiple verification behaviors. We recreate this example in PISCES, including the relevant combinator and contract. In Figure 7.3, we define a record to represent a tree node, a dependent tree contract combinator `treedc`, and, finally, define `bstc`, a tree contract parameterized by two strategies: `rec−strat`, which determines how to recursively enforce `bstc` on sub-trees, and `value−strat`, which determines how to enforce the *value* contract on each value in the tree.

As before, we may use this contract in various ways by changing its strategy parameters, varying these parameters to alter the contract's performance characteristics and guarantees. Using **eager** for both strategies, for example, will *completely* traverse the tree, checking each node (an $O(n)$ operation):

```
> (mon (bstc eager eager) eager tree blm)
=> <tree ...>
```

As before, this $O(n)$ traversal may be unsuitable as a contract for an $O(\log n)$ binary-search tree insert function, but we may recover our asymptotic behavior with the **semi** strategy:

```
> (mon (bstc semi eager) eager tree blm)
=> <tree ...>
```

As described in our previous discussions, we may also use `prom` at the top level verification form, creating a computational future that performs a concurrent $O(n)$ traversal and contract verification on the tree:

```
> (mon (bstc eager eager) prom tree blm)
=> <future ...>
```

Further, our two-strategy parameterization allows us to change the contract behavior along two axes by allowing us to control the recursive verification strategy *and* how to verify each node's value

```
(defrecord BinTree [val left right])

(defn treedc
  "Dependent tree contract combinator"
  [cleaf sleaf cval sval cleft cright srec]
  (fn  [tree blame]
    (if (nil? tree)
        (mon cleaf sleaf tree blame)
        (let [v (:val tree)]
          (BinTree.
            (mon cval       sval v              blame)
            (mon (cleft v)  srec (:left tree)   blame)
            (mon (cright v) srec (:right tree)  blame))))))

(defn bst-range [lo hi] (fn [v] (and (>= v lo) (<= v hi))))

(defn bstc
  "Binary search tree contract"
  [rec-strat value-strat]
  (letfn [(bstc [lo hi]
            (treedc
              anyc eager
              (predc (bst-range lo hi)) value-strat
              (fn [v] (bstc lo (extract v)))
              (fn [v] (bstc (extract v) hi))
              rec-strat))]
    (bstc Integer/MIN_VALUE Integer/MAX_VALUE)))
```

Figure 7.3. A strategy-parameterized contract for ensuring a binary tree is a binary-search tree, using the dependent tree contract combinator treedc. We assume leaves are nil values.

contract. For example, consider enforcing bstc with semi as the recursive strategy and eager as the node-value strategy on tree−wrong, which is *not* a binary-search tree:

```
(def tree-wrong
  (BinTree. 5
    (BinTree. 6 nil nil)
    (BinTree. 7 nil nil)))

> (extract
    (:left
      (extract (mon (bstc semi eager) semi tree-wrong blm))))
=> Contract Violation: ‘6‘ violated ‘bst-range‘
```

In this invocation, we extract the verification result before retrieving the left subtree, verifying the contract (bst−range Integer/MIN_VALUE 5) on 6 and signaling an error. If, however, we use semi for the node value contracts, the resultant structure further delays the node's value contract:

```
> (extract (:left (extract (mon (bstc semi semi) semi tree-wrong blm))))
=> <delay ...>

> (let [tree (extract (mon (bstc semi semi) semi tree-wrong blm))
        node (extract (:left tree))])
    (extract (:val node))
=> Contract Violation: '6' violated 'bst-range'
```

Instead of enforcing each value contract when extracting the node itself, this monitoring structure returns a BinTree node that contains a delayed value field, requiring the program to further extract the value to verify the contract.

This contract provides precise control over verification at each level, demonstrating the flexible nature of our first-class strategies, allowing programmers using Pisces (and λ^{π}_{cs}) to choose the best-fit strategy in each situation without redefining the contract in each case.

7.4.2. *State Machine Runtime Verification as User-Defined Meta-strategies*

Our second case study revisits the state-based transition system presented in Chapter 6. We recreate this behavior in Pisces to demonstrate the extensible nature of our Pisces library: all of the code in this case study is part of a *user program* in Clojure, separate from our Pisces library. We proceed by defining a pair of transition meta-strategies, using them to build our previous hasNext and next monitors for Java iterators, and using these monitors to verify a program always invokes hasNext on such iterators before retrieving the next value.

Transition Meta-Strategy System. We start with the transition and transition−as meta-strategies, given in Figure 7.4. We use a Clojure reference cell (created with ref) to track the current machine state, defining the transition and transition−as meta-strategies as operations over this reference as follows:

- The transition meta-strategy takes the strategy state (i.e., a reference to the current state), a from-state indicating the transition source, a to-state transition target, and a substrat sub-strategy indicating how to verify the contract.
- The transition−as meta-strategy takes the strategy state and a procedure transition-fn parameterized by the current strategy state and the contracted value. If the transition-fn returns the state :error, the meta-strategy signals an error; otherwise, we update the state with the resultant value and return the verification result to the user portion of the program.

These encodings directly correspond to Definition 6.5, utilizing Clojure's dosync operator to ensure atomic reference updates.

State-Transition Contracts. Using this transition meta-strategy system, we now develop a pair of contracts to verify a program correctly invokes hasNext on an iterator (and it returns true) before retrieving the next value, adhering to the state machine in Figure 6.1.

We present this implementation in Figure 7.5, recreating our λ^{π}_{cs} definitions using transition and transition−as:

- We model .next's behavior using transition — if the current state is :some, we transition to the :unknown state, and, if the current state is not :some when we call .next on the iterator, we produce an error.
- We model .hasNext's behavior based on its result (that is, as a flow-dependent transition based on .hasNext's output): after we run .hasNext, we use transition−as to inspect .hasNext's result, transitioning to the appropriate state.

Verifying State-Machine Behavior. Using the contracted nextm and hasNextm, we can now ensure our programs correctly check that a given iterator has additional values before attempting to retrieve them:

```
;; Example 1
> (let [iter (.iterator (.keySet (java.lang.System/getProperties)))]
    (while (hasNextm iter) (println (nextm iter))))
=> ...

;; Example 2
> (let [iter (.iterator (.keySet (java.lang.System/getProperties)))]
    (println (nextm iter)))
=> Program performed invalid operation:
   Current state: (:unknown)  Transition: :some -> :unknown
```

In the first example, the program correctly calls hasNextm before retrieving each element of the iterator, terminating as expected. In the second example, however, the program invokes next on the iterator before invoking hasNextm, an invalid state transition, and the transition verification mechanism signals an error.

Summary. Overall, this case study illustrates the extensible nature of PISCES: working from the PISCES record definitions, we have developed a new series of meta-strategies, creating additional, practical runtime verification mechanisms as natural extensions to the existing PISCES library.

```
(defn make-contract-state
  [start-state]
  (ref (list start-state)))

(defn in?
  [coll elm]
  (some #(= elm %) coll))

(defn transition-check
  [state-ref from-state to-state]
  (fn [contract sub-strat dterm blame]
    (let [res (mon contract sub-strat (extract dterm) blame)]
      (if (in? (deref state-ref) from-state)
          (dosync (ref-set state-ref
                           (if (list? to-state) to-state (list to-state)))
                  res)
          (throw (Exception. (str "Program performed invalid transition:\n"
                                  "  Current state: " (deref state-ref)
                                  "  Transition: " from-state
                                  " -> " to-state "\n"))))))))

(defn transition
  [state-ref from-state to-state strat]
  (Metastrat. (str "transition" from-state to-state)
              (transition-check state-ref from-state to-state)
              strat))

(defn transition-as-check
  [state-ref transition-fn]
  (fn [contract sub-strat dterm blame]
    (let [res (mon contract sub-strat (extract dterm) blame)
          to-state (transition-fn (deref state-ref) res)]
      (if (not (= to-state :error))
          (dosync (ref-set state-ref
                           (if (list? to-state) to-state (list to-state)))
                  res)
          (throw (Exception. (str "Program performed invalid operation:\n"
                                  "  Current state: "
                                  (deref state-ref) "\n"))))))))

(defn transition-as
  [state-ref transition-fn strat]
  (Metastrat. (str "transition-as")
              (transition-as-check state-ref transition-fn)
              strat))
```

Figure 7.4. State transition verification as meta-strategies. We have omitted the Exception messages due to length.

```
(def iter-state (ref (list :unknown)))

(def nextm
     (mon (func eager anyc
                (transition iter-state
                            :some
                            :unknown eager)
                anyc)
          eager
          #(.next %)
          blm))

(defn hasNextTrans
  [con-result cur-state]
  (if (extract con-result) :some :none))

(def hasNextm
     (mon (func eager anyc
                (transition-as iter-state
                               hasNextTrans
                               eager)
                anyc)
          eager
          #(.hasNext %)
          blm))
```

Figure 7.5. A flow-dependent finite-state machine PISCES implementation.

7.4.3. *Function Timing via Concurrent Logging*

Our final case studes follows the effects-as-process methodology we present in Section 6.6, using the **comm** strategy and a concurrent process that acts as a “timing report manager” to develop a series of operations that use our verification framework to achieve function time profiling. While this timer performs a similar operation to the timer implementation in §6.1, this approach follows our effect-process approach from Chapter 6, allowing us to use session-style effect operations that interact with—and supplement—contract verification.

We present this implementation in Figure 7.6, where we import Clojure's `core.async` library as `a`. Our definition proceeds as our state-effect example in Chapter 6, wherein we define an effect procedure and proceed with developing contracts that communicate with this procedure via meta-strategies.

We begin by defining `timer-task` to serve as the main loop of the concurrent timer process. This procedure takes two communication channel arguments, `in-chan` and `out-chan`, for receiving input and sending output (respectively), and `timer-info`, a list of timing information reported so far. As with our state manager in the previous chapter, each loop of the report manager reads in an *action*

```
(defn timer-task
  [in-chan out-chan timer-info]
  (let [action (a/<!! in-chan)]
      (cond
        (= action :result)
          (let [res (filter #(= ( first %) :time) timer-info)]
            (a/>!! out-chan
                   (/ (reduce + (map second res))
                      (float (count res)))))
        (and (list? action) (= (first action) :pre))
          (timer-task
            in-chan out-chan
            (cons (list :pre (second action)) timer-info))
        (and (list? action) (= (first action) :post))
          (timer-task
            in-chan out-chan
            (cons (list :time
                        (- (second action)
                           (second (first timer-info))))
                  (rest timer-info)))
        :else
          (timer-task in-chan out-chan timer-info))))

(defn start-timer-task
  [in-chan out-chan]
  (a/go (timer-task in-chan out-chan (list))))

(defn timer-func
  [in-chan out-chan]
  (do
    (start-timer-task in-chan out-chan)
    (func (comm in-chan
                (fn [_] (list :pre  (cur-time)))
                eager)
          anyc
          (comm in-chan
                (fn [_] (list :post (cur-time)))
                eager)
          anyc)))
```

Figure 7.6. Function timing with a concurrent process.

across the communication channel in–chan and then performs the appropriate behavior indicated by this action[5]:

- the "result" action computes the average performance time from the *timings* list and writes it across out–chan, recurring appropriately;

[5]We observe that this task is not thread-safe insofar as any number of processes may read from the same channel at once, and thus one process may receive another process's result. Modifying our timer task to provide thread-safety are beyond the scope of our presentation, but we refer the curious reader to the calculus presented by Orchard and Yoshida [70] for a solution.

- the "startTime" action accepts a new start time indicating the start time of a timing action, recurring with the start time and the flag *active*, indicating the timing is active;
- the "endTime" action accepts a new end time, recurring with a false *active* variable and updated timing list;
- the "endTask" action terminates the timer process with the value unit;
- and the task ignores any other action, recurring.

At each recursive invocation of timer−task, the report manager reads and performs the appropriate action (e.g., storing a function start time or end time, end time, retrieving the overall timing result, or terminating the timing process) and recurs on itself, maintaining a list of execution times between recursions (by storing each pre-condition enforcement time-stamp and computing its difference from the post-condition enforcement time-stamp). We also provide start−timer−task for ease of use in creating the new process.

In addition to this report manager, we also define the timer−func function contract which starts a timing task and uses comm to report the current (tagged) system timestamp to the timing process at each pre- and post-condition contract verification site.

Using these pieces in conjunction, we can time a factorial procedure as:

```
> (let [in (a/chan)
        out (a/chan)
        f (mon (timer-func in out) eager fact blm)]
    (f 15)
    (f 5)
    (a/put! in :result)
    (a/<!! out))
=> 104.33
```

In this usage example, we define input and output channels in and out (respectively); monitor the fact procedure with timer−func (which initiates a time-task via start−timer−task); invoke the monitored procedure a number of times; and, finally, retrieve the timing result. This program yields 104.33, indicating the average runtime of our two invocations.

This case study ultimately demonstrates the viability of our general approach, exhibiting how concurrent effect-style processes can collaborate with the contract verification system to perform general runtime inspection (and verification) in a real-world setting.

Summary—The PISCES *Library*

In this chapter, we developed the PISCES library, demonstrating how a modern programming language provides the necessary features to implement our multi-strategy and meta-strategy runtime verification approach whole-cloth in terms of existing language mechanisms. This insight suggests that our approach provides direct and actionable results for anyone hoping to use this style of runtime framework for anything from contract verification to function profiling, providing programmers with the tools to reason about their programs during execution.

CHAPTER 8

Related Works

SYNOPSIS

In this chapter, we discuss related works in the field of runtime verification and, in particular, software contract verification (§8.1), software contract metatheory (§8.2), contract verification blame (§8.3), contract system implementations (§8.4), and other runtime verification systems (§8.5).

The first language for specification and verification was Gypsy, introduced by Ambler et al. [5]. Much other work has gone into profiling and inspecting programs at runtime, leading to Meyer [66] introducing *software contracts* with the language Eiffel, along with our moderns notions of pre- and post-condition contract enforcement. Findler and Felleisen [36] brought software contracts into functional languages, where they have since thrived as runtime verification tools. Our work builds on this concept, generalizing software contracts and abstracting their verification mechanisms to support multiple verification strategies along with generic runtime verification and inspection operations.

8.1. Software Contracts

After Findler and Felleisen [36] brought software contracts into functional programming, a cottage industry of software contract verification techniques sprung up [15, 22, 26, 29, 37, 52, 68]. Swords et al. [80] provides a framework for expressing their interactions, and we extend and revise that work. Our presentation in Chapters 5-6 discuss multiple such strategies, accounting for each either in explicit detail or via implementation sketches.

In the rest of this section, we discuss four additional related portions of the literature: other surveys of contract verification strategies, alternative software contract verification techniques (including static enforcement), other systems that implement software contract verification with processes, and systems to provide contracts in lazy (e.g., call-by-name and call-by-need) programming languages.

8.1.1. *Surveys of Contract Verification*

Degen et al. [22] present descriptions of eager, semi-eager, and lazy software contract verification systems, characterizing each system's behavior in a call-by-need programming language via four properties (meaning reflection, meaning preservation, faithfulness, and idempotence). They ultimately conclude that "faithfulness is better than laziness." Dimoulas and Felleisen [25] go further, classifying different approaches through observational equivalence, based on when and how verification proceeds (with axes of static versus run-time and tight, loose, or shy-loose respectively). They also introduce the notion of a *shy* contract, which is analogous to the lazy verification presented by Degen et al. [22]: it is only allowed to inspect parts of the program the user program evaluates. Degen et al. [24] revisit different contract verification systems in call-by-need languages, evaluating them for completeness and meaning preservation, classifying a number of contract systems [16, 22, 36, 52, 86] by these properties. Conversely, our work follows Swords et al. [80]: instead of classifying verification strategies via secondary properties, we have presented a semantic-based comparison by directly encoding each verification strategy in λ^{π}_{cs}.

8.1.2. *Alternative Contract Enforcement Mechanisms*

Our presentation uses runtime enforcement of contracts that all exist at the value level. This is not, however, the only enforcement mechanism available. Ou et al. [71], Flanagan [38], and Greenberg et al. [44] each present a model of contract usage similar to refinement types [40] tracking these contracts at the type level and enforcing (and accruing) them as values flow through these types. The work by Greenberg et al. [44] in particular has recently been the subject of multiple extensions, including efficiency [43] and data-types [76]. While manifest contracts are closely related to standard contracts (and Racket's chaperone and impersonator system [79]), our work treats contracts as values instead of type-level terms.

Xu et al. [86] and Nguyen et al. [69] both present static verification models for contract verification, using static analyses to determine which contracts hold prior to running the program. While related, this style of static checking is orthogonal to our presentation here: we do not attempt to subsume static checking with our framework. Clojure's `core.spec` library [49] also supports enforcing contracts in different ways. Unlike our multi-strategy approach, however, the `core.spec` library allows users to determine which programming phase to check the provided specification (or "spec") at: the spec written for the program allows programmers to both instrument the runtime (via software contracts) *and* to generate sample data to probabilistically check functions, etc., without running the program itself. Combining our multi-strategy approach with multi-phase checking remains as future work.

Finally, Shinnar [77] presents contract assertions in the context of concurrency and software-transactional memory, focusing on effectful contracts interacting with Software Transactional Memory in Haskell, using Haskell's monadic effect system. They introduce the idea of *delimited checkpoints* for STM, allow the runtime to observe (and roll back) memory changes, supporting specification contracts alongside "framing" contracts, which ensure programs do not exhibit specific behaviors, and separation contracts, which combine these to provide separation logic-style behavior. The roll-back capabilities also allow the program to undo effects in the event of contract violations. Integrating such a mechanism into λ_{cs}^{π} may allow us to address the effect-related shortcomings of promise-based and concurrent contract verification, and this integration remains as future work.

8.1.3. *Contracts as Processes*

Dimoulas et al. [26] and Disney et al. [31] each explore the notion of contracts using message passing, and, further, the runtime verification literature contains a number of additional examples of using messages to verify program properties [7, 13, 46]. In each case, however, the work has different design goals from Swords et al. [80] and our work. Dimoulas et al. [26] model and explore concurrent contracts; Disney et al. [31] precisely model and explain temporal contracts with non-interference and trace completeness; our work models and explores how models contracts as secondary evaluators and varying the interactions with these evaluators allows us to encode multiple verification strategies in the same framework.

Dimoulas et al. [26] introduce concurrent contracts via *future contracts*, which send terms and their contracts, as messages, to a secondary evaluator for verification. Dimoulas et al. [26] observe that this approach is familiar in the broader runtime verification community. We extend the idea of a secondary contract evaluator to creating a separate evaluator for each individual contract, and vary this communication structure to provide programmers with myriad verification mechanisms that they may choose on a per-contract basis.

Disney et al. [31] utilize multiple contract verification processes to monitor and validate communications as (quasi-)recursive, long-running middle-man processes that mediate module interactions. In their system, contracts forward messages between these modules, inspecting constants and starting sub-monitors for structural contracts. This process separation uses isolation to ensure, a priori, contract non-interference. Our approach and focus differs in that we do not model "client" and "server" modules, instead presenting a user "program" interacting with contract processes directly; we do not require (or desire) non-interference in our contracts, allowing them to inspect modify, and otherwise manipulate contracted terms (e.g., by installing wrappers); and, finally, the focus of our work is allowing contracts to vary their method of verification the program on a per-contract basis and exploring these interaction patterns for programmer utility and semantic breadth.

Even so, these works share a notion of contracts as processes, suggesting that we may encode the temporal contracts outlined by Disney et al. [31] in λ^{π}_{cs}, and, similarly, that we might modify their calculus to support λ^{π}_{cs}-style multi-strategy verification (via adding delay operations and modifying their *guard* procedure) at the cost of some of their guarantees (such as non-interference).

8.1.4. *Contracts in Lazy Languages*

Chitil et al. [17] brought runtime verification to Haskell as *assertions*, which eschews blame and uses assertions closer to comprehensions than the contract structures we have seen here. Hinze et al. [52] introduced full contracts with blame and a strict assertion operation for Haskell, inducing our current notion of "semi-eager" verification. Chitil [15] also encode semi-eager verification in the Haskell language following by almost direct implementation of Findler and Felleisen [36] into a call-by-name language.

Degen et al. [22] introduce and compare semi-eager and eager verification in Haskell. They present a semi-eager contract system as a direct encoding of the contract verification presented by Findler and Felleisen [36], relying on Haskell's underlying semantics to add the appropriate laziness. Their eager implementation, conversely, utilizes Haskell's `seq` operator to subvert Haskell's evaluator into forcing eagerly-contracted expressions. Degen et al. [23] use a monadic meta-theory to describe semi-eager and lazy behavior in Haskell.

Reformulating our own contract interactions in a lazy variation of λ^{π}_{cs} follows similarly, but presents a unique challenge in the form of intermixing strategies: we must, at some point, *stop* the strictness behavior to avoid over-evaluating lazy subcontracts. This ultimately requires both forcing and "unforcing" operations in order to top strict evaluator positions from over-evaluation. For example, consider evaluating

$$\textbf{mon}\ (\mathsf{pair/c}\ \mathsf{nat/c}\ \textbf{semi}\ \mathsf{nat/c}\ \textbf{semi})\ \textbf{eager}\ (5, \text{-}1)B$$

The two interior contract must "unforce" the forcing operation of the outer semi if they are to ensure delayed verification behavior, mirroring our delay/force interactions in λ^{π}_{cs}.

8.2. Contract Metatheory

Blume and McAllester [11] present the first exploration of contract metatheory, introducing and exploring contract safety for eager contracts, and Findler and Blume [35] describe function contracts as *pairs of projections*. We diverge from contracts as pairs of projections, following Findler's later position that this view was too rigid [34].

Guha et al. [45] present the problem of parametric, polymorphic contracts as first-order values, and Ahmed et al. [3] revisit this problem and prove parametricity. We appeal to this notion in our type system, relying on polymorphic contract combinators.

Finally, Keil and Thiemann [57] provide denotation semantics for eager contract verification. While the system here is, in some sense, a metatheory for verification, we do not claim strong mathematical properties about any individual strategy (though it may be possible to recover them through further embedding proofs).

8.3. Complete Blame and Contract Monitoring

Multiple approaches to runtime verification blame assignment have been proposed [2, 27, 28, 44, 84], with special attention given to function contracts (and, in particular, dependent function contracts). Since our contract combinators are library functions, we can provide any blame assignment approach necessary (including *indy* semantics [27]).

As for correct verification, Dimoulas et al. [28] introduce the notion of *complete monitoring*, a fundamental correctness criterion for contract systems that generalizes correct blame assignments by ensuring they monitor each value that moves between components. Unfortunately, proving this property for λ^{π}_{cs} is particularly challenging for a number of reasons. First, their definition relies on multiple modules. We would need to first extend λ^{π}_{cs} with a module model (likely as interacting processes, following Disney et al. [31]). Second, Dimoulas et al. [28] use an ownership-and-obligation model to define complete monitoring, and we would need to replicate this in a multi-process setting where we explicitly model each contract verifier as a separate entity. In particular, we would need to address how value ownership changes when transferring value to monitors and, further, when transferring results (e.g., when values flow through multiple monitors toward the answer, such as for pairs, or forcing a delayed reference after communicating it between processes). Finally, complete monitoring is tells us little about some strategies, including **concurrent**: complete monitoring only requires that the program terminates with a value, diverges, or correctly detects a contract violation, and, if we verify every contract under **concurrent**, a λ^{π}_{cs} scheduler may always complete the user process without checking any outstanding contracts. This suggests that, in the context of "best-effort" contracts, complete monitoring is not a sufficient requirement.

8.4. Contract Implementations

Our own implementation deviates from our semantic framework in two ways: first, that we avoid processes and parallelism whenever possible to reduce contract enforcement overhead and, second that we introduce the notion of meta-strategies.

Racket's contract system provides multiple options when enforcing contracts by accepting additional, optional flags to its **check** form [37]. Our λ^{π}_{cs} and PISCES frameworks use a more direct approach, asking programmers to select strategies as primary, first-order directives.

Clojure's `core.spec` system uses the notion of "specifications" with different enforcement methods, including compile-time analysis and runtime checking. Unlike PISCES, where a function may be enforced at each use or spot-checked, higher-order function inputs are *always* spot-checked in `core.spec`. Combining the multi- and meta-strategy approach with `core.spec` remains as future work.

Cartwright and Felleisen [12] describe using communication to model effects, and the recent rise of algebraic effect models [10, 56, 61, 64] and their close connection to effects-as-processes [70] have opened the door for managing algebraic effects as process interactions, which is a natural fit with our process-based contract verification framework, and, ultimately our runtime instrumentation examples. As far as we know, this is the first work to leverage this connection in the context of contract verification.

8.5. Runtime Verification and Instrumentation

As stated at the start of this chapter, the first language for specification and verification was Gypsy, introduced by Ambler et al. [5]. Since then, runtime verification and instrumentation has been an active area of research, facilitating the inspection of program behavior to detect runtime issues including race conditions, exceptions, resource leaks, and uninitialized memory, and, further, to produce time and memory performance information for program executions.

Beyond Meyer's "Design by Contract" approach, this field of research had yielded been myriad proposals for verification systems, including generic runtime verification systems [46] and monitoring-oriented programming [13, 14, 54].

8.5.1. *Generic Runtime Verification Systems*

There has been myriad research projects in the field of runtime verification [8, 47, 78] yielding countless runtime verification systems [4, 9, 14, 21, 42, 46, 59, 63]. Moore et al. [68] outline the primary differences between this style of runtime verification and software contract systems as follows:

> Contract systems live inside of programming languages while runtime verification tools live outside. In other words, higher-order contract systems are extensions of programming languages, and run-time verification systems are elements of a language's external tool chain.

Our work presented here maintains this distinction, including contracts and strategies as first-class entities in the user program. In addition to living inside of the programming language itself, our contract system also provides programmers with the ability to write contracts that can appear to exist outside of it, i.e., in isolated, interacting processes. Our effectful tree fullness case study in Chapter 7 exhibits this secondary behavior, allowing us to replicate a sort of "external" verifier as a secondary process that interacts with the program through the **mon** form, allowing us to effectively emulate runtime verification tools that exist as part of an external tool chain.

8.5.2. *Monitoring-Oriented Programming*

Monitoring-oriented programming (MOP) [13, 14] is a form of specialized aspect-oriented programming [58], wherein the aspects are (formal) behavioral specifications, providing programmers with facilities to define program specifications that are automatically integrated into the underlying program. Our contract-based system takes a different tactic, requiring that programmers write individual contracts to verify the program's behavior. To summarize , monitoring-oriented programming (and aspect-oriented programming in general) reasons about *code*, whereas λ_{cs}^{π} reasons about *values*.

Even so, we demonstrate that λ_{cs}^{π} has the potential to replicate this MOP behavior similar approach to the first property in Chapter 6, defining the transition-based strategies to manage a formal state machine specification, and Dimoulas et al. [30] further demonstrate multiple applications of design-by-contract systems that use formal specification languages to define global system properties.

Chen and Roşu [13] also identify that contract systems always performing verification in the same process at the user program, whereas MOP may perform verification in a separate process, but Dimoulas et al. [26], Disney et al. [31], and our own λ_{cs}^{π} framework each address this divergence.

The third primary difference between standard contract verification and MOP is that MOP allows programmers to define violation handlers that may perform additional operations (such as correcting the issue) instead of raising errors. Since we treat contracts as black-box constructs, it is conceivable that we may construct additional contract combinators that perform these error-correcting computations instead of raising violations, e.g.:

$$\mathsf{pred/handle} := \lambda\ \mathit{pred}\ \mathit{handler}.\ \lambda\ x\ b.\ \mathsf{if}\ \mathit{pred}\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathit{handler}\ x\ b$$

Experimenting with these extended combinator forms to determine how much MOP behavior we can replicate in our λ_{cs}^{π} framework remains as future work.

CHAPTER 9

Summary & Future Work

Over the course of this dissertation, we have introduced and explored runtime verification as patterns of communication, starting with contract verification and growing our λ^{π}_{cs} framework to encompass generic runtime verification results. We have covered multiple aspect of runtime verification using λ^{π}_{cs} and our communication-centric account of verification, including exploring the fundamental nature of contract verification and resolving the existing "one-size-fits-none" problem often found in modern contract verification systems; ensuring global properties about data structures; verifying that programs adhere to behavioral specification (e.g., ensuring an iterator has a value before retrieving it); and even using our system for general function profiling.

Moreover, we have not exhaustively examined the possible strategies and meta-strategies in our newly-exposed design space—we are still far from taming the contract verification zoo. Even this early exploration, however, has resulted in multiple insights into the nature of runtime verification and the role it plays in programming, allowing us a natural view into verification through the lens of decoupled evaluation and flexible enforcement.

Ultimately, this multi-strategy, multi-process λ^{π}_{cs} framework supports my thesis:

> Runtime verification systems may be expressed as a collection of separate, concurrent processes that interact with the user program, and variations on verification systems may be encoded as variations on patterns of communication to provide programmers with general, practical runtime verification tools.

In addition to presenting a full account of the underlying mechanisms that drive contract enforcement in its myriad flavor, describing myriad verification strategies and meta-strategies in a single calculus, we have also provided proofs of embedding correctness and type safety; explored contracts and their interactions with session-style algebraic effects, using meta-strategies to account for contract-effect interactions; and outlined an implementation of our approach in Clojure. Our appendices also include a Haskell implementation of our calculus and a variant of our λ_{cs}^{π} calculus that elides force invocations on contract results.

Contracts and Effects. When we initially embarked upon this research, the goal was to quantify contract verification and its interaction with other effects. As Findler [34] observed in 2013, contracts with other effects provide programmers with immense utility for runtime inspection and verification. While we do not give a formal account of such effect interactions in this work, we lay the framework for understanding contracts as interactions with the main program in much the same way as one can view a reference cell, a tracing mechanism, or an error-reporting system as a separate entity that interacts with the underlying program. Indeed, in the time that we performed this research, researchers around the world began to account for other effects as such interactions [56, 61, 82] (including our own work, presented by Kiselyov et al. [60]), culminating in Orchard and Yoshida [70] demonstrating the clear connection between sessions and effects.

Thus, in some roundabout way, we have arrived where we intended all those years ago: we now have a reasoned, canonical mechanism to view contracts as additional, interacting processes just as other effects, and any formal treatment of effects in this way is amenable to supporting the framework presented here.

Future Works. The scope of this work is large enough that it may never be truly complete. Much of the strategy and meta-strategy design space is still unmapped, and one could wander it for years to come. Our implementation, too, could use improvements: a truly invisible chaperone-style system that eschews the need for **extract** would bring PISCES to every programmer, allowing them to precisely design and use our contract system in a modern language. Finally, the space for meta-theoretical contributions here is far from exhausted: proofs of complete blame, further correct strategy embeddings, and embedding λ_{cs}^{π} into effect-sensitive calculi are all future possibilities, with their own challenges and insights. Finally, another major potential for future work involves encoding our λ_{cs}^{π} system in a call-by-push-value calculus, giving a detailed treatment of each step of contract evaluation. In addition to providing a fine-grained behavioral account, this would also pave the way for adding such a framework to, e.g., the Frank programming language [64], which utilizes a call-by-push-value core calculus.

All told, there is still a life's work to be done for runtime verification metatheory.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition*. The MIT Press, 1996. ISBN 0262510871.

[2] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.

[3] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. In *Proceedings of the 22th International Conference on Functional Programming*, ICFP '17, 2017.

[4] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 345–364, New York, NY, USA, 2005. ACM.

[5] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Choen, C. G. Hoch, and R. E. Wells. Gypsy: a language for specification and implementation of verifiable programs. *SIGPLAN*, pages 1–10, 1977. URL `http://doi.acm.org/10.1145/800022.808306`. Proceedings of the ACM Conference on Language Design for Reliable Software.

[6] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, New York, NY, USA, 1995. ACM.

[7] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS '04. Springer-Verlag, 2005.

[8] Howard Barringer, Bernd Finkbeiner, Henny Sipma, and Yuri Gurevich. Runtime Verification (RV '05). Elsevier, 2005, 2005. ENTCS 144.

[9] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From eagle to ruler. In *Proceedings of the 7th International Conference on Runtime Verification*, RV'07, pages 111–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. arXiv:1203.1539 [cs.PL], 2012.

[11] Matthias Blume and David McAllester. A sound (and complete) model of contracts. In *Proceedings of the 9th International Conference on Functional Programming*, ICFP '04. ACM, 2004.

[12] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '94, London, UK, UK, 1994. Springer-Verlag.

[13] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22Nd Annual Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07. ACM, 2007.

[14] Feng Chen and Grigore Rou. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proceedings of the 3rd Run-time Verification*, RV '03, pages 108 – 127, 2003.

[15] Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th International Conference on Functional Programming*, ICFP '12. ACM, 2012.

[16] Olaf Chitil and Frank Huch. A pattern logic for prompt lazy assertions in haskell. In *Symposium on Implementation and Application of Functional Languages*, IFL '06. Springer, 2006.

[17] Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy assertions. In *Symposium on Implementation and Application of Functional Languages*, IFL '03. Springer-Verlag, 2003.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001. ISBN 0262032937.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition (MIT Press).* The MIT Press, 2009. ISBN 0262033844.

[20] George F. Coulouris. *Distributed Systems.* Pearson Education, 2011. ISBN 0273760599.

[21] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005. ISSN 0163-5948.

[22] Markus Degen, Peter Thiemann, and Stefan Wehr. True Lies: lazy contracts for lazy languages (faithfulness is better than laziness). In *Arbeitstagung Programmiersprachen (ATPS)*, ATPS '09. Springer, 2009.

[23] Markus Degen, Peter Thiemann, and Stefan Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *The Journal of Logic and Algebraic Programming*, 79, 2010.

[24] Markus Degen, Peter Thiemann, and Stefan Wehr. The interaction of contracts and laziness. In *Proceedings of the 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12. ACM, 2012.

[25] Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, 33(5), November 2011. ISSN 0164-0925.

[26] Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. Future contracts. In *Proceedings of the 11th Conference on Principles and Practice of Declarative Programming*, PPDP '09. ACM, 2009.

[27] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.

[28] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming*, ESOP '12. Springer-Verlag, 2012.

[29] Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. Option contracts. In *Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13. ACM, 2013.

[30] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. Oh lord, please don't let contracts be misunderstood (functional pearl). In *Proceedings of the 2st International Conference on Functional Programming*, ICFP '16, 2016.

[31] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th International Conference on Functional Programming*, ICFP '11. ACM, 2011.

[32] Paul Erdős. On pseudoprimes and carmichael numbers. In *Publicationes Mathematicae Debrecen*. Institute of Mathematics, University of Debrecen, 1956.

[33] Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, STOC '98. ACM, 1998.

[34] Robert Bruce Findler. Behavioral software contracts. In *Proceedings of the 19th International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[35] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS '06. Springer-Verlag, 2006.

[36] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th International Conference on Functional Programming*, ICFP '02. ACM, 2002.

[37] Robert Bruce Findler, Shu-Yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In *Implementation and Application of Functional Languages*. Springer-Verlag, 2008. ISBN 978-3-540-85372-5.

[38] Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd Symposium on Principles of Programming Languages*, POPL '06. ACM, 2006.

[39] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[40] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, 1994.

[41] Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. Technical Report 52, Indiana University, Computer Science Department, 1976.

[42] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[43] Michael Greenberg. Space-efficient manifest contracts. In *Proceedings of the 42nd Symposium on Principles of Programming Languages*, POPL '15. ACM, 2015.

[44] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th Symposium on Principles of Programming Languages*, POPL '10. ACM, 2010.

[45] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 Symposium on Dynamic languages*, DLS '07. ACM, 2007.

[46] Klaus Havelund and Grigore Rosu. Monitoring java programs with java pathexplorer. Technical report, NASA Ames Research Center, 2001.

[47] Klaus Havelund and Grigore Rosu. Runtime Verification (RV '01, RV '02, RV '04). Elsevier, 2001, 2002, 2004, 2001,2002,2004. ENTCS 55, 70, 113.

[48] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 13–26, New York, NY, USA, 1987. ACM.

[49] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, New York, NY, USA, 2008. ACM.

[50] Rich Hickey. Clojure core.spec Documentation. `https://clojure.org/guides/spec`, 2018. [Online; accessed 18-February-2018].

[51] Daniel Higginbotham. *Clojure for the Brave and True: Learn the Ultimate Language and Become a Better Programmer.* No Starch Press, 2015.

[52] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS '06. Springer-Verlag, 2006.

[53] P. Z. Ingerman. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1), January 1961. ISSN 0001-0782.

[54] Omar Javed, Yudi Zheng, Andrea Rosà, Haiyang Sun, and Walter Binder. Extended code coverage for aspectj-based runtime verification tools. In Yliès Falcone and César Sánchez, editors, *Runtime Verification: 16th International Conference*, RV '16. Springer, 2016.

[55] A. Jeffrey. Semantics for core Concurrent ML using computation types. In *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1998.

[56] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, 2013.

[57] Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *Proceedings of the 20th International Conference on Functional Programming*, ICFP '15. ACM, 2015.

[58] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, ECOOP '97, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[59] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, Mar 2004. ISSN 1572-8102. doi: 10.1023/B:FORM.0000017719.43755.7c. URL `https://doi.org/10.1023/B:FORM.0000017719.43755.7c`.

[60] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Haskell Symposium*, 2013.

[61] Daan Leijen. Koka: A language with row-polymorphic effect inference. In *1st Workshop on Higher-Order Programming with Effects (HOPE 2012)*, September 2012.

[62] Tim Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley, Upper Saddle River, NJ, 2014. ISBN 978-0133905908.

[63] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[64] Conor McBride. The Frank manual. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/`, 2012.

[65] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 1960.

[66] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.

[67] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[68] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. Extensible access control with authorization contracts. In *Proceedings of the 2016 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16. ACM, 2016.

[69] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *Proceedings of the 19th International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[70] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proceedings of the 43th Annual Symposium on Principles of Programming Languages*, POPL '16, New York, NY, USA, 2016. ACM.

[71] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US.

[72] Zachary Owens. Contract monitoring as an effect. In *Proceedings of the 1st ACM SIGPLAN Workshop on Higher-Order Programming with Effects*, HOPE '12. ACM, 2012.

[73] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[74] John H. Reppy. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*. Springer-Verlag, 1993. ISBN 3-540-56883-2.

[75] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. ISBN 0-521-48089-2.

[76] Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42Nd Symposium on Principles of Programming Languages*, POPL '15. ACM, 2015.

[77] Avraham Shinnar. Safe and effective contracts. Technical report, Harvard University, 2011.

[78] Oleg Sokolsky, Klaus, and Mahesh Viswanathan. Runtime Verification (RV '03). Elsevier, 2003, 2005. ENTCS 89.

[79] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. ACM, 2012.

[80] Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. Expressing contract monitors as patterns of communication. In *Proceedings of the 20th International Conference on Functional Programming*, ICFP '15. ACM, 2015.

[81] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th International Conference on Functional Programming*, ICFP '10. ACM, 2010.

[82] Sjoerd Visscher. Control.effects. `http://github.com/sjoerdvisscher/effects`, 2012.

[83] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open world soundness and collaborative blame for gradual type system. In *POPL*, 2017.

[84] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09. Springer-Verlag, 2009.

[85] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. ISSN 0890-5401.

[86] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages*, POPL

’09. ACM, 2009.

CHAPTER A

Embedding Findler and Felleisen [36] into λ^{π}_{cs}

We set out to provide a unifying framework for contract semantics, a sort of "assembly language" target for recreating, understanding, and comparing contract strategies. To demonstrate our accomplishment, we now prove that **eager** in λ^{π}_{cs} simulates the runtime verification of λ^{CON} from Findler and Felleisen [36] (given in Figure A.1), up to alpha-equivalence and unit elimination.

The language give in Figure A.1 elides a handful of forms from the version presented in λ^{CON}, including list and fixpoint operations (since neither are relevant to the discussion) and, more important, their outer $\mathsf{val\ rec}$ form, defined as (with appropriate evaluation contexts to match, which

Findler and Felleisen, 2002

$$\begin{array}{rcl}
c & = & \lambda\, x.\, c \mid c\ c \mid x \mid \mathsf{if}\ c\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{true} \mid \mathsf{false} \mid n \mid c\ op\ c \\
 & \mid & c \mapsto c \mid \mathsf{contract}(c) \mid \mathsf{blame}(c) \mid c^{c,x,x} \\
C & = & \square \mid C\ c \mid V\ C \mid C\ op\ c \mid V\ op\ C \mid \mathsf{if}\ C\ \mathsf{then}\ c\ \mathsf{else}\ c \\
 & \mid & C \mapsto c \mid \mathsf{contract}(C) \mid \mathsf{blame}(C) \mid c^{C,x,x} \mid C^{V,x,x} \\
V & = & \lambda\, x.\, c \mid n \mid \mathsf{true} \mid \mathsf{false} \mid V \mapsto V \mid \mathsf{contract}(V) \mid V^{V\mapsto V,x,x}
\end{array}$$

$$\begin{aligned}
C[V_1^{\mathsf{contract}(V_2),p,n}] &\longrightarrow C[\mathsf{if}\ V_2\ V_1\ \mathsf{then}\ V_1\ \mathsf{else}\ \mathsf{blame}(p)] \\
C[(V_1^{V_3\mapsto V_3,p,n})\ V_2] &\longrightarrow C[(V_1\ V_2^{V_3,n,p})^{V_4,p,n}] \\
C[e] &\longrightarrow C[e']\ (\mathit{if}\ e \rightsquigarrow e')
\end{aligned}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\lambda\, x.\, c\ V & \rightsquigarrow & c[x/V] & \ulcorner n_1 \urcorner + \ulcorner n_2 \urcorner & \rightsquigarrow & \ulcorner n_1 + n_2 \urcorner \\
\mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2 & \rightsquigarrow & c_1 & \ulcorner n_1 \urcorner \leq \ulcorner n_2 \urcorner & \rightsquigarrow & \mathsf{true}\ (\text{if}\ n_1 \leq n_2) \\
\mathsf{if\ false\ then}\ c_1\ \mathsf{else}\ c_2 & \rightsquigarrow & c_2 & \ulcorner n_1 \urcorner \leq \ulcorner n_2 \urcorner & \rightsquigarrow & \mathsf{false}\ (\text{if}\ n_1 \not\leq n_2) \\
 & & & & \vdots &
\end{array}$$

Figure A.1. A subset of the λ^{CON} language from Findler and Felleisen [36]. We have renamed their E to C and e to c to avoid ambiguity.

Embedding Translation

$$\boxed{c \rightsquigarrow (K,\ P,\ e)}$$

$$\frac{}{\mathsf{true} \rightsquigarrow (\emptyset,\ \emptyset,\ \mathsf{true})} \qquad \frac{}{\mathsf{false} \rightsquigarrow (\emptyset,\ \emptyset,\ \mathsf{false})} \qquad \frac{}{n \rightsquigarrow (\emptyset,\ \emptyset,\ n)}$$

$$\frac{}{x \rightsquigarrow (\emptyset,\ \emptyset,\ x)} \qquad \frac{c \twoheadrightarrow_e e}{\lambda\ x.\ c \rightsquigarrow (\emptyset,\ \emptyset,\ \lambda\ x.\ e)}$$

$$\frac{c_1 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e_2)}{c_1\ c_2 \rightsquigarrow (K,\ P,\ e_1\ e_2)} \qquad \frac{c_1 \notin V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2}{c_1\ c_2 \rightsquigarrow (K,\ P,\ e_1\ e_2)}$$

$$\frac{c_1 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e_2)}{c_1\ op\ c_2 \rightsquigarrow (K,\ P,\ e_1\ op\ e_2)} \qquad \frac{c_1 \notin V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2}{c_1\ op\ c_2 \rightsquigarrow (K,\ P,\ e_1\ op\ e_2)}$$

$$\frac{c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad c_3 \twoheadrightarrow_e e_3}{\mathsf{if}_\circ\ c_1\ \mathsf{then}\ c_2\ \mathsf{else}\ c_3 \rightsquigarrow (K,\ P,\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e3)}$$

$$\frac{\begin{array}{c} c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad c_3 \twoheadrightarrow_e e_3 \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad p = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)))\rangle^\pi \end{array}}{\mathsf{if}_\bullet\ c_1\ \mathsf{then}\ c_2\ \mathsf{else}\ c_3 \rightsquigarrow (\{\iota\} \uplus K,\ \{p\} \uplus P,\ \mathit{conres}\ (\mathsf{read}\ \iota))}$$

$$\frac{c_1 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e_2) \quad \mathit{fresh}\ f \quad \mathit{fresh}\ b \quad \mathit{fresh}\ x}{(c_1 \mapsto c_2) \rightsquigarrow (K,\ P,\ \lambda\ f\ b.\ \lambda\ x.\ \mathbf{mon}\ e_2\ \mathbf{eager}\ (f\ (\mathbf{mon}\ e_1\ \mathbf{eager}\ x\ (\mathit{invert}\ b)))\ b)}$$

$$\frac{c_1 \notin V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad \mathit{fresh}\ f \quad \mathit{fresh}\ b \quad \mathit{fresh}\ x}{(c_1 \mapsto c_2) \rightsquigarrow (K,\ P,\ \lambda\ f\ b.\ \lambda\ x.\ \mathbf{mon}\ e_2\ \mathbf{eager}\ (f\ (\mathbf{mon}\ e_1\ \mathbf{eager}\ x\ (\mathit{invert}\ b)))\ b)}$$

$$\frac{c \rightsquigarrow (K,\ P,\ e) \quad \mathit{fresh}\ x \quad \mathit{fresh}\ b}{\mathsf{contract}(c) \rightsquigarrow (K,\ P,\ \lambda\ x\ b.\ \mathsf{if}\ e\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{raise}\ b)}$$

$$\frac{c \rightsquigarrow (K,\ P,\ e) \quad B \approx e}{\mathsf{blame}(c) \rightsquigarrow (K,\ P,\ \mathsf{raise}\ B)}$$

$$\frac{c_1 \notin V \quad c_2 \notin V \quad c_1 \twoheadrightarrow_e e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e) \quad B \approx (p, n)}{c_1^{c_2,p,n} \rightsquigarrow (K,\ P,\ \mathbf{mon}\ e_2\ \mathbf{eager}\ e_1\ B)}$$

$$\frac{\begin{array}{c} c_1 \notin V \quad c_2 \in V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_v e_2 \quad B \approx (p, n) \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad p = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_2\ (\mathsf{read}\ \iota)\ B)))\rangle^\pi \end{array}}{c_1^{c_2,p,n} \rightsquigarrow (\{\iota\} \uplus K,\ \{p\} \uplus P,\ \mathsf{seq}\ (\mathsf{write}\ \iota\ e_1)\ (\mathit{conres}\ (\mathsf{read}\ \iota)))}$$

$$\frac{\begin{array}{c} c_1 \in V \quad c_2 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad \mathsf{contract}(c_2) \twoheadrightarrow_v e_2 \quad B \approx (p, n) \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad p = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_2\ e_1\ B)))\rangle^\pi \end{array}}{c_1^{\mathsf{contract}(c_2),p,n} \rightsquigarrow (\{\iota\},\ \{p\},\ \mathit{conres}\ (\mathsf{read}\ \iota))}$$

$$\frac{\begin{array}{c} c_1 \in V \quad c_2 \in V \quad c_3 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \twoheadrightarrow_v e_2 \quad c_3 \twoheadrightarrow_v e_3 \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad \mathit{fresh}\ x \quad B \approx (p, n) \end{array}}{c_1^{c_2 \mapsto c_3,p,n} \rightsquigarrow (\emptyset,\ \emptyset,\ \lambda\ x.\ \mathbf{mon}\ c_3\ \mathbf{eager}\ (e_1\ (\mathbf{mon}\ c_2\ \mathbf{eager}\ x\ (\mathit{invert}\ B)))\ B)}$$

Figure A.2. Embedding procedure for λ^{CON} into λ^{π}_{cs}.

Embedding Translations

$$\boxed{c \twoheadrightarrow_e e}$$

$$\frac{}{\mathsf{true} \twoheadrightarrow_e \mathsf{true}} \qquad \frac{}{\mathsf{false} \twoheadrightarrow_e \mathsf{false}} \qquad \frac{}{x \twoheadrightarrow_e x} \qquad \frac{}{n \twoheadrightarrow_e n}$$

$$\frac{c \twoheadrightarrow_e e}{\lambda\, x.\, c \twoheadrightarrow_e \lambda\, x.\, e} \qquad \frac{c_1 \twoheadrightarrow_e e_1 \quad c_2 \twoheadrightarrow_e e_2}{c_1\ c_2 \twoheadrightarrow_e e_1\ e_2} \qquad \frac{c_1 \twoheadrightarrow_e e_1 \quad c_2 \twoheadrightarrow_e e_2}{c_1\ op\ c_2 \twoheadrightarrow_e e_1\ op\ e_2}$$

$$\frac{c_1 \twoheadrightarrow_e e_1 \quad c_2 \twoheadrightarrow_e e_2 \quad c_3 \twoheadrightarrow_e e_3}{\mathsf{if}_\circ\ c_1\ \mathsf{then}\ c_2\ \mathsf{else}\ c_3 \twoheadrightarrow_e \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e3}$$

$$\frac{c \twoheadrightarrow_e e \quad fresh\ x \quad fresh\ b}{\mathsf{contract}(c) \twoheadrightarrow_e \lambda\ x\ b.\ \mathsf{if}\ e\ x\ \mathsf{then}\ x\ \mathsf{else\ raise}\ b} \qquad \frac{c \twoheadrightarrow_e e}{\mathsf{blame}(c) \twoheadrightarrow_e \mathsf{raise}\ e}$$

$$\frac{c_1 \twoheadrightarrow_e e_1 \quad c_2 \twoheadrightarrow_e e_2 \quad B \approx (p,n)}{c_1^{c_2,p,n} \twoheadrightarrow_e \mathbf{mon}\ e_2\ \mathbf{eager}\ e_1\ B}$$

$$\boxed{V \twoheadrightarrow_v v}$$

$$\frac{}{\mathsf{true} \twoheadrightarrow_v \mathsf{true}} \qquad \frac{}{\mathsf{false} \twoheadrightarrow_v \mathsf{false}} \qquad \frac{}{n \twoheadrightarrow_v n} \qquad \frac{c \twoheadrightarrow_e e}{\lambda\, x.\, c \twoheadrightarrow_v \lambda\, x.\, e}$$

$$\frac{c \twoheadrightarrow_v e \quad fresh\ x \quad fresh\ b}{\mathsf{contract}(e) \twoheadrightarrow_v \lambda\ x\ b.\ \mathsf{if}\ e\ x\ \mathsf{then}\ x\ \mathsf{else\ raise}\ b} \qquad \frac{c \twoheadrightarrow_v e}{\mathsf{blame}(c) \twoheadrightarrow_v \mathsf{raise}\ e}$$

$$\frac{c_1 \twoheadrightarrow_v e_1 \quad c_2 \twoheadrightarrow_v e_2 \quad c_3 \twoheadrightarrow_v e_3 \quad B \approx (p,n) \quad fresh\ x}{c_1^{c_2 \mapsto c_3,p,n} \twoheadrightarrow_v \lambda\ x.\ \mathbf{mon}\ c_3\ \mathbf{eager}\ (e_1\ (\mathbf{mon}\ c_2\ \mathbf{eager}\ x\ (invert\ B)))\ B}$$

Figure A.3. Sub-translation relations $\twoheadrightarrow_e$ and $\twoheadrightarrow_v$ for embedding λ^{CON} into λ^{π}_{cs}.

evaluate the bindings before the body):

$$\begin{aligned} p &= d \cdots c \\ d &= \mathsf{val\ rec}\ x : c = c \end{aligned}$$

Here, each binding has two values associated as $x : V_1 = V_2$, where the first represents a contract on the second. Findler and Felleisen install these contracts on each occurrence of x in the program via their I operator (given in Figure 14 of their technical report) such that, if the program binds x as $\mathsf{val\ rec}\ x : e_1 = e2$, then each usage site of x is rewritten as $x^{e_1,x,n}$ to enforce the contract e_1. Findler and Felleisen provide this machinery to more closely match the module interaction system and blame coordination in Racket, which is unnecessary for demonstrating that their individual monitors proceed via **eager** verification. As such, our simulation assumes that each clause in the outer val rec form is completely evaluated and substituted in, removing the need for the "$\xrightarrow{rec}$" reduction along with syntax forms p and d and contexts P.

Our simulation begins with three translation relations from λ^{CON} to λ^{π}_{cs}, defined as "$\rightsquigarrow$", "$\twoheadrightarrow_e$", and "$\twoheadrightarrow_v$", defined in Figure A.2 and Figure A.3. The $\rightsquigarrow$ operator relates a term c with an expression e, a set of new channels K, and a set of process P, where the translated expression will fill process π_0, which $\twoheadrightarrow_e$ and $\twoheadrightarrow_v$ translate terms and values in λ^{CON} into equivalent term-level expressions in λ^{π}_{cs}.

The intuition is that $\rightsquigarrow$ acts as the main translator, focusing in on the next redex of the program, while $\twoheadrightarrow_e$ evaluates each unevaluated portion of the program (i.e., c occurrences in context C) and $\twoheadrightarrow_v$ translates each evaluated portion of the term (i.e., V occurrences in context C). Next, we define a series of lemmas and finally prove our simulation result:

Lemma A.1 (Translating Values Yield No Channels or Processes). *If $c \in V$ and $c \rightsquigarrow (K,\ P,\ e)$, then $e \in v$, $K = \emptyset$, and $P = \emptyset$.*

Proof (Sketch). This proof establishes that we relate values in λ^{CON} to values in λ^{π}_{cs}.

Proof proceeds by induction on c and our constraint that it is a value, then inversion on the translation relation. □

Lemma A.2 (Embedding Translation Focuses on Redex). *If $c = C[c_0]$ such that c_0 is the next redex, then $c \rightsquigarrow (K,\ P,\ e)$ has some derivation tree as*

$$\frac{\mathcal{D}}{c \rightsquigarrow (K,\ P,\ e)}$$

then $\mathcal{D}$ contains $c_0 \rightsquigarrow (K_0,\ P_0,\ e_0)$.

Proof (Sketch). Findler and Felleisen [36] prove unique decomposition for c terms, and proof proceeds by induction on the structure of C and inversion on $\mathcal{D}$. □

Lemma A.3 (Embedding Reduction). *If $c \in \lambda^{CON}$ such that $\cdot \vdash c : t$ (that is, c is well-typed), $c \longrightarrow c'$, and $c \rightsquigarrow (K,\ P,\ e)$ and $c' \rightsquigarrow (K',\ P',\ e')$, then $K, \{\pi_0\}, \{\langle e \rangle^{\pi_0}\} + P \Rightarrow^* K'',\ \{\pi_0\},\ P''$ such that*
$K'',\ \{\pi_0\},\ P'' =_{\alpha,\mathsf{unit}} K', \{\pi_0\}, \{\langle e' \rangle^{\pi_0}\} + P'$.

Proof. This proof establishes that each evaluation step in λ^{CON} has a corresponding set of evaluation steps in λ^{π}_{cs}. We make the following simplifying assumptions:

- We appeal to alpha equivalence for our reduction in order to avoid the need to explicitly track and name channels and process identifiers in terms of their creation, so that we do not need to maintain the *exact* channel name or process identifier between steps, only that we use each name alpha-equivalently.

- We disregard processes of the form $\langle \mathsf{unit} \rangle^{\pi}$ in our embedding; these processes are the product of contract verification, and ensuring each exists would require keeping track of each contract checked *up until the current point in the derivation.*

Next, we note that the termination set will always be precisely $\{\pi_0\}$ since neither our translation relation or eager monitors will introduce a spawn_f term, and thus we do not need to consider the termination set for the proof.

To differentiate between contract-checking terms and other conditional branching operations, we "recolor" the if expressions in c to indicate their origin: we mark if expressions that originate in the program (i.e., c) with $\circ$, and we modify $\rightarrow_{\lambda CON}$ to color contract-introduced if statements as:

$$D[V^{\mathsf{contract}(V_2),p,n}] \xrightarrow{\mathit{flat}} D[\mathsf{if}_{\bullet}\ V_2\ (V)\ \mathsf{then}\ V\ \mathsf{else}\ \mathsf{blame}(p)]$$

Evaluation for both $\mathsf{if}_{\circ}\ c\ \mathsf{then}\ c\ \mathsf{else}\ c$ and $\mathsf{if}_{\bullet}\ c\ \mathsf{then}\ c\ \mathsf{else}\ c$ otherwise proceed as $\mathsf{if}\ c\ \mathsf{then}\ c\ \mathsf{else}\ c$.

We perform induction on "$c \longrightarrow c''$. Except for contract enforcement operations, our translation directly preserves the source language syntax, and thus we only present the contract-related cases in detail:

<u>*Case:*</u>. $C[V_1^{\mathsf{contract}(V_2),p,n}] \longrightarrow C[\mathsf{if}_{\bullet}\ V_2\ V_1\ \mathsf{then}\ V_1\ \mathsf{else}\ \mathsf{blame}(p)]$

Using $\rightsquigarrow$, we have:

$$C[V_1^{\mathsf{contract}(V_2),p,n}] \rightsquigarrow (P,\ K,\ e)$$

$$C[\mathsf{if}_{\bullet}\ V_2\ V_1\ \mathsf{then}\ V_1\ \mathsf{else}\ \mathsf{blame}(p)] \rightsquigarrow (P',\ K',\ e')$$

Since C is static, the term inside of C is the next redex and thus is translated by $\twoheadrightarrow_v$ and $\twoheadrightarrow_e$, whereas we translate $V_1^{\mathsf{contract}(V_2),p,n}$ by $\rightsquigarrow$ (via Lemma A.2). Furthermore,

$$\frac{\begin{array}{ll} V_1 \in V \quad V_2 \in V & V_2 \twoheadrightarrow_v e_2 \\ B \approx (p,n) \quad V_1 \twoheadrightarrow_v e_1 & \mathsf{contract}(V_2) \twoheadrightarrow_v \lambda\ x\ b.\ \mathsf{if}\ e_2\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{raise}\ b \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi & p = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (V_2\ V_1\ B)))\rangle^{\pi} \end{array}}{V_1^{\mathsf{contract}(V_2),p,n} \rightsquigarrow (\{\iota\},\ \{p\},\ \mathit{conres}\ (\mathsf{read}\ \iota))}$$

and

$$\frac{\begin{array}{c} c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad c_3 \twoheadrightarrow_e e_3 \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi' \quad p' = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)))\rangle^{\pi'} \end{array}}{\mathsf{if}_{\bullet}\ V_2\ V_1\ \mathsf{then}\ V_2\ \mathsf{else}\ \mathsf{blame}(p) \rightsquigarrow (\{\iota'\} \uplus K,\ \{p'\} \uplus P,\ \mathit{conres}\ (\mathsf{read}\ \iota))}$$

Thus $P = P_0 \uplus \{p\}$ and $K = K_0 \uplus \{\iota\}$ (taking $\iota = \iota'$ and $\pi = \pi'$). Then it is sufficient to show that

$$\begin{array}{rl} & K \uplus \{\iota\}, \{\pi_0\}, P_0 \uplus \{p\} \uplus \langle \mathcal{E}[\mathsf{conres}\ (\mathsf{read}\ \iota)]\rangle^{\pi} \\ \Rightarrow & K \uplus \{\iota\}, \{\pi_0\}, P_0 \uplus \{p'\} \uplus \langle \mathcal{E}[\mathsf{conres}\ (\mathsf{read}\ \iota)]\rangle^{\pi} \end{array}$$

This follows directly because p reduces to p' by our definition of term reduction relation "$\longmapsto$" and [ProcStep] in the "$\Rightarrow$" relation.

<u>*Case:*</u>. $C[(V_1^{V_3 \mapsto V_4, p, n})\ V_2] \longrightarrow C[(V_1\ V_2^{V_3,n,p})^{V_4,p,n}]$

We use the same argument to deal with context C as in the previous case, yielding some K_0 and P_0 for the reduction. Then:

$$\frac{V_1^{V_3 \mapsto V_4,p,n} \in V \quad V_1^{V_3 \mapsto V_4,p,n} \twoheadrightarrow_v e_{fc} \quad V_2 \rightsquigarrow (\emptyset,\ \emptyset,\ e_2)}{V_1^{V_3 \mapsto V_4,p,n}\ V_2 \rightsquigarrow (\emptyset,\ \emptyset,\ e_{fc}\ e_2)}$$

where the channel and processes in the premises are empty by Lemma A.1 and e_{fc} is the result of the translation

$$\frac{V_1 \twoheadrightarrow_v e_1 \quad V_3 \twoheadrightarrow_v e_3 \quad V_4 \twoheadrightarrow_v e_4 \quad B \approx (p,n) \quad \mathit{fresh}\ x}{V_1^{V_3 \mapsto V_4,p,n} \twoheadrightarrow_v \lambda\ x.\ \mathbf{mon}\ e_4\ \mathbf{eager}\ (e_1\ (\mathbf{mon}\ e_3\ \mathbf{eager}\ x\ (\mathit{invert}\ B)))\ B}$$

The right-hand side of the reduction proceeds as:

$$\frac{\begin{array}{c}(V_1\ V_2^{V_3,n,p}) \notin V \quad V_4 \in V \quad (V_1\ V_2^{V_3,n,p}) \rightsquigarrow (K',\ P',\ e_1\ e_2') \quad V_4 \twoheadrightarrow_v e_2 \quad B \approx (p,n) \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad p = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_4\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi}\end{array}}{(V_1\ V_2^{V_3,n,p})^{V_4,p,n} \rightsquigarrow (\{\iota\} \uplus K',\ \{p\} \uplus P',\ \mathsf{seq}\ (\mathsf{write}\ \iota\ e_1\ e_2')\ (\mathsf{conres}\ (\mathsf{read}\ \iota)))}$$

where

$$\frac{V_1 \in V \quad V_1 \twoheadrightarrow_v e_1 \quad V_2^{V_3,n,p} \rightsquigarrow (K',\ P',\ e_2')}{(V_1\ V_2^{V_3,n,p}) \rightsquigarrow (K,\ P,\ e_1\ e_2')}$$

The exact shape of e_2' depends upon the shape of V_3. Since c is well-typed, it is either a flat or function contract. We proceed by consider each, appealing to this series of reductions to prove each:

Subcase: $V_3 = \mathsf{contract}(V_3')$

$$\frac{\begin{array}{c}V_3 \in V \quad V_3 \in V \quad V_3 \twoheadrightarrow_v e_2 \quad \mathsf{contract}(V_3) \twoheadrightarrow_v e_3 \quad B \approx (p,n) \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad p = \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_3\ e_2\ B)))\rangle^{\pi}\end{array}}{V_3^{\mathsf{contract}(V_3),p,n} \rightsquigarrow (\{\iota\},\ \{p\},\ \mathsf{conres}\ (\mathsf{read}\ \iota))}$$

Thus it is sufficient to show

$$\begin{array}{rl} & K_0, P_0 + \langle \mathcal{E}[(\mathbf{mon}\ e_4\ \mathbf{eager}\ (e_1\ (\mathbf{mon}\ e_3\ \mathbf{eager}\ x\ (\mathit{invert}\ B)))\ B)\ e_2]\rangle^{\pi_0} \\ \Rightarrow^* & K_0 \uplus \{\iota, \iota'\}, P_0 + \langle \mathcal{E}[\mathsf{seq}\ (\mathsf{write}\ \iota\ (e_1\ (\mathsf{conres}\ (\mathsf{read}\ \iota'))))\ (\mathsf{conres}\ (\mathsf{read}\ \iota))]\rangle^{\pi_0} \\ & + \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_4\ (\mathsf{read}\ \iota)\ B)))\rangle^{\pi} \\ & + \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_3\ e_2\ \bar{B})))\rangle^{\pi} \end{array}$$

which follows from our reduction semantics in λ_{cs}^{π}.

Subcase: $V_3 = V_{3i} \mapsto V_{3o}$

Since both V_2 and V_3 are values, we translate this as:

$$\frac{\begin{array}{c} V_2 \in V \quad V_{3i} \in V \quad V_{3o} \in V \quad V_2 \twoheadrightarrow_v e_2 \quad V_{3i} \twoheadrightarrow_v e_{3i} \quad V_{3o} \twoheadrightarrow_v e_{3o} \\ \mathit{fresh}\ \iota \quad \mathit{fresh}\ \pi \quad \mathit{fresh}\ x \quad \bar{B} \approx (n, p) \end{array}}{V_2^{V_{3i} \mapsto V_{3o}, n, p} \rightsquigarrow (\emptyset,\ \emptyset,\ \lambda\ x.\ \mathsf{mon}\ e_{3o}\ \mathsf{eager}\ (e_2\ (\mathsf{mon}\ e_{3i}\ \mathsf{eager}\ x\ B))\ \bar{B})}$$

Thus it is sufficient to show

$$\begin{aligned} & K_0, P_0 + \langle \mathcal{E}[(\mathsf{mon}\ e_4\ \mathsf{eager}\ (e_1\ (\mathsf{mon}\ e_3\ \mathsf{eager}\ x\ (\mathit{invert}\ B)))\ B)\ e_2] \rangle^{\pi_0} \\ \Rightarrow^* \ & K_0 \uplus \{\iota\}, P_0 + \langle \mathcal{E}[\mathsf{seq}\ (\mathsf{write}\ \iota\ (e_1\ e_{2v}))\ (\mathit{conres}\ (\mathsf{read}\ \iota))] \rangle^{\pi_0} \\ & + \langle \mathsf{write}\ \iota\ (\mathsf{catch}\ \mathsf{inl}\ (\mathsf{inr}\ (e_4\ (\mathsf{read}\ \iota)\ B))) \rangle^{\pi} \end{aligned}$$

where

$$e_{2v} = \lambda\ x.\ \mathsf{mon}\ e_{3o}\ \mathsf{eager}\ (e_1\ (\mathsf{mon}\ e_{3i}\ \mathsf{eager}\ x\ B))\ \bar{B}$$

which follows from our reduction semantics and unit equality to account for the missing unit-value process created by constructing the contracted function value for V_2. □

Finally, we state the embedding theorem as:

THEOREM A.1 (Embedding Correctness). *If* $c \in \lambda^{CON}$ *such that* $\cdot \vdash c : t$, $c \longrightarrow^* V$, $c \rightsquigarrow (K,\ P,\ e)$, *and* $V \twoheadrightarrow_v v$,
then $K, \{\pi_0\}, \{\langle e \rangle^{\pi_0}\} + P \longmapsto^* K', \{\pi_0\}, \langle v \rangle^{\pi_0} + P'$.

PROOF. First, no translation will produce an spawn_f form, and thus T remains constant. Then the proof proceeds by induction on the length of $\longrightarrow^*$ and Lemma A.3. □

This proof demonstrates that our approach to eager monitoring faithfully recreates the original presentation and, more generally, that defining contracts as patterns of communication maintains previous models while exposing their internal workings at a finer granularity.

Moreover, this proof is, in a sense, straightforward: we are, in essence, doing nothing more than piecing apart and tracking the individual evaluators that go into contract monitoring, separating the user portions from the monitoring ones. These two approaches may appear distinct on the surface, but they are, in actuality, closely connected. Moreover, additional simulation proofs will be more complex, but follow this same approach: syntactic munging plus a little process management.

CHAPTER B

λ^{π}_{cd}: A delay-less Variant of λ^{π}_{cs}

The chaperone-based λ^{π}_{cd} differs from λ^{π}_{cs} in its inclusion of chaperone values of the form $\langle e_1\rangle^{e_2}_{e_3}$, where e_1 is the chaperoned value, e_2 is the *chaperone*, indicating how enforcement should proceed then the chaperoned value is required, and e_3 contains blame information. After adding these chaperones, we can remove force and delay, and thus do so. To automatically force chaperones, we reclassify those contexts which must perform forcing as C contexts in order to force operations in, e.g., operator position. These C contexts are a strict subset of the D contexts, and are embedded as such.

Finally, we modify our reduction relation "$\rightarrow$" to support evaluating a chaperone in a C context. This reduction uses the chaperone to yield the underlying value to the program:

$$C[\langle e_1\rangle^{e_2}_{e_3}] \;\rightarrow\; C[e_2\ e_1\ e_3]$$

Now, we can define semi and prom verification in terms of such chaperones:

```
mon con semi exp B  →  let f = λ exp b.let i  =  chan
                                         in seq
                                              (spawn (write i (catch inl (inr (con (read i) b)))))
                                              (write i exp)
                                              (conres (read i))
                       in ⟨exp⟩^f_B

mon con prom exp B  → let i  =  chan
                       in seq
                           (spawn (write i (catch inl (inr (con (read i) B)))) )
                           (write i exp)
                           ⟨exp⟩_B^(λ _ _. conres (read i))
```

In each case, we construct the appropriate procedure for extraction and construct the appropriate chaperone result. For example, in the semi-eager case, checking nat/c on 5 will result in a value $\langle 5\rangle^{\dots\ \text{check nat/c}\ \dots}_{B}$ that will ensure 5 is a natural number when it appears in a forcing position, i.e., C contexts.

Syntax

$$
\begin{array}{llll}
\textsc{Exprs} & e & := & x \mid v \mid e\ e \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \\
 & & \mid & e\ \mathit{binop}\ e \mid \mathit{unop}\ e \mid (e, e) \mid \textsf{fst}\ e \mid \textsf{snd}\ e \\
 & & \mid & \textsf{case}\ (e;\ x \rhd e;\ x \rhd e) \mid \textsf{inl}\ e \mid \textsf{inr}\ e \\
 & & \mid & \textsf{raise}\ e \mid \textsf{catch}\ e\ e \\
 & & \mid & \textsf{spawn}\ e \mid \textsf{spawn}_f\ e \mid \textsf{chane} \mid \textsf{read}\ e \mid \textsf{write}\ e\ e \\
 & & \mid & \textbf{mon}\ e\ e\ e\ e \\
 & & & \\
\textsc{Values} & v & := & \lambda\ x.\ e \mid \iota \mid (v, v) \mid \textsf{inl}\ v \mid \textsf{inr}\ v \\
 & & \mid & n \mid \textsf{true} \mid \textsf{false} \mid \textsf{unit} \mid B \mid s \\
 & & \mid & \langle e \rangle_v^v \\
\textsc{Strategies} & s & := & \textbf{eager} \mid \textbf{semi} \mid \textbf{prom} \mid \cdots \\
 & & & \\
\textsc{C-Contexts} & C & := & \square \mid C\ e \mid \textsf{if}\ C\ \textsf{then}\ e\ \textsf{else}\ e \\
 & & \mid & C\ \mathit{binop}\ e \mid v\ \mathit{binop}\ C \mid \mathit{unop}\ C \\
 & & \mid & \textsf{fst}\ C \mid \textsf{snd}\ C \\
 & & \mid & \textsf{case}\ (C;\ x \rhd e;\ x \rhd e) \\
 & & \mid & \textsf{catch}\ C\ e \\
 & & \mid & \textsf{read}\ C \mid \textsf{write}\ C\ e \\
 & & \mid & \textbf{mon}\ C\ e\ e\ e \mid \textbf{mon}\ v\ C\ e\ e \mid \textbf{mon}\ v\ v\ e\ C \\
 & & & \\
\textsc{D-Contexts} & D & := & \square \mid C \mid C[D] \mid D\ e \mid v\ D \mid \textsf{if}\ D\ \textsf{then}\ e\ \textsf{else}\ e \\
 & & \mid & D\ \mathit{binop}\ e \mid v\ \mathit{binop}\ D \mid \mathit{unop}\ D \\
 & & \mid & (D, e) \mid (v, D) \mid \textsf{fst}\ D \mid \textsf{snd}\ D \\
 & & \mid & \textsf{case}\ (D;\ x \rhd e;\ x \rhd e) \mid \textsf{inl}\ D \mid \textsf{inr}\ D \\
 & & \mid & \textsf{raise}\ D \mid \textsf{catch}\ D\ e \\
 & & \mid & \textsf{chan}D \mid \textsf{read}\ D \mid \textsf{write}\ D\ e \mid \textsf{write}\ v\ D \\
 & & \mid & \textbf{mon}\ D\ e\ e\ e \mid \textbf{mon}\ v\ D\ e\ e \mid \textbf{mon}\ v\ v\ e\ D \\
 & & & \\
\mathcal{E}\textsc{-Contexts} & \mathcal{E} & := & D \mid D[\mathcal{E}] \mid \textsf{catch}\ v\ \mathcal{E} \\
 & & & \\
\textsc{ProcId} & \pi & \in & \mathbb{N} \\
\textsc{Process} & \mathit{proc} & = & \langle e_i \rangle^{\pi_i} \\
\textsc{Proc. Set} & P & \in & \mathit{Fin}(\mathit{proc}) \\
 & & & \\
\textsc{Proc. Decomp.} & P + \langle e \rangle^{\pi} & \equiv & P \cup \{\langle e \rangle^{\pi}\} \\
 & & & \\
\textsc{Proc. Config.} & K, T, P & & K \in \mathit{Fin}(\text{channel names}) \\
 & & & T \in \mathit{Fin}(\text{process ids})
\end{array}
$$

Figure B.1. Syntax definitions for λ^{π}_{cs}.

Dynamic Semantics

$\boxed{e \to e'}$

$(\lambda\ x.\ e)\ v$	$\to$	$e[v/x]$	
$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2$	$\to$	e_1	
$\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2$	$\to$	e_2	
$v_1\ \mathit{binop}\ v_2$	$\to$	v	*where* $\delta(\mathit{binop}, v_1, v_2) = v$
$\mathit{unop}\ v'$	$\to$	v	*where* $\delta(\mathit{unop}, v') = v$
$\mathsf{case}\ (\mathsf{inl}\ v;\ x_1 \rhd e_1;\ x_2 \rhd e_2)$	$\to$	$e_1[v/x_1]$	
$\mathsf{case}\ (\mathsf{inr}\ v;\ x_1 \rhd e_1;\ x_2 \rhd e_2)$	$\to$	$e_2[v/x_2]$	
$\mathsf{force}\ (\mathsf{delay}\ e)$	$\to$	e	
$\mathsf{force}\ v$	$\to$	v	*where* $v \neq \mathsf{delay}\ e$
$\mathsf{catch}\ v_1\ v_2$	$\to$	v_2	
$\mathsf{catch}\ v_1\ (\mathsf{raise}\ v_2)$	$\to$	$v_1\ v_2$	
$C[\langle e_1\rangle_{e_3}^{e_2}]$	$\to$	$C[e_2\ e_1\ e_3]$	

$\boxed{e \longmapsto e'}$

$$\frac{e \to e'}{\mathcal{E}[e] \longmapsto \mathcal{E}[e']} \qquad \frac{}{\mathcal{E}[D[\mathsf{raise}\ v_2]] \longmapsto \mathcal{E}[\mathsf{raise}\ v_2]}$$

$\boxed{e_1 \overset{\iota}{\overset{\frown}{\smile}} e_2 \text{ with } (e_1', e_2')}$

$$\frac{}{\mathsf{write}\ \iota\ v \overset{\iota}{\overset{\frown}{\smile}} \mathsf{read}\ \iota \text{ with } (\mathsf{unit}, v)} \qquad \frac{e_1 \overset{\iota}{\overset{\frown}{\smile}} e_2 \text{ with } (e_1', e_2')}{e_2 \overset{\iota}{\overset{\frown}{\smile}} e_1 \text{ with } (e_2', e_1')}$$

$\boxed{K, T, P \Rightarrow K', T, P'}$

$$\frac{e \longmapsto e'}{K, T, P + \langle e\rangle^{\pi} \Rightarrow K, T, P + \langle e'\rangle^{\pi}}$$

$$\frac{\pi' \notin \mathrm{dom}(P)}{K, T, P + \langle \mathcal{E}[\mathsf{spawn}\ e\ ]\rangle^{\pi} \Rightarrow K, T, P + \langle \mathcal{E}[\mathsf{unit}]\rangle^{\pi} + \langle e\rangle^{\pi'}}$$

$$\frac{\pi' \notin \mathrm{dom}(P)}{K, T, P + \langle \mathcal{E}[\mathsf{spawn}_f\ e\ ]\rangle^{\pi} \Rightarrow K,\ T \cup \{\pi'\},\ P + \langle \mathcal{E}[\mathsf{unit}]\rangle^{\pi} + \langle e\rangle^{\pi'}}$$

$$\frac{\iota \notin K}{K, T, P + \langle \mathcal{E}[\mathsf{chan}]\rangle^{\pi} \Rightarrow K \cup \{\iota\}, T, P + \langle \mathcal{E}[\iota]\rangle^{\pi}}$$

$$\frac{e_1 \overset{\iota}{\overset{\frown}{\smile}} e_2 \text{ with } (e_1', e_2') \quad \iota \in K}{K, T, P + \langle \mathcal{E}_1[e_1]\rangle^{\pi_1} + \langle \mathcal{E}_2[e_2]\rangle^{\pi_2} \Rightarrow K, T, P + \langle \mathcal{E}_1[e_1']\rangle^{\pi_1} + \langle \mathcal{E}_2[e_2']\rangle^{\pi_2}}$$

Figure B.2. Dynamic semantics for λ_{cs}^{π}.

CHAPTER C

The Pisces Source Code

```clojure
(ns mcon.core
  (:require [clojure.core.async :as a :refer [go put!]])
  (:require [clojure.test.check.generators :as gen :refer [sample]])
  (:require [clojure.main :as m]))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Check Definition                                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defrecord Blame [server client contract])

(def blm (Blame. "server" "client" "contract"))

(defn invert-blame
  [blame]
  (Blame. (:client blame) (:server blame) (:contract blame)))

(defn indy-blame
  [blame]
  (Blame. (:client blame) (:contract blame) (:contract blame)))

(defrecord Strategy [sname impl])

(defrecord Metastrat [sname impl substrat])

(defn mon-flat
  [contract strat dterm blame]
  (cond
    (instance? Strategy strat)
      ((:impl strat) contract dterm blame)
    :else (Exception. (str "Invalid strategy: " strat "\n"
                           "  contract: " contract "\n"
                           "  input:" dterm "\n"
                           "Blame: " blame))))

(defn mon-meta
  [contract strat dterm blame]
  (cond
    (instance? Metastrat strat)
      ((:impl strat) contract dterm (:substrat strat) blame)
    (instance? Strategy strat)
      (mon-flat contract strat dterm blame)
    :else (Exception. (str "Invalid strategy: " strat "\n"
                           "  contract: " contract "\n"
                           "  input:" dterm "\n"
```

```
                            "Blame: " blame))))

(defmacro mon
  "Check a contract with a specific strategy"
  [contract strat value blame]
  `(mon-meta ~contract ~strat (delay ~value) ~blame))

(defn extract
  [exp] (if (or (delay? exp) (future? exp)) @exp exp))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Strategy Definitions                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Skip Verification
(defn skip-check
  [contract dterm blame]
  (extract dterm))

(def skip  (Strategy. "skip" skip-check))

; Eager Verification
(defn eager-check
  [contract dterm blame]
  (contract (extract dterm) blame))

(def eager (Strategy. "eager" eager-check))

; Semi Verification
(defn semi-check
  [contract dterm blame]
  (delay (contract (extract dterm) blame)))

(def semi  (Strategy. "semi-eager" semi-check))

; Prom Verification
(defn future-check
  [contract dterm blame]
  (future (contract (extract dterm) blame)))

(def futur  (Strategy. "future" future-check))

; Conc Verification
(defn conc-check
  [contract dterm blame]
  (do (a/go (contract (extract dterm) blame)) (extract dterm)))

(def conc (Strategy. "conc" conc-check))

; Spot-Checking Verification (functions only)
(defn spot-check
  [generator]
  (fn [contract dterm blame]
    (let [f (extract dterm)]
      (if (not (fn? f))
          (throw (Exception. (str f " is not a function"))))
      (let [c (contract f blame)]
        (doall (map (fn [x] (c x)) (gen/sample generator 20)))
        f))))

(defn spot
  "spot-checker strategy"
  [g]
  (Strategy. "spot" (spot-check g)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Metastrategy Definitions                                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; With-Operator Verification
(defn with-check
  [fun]
  (fn [contract dterm sub-strat blame]
    (let [val (extract dterm)
          res (mon contract sub-strat val blame)]
      (do (fun (mon contract sub-strat val (indy-blame blame)))
          res))))

(defn with
   "with meta-strategy, expects a function and a strategy"
   [fun strat]
   (Metastrat. "with" (with-check fun) strat))

; Random Verification
(defn random-check
  [rate]
  (fn [contract dterm sub-strat blame]
      (if (< (rand) rate)
          (mon contract sub-strat (extract dterm) blame)
          (extract dterm))))

(defn random
  "randomizer meta-strategy, excepts a monitor rate and a strategy"
  [rate strat]
  (Metastrat. "rand" (random-check rate) strat))

; Communicating Verification
(defn comm-check
  [channel fun]
  (fn [contract dterm sub-strat blame]
    (let [val (extract dterm)
          res (mon contract sub-strat val blame)]
    (do (a/put! channel (fun (mon contract sub-strat val (indy-blame blame))))
        res))))

(defn comm
   "communication meta-strategy, expects a channel and a strategy"
   [chan fun strat]
   (Metastrat. "comm" (comm-check chan fun) strat))

; Memoizing Verification
(defn memo-check
  [contract dterm sub-strat blame]
    (mon (memoize contract) sub-strat (extract dterm) blame))

(defn memo
  "memoizer meta-strategy, excepts a ref to a map and a strategy"
  [strat]
  (Metastrat. "memo" memo-check strat))

; State Contracts
(defn make-contract-state
  [start-state]
  (ref (list start-state)))

(defn in?
  [coll elm]
  (some #(= elm %) coll))

(defn transition-check
  [state-ref from-state to-state]
  (fn [contract dterm sub-strat blame]
    (let [res (mon contract sub-strat (extract dterm) blame)]
      (if (in? (deref state-ref) from-state)
          (dosync
            (ref-set state-ref
                     (if (list? to-state) to-state (list to-state)))
            res)
          (throw
            (Exception.
              (str "Program performed invalid state transition:\n"
                   "  Current state: " (deref state-ref)
                   "  Transition: " from-state " -> " to-state "\n")))))))

(defn transition
  [state-ref from-state to-state strat]
  (Metastrat. (str "transition" from-state to-state)
              (transition-check state-ref from-state to-state)
              strat))

(defn transition-as-check
  [state-ref transition-fn]
  (fn [contract dterm sub-strat blame]
    (let [res (mon contract sub-strat (extract dterm) blame)
          to-state (transition-fn (deref state-ref) res)]
      (if (not (= to-state :error))
          (dosync
            (ref-set state-ref
                     (if (list? to-state) to-state (list to-state)))
            res)
          (throw
            (Exception.
              (str "Program performed invalid operation:\n"
                   "  Current state: " (deref state-ref) "\n")))))))

(defn transition-as
  [state-ref transition-fn strat]
  (Metastrat. (str "transition-as")
              (transition-as-check state-ref transition-fn)
              strat))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Contract Definitions                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defn pretty-demunge
  [fn-object]
  (let [dem-fn (m/demunge (str fn-object))
        pretty (second (re-find #"(.*?\/.*?)[\-\-|@].*" dem-fn))]
    (if pretty pretty dem-fn)))

(defn predc
  "Build a predicate contract"
    [f]
      (fn [x blame]
        (if (f x)
            x
            (throw (Exception.
                     (str "Contract violation: " x
                          " violated " f "\n"
                          "Blame: " blame))))))

(def anyc (predc (fn [x] true)))

(def natc (predc (fn [x] (>= x 0))))

(defn pairc
  "Build a predicate contract"
    [c1 c2 s]
    (fn [pair blame]
        [(mon c1 s (first pair) blame)
         (mon c2 s (second pair) blame)]))

(defn con-ravel
  [args ins]
  (if (empty? ins)
      (list)
      (cons (concat (take 2 args) (list (first ins)))
            (con-ravel (drop 2 args) (rest ins)))))

(defn func
  "Build a function contract"
  [& scs]
  (fn [f blame]
      (fn [& ins]
        (let [l  (* 2 (count ins))
              cl (count scs)]
          (if (not (= (+ 2 l) cl))
            (throw (Exception. "Invalid number of arguments for contracts")))
          (let [mon-sets (con-ravel scs ins)
                posts    (drop l scs)]
            (mon (first posts)
                 (second posts)
                 (apply f
                                (map (fn [x]
                                         (mon (first x)
                                              (second x)
                                              (second (rest x))
                                              (invert-blame blame)))
                                     mon-sets))
                         blame))))))
```

Cameron Swords — Curriculum Vitae

cameronswords@gmail.com — *http://cswords.com*

Research Interests

I am a computer scientist studying programming technology for programmatic analysis and performance. My expertise is in all aspects of programming language implementation and design, particularly for runtime verification, effectful and concurrent computation, and compiler implementation. My previous work includes: (1) concurrency-based models of runtime analysis to ensure program correctness; (2) meta-programming facilities for programming languages, with a focus on hygienic macros; (3) modeling programmatic effects in semantic models; and (4) exploring logic-based programming mechanisms.

Education

Indiana University, Bloomington, IN — 2011–2019
Ph.D. in Computer Science, *February 2019*
Dissertation: *A Unified Characterization of Runtime Verification Systems as Patterns of Communication.*
Advisor: Amr Sabry
Committee: Sam Tobin-Hochstadt, Jeremy Siek, Lawrence S. Moss.

M.S. in Computer Science, *March 2016*

Oregon Programming Language Summer School — 2012

Trinity University, San Antonio, TX — 2007–2011
Bachelor of Science in Computer Science (with Honors), *May 2011*
Thesis: *Pocketwatch: A Parallel Language.*
Advisor: Maurice Eggen

Employment History

Indiana University, Bloomington, IN — 2011–2018
Associate Instructor (with Amr Sabry), *Spring 2016–Spring 2017*

Graduate Research Assistant (with Amr Sabry), *Fall 2012–Spring 2016*

Associate Instructor (with Daniel P. Friedman), *Spring 2012–Fall 2012*

Mozilla, San Francisco, CA — Summer 2016
Research Intern on the Rust Programming Language team (with Nick Cameron)

Southwest Research Institute, San Antonio, TX — Summer 2011
Graduate Student Intern with the Defense & Intelligence Solutions Division

Trinity University, San Antonio, TX — Summer 2009, 2010
Research Assistant (with Mark Lewis)

Publications

Conference & Journal Publications

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. An extended account of contract monitoring strategies as patterns of communication. *Journal of Functional Programming*, 28, 2018.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, 2017. ACM.

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. Expressing contract monitors as patterns of communication. In *International Conference on Functional Programming*, 2015.

Mark Lewis and Cameron Swords. Lock-graph: A tree-based locking method for parallel collision handling with diverse particle populations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 11, pages 157–161, 2011.

Workshop Publications

Jason Hemann, Cameron Swords, and Lawrence Moss. Two advances in the implementations of extended syllogistic logics. In *Workshop on Natural Language Processing and Automated Reasoning*, 2015.

Cameron Swords and Dan Friedman. rKanren: Guided search in minikanren. In *Scheme and Functional Programming Workshop*, 2013.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Haskell Symposium*, 2013.

Theses

A Unified Characterization of Runtime Verification Systems as Patterns of Communication — February 2019
Ph.D. dissertation

Pocketwatch: A Parallel Language — May 2011
Honors thesis

Selected Talks

Procedural Macros in Rust — July 2016
Mozilla Corporation, San Francisco, CA.

Expressing Contract Monitors as Patterns of Communication — September 2015
International Conference on Functional Programming (ICFP '15), Vancouver, BC.

Lightning Talk: JITs for Haskell — September 2015
Haskell Implementors Workshop (HIW '15), Vancouver, BC.

rKanren: Guided search in miniKanren — November 2013
Scheme and Functional Programming Workshop, Alexandria, VA.

CUDA Event-Driven Particle Simulations — April 2010
National Conferences on Undergraduate Research (NCUR '10), Missoula, MT.

Software Projects

Pisces — *github.com/cgswords/dissertation*
A software contract library for Clojure with a focus on multi-strategy and meta-strategy contracts, recreating my dissertation work in a real-world programming language.

Preliminary proc_macro Implementation — *Rust Programming Language*
Implemented a preliminary version of procedural macro programming facilities for the Rust programming language, including a token-stream representation of input syntax and meta-programming tools for syntactic macro writers (i.e., a syntactic quasiquoter). This work was merged into the Rust language as part of proc_macro crate.

Scheme-to-x86_64 Compiler *github.com/cgswords/sgc*
A nanopass-style, functional compiler written in Scheme, complete with tail-call optimization and run-time garbage collection.

rKanren *github.com/cgswords/rkanren*
A uniform-cost search dialect of miniKanren, a declarative, logic programming language embedded in Scheme.

Teaching

Teaching Assistant, Programming Language Foundations Spring 2017
Graduate course on programming language theory. Taught by Amr Sabry.

Teaching Assistant, Intro to Computer Science (Honors) Fall 2016
Undergraduate course on introductory programming. Taught by Amr Sabry.

Teaching Assistant, Discrete Structures for Computer Science (Honors) Spring 2016
Undergraduate course on discrete mathematics and formal logic. Taught by Amr Sabry. (Unofficial Position)

Teaching Assistant, Principles of Programming Languages Fall 2012
Graduate course on programming language design. Taught by Daniel P. Friedman. (Unofficial Position)

Teaching Assistant, Principles of Programming Languages Spring 2012
Undergraduate course on programming language design. Taught by Daniel P. Friedman.

Service

External Reviewer for Haskell Symposium 2014, PEPM 2014, COMLAN 2016

External Subreviewer for ICFP 2016, POPL 2017

Organizer, Indiana University PL Colloquium Series 2013–2017

Administrator, Indiana University PL Colloquium Mailing List and Website 2013–2017

Select Graduate Coursework

Programming Language Theory	Compiler Design and Implementation
Meta-programming	Formal Verification
Theory of Computation	Domain Specific Languages
Set Theory	Recursion Theory
Software Engineering	Operating Systems