Under consideration for publication in J. Functional Programming

An Extended Account of Contract Monitoring Strategies as Patterns of Communication

CAMERON SWORDS and AMR SABRY and SAM TOBIN-HOCHSTADT
Indiana University, Indiana, United States
(*e-mail:* {cswords,sabry,samth}@indiana.edu)

Abstract

Contract systems have come to play a vital role in many aspects of software engineering. This has resulted in a wide variety of approaches to enforcing contracts—ranging from the straightforward precondition and postcondition checking of Eiffel to lazy, optional, and parallel enforcement strategies. Each of these approaches has its merits, but each has required ground-up development of an entire contract monitoring system.

We present a unified approach to understanding this variety, 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 enforcement strategies, including strict and lazy enforcement as well as concurrent approaches, including new contracts and strategies. We show how each of these can be embedded into a core calculus, and demonstrate a proof (via simulation) of correctness for one such encoding. Finally, we show that our approach suggests new monitoring approaches and contracts not previously expressible.

1 Introduction

Behavioral software contracts, originally introduced by Meyer (1992), have become an integral part of modern programming practice, where they are used to specify and ensure program correctness. In the first-order, functional setting, programmers write predicates (i.e., functions that take an input and return a Boolean value as a result) to check program properties at function boundaries: the pre-condition predicate is applied to the function input; the function is run on its input (if the pre-condition predicate holds); and the post-condition predicate is applied to the function result. If either predicate returns false, the program terminates with an error.

Programming, however, is more complex: programs utilize higher-order functions, effectul operations, massive data structures, lazy evaluation mechanisms, and more. This added complexity must be addressed by the contract system: for higher-order functions, we must delay contract enforcement (Findler & Felleisen, 2002); 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.

Researchers have proposed myriad contract enforcement strategies in response to these rising complexities, including lazy monitors (Degen *et al.*, 2009; Chitil, 2012), concurrent

monitoring systems (Dimoulas *et al.*, 2009), optional enforcement (Dimoulas *et al.*, 2013), and probabilistic contract verification (Clojure, 2017; Moore *et al.*, 2016).

While many of these systems initially appear incompatible, each of these approaches to contract enforcement 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 original program fragment. These two pieces of code proceed independently, synchronizing at specific points. Prior contract designs blur this distinction, fixing the evaluator interaction pattern in the host language, providing a *single* enforcement strategy to the programmer. If we preserve this distinction, however, these variations on evaluator interaction may be directly encoded, allowing programmers to vary monitor behavior on a *per-contract* basis, choosing the appropriate behavior for each contract.

Contributes & Outline. This paper extends and refines the work presented by Swords *et al.* (2015) in the following ways: we simplify the underlying calculus presented by Swords *et al.* (2015), eschewing process tags (which governed termination behavior); present a modified encoding of contracts as patterns of communication which ensures each monitored term is evaluated in the correct process (whereas Swords *et al.* (2015) always evaluate monitored terms in the monitoring process) include a number of additional examples; introduce an additional monitoring strategy, **fconc**, for finally-concurrent monitoring; and present an extended proof that our **eager** verification strategy simulates the λ^{CON} calculus presented by (Findler & Felleisen, 2002).

The paper proceeds as follows: we describe and discuss previous contract verification variants (Section 2); describe how these variations may be verified by viewing contract monitors as separate evaluators (Section 3); present a concurrent process calculus as a unified system encoding multiple contract verification strategies (Section 5); encode a number of modern verification strategies in this framework (Section 6); demonstrate the multi-strategy approach to contract monitoring (Section 7); present a simluation of Findler & Felleisen (2002) as eager verification (Section 8); and present related works and conclude.

2 Background

In the conventional study of software contracts, a language designer extends a core calculus with monitoring facilities that adhere to a specific *monitoring strategy*, describing how contract verification interacts with the user program in precise semantics. This semantic specification is a permanent fixture of the language, dictating the behavior of contract verification across the entire program.

While modern contract software verification literature introduces a slew of such specifications (Findler & Felleisen, 2002; Hinze *et al.*, 2006; Degen *et al.*, 2009; Dimoulas & Felleisen, 2011; Disney *et al.*, 2011; Clojure, 2017; Dimoulas *et al.*, 2013; Moore *et al.*, 2016; Ergün *et al.*, 1998; Keil & Thiemann, 2015; Chitil, 2012; Chitil *et al.*, 2003), each variation makes distinct decisions about the various trade-offs for verification, including:

- Should we treat contracts as specifications we must verify, ensuring values adhere to these contracts regardless of their usage in the program?
- Should the user program wait while verification occurs?
- Should the user program have a role in verification (i.e., retrieving contract results)?

These questions do not have definitive "yes-or-no" answers, but each represent a gradient of answers has led to a number of semantic specifications for verification monitors that each take different stances on these questions.

For example, the option contract system presented by Dimoulas *et al.* (2013) suggests that we should *sometimes* treat contracts as specifications we must verify, waiting while we do, and the user program plays a role in verification by choosing to postpone or eschew contracts. At the other end of the spectrum, we can eschew complete verification and removing the user's role in verification, but still suspend the program to check contracts, while we get spot-checking systems similar to those described by Ergün *et al.* (1998).

To further explore these variations, we introduce a language (similar to the core calculus presented by Degen *et al.* (2009) and *CPCF* presented by Dimoulas & Felleisen (2011)):

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

Most of these operations act as expected: we have variables x, values V (including contracts), lambda abstractions and application, conditional branching, binary operations, pairs with accessors, and errors (which raise "empty" errors for now, to simplify presentation—they would normally carry blame information). E also includes:

- *monitoring* **mon** E_1 E_2, which installs a runtime monitor that verifies the contract E_1 on the expression E_2 such that the monitor either returns the evaluated term (with potentially embedded monitors[1]) or raises an error;
- the *predicate contract combinator* **pred/c** E, where E must be a predicate function (i.e., returns a boolean based on a single argument);
- and the *pair contract combinator* **pair/c** E_1 E_2, where E_1 and E_2 are subcontracts to check on the first and second elements of the monitored pair (respectively).

We may use these combinators to construct new contracts, such as a contract that verifies its input is a natural number, and, further, use this definition and the pair contract combinator to define a contract that will verify its input is a pair of natural numbers:

$$\textbf{nat/c} := \textbf{pred/c}\ (\lambda n.\ n \geq 0) \tag{1}$$

$$\textbf{nat-pair/c} := \textbf{pair/c nat/c nat/c} \tag{2}$$

Then, we may ask: what does it mean to evaluate **mon nat/c** 5? What about **mon nat-pair/c** $(5,\text{-}1)$? To determine the answer, we must fix a *semantic contract verification strategy*, choosing how to answer each of the questions above (and, as we shall see in Section 2.4, any further questions these answers may raise).

Next, we examine the behavior of five different contract verification strategies, which each take different positions on these questions. We have selected these five because they are either well-represented in the literature (in the case of eager, semi-eager, and promise-based checking), have elegant encodings into our model (which directly correspond to the

[1] As we will see, contracts over structures may embed additional monitors in the resultant term that are enforced as the evaluator explores the structure itself.

ownership flow model identified by Dimoulas & Felleisen (2011)), or explore a various possible answers to our questions about the nature of verification. These are not, however, the only possible strategies; we discuss a number of additional strategies in Section 9.

2.1 Eager Verification.

Eager contract enforcement for software contracts, first presented by Meyer (1992), brought into the functional world by Findler & Felleisen (2002), and repeatedly refined (Findler & Blume, 2006; Degen *et al.*, 2009; 2010; Dimoulas *et al.*, 2011; Ou *et al.*, 2004; Flanagan, 2006; Greenberg *et al.*, 2010), presents the idea that contracts are *fully-verified* specifications, over-evaluating their input to ensure the contract holds while the user program waits for the verification result. To demonstrate this behavior, consider eagerly monitoring **nat/c**:

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

When the evaluator encounters the monitoring form (line 3), the user program (attempting to apply $(\lambda x.\ 10)$ to $(2\ +\ 3)$) is suspended while the monitor evaluates its input and verifies the contract. After the monitoring reduction ensures the contract holds, it yields control back to the user portion of the program with the *monitored value* (in this case, 5). If 5 had violated the contract, the monitor would subvert the user program to raise an error.

This contract enforcement strategy treats contracts as concrete specifications to *fully* verify, regardless of the program's execution trace (e.g., how it uses the contracted values). For example, consider taking the first element of a **nat-pair/c**-contracted pair:

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

Enforcing a pair contract with an eager verification strategy immediately and completely enforces each subcontract. In this case, even though the second element of the pair (-1) is unused in the program's result, the contract still detects that it is not a natural number and signals an error.

As we can see, eager contract verification has a number of drawbacks:

- Treating contracts as fully-verified specifications may inhibit 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.
- Suspending the user program while the monitor proceeds can be computationally inhibitive. For example, consider

  $$\textbf{mon prime/c}\ (237^{63} + 567) \tag{3}$$

  The user program waits while the contract monitor performs this check, which may bottleneck applications. 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 check many rich properties of their programs.

- Interrupting the user evaluator and over-evaluating inputs may produce different program behavior, and may not always preserve the underling program's meaning (Owens, 2012). For example, consider the following predicate (meant to be monitored on a function):

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

  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: that contracts are concrete specifications that *must* be totally enforced as the user program encounters them.

2.2 Semi-Eager Verification.

The over-evaluation in eager verification suggests an immediate alternative: we may maintain the user program suspension during verification, but postpone each individual contract's verification until the program demands the contracted value. This *semi-eager* verification strategy, originally presented by Hinze *et al.* (2006) and later refined (Degen *et al.*, 2009; Findler *et al.*, 2008; Degen *et al.*, 2010; Chitil, 2012; Dimoulas & Felleisen, 2011), specifies that only those values the program uses will have their contracts verified.

To this end, the monitor reduction "boxes up" the contract and the monitored expression as a value, suspending contract enforcement. When the user evaluator demands the value (e.g., the boxed expression occurs in evaluation position), the evaluator suspends the user program and enforces the contract, after which the program resumes with the monitored value (if the contract holds) or raises an error (if it does not).

To illustrate this behavior, consider evaluating the previous pair example with semi-eager monitoring (where we represent contract-expression boxes as $\langle \mathit{contract} \mid \mathit{expression} \rangle$ and reduction handles these as special forms when they occur in evaluation positions):

$$\begin{array}{ll} & \textbf{fst}\ (\textbf{mon nat-pair/c}\ (5, \text{-}1)) \\ \rightarrow^* & \textbf{fst}\ \langle \textbf{nat-pair/c} \mid (5, \text{-}1) \rangle \\ \rightarrow^* & \textbf{fst}\ (\textbf{mon nat/c}\ 5, \textbf{mon nat/c}\ \text{-}1) \\ \rightarrow^* & \textbf{fst}\ (\langle \textbf{nat/c} \mid 5 \rangle, \langle \textbf{nat/c} \mid \text{-}1 \rangle) \\ \rightarrow & \langle \textbf{nat/c} \mid 5 \rangle \\ \rightarrow & 5 \end{array}$$

This example illustrates the primary behavioral difference from eager verification: in semi-eager verification, contracts are no longer strict specifications, but any value the program uses is correctly monitored. Such enforcement may prove invaluable: we may check *only those values we use* out of a infinite stream or hash-map, preserving program behavior, performance, *and* localized contract verification. Even so, semi-eager verification has its own drawbacks:

- First, semi-eager enforcement is not faithful to the contract specification (Degen *et al.*, 2009): we will not detect errors in unused values, and thus cannot always trust

our contracts as full specifications. This may also lead to programmer evaluating values, such as taking the unused second element of the pair, expressly for contract enforcement.

- Second, semi-eager enforcement is not idempotent. Degen *et al.* (2009) observe verifying a contract in semi-eager verification may inadvertantly cause other, pending verifications to be performed on that value. In this situation, an unused value that violates its contract may raise an error if that contract is applied a second time.
- Finally, this verification technique still suspends the user evaluator while verification proceeds, and thus some contracts may still prove too expensive.

As we can see, semi-eager verification is also not a catch-all solution.

A Remark on Function Contracts. Function 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 an infinite input space). To avoid this problem, Findler & Felleisen (2002) propose an alternative approach to function contracts wherein a function contract yields a *new* function, wrapping the input function in pre- and post-condition checks:

$$\textbf{mon}\ (\textbf{fun/c nat/c nat/c})\ (\lambda n.\ n\ +\ 5) \rightarrow \lambda x.\ \textbf{mon nat/c}\ ((\lambda n.\ n\ +\ 5)\ (\textbf{mon nat/c}\ x))$$

This definition suggests that function contracts are *natively* semi-eager, utilizing the implicit delaying nature of λ to check values as they flow in and out of the function (since checking the function's entire domain and range is, in general, impossible). Other, more "eager" approaches would entail either probabilistic or static analysis for verification (Ergün *et al.*, 1998; Xu *et al.*, 2009; Nguyen *et al.*, 2014). We postpone discussing these alternatives until Section 9 in order to keep function contracts similar, in nature, to their structural counterparts: each takes some structure as input and returns the same structure with contracts nested in it.

2.3 Promise-Based Verification.

In order to address the concern of evaluator interruption, Dimoulas *et al.* (2009) introduce the notion of *Future Contracts*, presenting a concurrency model with a user process and a monitoring process wherein the user process communicates contracts and expressions to the monitoring process and the monitoring process concurrently performs contract verification, reporting errors to the user process at pre-determined synchronization points (Dimoulas *et al.* (2009) choose effectful operations to perform this synchronization).

We may utilize a similar approach with inter-process communication and computational *promises* (Friedman & Wise, 1976), written $\langle\!\langle e \rangle\!\rangle$ where e is the expression to retrieve the result when the promise is demanded by the user process. During monitor insertion, the user evaluator communicates the contract and expression to a concurrent process and resumes its computation while the concurrent monitoring process proceeds with verification and communicates the result, fulfilling the computational promise.

For example, consider a program that expects a sorted list, generates a new encryption key, and then uses that key to encrypt the list. If generating a new encryption key is computationally intensive, it may be worthwhile to ensure the list is sorted concurrently[2]:

	User Process		*Promise Process*
	let *list* = **mon sorted/c** *ls*		
	in *encrypt* (*gen-new-key*) (**retrieve** *list*)		
$\rightarrow^*$	**let** *list* = ⟦**read** ι⟧		**if** (*sorted?*) *ls*
	in *encrypt* (*gen-new-key*) (**retrieve** *list*)		**then write** ι *ls*
			else write ι **error**
$\rightarrow^*$	*encrypt* (*gen-new-key*) (**retrieve** ⟦**read** ι⟧)	$\rightarrow^*$	**write** ι **error**
$\rightarrow^*$	*encrypt V* (**retrieve** ⟦**read** ι⟧)		
$\rightarrow^*$	**error**		

Similar to semi-eager verification, this promise-based verification model forces promises when they occur in evaluation position and, as a result, only contract results used in the final program output are communicated to the user evaluator.

Unlike eager and semi-eager verification, the user evaluator is not interrupted during monitoring, and may proceed with its own computation separate from contract verification. As Dimoulas *et al.* (2009) observe, 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.

The decision to eschew contracts as concrete specifications and remove monitoring suspensions, unfortunately, has its own potential issues:

- First, promise-based verification falls victim to many of the previous concerns of semi-eager verification (including lack of idempotence and verification anti-patterns).
- Second, the user evaluator may end up "wasting cycles" on speculative computation, performing operations that are unused after a contract signals an error (or worse, must be rolled back, such as in the case of side effects such as writing data to a file).
- Third, the cost of communication and promises may dominate program performance.
- Finally, effectful contracts (e.g., a contract that maintains internal state) no longer have a guaranteed execution order, and may yield unpredictable results.

As with eager and semi-eager verification before it, we can see that promise-based verification is also not a perfect-fit solution.

2.4 Concurrent Verification.

In an attempt to relax and address some of the semantic complexity of the previous verification approaches, we now consider an alternative, concurrent verification technique that forgoes reporting the result to the user evaluator. Instead, the monitoring process enforces

[2] In a real-world system, we would not write an error across a channel; we take this liberty here for presentation purposes

the contract concurrently, either completing silently or detecting and reporting an error (and halting the entire computation). This concurrent enforcement may proceed as:

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

This program will either result in 5 or raise an error; which should we expect? By removing the secondary synchronization point, we expose a further verification question:

> 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 **raise** as the final answer, dependent upon process scheduling. While ultimately weaker than the previous verification techniques, this best-effort approach may be ideal for enforcing expensive properties (such as probabilistic primality checking) during a program. This concurrent verification strategy, however, has its own pitfalls:

- First, concurrent verification may prove too weak to be reliable: a best-effort approach to verification may inhibit programmers from reliably ensuring program properties. For example, a probabilistic primality check contract may not find a counter-example, even if the number is not prime.
- Second, scheduler-dependent results may only detect contract violations on some program executions.
- Finally, effectful contracts may have unpredictable behavior.

To address the first two issues, we may instrument the language runtime to wait for these monitoring processes to run to completion before reporting the result of the user evaluator. This alternative, "finally-concurrent" verification approach recovers the guarantees lost in concurrent verification by ensuring that every contract we monitor is fully enforced before the user program terminates. In this above example, this means the error will be *always* reported before the user program can yield 5 as an answer. Even so, this finally-concurrent approach has its own issues:

- First, finally-concurrent verification yields a similar flavor of over-evaluation as eager monitoring: enforcing a contract on a stream will never terminate, and as a result, it is possible to create contract monitoring processes that never terminate.
- Second, effectful contracts suffer the same problems that concurrent and promise-based verification exhibit.

3 Unifying Variations on Verification

Each of the contract verification strategies we have seen have their own strengths and weaknesses based on how, precisely, each chooses to interpret and answer our verification

questions. In light of these pros and cons, Degen *et al.* (2009) declare that "faithfulness is better than laziness" for lazy languages, advocating that concrete contract assurances are more valuable than any other properties. Conversely, Findler *et al.* (2008) identify a number of contracts where semi-eager enforcement is critical to maintain standard program asymptotics.

Swords *et al.* (2015) suggest that, for the practical programmer, none of these strategies is going to fit every use case in a program, and that programmers should be able to choose their verification strategy on a contract-by-contract basis to address different needs 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 in the same program:

- via eager monitoring after initial construction, to ensure that a *list-to-tree* procedure produces the correct structure;
- via semi-eager monitoring, to check that each element touched 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 is sorted.

To facilitate this flexibility, we must give programmers a mechanism to select which strategy they would like at a per-contract basis, and move between them freely. To this end, we introduce a new **check** form:

$$E := ... \mid \textbf{check}\ E\ E\ E\ E$$

As with the **mon** operator, the **check** operation takes a contract and an expression to monitor. In addition to these arguments, **check** also takes a *strategy* argument – a value describing which enforcement strategy the monitor should use:

$$V := \quad ... \mid S \qquad S := \quad \textbf{eager} \mid \textbf{semi} \mid \textbf{prom} \mid \textbf{conc} \mid \textbf{fconc}$$

We may use these strategies in combination with the previously-defined contracts to produce each monitoring behavior. This revised **check** form also takes a blame argument b, which we elided for **mon**. This blame argument is the standard, *indy*-style three-tuple (Dimoulas *et al.*, 2011), and thus we elide its precise definition and further explanation. Blame inversion and dependent-style blame follow directly.

To demonstrate its immediate utility, consider a **bst/c** contract over binary trees, parameterized by a strategy indicating how to enforce each recursive check. This parameterization will allow us to write each of the contracts described above:

- **check eager** (**bst/c eager**) *tree B* will eagerly ensure that a tree is a binary-search tree;
- **check semi** (**bst/c semi**) *tree B* will return a tree that will check that each subtree is correctly-ordered as it is explored;
- **check prom** (**bst/c prom**) *tree B* will create a cascading chain of monitoring processes for each node, and exploring the tree will synchronize with the appropriate processes at each level;
- **check conc** (**bst/c conc**) *tree B* will concurrently enforce that the tree is a binary-search tree using a similar set of cascading processes, but as a "best-effort" check;

- and **check fconc** (**bst/c fconc**) *tree B* will do the same as the **conc** contract, but force the program to wait for it to complete before terminating.

These are not the only possibilities, however: the strategy argument to **check** describes how the *top-level* monitor should behave while the strategy argument to **bst/c** describes how each recursive contract (i.e., the contract applied to each child node) behaves. This fine-grained control allows us to freely intermix strategies to produce additional verification patterns. For example, we may create a single promise-contained check that eagerly enforces each subcontract as **check prom** (**bst/c eager**) *tree B*.

4 Unifying Verification

Our strategy descriptions thus far appear to be specialized semantic forms, each designed from the ground up to provide unique behavior. On the surface, this suggests that each strategy requires unique implementation facilities. This perception, however, is incorrect.

While it may be possible to encode each strategy as a custom, unique entity as part of a specialized case for **check**, each of these strategies is, fundamentally, a small variation on the same theme: a monitoring evaluator subverts the user program, suspending it (or working concurrently) while it enforces the contract on the monitored expression. Each strategy variation directly corresponds to *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 idea, consider evaluating the expression

$$5 + \textbf{check eager nat/c}\ (1+2)$$

First, the user program will evaluate $(1+2)$, yielding 3. Next, the monitoring expression **check eager nat/c** 3 will suspend the user program while the monitoring evaluator ensures that 3 is a natural number. Finally, the monitoring expression yields 3 and the user evaluator resumes, evaluating $5+3$ to 8 as the final result. This derivation is presented in the top half of Figure 1, where we have indicated the user portions of the evaluation in blue and the monitoring portions in red. This value flow between evaluators is reminiscent the ownership model described by Dimoulas *et al.* (2011), where we account for the contract itself as an additional party.

Further, observe that we may separate out the monitoring portion of the program from the user portion and explicitly model these evaluators and interactions (Disney *et al.*, 2011; Swords *et al.*, 2015). We can apply this separation to our previous example, yielding the derivation in the bottom half of Figure 1, where the user and monitoring programs *explicitly interact* to compute the final result.

This revised derivation reveals the fundamental nature of contract monitoring: a contract monitor is a *separate evaluator* communicating with the user program. And now we may *vary* the pattern of communication to produce our varied contract enforcement strategies. This explicit account of interactions allows us to to explore the enforcement design space, expressing contract verification strategies in a single, unified framework using evaluator interactions in order to examine their unique behavior and interactions in a uniform system.

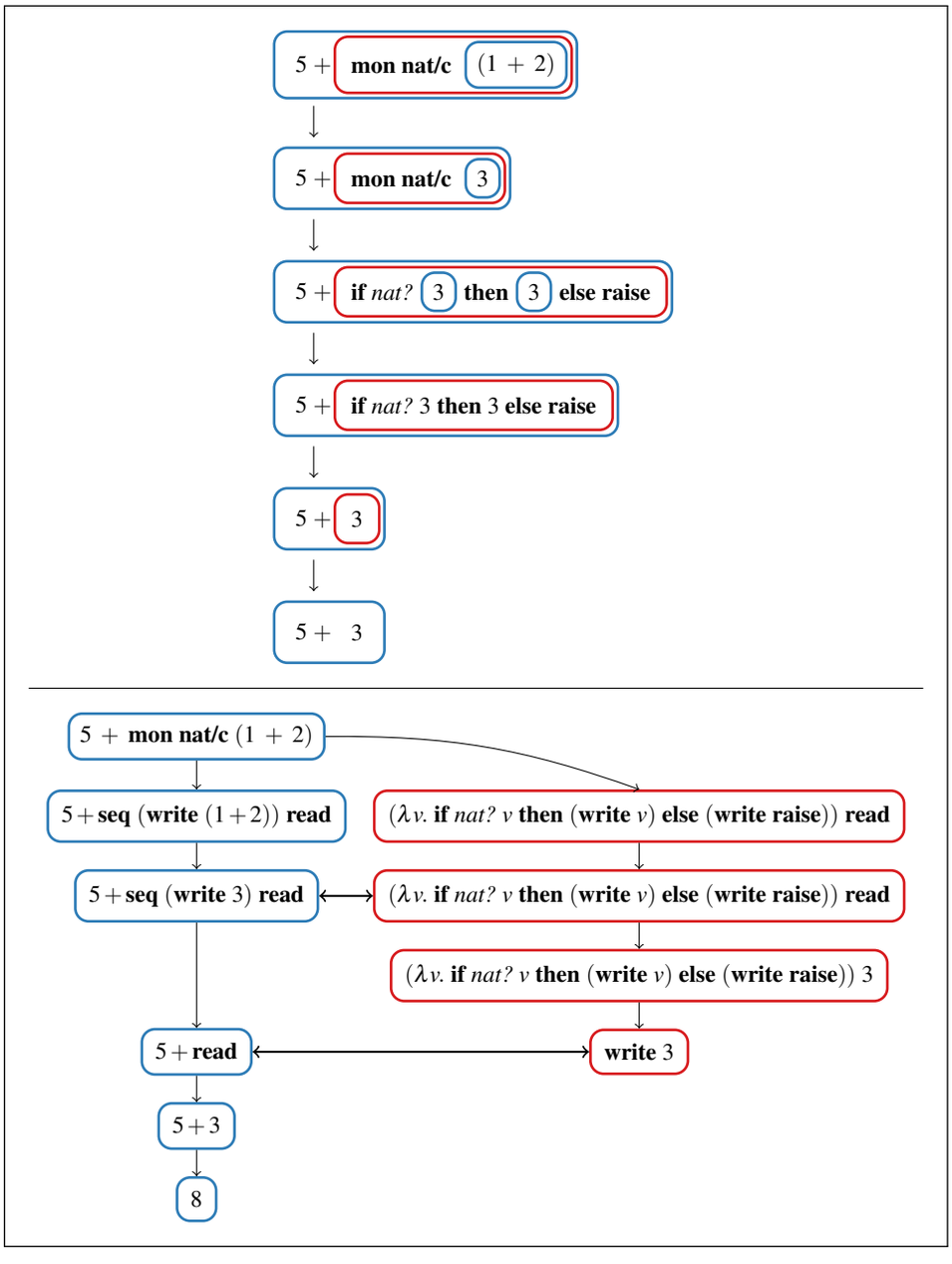

Fig. 1: Separating an eager, flat contract into a pattern of communication. (We take *nat?* to mean $\lambda x.\ x \geq 0$ to simplify our presentation.) The first image depicts a single evaluator indicating the different evaluation components of a software contract system, where the user components are colored blue and the software contract system component is colored red. The second image depicts peforming verification in an explicitly separate evaluator.

5 A Calculus for Contract Monitors

To solidify this notion of monitors-as-communication, we first need a calculus that will let us reason about multiple evaluators and their interactions. We start with a calculus based on Concurrent ML (Reppy, 1999; Jeffrey, 1998; Reppy, 1993), given in Figures 2 and 3, including a term language e with values v and term reduction "$\longmapsto$", and a concurrent reduction relation "$\Rightarrow$" which extends term reduction to a finite set of interacting processes. This calculus is, in general, unremarkable: as with most modern programming language, our calculus supports process creation and communication events, raising and catching errors, and delaying and forcing individual terms.

Term Language. Expressions e include *variables*, *values* (including λ-abstractions and *communication channels* ι), application, and a number of canonical operations (Pierce, 2002). A full type system, complete with a proof of type soundness, is provided as an electronic appendix[3]. Further, we elide the definition of δ, which is a partial function that encodes binary and unary operator behavior. We also include:

- **delay**, a delaying construct that produces thunk-like objects (Ingerman, 1961);
- **force**, which evaluates delayed expressions (and any other term to a value);
- **raise** and **catch**, specialized to blame values:
  - — if **raise** v occurs without an error handler (i.e., in a D-CONTEXT), we discard the surrounding context;
  - — if **raise** v occurs under a handler **catch** v_1 $D[\textbf{raise}\ v]$, we discard the intermediate, non-catching D-CONTEXT and apply the handler to the error argument (as $v_1\ v$);
  - — if the evaluation under an error handler terminates with a value v, we discard the handler and proceed with v.
- and opaque blame values B, which represent blame information such as positive, contract, and negative positions (following Dimoulas *et al.* (2011)).

We postpone defining **check** and our contract combinators until Section 6.

Processes & Communication. The concurrent evaluation relation "$\Rightarrow$" extends the "$\longmapsto$" relation with a finite set of processes (i.e., terms with associated process identification numbers) with additional rules to handle the process-level operations for channel creation (**chan**), process creation (**spawn**), and process communication (**read**, and **write**). We define process identification numbers π such that $\langle e \rangle_\pi$ constitutes a single process. A process configuration K, T, P has three components:

- K is a set of channel names for the configuration;
- T is a set of process identification numbers π that indicate the *termination set* of a configuration, allowing us to define *answer configurations*:

  Definition 5.1 (*Answer Configuration*)
  A configuration K, T, P is considered an *answer configuration* if, for every $\pi \in T$, there is a process $\langle e \rangle_\pi \in P$ and $e \not\longmapsto$.

[3] `http://cswords.com/jfp17-type-safety.pdf`

- and P denotes the set of processes in the configuration.

When convenient, we elide K and T from our traces and write $P + \langle e \rangle_\pi$ to mean $P \cup \{\langle e \rangle_\pi\}$ to simplify presentation. The process reduction rules proceed as:

- [PROCSTEP] describes internal process reduction, lifting the term evaluation relation "$\longmapsto$" to configurations.
- [SPAWN] and [FSPAWN] describe process creation, which selects a new process identification number π and create a new process with the provided expression. The originating process proceed with **unit**. In the case of [FSPAWN], the new process identification number is also added to T, ensuring the process will complete before the configuration is an answer configuration.
- [CHANNEL] describes channel creation, where we select a new channel name ι, add it to K, and continue with it in the process.
- [SYNCHRONIZE] describes process synchronization, defined in terms of *matched events*, where event matching is defined as

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

  in Figure 3. If two processes are matched, they may communicate as a reduction.

In $\lambda^{\Rightarrow}_{/c}$, processes are immortal—if a process evaluates to a value v, it will remain in the process set without further reductions.

6 Contracts as Patterns of Communication

With our core calculus in place, we may now define contract monitoring strategies in a single, unified system, expressing each in terms of the language primitives we have introduced in $\lambda^{\Rightarrow}_{/c}$. To begin, we define three contract combinators, **pred/c**, **pair/c**, and **fun/c**, to use for our discussion:

$$\textbf{pred/c} \triangleq \lambda\, pred.\ \lambda\, val\ blame.\ \textbf{if}\ pred\ val\ \textbf{then}\ val\ \textbf{else raise}\ blame \tag{4}$$

$$\begin{aligned}\textbf{pair/c} \triangleq\ & \lambda\, con_1\ strat_1\ con_2\ strat_2. \\ & \lambda\, pair\ blame.\ (\textbf{check}\ con_1\ strat_1\ (\textbf{fst}\ pair)\ blame, \\ & \qquad \textbf{check}\ con_2\ strat_2\ (\textbf{snd}\ pair)\ blame)\end{aligned} \tag{5}$$

$$\begin{aligned}\textbf{fun/c} \triangleq\ & \lambda\, con_1\ strat_1\ con_2\ strat_2. \\ & \lambda\, f\ blame. \\ & \lambda x.\ \textbf{check}\ con_2\ strat_2\ (f\ (\textbf{check}\ con_1\ strat_1\ x\ (invert\ blame)))\ blame\end{aligned} \tag{6}$$

In this encoding, a *contract* is a procedure that takes a value and a set of blame information, yielding a term of the same type or raising an error. As before, the **pred/c** contract combinator take a predicate and produces a contract that checks that predicate. Similarly, the **pair/c** contract combinator takes *two* contracts and associated strategies to enforce on each element of a pair (under the respective strategies), yielding a pair contract. Finally, the **fun/c** contract combinator takes two contracts and associated strategies and produces

Syntax

EXPRS	e	:=	$x \mid v \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$
		$\mid$	$e\ \mathit{binop}\ e \mid \mathit{unop}\ e \mid (e,e) \mid \textbf{fst}\ e \mid \textbf{snd}\ e$
		$\mid$	$\textbf{case}\ (e;\ \textbf{inl}\ x \rhd e;\ \textbf{inr}\ x \rhd e) \mid \textbf{inl}\ e \mid \textbf{inr}\ e$
		$\mid$	$\textbf{force}\ e \mid \textbf{raise}\ e \mid \textbf{catch}\ e\ e$
		$\mid$	$\textbf{spawn}\ e \mid \textbf{fspawn}\ e \mid \textbf{chan} \mid \textbf{read}\ e \mid \textbf{write}\ e\ e$
VALUES	v	:=	$\lambda x.\ e \mid \iota \mid \textbf{delay}\ e \mid (v,v) \mid \textbf{inl}\ v \mid \textbf{inr}\ v$
		$\mid$	$n \mid \textbf{true} \mid \textbf{false} \mid \textbf{unit} \mid B$
D-CONTEXTS	D	:=	$\Box \mid D\ e \mid v\ D \mid \textbf{if}\ D\ \textbf{then}\ e\ \textbf{else}\ e$
		$\mid$	$D\ \mathit{binop}\ e \mid v\ \mathit{binop}\ D \mid \mathit{unop}\ D$
		$\mid$	$(D,e) \mid (v,D) \mid \textbf{fst}\ D \mid \textbf{snd}\ D$
		$\mid$	$\textbf{case}\ (D;\ \textbf{inl}\ x \rhd e;\ \textbf{inr}\ x \rhd e) \mid \textbf{inl}\ D \mid \textbf{inr}\ D$
		$\mid$	$\textbf{force}\ D \mid \textbf{raise}\ D \mid \textbf{catch}\ D\ e$
		$\mid$	$\textbf{read}\ D \mid \textbf{write}\ D\ e \mid \textbf{write}\ v\ D$
$\mathscr{E}$-CONTEXTS	$\mathscr{E}$	:=	$D \mid D[\mathscr{E}] \mid \textbf{catch}\ v\ \mathscr{E}$
PROCID	π	$\in$	$\mathbb{N}$
PROCESS	proc	=	$\langle e_i \rangle_{\pi_i}$
PROC. SET	P	$\in$	$\mathit{Fin}(\mathit{proc})$
PROC. DECOMP.	$P + \langle e \rangle_\pi$	$\equiv$	$P \cup \{\langle e \rangle_\pi\}$
PROC. CONFIG.	K,T,P		$K \in \mathit{Fin}(\text{channel names})$ $T \in \mathit{Fin}(\text{process ids})$

Fig. 2: Syntax definitions for $\lambda^{\Rightarrow}_{/c}$.

a function contract. When we check this contract on some function f, the combinator yields a new function (to "stand in" for the previous function), checking the first contract, or *pre-condition*, on each input to this contracted f and checking the second contract, or *post-condition*, on each of f's results (Findler & Felleisen, 2002; Strickland *et al.*, 2012).

With these combinators in place, we turn our attention to the contract monitoring strategies described in Section 2, providing semantic definitions for each in terms of $\lambda^{\Rightarrow}_{/c}$.

6.1 Eager Contract Monitoring—Interrupting the User Evaluator

We begin with eager monitoring, where each contract is completely verified at assertion time. To model this strategy as a *pattern of communication*, the initiating process:

(E_{U1}) creates a new communication channel ι;
(E_{U2}) spawns a monitoring process that will evaluate the contract and communicate the result across ι;
(E_{U3}) provide the (evaluated) subject value to the monitoring process across ι;

Dynamic Semantics

$\boxed{e \to e'}$

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

$\boxed{e \longmapsto e'}$

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

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

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

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

$$\frac{e \longmapsto e'}{K, T, P + \langle e \rangle_\pi \Rightarrow K, T, P + \langle e' \rangle_\pi}\ [\textsc{ProcStep}]$$

$$\frac{\pi' \notin \mathrm{dom}(P)}{K, T, P + \langle \mathscr{E}[\textbf{spawn}\ e] \rangle_\pi \Rightarrow K, T, P + \langle \mathscr{E}[\textbf{unit}] \rangle_\pi + \langle e \rangle_{\pi'}}\ [\textsc{Spawn}]$$

$$\frac{\pi' \notin \mathrm{dom}(P)}{K, T, P + \langle \mathscr{E}[\textbf{fspawn}\ e] \rangle_\pi \Rightarrow K,\ \ T \cup \{\pi'\},\ \ P + \langle \mathscr{E}[\textbf{unit}] \rangle_\pi + \langle e \rangle_{\pi'}}\ [\textsc{FSpawn}]$$

$$\frac{\iota \notin K}{K, T, P + \langle \mathscr{E}[\textbf{chan}] \rangle_\pi \Rightarrow K \cup \{\iota\}, T, P + \langle \mathscr{E}[\iota] \rangle_\pi}\ [\textsc{Channel}]$$

$$\frac{e_1 \overset{\iota}{\frown} e_2 \text{ with } (e_1', e_2') \qquad \iota \in K}{K, T, P + \langle \mathscr{E}_1[e_1] \rangle_{\pi_1} + \langle \mathscr{E}_2[e_2] \rangle_{\pi_2} \Rightarrow K, T, P + \langle \mathscr{E}_1[e_1'] \rangle_{\pi_1} + \langle \mathscr{E}_2[e_2'] \rangle_{\pi_2}}\ [\textsc{Synchronize}]$$

Fig. 3: Dynamic semantics for $\lambda^{\Rightarrow}_{fc}$.

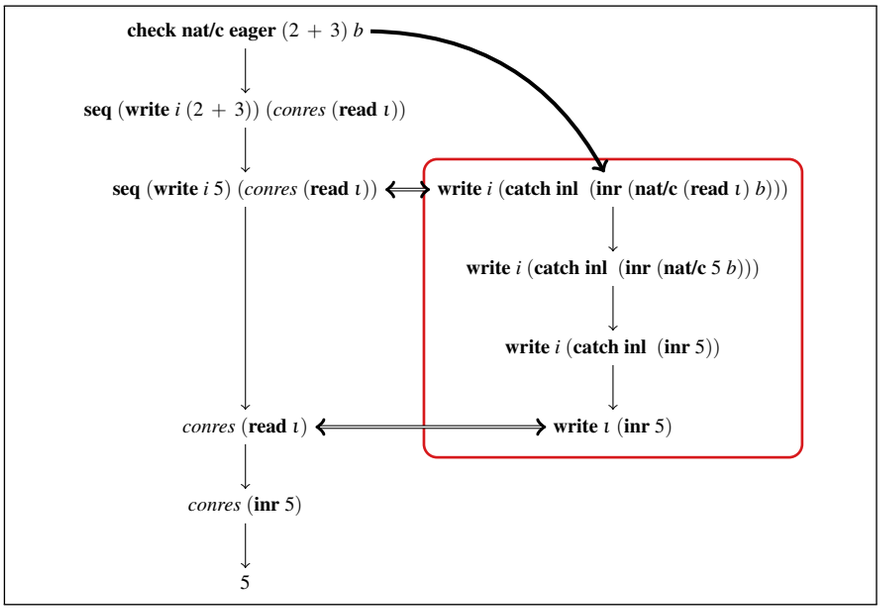

Fig. 4: Checking **nat/c** with **eager**.

(E_{U4}) and retrieve the result from the monitoring process across ι and handle the results (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}) next, if the contract returns a value, the monitoring process injects it right and, similarly, if the contract raises an error, the monitoring process catches that error and injects it to the left;
(E_{M4}) and writes the injected value across ι to the user process.

This interaction is presented in Figure 4 (with the monitoring process colored red). 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. We begin our definition of **check** by encoding this interaction as **eager** verification[4]

$$e ::= \cdots \mid \textbf{check}\ e\ e\ e\ e$$

$$D ::= \cdots \mid \textbf{check}\ D\ e\ e\ e \mid \textbf{check}\ v\ D\ e\ e \mid \textbf{check}\ v\ v\ e\ D$$

[4] We use **seq** to mean $\lambda x\, y.\ y$ and **let** $x = e_1$ **in** e_2 as $(\lambda x.\ e_2)\ e_1$, and use their extended forms that deal with additional arguments in the usual way.

$$\begin{array}{l}\textbf{check}\ con\ \textbf{eager}\ exp\ B \rightarrow \textbf{let}\ i\ =\ \textbf{chan}\\ \qquad\qquad\qquad\qquad\quad \textbf{in seq}\\ \qquad\qquad\qquad\qquad\qquad (\textbf{spawn}\ (\textbf{write}\ i\ (\textbf{catch inl}\ (\textbf{inr}\ (con\ (\textbf{read}\ i)\ B)))))\\ \qquad\qquad\qquad\qquad\qquad (\textbf{write}\ i\ exp)\\ \qquad\qquad\qquad\qquad\qquad (conres\ (\textbf{read}\ i))\end{array} \tag{7}$$

Recall that the **check** language form accepts the contract to check, the strategy to check it with, the expression to check it on, and blame information. Note, however, that we do not evaluate the monitored term until after the constructing the correct monitoring pattern, which allows individual strategies to control (or delay) when to evaluate the contracted term (such as in the case of **semi** monitoring).

The *conres* helper interprets the monitor result in the user process (as indicated in E_{M4}):

$$conres \stackrel{\Delta}{=} \lambda x.\ \textbf{case}\ (x;\ \textbf{inl}\ y \rhd \textbf{raise}\ y;\ \textbf{inr}\ z \rhd z) \tag{8}$$

If the value is left-injected (indicating a contract violation), we re-raise the error in the process expecting the contract result, and, similarly, if the value is right-injected (indicating that **check** did not raise an error), we return it to the process[5].

To demonstrate this monitoring structure, consider the program in Figure 4, which checks that $(2\ +\ 3)$ is a natural number. The monitor creates a process, communicates the evaluated version of the term, and retrieves the results. If the monitored expression had been -1 instead, the user process would instead terminate with an error.

$$\begin{array}{ll}\Rightarrow^* & \{\langle(conres\ (\textbf{read}\ \iota))\rangle_{\pi_0}, \langle\textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inl}\ (\textbf{nat/c}\ \text{-}1\ B)))\rangle_{\pi_1}\}\\ \Rightarrow^* & \{\langle(conres\ (\textbf{read}\ \iota))\rangle_{\pi_0}, \langle\textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{raise}\ B))\rangle_{\pi_1}\}\\ \Rightarrow^* & \{\langle(conres\ (\textbf{read}\ \iota))\rangle_{\pi_0}, \langle\textbf{write}\ \iota\ (\textbf{inl}\ B)\rangle_{\pi_1}\\ \Rightarrow^* & \{\langle(conres\ (\textbf{inl}\ B))\rangle_{\pi_0}\ , \langle\textbf{unit}\rangle_{\pi_1}\}\\ \Rightarrow^* & \{\langle\textbf{raise}\ B\rangle_{\pi_0}\ , \langle\textbf{unit}\rangle_{\pi_1}\}\end{array}$$

Pair and Function Contracts. Recall that eager monitors may "over-explore" their input, detecting and signaling contract violations for values that are unused in the user program. 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 Section 2 as:

$$\textbf{nat-pair/c}^{eager} \stackrel{\Delta}{=} \textbf{pair/c nat/c eager nat/c eager} \tag{9}$$

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

A partial trace is given in Figure 5. This interaction results with an error in the user process (π_0) because -1 is not a natural number. Notice that there are *three* monitoring evaluators in this derivation: 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

[5] 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 **check**/*conres* mechanism will also ensure this error is properly propagated to the initiating evaluator. Our pair contract example utilizes this behavior to propagate subcontract errors, too.

$$
\begin{array}{ll}
 & \{\langle \textbf{fst}\ (\textbf{check nat-pair/c}^{eager}\ \textbf{eager}\ (5,\text{-}1)\ B)\rangle_{\pi_0}\} \\
\Rightarrow^* & \{\langle \textbf{fst}\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0} \\
 & \ ,\langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{nat-pair/c}^{eager}\ (5,\text{-}1)\ B)))\rangle_{\pi_1}\} \\
\Rightarrow^* & \{\langle \textbf{fst}\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0} \\
 & \ ,\langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{check nat/c eager}\ (\textbf{fst}\ (5,\text{-}1))\ B\quad)))\rangle_{\pi_1} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad ,\textbf{check nat/c eager}\ (\textbf{snd}\ (5,\text{-}1))\ B) \\
\Rightarrow^* & \{\langle \textbf{fst}\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0}, \langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (5, \textbf{raise}\ B)))\rangle_{\pi_1}, \langle \textbf{unit}\rangle_{\pi_2}\ ,\ \langle \textbf{unit}\rangle_{\pi_3}\} \\
\Rightarrow^* & \{\langle \textbf{fst}\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0}, \langle \textbf{write}\ \iota\ (\textbf{inl}\ B)\rangle_{\pi_1} \qquad\qquad\qquad\qquad\qquad\qquad ,\langle \textbf{unit}\rangle_{\pi_2}\ ,\ \langle \textbf{unit}\rangle_{\pi_3}\} \\
\Rightarrow^* & \{\langle \textbf{fst}\ (\mathit{conres}\ (\textbf{inl}\ B))\rangle_{\pi_0}\ \ ,\langle \textbf{unit}\rangle_{\pi_1} \qquad\qquad\qquad\qquad\qquad\qquad ,\langle \textbf{unit}\rangle_{\pi_2}\ ,\ \langle \textbf{unit}\rangle_{\pi_3}\}
\end{array}
$$

Fig. 5: Enforcing $\textbf{nat-pair/c}^{eager}$ on the pair $(5,\text{-}1)$.

$$
\begin{array}{ll}
 & \{\langle(\textbf{check nat-fun/c}^{eager}\ \textbf{eager}\ (\lambda x.\ 1)\ B)\ 5\rangle_{\pi_0}\} \\
\Rightarrow^* & \{\langle(\lambda x.\ \textbf{check nat/c eager}\ (\lambda x.\ 1)\ (\textbf{check nat/c eager}\ x\ (\mathit{invert}\ B))\ B)\ 5\rangle_{\pi_0} \\
 & \ ,\langle \textbf{unit}\rangle_{\pi_1}\} \\
\Rightarrow^* & \{\langle \textbf{check nat/c eager}\ B\ \rangle_{\pi_0}\} \\
\Rightarrow^* & \{\langle \textbf{seq}\ (\textbf{write}\ \iota\ ((\lambda x.\ 1)\ (\textbf{check nat/c eager}\ 5\ (\mathit{invert}\ B))))\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0} \\
 & \ ,\langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{nat/c}\ (\textbf{read}\ \iota)\ \ B))))\rangle_{\pi_2}\} \\
\Rightarrow^* & \{\langle \textbf{seq}\ (\textbf{write}\ \iota\ ((\lambda x.\ 1)\ (\mathit{conres}\ (\textbf{read}\ \iota'))))\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0} \\
 & \ ,\langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{nat/c}\ (\textbf{read}\ \iota)\ \ B))))\rangle_{\pi_2} \\
 & \ ,\langle \textbf{write}\ \iota'\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{nat/c}\ 5\ (\mathit{invert}\ B)))))\rangle_{\pi_3}\} \\
\Rightarrow^* & \{\langle \textbf{seq}\ (\textbf{write}\ \iota\ ((\lambda x.\ 1)\ 5))\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0} \\
 & \ ,\langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{nat/c}\ (\textbf{read}\ \iota)\ \ B))))\rangle_{\pi_2}\} \\
\Rightarrow^* & \{\langle \textbf{seq}\ (\textbf{write}\ \iota\ 1)\ (\mathit{conres}\ (\textbf{read}\ \iota))\rangle_{\pi_0} \\
 & \ ,\langle \textbf{write}\ \iota\ (\textbf{catch inl}\ (\textbf{inr}\ (\textbf{nat/c}\ (\textbf{read}\ \iota)\ \ B))))\rangle_{\pi_2}\} \\
\Rightarrow^* & \{\langle 1\rangle_{\pi_0}\quad ,\quad \langle \textbf{unit}\rangle_{\pi_1}\quad ,\quad \langle \textbf{unit}\rangle_{\pi_2}\quad ,\quad \langle \textbf{unit}\rangle_{\pi_3}\}
\end{array}
$$

Fig. 6: Enforcing $\textbf{nat-fun/c}^{eager}$ on the function $(\lambda x.\ x)$ with input 5, eliding process of the form $\langle \textbf{unit}\rangle_{\pi}$ except in the last step.

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.

Function contracts proceed similarly, but without the nesting monitors, illustrating one of the main revisions over the work presented by Swords *et al.* (2015): while their work directly embeds each monitored term in the monitoring process, this reformulation explicitly evaluates the monitored term in the *user process* before it is given to the monitor. For example, consider monitoring the following function contract on $\lambda x.\ 1$:

$$\textbf{nat-fun/c}^{eager} \triangleq \textbf{fun/c nat/c eager nat/c eager} \tag{10}$$

We present a trace of its usage in Figure 6. As evaluation proceeds, the term

$$((\lambda x.\ 1)\ (\textbf{check nat/c eager}\ 5\ (\mathit{invert}\ B)))$$

occurs in the user process, triggering the pre-condition check while the post-condition check awaits the function result. After the pre-condition is complete, the *user evaluator* performs the actual function application $((\lambda x.\ 1)\ 5)$, reflecting the value flow in Figure 1.

Even so, these last two examples illustrate the over-eager nature of eager verification: while we did not inspect the second element of the pair $(5, \text{-}1)$ and the function $\lambda x.\ 1$ did not use its argument, we still checked that each was a natural number. We can see, now, that the fundamental problem is *preemption*: eager evaluation explicitly suspends the initiating evaluator while verifying the contract, only resuming the initiating evaluator once it is complete. As a result, the overall computation performance is ultimately tied to contract performance, and to alleviate this situation, we merely need to vary the *pattern of communication* between these evaluators.

6.2 Semi-Eager Contract Monitoring—Postponing Contract Verification

Following the order in Section 2, we next encode semi-eager monitoring, indicated with the **semi** strategy. Recall that, in semi-eager verification, the monitor must suspend enforcement until the user evaluator demands the result, which we previously described as "boxing up" the contract and value. To model this monitoring strategy as patterns of communication, the initiating process:

(S_{U1}) creates a delayed expression d and returns it to the user.

When the delayed expression is forced, the forcing process (*not necessarily* the initiating process):

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

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

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

(S_{U5}) and retrieve the result across ι and handle the results (via *conres*).

Dually, 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 this entire computation is captured in a delayed expression (at S_{U1}), giving the initiating evaluator freedom to invoke (or ignore) the verification, as it chooses. This interaction is presented in Figure 7 (with).

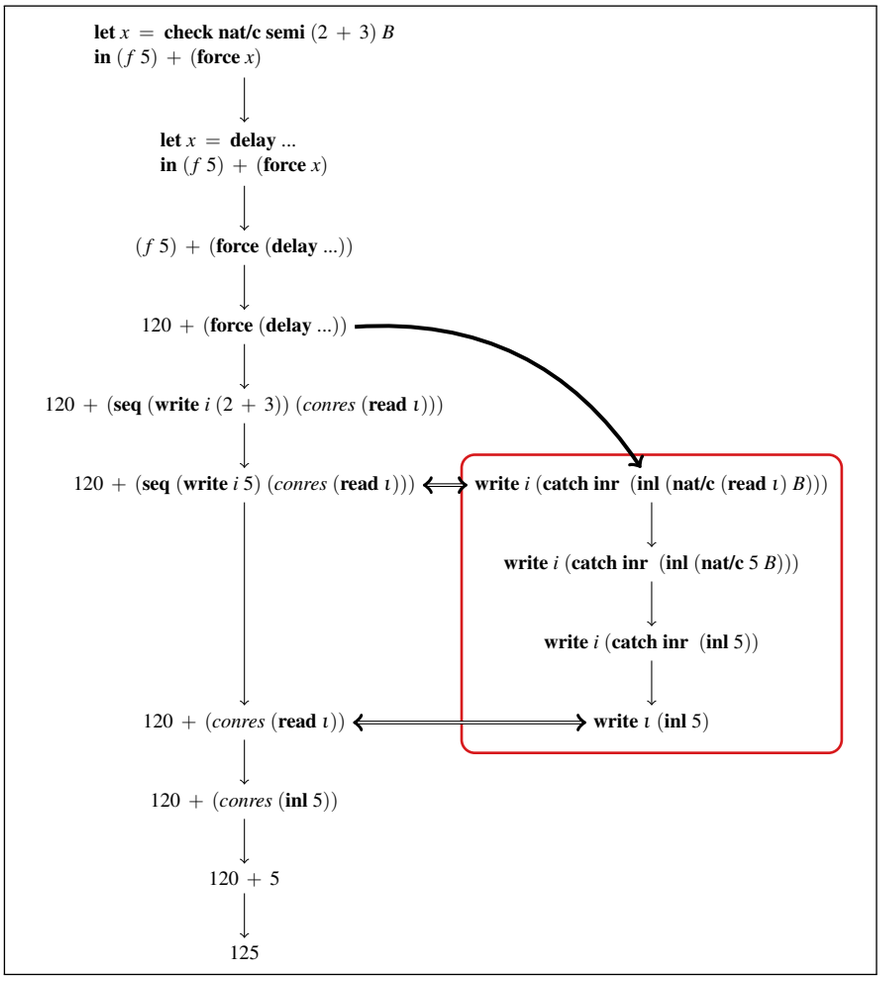

Fig. 7: Checking **nat/c** with **semi**. The monitoring process is not created until the initiating evaluator forces the **check** result. The monitoring process is colored red.

We extend our definition of **check** with this encoding:

$$
\begin{aligned}
\textbf{check}\ \mathit{con}\ \textbf{semi}\ \mathit{exp}\ B \rightarrow\ & \textbf{delay} \\
& (\textbf{let}\ i\ =\ \textbf{chan} \\
& \ \textbf{in seq} \\
& \quad (\textbf{spawn}\ (\textbf{write}\ i\ (\textbf{catch inl}\ (\textbf{inr}\ (\mathit{con}\ (\textbf{read}\ i)\ B))))) \\
& \quad (\textbf{write}\ i\ \mathit{exp}) \\
& \quad (\mathit{conres}\ (\textbf{read}\ i)))
\end{aligned}
\tag{11}
$$

This implementation directly corresponds to the **eager** implementation in Equation 7, except the entire verification expression is delayed (highlighted in yellow to show its addition), packaging up the verification computation until an evaluator forces it. For example,

we may semi-eagerly verify that $(2 + 3)$ is a natural number. We sketch this verification Figure 7: when we force the delayed cell, the evaluator creates a new process, performs the contract verification, and continues the program with the result, yielding 125.

For structural contracts, this system of delaying and forcing contracts gives programmers immense control over *which parts* of the structure are monitored. For example, a semi-eager pair contract that ensures each element is a natural number may forgo checking unused parts of the pair:[6]

$$\begin{array}{ll} & \{\langle \textbf{force}\ (\textbf{fst}\ (\textbf{force}\ (\textbf{check nat-pair/c}^{semi}\ \textbf{semi}\ (5, \text{-}1)\ ))))\rangle_{\pi_0}\} \\ \Rightarrow^* & \{\langle \textbf{force}\ (\textbf{fst}\ (\textbf{delay}\ \text{...“check nat/c on 5”...}\ ,\ \textbf{delay}\ \text{...“check nat/c on -1”...}))\rangle_{\pi_0}, \ldots\} \\ \Rightarrow^* & \{\langle \textbf{force}\ (\textbf{delay}\ \text{...check nat/c on 5...})\rangle_{\pi_0}, \langle \textbf{unit}\rangle_{\pi_1}\} \\ \Rightarrow^* & \{\langle 5\rangle_{\pi_0}, \langle \textbf{unit}\rangle_{\pi_1}, \langle \textbf{unit}\rangle_{\pi_2}\} \end{array}$$

Since the user evaluator never forces the second **nat/c** contract (that would have checked if -1 is a natural number), the program completes without enforcing the contract. In general, semi-eager verification allows users to verify precisely those values they require.

On the Use of Delay and Force. In **semi** (and **prom**, below) monitoring, we use **delay** to produce delayed 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 explicitly clarify the nature of the evaluator interactions for these strategies, allowing **force** to propagate across our user programs. We tolerate this intrusion in our presentation to better explain the nature of these synchronizing monitors in order 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 may clutter the program and inconvenience the programmer. As such, we suggest that they should be concealed in a language implementation, using implicit forcing at evaluation sites (and some transparent structure, such as, e.g., *chaperones* (Strickland *et al.*, 2012) for delaying structure) to remove this complexity.

6.3 Promise-Based Contract Monitoring—Concurrent Checking with Synchronization

Our next contract verification strategy is promise-based verification, indicated by **prom**, which returns a promise to the initiating evaluator while verification proceeds concurrently. Unlike the box-driven description in Section 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 will block until the verification is complete (if the verification has previously finished, the initiating process will receive this result immediately). To model this strategy via patterns of communication, the initiating process:

(P_{U1}) creates a new communication channel ι;
(P_{U2}) spawns a monitoring process that will evaluate the contract;

[6] The **nat-pair/c**^{semi} contract is similar to **nat-pair/c**^{eager}, replacing **eager** with **semi**.

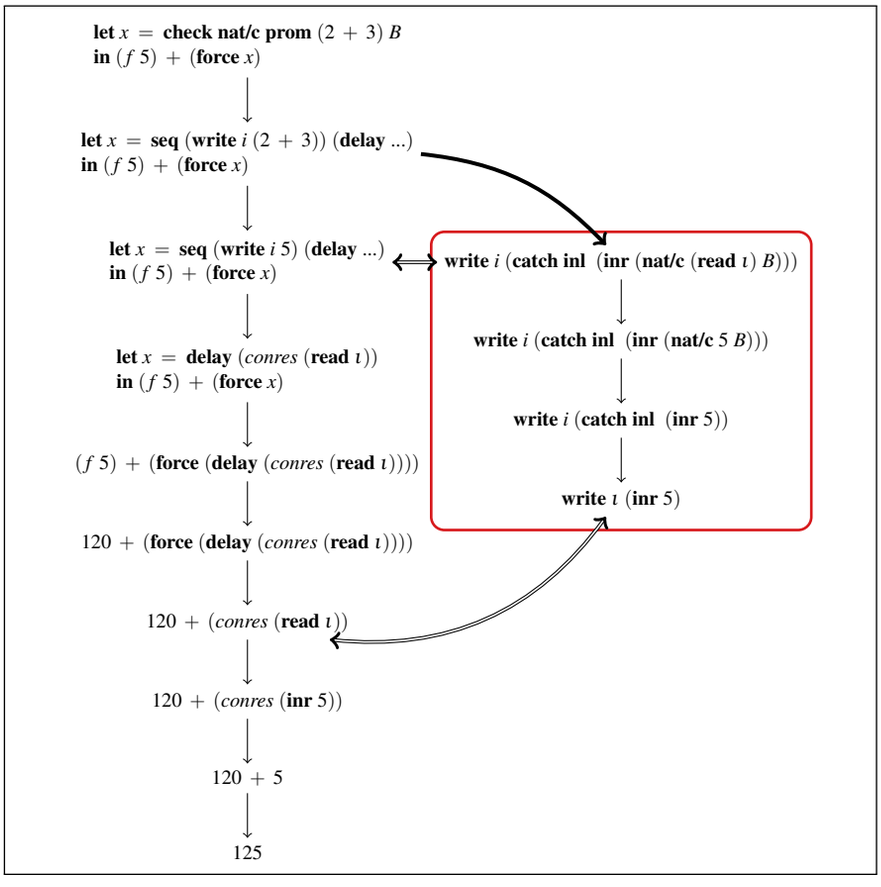

Fig. 8: Checking **nat/c** with **prom**. The monitoring process does not communicate its final result until the initiating evaluator forces the **check** result. The expression $(f\ 5)$ is a stand-in for additional computation in the initiation process before synchronization. The monitoring process is colored red

(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 with the provided blame information;
(P_{M3}) injects the result appropriately;
(P_{M4}) and writes the injected value across ι to the user process.

These two evaluators synchronize twice: first at (P_{U3}, P_{M1}), and later at (P_{M4}) when the user process forces the delayed expression created in (P_{U4}). The forced expression will perform a blocking **read** across ι, receiving the contract result via *conres*. This interaction

is presented in Figure 8, and we extend **check** to support **prom** as:

$$\begin{array}{l}\textbf{check}\ con\ \textbf{prom}\ exp\ B\ \rightarrow \textbf{let}\ i\ =\ \textbf{chan}\\ \qquad\qquad\qquad\qquad\quad \textbf{in seq}\\ \qquad\qquad\qquad\qquad\qquad (\textbf{spawn}\ (\textbf{write}\ i\ (\textbf{catch inl}\ (\textbf{inr}\ (con\ (\textbf{read}\ i)\ B)))))\\ \qquad\qquad\qquad\qquad\qquad (\textbf{write}\ i\ exp)\\ \qquad\qquad\qquad\qquad\qquad (\ \textbf{delay}\ \ (conres\ (\textbf{read}\ i))))\end{array} \tag{12}$$

As with **semi**, this implementation directly corresponds to the **eager** implementation in Equation 7, 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 monitoring continues concurrently. An example of this enforcement interaction is given in Figure 8. Moreover, if these concurrent processes are run in parallel, the **prom** strategy allows contract verification to happen *while* speculatively performing additional computation.

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

6.4 Concurrent Contract Monitoring—Complete Evaluator Decoupling

Our next strategy, **conc**, eschews having a process retrieve the monitor result to provide concurrent contract verification. Instead, the monitoring evaluator proceeds concurrently without reporting its result. Modeled as patterns of communication, the initiating process:

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

Dually, the monitoring process:

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

These two evaluators synchronize once, at (A_{U3}, A_{M1}), to communicate the subject value. This interaction is presented in Figure 9 (with the monitoring process colored red). We extend **check** with this encoding of **conc**:

$$\begin{array}{l}\textbf{check}\ con\ \textbf{conc}\ exp\ B\ \rightarrow \textbf{let}\ i\ =\ \textbf{chan}\\ \qquad\qquad\qquad\qquad\quad \textbf{in seq}\ (\textbf{spawn}\ (con\ (\textbf{read}\ i)\ B))\\ \qquad\qquad\qquad\qquad\qquad\quad (\textbf{let}\ x\ =\ exp\ \textbf{in seq}\ (\textbf{write}\ i\ x)\ x)\end{array} \tag{13}$$

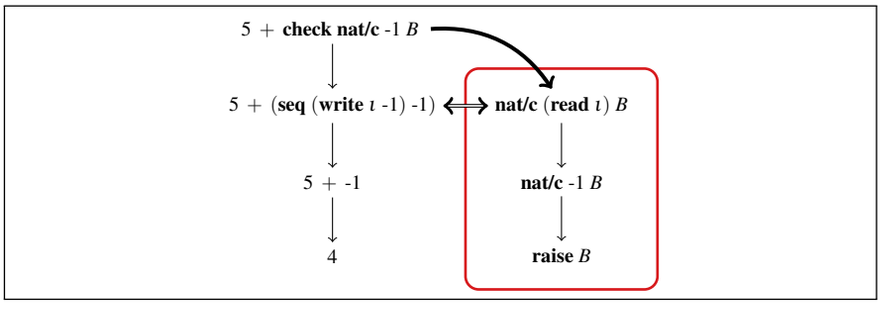

Fig. 9: Checking **nat/c** with **conc**. The monitoring process continues concurrently while the initiating process computes the result.

Using our previous definition of *answer configurations*, we see that **conc** contracts represent contracts that *may not* finish; if the contract is still running when π_0 terminates, its result is ignored. For example, consider concurrently enforcing **nat/c** on -1:

$$\{\langle 5 + (\textbf{check nat/c conc } \text{-1 } B\rangle_{\pi_0}\} \Rightarrow^* \{\langle 4\rangle_{\pi_0}, \langle \textit{... check in progress ...}\rangle_{\pi_1}\} \textit{ or } \{\langle 4\rangle_{\pi_0}, \langle \textbf{raise } B\rangle_{\pi_1}\} \tag{14}$$

Our $\Rightarrow$ relation is nondeterministic, and thus this monitor may or may not complete before the initiating evaluator, resulting in "best-effort" checking.

For structural and functional contracts, this approach will yield numerous, nondeterministic processes, meaning errors may be reported nondeterministically if there are multiple contract violations. This has the downside of so-called "heisenbug"-style errors, but the upside is that programmers may utilize concurrent behavior for weak, long-lived contracts. It is also imaginable that these violations may be reported as "warnings" to the programmer instead, indicating problematic values without bringing the program to a halt.

As a further example of nondeterministic enforcement, consider this function contract enforcement:

$$\begin{aligned}
&\{\langle(\textbf{check } (\textbf{fun/c nat/c conc nat/c conc}) \textbf{ eager } (\lambda x.\ x\ +\ 1)\ B)\ 5\rangle_{\pi_0}\} \\
\Rightarrow^* &\{\langle(\lambda x.\ \textbf{check nat/c conc } ((\lambda x.\ x\ +\ 1)\ (\textbf{check nat/c conc } x\ (\textit{invert } B)))\ B)\ 5\rangle_{\pi_0} \\
&\ ,\ \langle\textbf{unit}\rangle_{\pi_1}\} \\
\Rightarrow^* &\{\langle\textbf{check nat/c conc } ((\lambda x.\ x\ +\ 1)\ (\textbf{check nat/c conc } 5\ (\textit{invert } B)))\ B\rangle_{\pi_0} \\
&\ ,\ \langle\textbf{unit}\rangle_{\pi_1} \\
&\ ,\ \langle\textbf{nat/c } 5\ (\textit{invert } B))\rangle_{\pi_2}\} \\
\Rightarrow^* &\{\langle\textbf{check nat/c conc } ((\lambda x.\ x\ +\ 1)\ 5)\ B\rangle_{\pi_0}\ ,\ \langle\textbf{unit}\rangle_{\pi_1}\ ,\ \langle\textbf{nat/c } 5\ (\textit{invert } B))\rangle_{\pi_2}\} \\
\Rightarrow^* &\{\langle\textbf{check nat/c conc } 6\ B\rangle_{\pi_0}\ ,\ \langle\textbf{unit}\rangle_{\pi_1}\ ,\ \langle 5\rangle_{\pi_2}\} \\
\Rightarrow^* &\{\langle 6\rangle_{\pi_0}\ ,\ \langle\textbf{unit}\rangle_{\pi_1}\ ,\ \langle 5\rangle_{\pi_2}\ ,\ \langle\textbf{nat/c } 6\ B)\rangle_{\pi_3}\}
\end{aligned}$$

This enforcement has a few oddities:

1. We call the top-level **check** with **eager**: if we had used **conc**, the function contract combinator would have produced the monitored procedure in the monitoring process, and the user program would have proceeded with $\lambda x.\ x\ +\ 1$. It is conceivable that checking a function contract with **conc** may perform some sort of enumerative analysis, run concurrently, checking the function against inputs for the remainder of

the program's run. This approach, however, has a number of drawbacks, including input-based dispatched to determine if the value is a procedure (via, e.g., Racket's `procedure?` operation (Flatt & PLT, 2010)), introducing non-uniform strategy behavior. Clojure's `core.spec` (Clojure, 2017) adopts a similar approach for higher-order function contracts, where higher-order function inputs are randomly checked with sampled values to ensure they conform to the specification. We encode this strategy in the Section 9.

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

As previously discussed, this verification technique may often be "too weak" for many properties that a programmer must rely on.

6.5 Finally-Concurrent Contract Monitoring—Verification Without Synchronization

In order to provide programmers with "start and forget" verification with stronger guarantees, we introduce **fconc** verification. Similar to **conc** verification, **fconc** monitoring processes elides secondary synchronization with the initiating process. Unlike **conc**, however, we ensure the monitor completes before the configuration is considered "done." To this end, we us the spawn variant **fspawn**, 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. With this system in place, the initiating **fconc** process:

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

Dually, the *final* monitoring process:

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

This interaction is presented in Figure 10 (with the monitoring process colored red), which is nearly identical to Figure 9, with the addition of the answer configuration in green. The *only* difference between **fconc** and **conc** is this notion of process finality, and thus its implementation simply exchanges **spawn** for **fspawn**:

$$\begin{aligned}\textbf{check}\ \mathit{con}\ \textbf{fconc}\ \mathit{exp}\ B\ \rightarrow\ &\textbf{let}\ i\ =\ \textbf{chan}\\ &\textbf{in seq}\ (\textbf{fspawn}\ (\mathit{con}\ (\textbf{read}\ i)\ B))\\ &\qquad(\textbf{let}\ x\ =\ \mathit{exp}\ \textbf{in seq}\ (\textbf{write}\ i\ x)\ x)\end{aligned} \tag{15}$$

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*:

$$\begin{aligned}&\emptyset,\{\pi_0\};\{\langle(\textbf{check nat/c fconc}\ \text{-}1\ B)+(\textbf{check nat/c fconc}\ 3\ B)\rangle_{\pi_0}\}\\ \Rightarrow^*\ &\{\iota\},\{\pi_0,\pi_1,\pi_2\};\{\langle\text{-}1+3\rangle_{\pi_0},\langle\textbf{nat/c}\ \text{-}1\ B\rangle_{\pi_1},\langle\textbf{nat/c}\ 3\ B\rangle_{\pi_2}\}\\ \Rightarrow^*\ &\{\iota\},\{\pi_0,\pi_1\};\{\langle 8\rangle_{\pi_0}\{\langle\textbf{raise}\ B\rangle_{\pi_1},\langle\textbf{nat/c}\ 3\ B\rangle_{\pi_2}\}\\ \Rightarrow^*\ &\{\iota\},\{\pi_0,\pi_1\};\{\langle 8\rangle_{\pi_0}\{\langle\textbf{raise}\ B\rangle_{\pi_1},\langle 3\rangle_{\pi_2}\}\end{aligned}$$

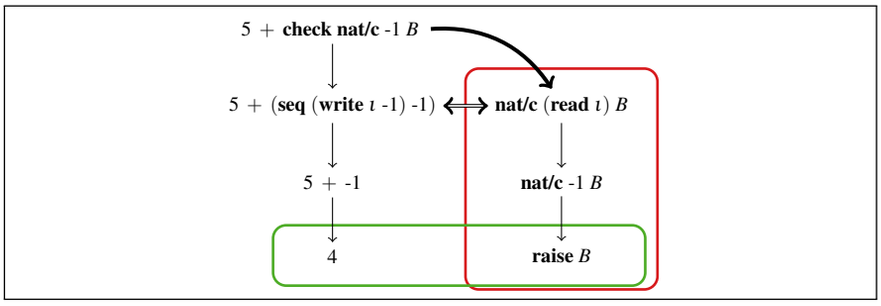

Fig. 10: Checking **nat/c** with **fconc**. The initiating and monitoring processes each proceed concurrently, and the green box indicate an *answer configuration*.

Even though one contract raised an error, we must still wait for each **fconc** contract to complete before termination. This "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 is done, they will know the data is correct.

7 Multi-Strategy Contracts

Beyond choosing which strategy to use for each contract, programmers may also freely intermix strategies in $\lambda^{\Rightarrow}_{/c}$, yielding flexibility and utility beyond traditional contract systems. First, we summarize our strategies so far, giving their implementations together in Figure 11.

7.1 A Flexible Binary-Search Tree Contract.

Our first example is our strategy-parameterized binary tree contract from Section 3. While it is possible to construct this contract in $\lambda^{\Rightarrow}_{/c}$, we take some liberties here for simplicity of presentation, namely:

- We assume we may directly match on trees with named constructors:

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

  (otherwise, we would have to define a binary tree as a sum type);
- and we assume a fixpoint operator μ as

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

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

- **check eager** (**bst/c eager**) *tree B* will eagerly ensure its input is a binary-search tree;
- **check semi** (**bst/c semi**) *tree B* will return a tree that will check that each node is correctly-ordered as it is explored;
- **check prom** (**bst/c prom**) *tree B* will create a cascading chain of monitoring processes for each node, and exploring the tree will synchronize with the appropriate processes at each level;

```
check 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))

check 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)))

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

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

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

Fig. 11: The aggregate contract verification strategies presented in Section 6.

- and **check conc** (**bst/c conc**) *tree B* will concurrently enforce that the tree is a binary-search tree using a similar set of cascading processes.
- and **check fconc** (**bst/c fconc**) *tree B* will do the same as the **conc** contract, but force the program to wait for it to complete before terminating.

To define **bst/c**, we must check that each value in the left sub-tree is less than (or equal to) the node's value and each value in the right sub-tree is greater (or equal to) than the node's value. To do this, we must propagate node values downward through subcontracts. This

requires a *dependent* tree contract, wherein each sub-contract is given the current node's value as an input before enforcement. We define this combinator, **tree/dc**, as:

$$\begin{aligned}
\textbf{tree/dc} := \ & \lambda\ c_{leaf}\ s_{leaf}\ c_{node}\ s_{node}\ c_{left}\ c_{right}\ s_{rec}. \\
& \lambda tree\ b. \\
& \quad \textbf{case}_{tree}\ (tree; \\
& \qquad \textbf{leaf} \qquad\quad \triangleright \textbf{check}\ c_{leaf}\ s_{leaf}\ \textbf{leaf}\ b; \\
& \qquad \textbf{node}\ v\ tl\ tr \triangleright \textbf{node}\ (\textbf{check}\ c_{node}\ s_{node}\ v\ b) \\
& \qquad\qquad\qquad\qquad\quad (\textbf{check}\ (c_{left}\ v)\ s_{rec}\ tl\ b) \\
& \qquad\qquad\qquad\qquad\quad (\textbf{check}\ (c_{right}\ v)\ s_{rec}\ tr\ b))
\end{aligned} \tag{16}$$

This combinator takes seven arguments[7]

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 subtrees.

We can use this dependent contract combinator to define **bst/c** as[8]:

$$\textbf{any/c} := \textbf{pred/c}\ (\lambda x.\ \textbf{true}) \tag{17}$$

$$\begin{aligned}
\textbf{bst/c} := \ & \lambda s.\ \mu\ \textbf{bst/c}\ s\ lo\ hi. \\
& \quad \textbf{tree/dc any/c} \qquad\qquad\qquad\qquad\qquad\quad \textbf{eager} \\
& \qquad\quad (\textbf{pred/c}\ (\lambda x.\ (lo \leq x)\ \&\&\ (x \leq hi))) \ \textbf{eager} \\
& \qquad\quad (\lambda v.\ \textbf{bst/c}\ s\ lo\ v)\ (\lambda v.\ \textbf{bst/c}\ s\ v\ hi) \quad s
\end{aligned} \tag{18}$$

This contract ensures that each leaf of the tree is any value and each internal value in the tree is within the correct numeric bounds. As **bst/c eager**, this contract must traverse the entire tree to enforce this constraint, requiring O(n) time (whereas a insertion algorithm would require O(log n) time in a sorted tree). We can forgo such a strong guarantee, however, and use **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 **bst/c prom**, 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 **bst/c conc**, and even use **fconc** for finally-concurrent, start-and-forget verification. Further, we can check **bst/c eager** under **prom**, constructing a promise that will concurrently enforce the entire contract. And each of these different variations sprout forth from the same definition of **bst/c**. Even further, we may define a secondary version of **bst/c**, called call **bst/c**^{s}, which takes an additional strategy s_2 to use for node

[7] 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 subtrees. These alternatives, however, trade expressiveness for programmatic ease.

[8] We could also ensure the tree's values are *int?* by adding it to the value contract conjunction.

value contract enforcement, allowing us to control exactly when to verify each *value* in addition to the general recursive verification scheme:

$$\begin{array}{l}\textbf{bst/c}^s := \mu\ \textbf{bst/c}^s\ s_1\ s_2\ lo\ hi. \\ \quad \textbf{tree/dc any/c} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{eager} \\ \qquad\quad (\textbf{pred/c}\ (\lambda x.\ (lo \leq x)\ \&\&\ (x \leq hi))) \qquad\qquad s_2 \\ \qquad\quad (\lambda v.\ \textbf{bst/c}^s\ s_1\ s_2\ lo\ v)\ (\lambda v.\ \textbf{bst/c}^s\ s_1\ s_2\ v\ hi)\ s_1\end{array} \tag{19}$$

In this definition, we use s_2 to enforce the node predicate at each level.

7.2 A Lazy Tree Fullness Contract.

Our second example tackles the problem of lazily ensuring that a binary tree is full (that is, each node's subtrees have the same height). Findler *et al.* (2008) identify such checks as requiring *upward value propagation* through monitored structures, which is generally impossible with structural contracts. To illustrate this problem, we define the predicate **full?** and use it to create a predicate contract:

$$\begin{array}{l}\textbf{full?} := \textbf{let}\ f\ =\ \mu\ full\ tree\ . \\ \qquad\qquad \textbf{case}_{tree}\ (tree; \\ \qquad\qquad\qquad \textbf{leaf} \qquad\quad \triangleright 0; \\ \qquad\qquad\qquad \textbf{node}\ v\ tl\ tr \triangleright \textbf{let}\ hl = full\ tl \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad hr = full\ tr \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{in if}\ (hr = hl)\ \&\&\ (hl \geq 0)\ \&\&\ (hr \geq 0) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{then}\ 1 + hl \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{else}\ \text{-}1 \\ \qquad \textbf{in}\ \lambda t.\ 0\ \leq\ (f\ t)\end{array} \tag{20}$$

$$\textbf{full/c} := \textbf{pred/c full?} \tag{21}$$

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, propagating heights upward to ensure the property. 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.

Alternatively, we can adopt the side-channel style of contract definition presented by Swords *et al.* (2015), which allow multiple contracts to collaborate to ensure global properties. Using the **conc** strategy and communication, we may postpone checking any contract until its subcontracts are complete by having each contract communicate with its subcontracts. While complex in concept, the only additional facility we require is a choice-based reading operation (to allow us to communicate with multiple subcontracts at once).

This choice operator is a straightforward addition to $\lambda^{\Rightarrow}_{/c}$: we extend the "matches" relation from Figure 3 to support choice as:

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

The [SYNCHRONIZE] operation in Figure 3, in conjunction with this extended definition, allow us to perform choice-based communication via **choice**. We may now use this oper-

ation and a dependent tree contract with this "callback"-style communication to create a semi-eager fullness contract, given as:

$$
\begin{array}{l}
\textbf{full/fc} := \mu \mathit{full}\ i\ . \qquad (22)\\
\quad \textbf{let}\ i_l = \textbf{chan}\\
\quad\quad\ \ i_r = \textbf{chan}\\
\quad \textbf{in tree/dc}\ (\textbf{pred/c}\ (\lambda\_.\ \textbf{seq}\ (\textbf{write}\ i\ 0)\ \textbf{true}))\qquad\qquad \textbf{eager}\\
\quad\quad\quad (\textbf{pred/c}\ (\lambda\_.\ \textbf{let}\ h1 = \textbf{choice}\ (\textbf{read}\ i_l)\ (\textbf{read}\ i_r)\\
\quad\quad\quad\quad\quad\quad\quad\quad\ \ h1 = \textbf{choice}\ (\textbf{read}\ i_l)\ (\textbf{read}\ i_r)\\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in if}\ h1\ =\ h2\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{then}\ (\textbf{seq}\ (\textbf{write}\ i\ (1+h1))\ \textbf{true})\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else false}))\\
\quad\quad\quad\quad\quad\quad \textbf{conc}\\
\quad\quad\quad\quad\quad\quad (\lambda\_.\ \mathit{full}\ i_l)\\
\quad\quad\quad\quad\quad\quad (\lambda\_.\ \mathit{full}\ i_r)\\
\quad\quad\quad\quad\quad\quad \textbf{semi}))
\end{array}
$$

Each contract invocation is parameterized by a communication channel i, indicating where to write the current node's 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 subtrees respectively. Next we assert $(\mathit{full}\ i_l)$ and $(\mathit{full}\ i_r)$ on the appropriate subtrees, utilizing the dependent contract to delay these invocations until usage time (to prevent divergence). Each of these subcontracts are monitored with **semi** and thus will not be verified until the initiating process demands these subtrees. Finally, the node value contract, monitored with **conc**, retrieves its subtree 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 full separation of the monitor evaluator from the user evaluator and exposing communication tools to contract writers, facilitating contract verification in a custom-crafted traversal of the monitored structure.

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. This secondary process will then raise a contract violation if it ever received two disagreeing depths, indicating the tree is not full. This style of effectful contract verification allows programmers to check global properties with less overhead.

8 Embedding Findler & Felleisen (2002) into $\lambda^{\Rightarrow}_{/c}$

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 accomplishhment, we now prove that **eager** in $\lambda^{\Rightarrow}_{/c}$ simulates the eager contract

Findler and Felleisen, 2002

$$
\begin{array}{rcl}
c & = & \lambda x.\, c \mid c\ c \mid x \mid \textbf{if } c \textbf{ then } c \textbf{ else } c \mid \textbf{true} \mid \textbf{false} \mid n \mid c\ op\ c \\
 & \mid & c \mapsto c \mid \textbf{contract}(c) \mid \textbf{blame}(c) \mid c^{c,x,x} \\
C & = & \square \mid C\ c \mid V\ C \mid C\ op\ c \mid V\ op\ C \mid \textbf{if } C \textbf{ then } c \textbf{ else } c \\
 & \mid & C \mapsto c \mid \textbf{contract}(C) \mid \textbf{blame}(C) \mid c^{C,x,x} \mid C^{V,x,x} \\
V & = & \lambda x.\, c \mid n \mid \textbf{true} \mid \textbf{false} \mid V \mapsto V \mid \textbf{contract}(V) \mid V^{V \mapsto V,x,x}
\end{array}
$$

$$
\begin{array}{c}
C[V_1^{\textbf{contract}(V_2),p,n}] \longrightarrow C[\textbf{if } V_2\ V_1 \textbf{ then } V_1 \textbf{ else blame}(p)] \\
C[(V_1^{V_3 \mapsto V_3,p,n})\ V_2] \longrightarrow C[(V_1\ V_2^{V_3,n,p})^{V_4,p,n}] \\
C[e] \longrightarrow C[e'] \quad (\textit{if } e \rightsquigarrow e')
\end{array}
$$

$$
\begin{array}{ll}
\lambda x.\, c\ V \rightsquigarrow c[x/V] & \ulcorner n_1 \urcorner + \ulcorner n_2 \urcorner \rightsquigarrow \ulcorner n_1 + n_2 \urcorner \\
\textbf{if true then } c_1 \textbf{ else } c_2 \rightsquigarrow c_1 & \ulcorner n_1 \urcorner \leq \ulcorner n_2 \urcorner \rightsquigarrow \textbf{true} \quad (\text{if } n_1 \leq n_2) \\
\textbf{if false then } c_1 \textbf{ else } c_2 \rightsquigarrow c_2 & \ulcorner n_1 \urcorner \leq \ulcorner n_2 \urcorner \rightsquigarrow \textbf{false} \quad (\text{if } n_1 \not\leq n_2) \\
 & \vdots
\end{array}
$$

Fig. 12: A subset of the λ^{CON} language from Findler & Felleisen (2002). We have renamed their E to C and e to c to avoid ambiguity.

verification semantics presented by Findler & Felleisen (2002), up to alpha-equivalence and unit elimination. This proof is, in a sense, straightforward: our work follows Findler & Felleisen (2002) to recreate **eager** verification in $\lambda^{\Rightarrow}_{/c}$ in terms of interacting evaluators, and thus our proof is primarily concerned with extracting and recovering the individual evaluators for contract monitoring, separating the user portions of the program from the monitoring ones.

We start with the core language λ^{CON} presented by Findler & Felleisen (2002), given in Figure 12. This λ^{CON} definition elides a handful of forms from the version presented by Findler & Felleisen (2002), including list and fixpoint operations (since neither are relevant to the discussion) and, more importantly, their outer **val rec** form, defined as follows (with appropriate evaluation contexts to match, which evaluate the bindings before the body):

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

Here, each binding has two values associated as $x : V_1 = V_2$, where the first represents a contract on the second. Findler & Felleisen (2002) 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 x is bound as **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 (where x indicates the positive blame party, which is the binding itself, and n represents the negative blame party, which is defined by the variable's context). Findler & Felleisen (2002) 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 has been completely evaluated and substituted in, eschewing p and d syntax forms (and their associated evaluation contexts).

Our simulation works via three translation relations from λ^{CON} to $\lambda^{\Rightarrow}_{/c}$, defined as "$\rightsquigarrow$", "$\twoheadrightarrow_e$", and "$\twoheadrightarrow_v$" in Figure 13 and Figure 14. This translation relies on one additional modification to λ^{CON}: Findler & Felleisen (2002) 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 originate in the user program with $\circ$, indicating it is part of the user program.
- We modify $\rightarrow$ in λ^{CON} to produce $\textbf{if}_\bullet$ forms as:

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

Evaluation for both $\textbf{if}_\circ\ c\ \textbf{then}\ c\ \textbf{else}\ c$ and $\textbf{if}_\bullet\ c\ \textbf{then}\ c\ \textbf{else}\ c$ otherwise proceed as **if** c **then** c **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 $\lambda^{\Rightarrow}_{/c}$.

The $\rightsquigarrow$ operator relates a term c with an expression e, a set of new channels K, and a set of processes P, where the translated expression is meant to fill process π_0. The $\twoheadrightarrow_e$ and $\twoheadrightarrow_v$ are helper relations that translate terms and values in λ^{CON} into equivalent term-level expressions in $\lambda^{\Rightarrow}_{/c}$ respectively.

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 8.1
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 values in λ^{CON} are related to values in $\lambda^{\Rightarrow}_{/c}$.

Proof proceeds by induction on c and our constraint that it is a value, then inversion on the translation relation. □

Lemma 8.2
If $c = C[c_0]$ such that c_0 is the next redex, then $c \rightsquigarrow (K,\ P,\ e)$ has some derivation tree as

$$\frac{\mathscr{D}}{c \rightsquigarrow (K,\ P,\ e)}$$

then $\mathscr{D}$ contains $c_0 \rightsquigarrow (K_0,\ P_0,\ e_0)$.

Proof (Sketch)
Findler & Felleisen (2002) prove unique decomposition for c terms, and proof proceeds by induction on the structure of C and inversion on $\mathscr{D}$. □

Lemma 8.3 (*Embedding Reduction*)

Embedding Translation

$\boxed{c \rightsquigarrow (K,\ P,\ e)}$

$$\frac{}{\mathbf{true} \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{true})} \qquad \frac{}{\mathbf{false} \rightsquigarrow (\emptyset,\ \emptyset,\ \mathbf{false})} \qquad \frac{}{n \rightsquigarrow (\emptyset,\ \emptyset,\ n)}$$

$$\frac{}{x \rightsquigarrow (\emptyset,\ \emptyset,\ x)} \qquad \frac{c \twoheadrightarrow_e e}{\lambda x.\, c \rightsquigarrow (\emptyset,\ \emptyset,\ \lambda x.\, e)}$$

$$\frac{c_1 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e_2)}{c_1\ c_2 \rightsquigarrow (K,\ P,\ e_1\ e_2)} \qquad \frac{c_1 \notin V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2}{c_1\ c_2 \rightsquigarrow (K,\ P,\ e_1\ e_2)}$$

$$\frac{c_1 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e_2)}{c_1\ op\ c_2 \rightsquigarrow (K,\ P,\ e_1\ op\ e_2)} \qquad \frac{c_1 \notin V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2}{c_1\ op\ c_2 \rightsquigarrow (K,\ P,\ e_1\ op\ e_2)}$$

$$\frac{c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad c_3 \twoheadrightarrow_e e_3}{\mathbf{if}_\circ\ c_1\ \mathbf{then}\ c_2\ \mathbf{else}\ c_3 \rightsquigarrow (K,\ P,\ \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e3)}$$

$$\frac{\begin{array}{c}c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad c_3 \twoheadrightarrow_e e_3 \\ fresh\ \iota \quad fresh\ \pi \quad p = \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ \ (\mathbf{inr}\ (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)))\rangle_\pi\end{array}}{\mathbf{if}_\bullet\ c_1\ \mathbf{then}\ c_2\ \mathbf{else}\ c_3 \rightsquigarrow (\{\iota\} \uplus K,\ \{p\} \uplus P,\ conres\ (\mathbf{read}\ \iota))}$$

$$\frac{c_1 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e_2) \quad fresh\ f \quad fresh\ b \quad fresh\ x}{(c_1 \mapsto c_2) \rightsquigarrow (K,\ P,\ \lambda f\ b.\ \lambda x.\ \mathbf{check}\ e_2\ \mathbf{eager}\ (f\ (\mathbf{check}\ e_1\ \mathbf{eager}\ x\ (invert\ b)))\ b)}$$

$$\frac{c_1 \notin V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_e e_2 \quad fresh\ f \quad fresh\ b \quad fresh\ x}{(c_1 \mapsto c_2) \rightsquigarrow (K,\ P,\ \lambda f\ b.\ \lambda x.\ \mathbf{check}\ e_2\ \mathbf{eager}\ (f\ (\mathbf{check}\ e_1\ \mathbf{eager}\ x\ (invert\ b)))\ b)}$$

$$\frac{c \rightsquigarrow (K,\ P,\ e) \quad fresh\ x \quad fresh\ b}{\mathbf{contract}(c) \rightsquigarrow (K,\ P,\ \lambda x\ b.\ \mathbf{if}\ e\ x\ \mathbf{then}\ x\ \mathbf{else\ raise}\ b)}$$

$$\frac{c \rightsquigarrow (K,\ P,\ e) \quad B \approx e}{\mathbf{blame}(c) \rightsquigarrow (K,\ P,\ \mathbf{raise}\ B)}$$

$$\frac{c_1 \notin V \quad c_2 \notin V \quad c_1 \twoheadrightarrow_e e_1 \quad c_2 \rightsquigarrow (K,\ P,\ e) \quad B \approx (p,n)}{c_1^{c_2,p,n} \rightsquigarrow (K,\ P,\ \mathbf{check}\ e_2\ \mathbf{eager}\ e_1\ B)}$$

$$\frac{\begin{array}{c}c_1 \notin V \quad c_2 \in V \quad c_1 \rightsquigarrow (K,\ P,\ e_1) \quad c_2 \twoheadrightarrow_v e_2 \quad B \approx (p,n) \\ fresh\ \iota \quad fresh\ \pi \quad p = \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ \ (\mathbf{inr}\ (e_2\ (\mathbf{read}\ \iota)\ B)))\rangle_\pi\end{array}}{c_1^{c_2,p,n} \rightsquigarrow (\{\iota\} \uplus K,\ \{p\} \uplus P,\ \mathbf{seq}\ (\mathbf{write}\ \iota\ e_1)\ (conres\ (\mathbf{read}\ \iota)))}$$

$$\frac{\begin{array}{c}c_1 \in V \quad c_2 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad \mathbf{contract}(c_2) \twoheadrightarrow_v e_2 \quad B \approx (p,n) \\ fresh\ \iota \quad fresh\ \pi \quad p = \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ \ (\mathbf{inr}\ (e_2\ e_1\ B)))\rangle_\pi\end{array}}{c_1^{\mathbf{contract}(c_2),p,n} \rightsquigarrow (\{\iota\},\ \{p\},\ conres\ (\mathbf{read}\ \iota))}$$

$$\frac{\begin{array}{c}c_1 \in V \quad c_2 \in V \quad c_3 \in V \quad c_1 \twoheadrightarrow_v e_1 \quad c_2 \twoheadrightarrow_v e_2 \quad c_3 \twoheadrightarrow_v e_3 \\ fresh\ \iota \quad fresh\ \pi \quad fresh\ x \quad B \approx (p,n)\end{array}}{c_1^{c_2 \mapsto c_3,p,n} \rightsquigarrow (\emptyset,\ \emptyset,\ \lambda x.\ \mathbf{check}\ c_3\ \mathbf{eager}\ (e_1\ (\mathbf{check}\ c_2\ \mathbf{eager}\ x\ (invert\ B)))\ B)}$$

Fig. 13: Embedding procedure for λ^{CON} into $\lambda^{\Rightarrow}_{/c}$.

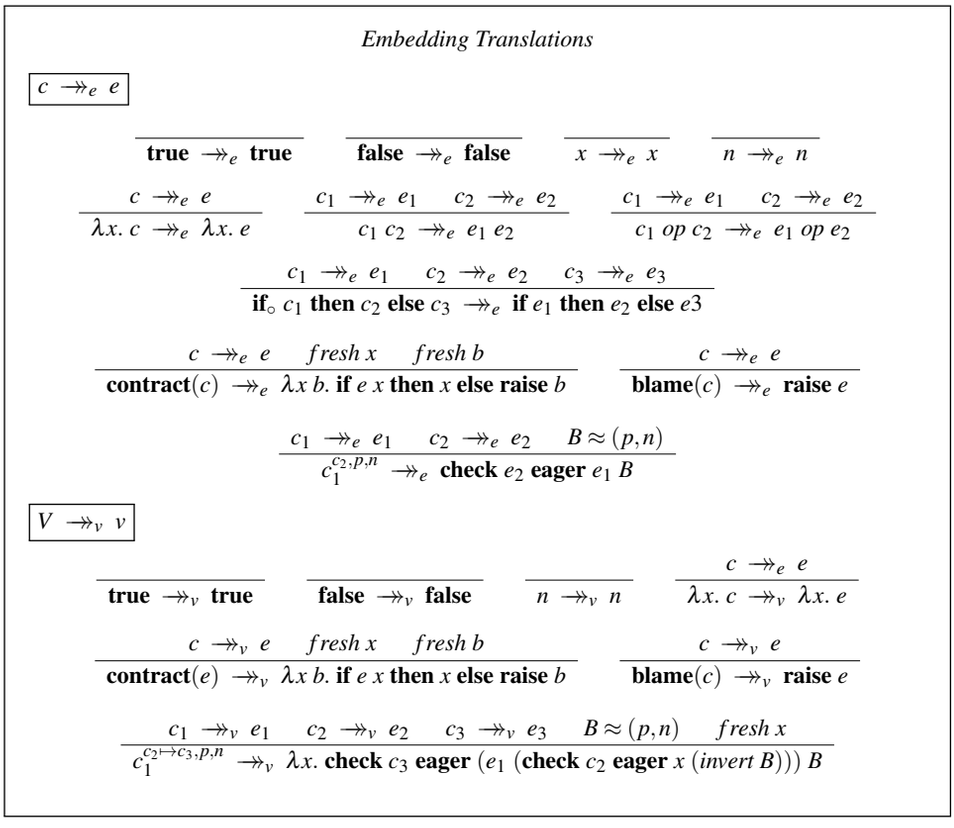

Fig. 14: Sub-translation relations $\twoheadrightarrow_e$ and $\twoheadrightarrow_v$ for embedding λ^{CON} into $\lambda^{\Rightarrow}_{/c}$.

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 $K'',\ \{\pi_0\},\ P'' =_{\alpha,\mathbf{unit}} K', \{\pi_0\}, \{\langle e' \rangle_{\pi_0}\} + P'$.

Proof

This proof establishes that each evaluation step in λ^{CON} corresponds to a sequence of evaluation steps in $\lambda^{\Rightarrow}_{/c}$. 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 worry that the *exact* channel name or process identifier is used between steps, only that each name is used alpha-equivalently.
- We disregard processes of the form $\langle \mathbf{unit} \rangle_\pi$ in our embedding; many of these are left over when contracts are done checking, 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 nor eager monitors will introduce a **fspawn** term, and thus we do not

need to consider the termination set for the proof. Recall, also, that we distinguish contract-checking **if** expressions from others for translation purposes.

We proceed by 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:

Case: $C[V_1^{\mathbf{contract}(V_2),p,n}] \longrightarrow C[\mathbf{if}_\bullet\ V_2\ V_1\ \mathbf{then}\ V_1\ \mathbf{else}\ \mathbf{blame}(p)]$
Using $\rightsquigarrow$, we have:

$$\frac{C[V_1^{\mathbf{contract}(V_2),p,n}] \rightsquigarrow (P,\ K,\ e)}{C[\mathbf{if}_\bullet\ V_2\ V_1\ \mathbf{then}\ V_1\ \mathbf{else}\ \mathbf{blame}(p)] \rightsquigarrow (P',\ K',\ e')}$$

Since C is static, the term inside of C is the next redex and thus it will be translated by $\twoheadrightarrow_v$ and $\twoheadrightarrow_e$, whereas $V_1^{\mathbf{contract}(V_2),p,n}$ will be translated by $\rightsquigarrow$ (via Lemma 8.2). Furthermore,

$$\frac{\begin{array}{lll} V_1 \in V & V_2 \in V & V_2 \twoheadrightarrow_v e_2 \\ B \approx (p,n) & V_1 \twoheadrightarrow_v e_1 & \mathbf{contract}(V_2) \twoheadrightarrow_v \lambda x\, b.\ \mathbf{if}\ e_2\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{raise}\ b \\ fresh\ \iota & fresh\ \pi & p = \langle \mathbf{write}\ \iota\ (\mathbf{catch}\ \mathbf{inl}\ \ (\mathbf{inr}\ (V_2\ V_1\ B)))\rangle_\pi \end{array}}{V_1^{\mathbf{contract}(V_2),p,n} \rightsquigarrow (\{\iota\},\ \{p\},\ conres\ (\mathbf{read}\ \iota))}$$

and

$$\frac{\begin{array}{lll} c_1 \rightsquigarrow (K,\ P,\ e_1) & c_2 \twoheadrightarrow_e e_2 & c_3 \twoheadrightarrow_e e_3 \\ fresh\ \iota & fresh\ \pi' & p' = \langle \mathbf{write}\ \iota\ (\mathbf{catch}\ \mathbf{inl}\ \ (\mathbf{inr}\ (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)))\rangle_{\pi'} \end{array}}{\mathbf{if}_\bullet\ V_2\ V_1\ \mathbf{then}\ V_2\ \mathbf{else}\ \mathbf{blame}(p) \rightsquigarrow (\{\iota'\} \uplus K,\ \{p'\} \uplus P,\ conres\ (\mathbf{read}\ \iota))}$$

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

$$\begin{array}{rl} & K \uplus \{\iota\}, \{\pi_0\}, P_0 \uplus \{p\} \uplus \langle \mathscr{E}[conres\ (\mathbf{read}\ \iota)]\rangle_\pi \\ \Rightarrow & K \uplus \{\iota\}, \{\pi_0\}, P_0 \uplus \{p'\} \uplus \langle \mathscr{E}[conres\ (\mathbf{read}\ \iota)]\rangle_\pi \end{array}$$

This follows directly because p reduces to p' by our definition of term reduction relation "$\longmapsto$" and [PROCSTEP] in the "$\Rightarrow$" relation.

Case: $C[(V_1^{V_3 \mapsto V_4,p,n})\ V_2] \longrightarrow C[(V_1\ V_2^{V_3,n,p})^{V_4,p,n}]$
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:

$$\frac{V_1^{V_3 \mapsto V_4,p,n} \in V \qquad V_1^{V_3 \mapsto V_4,p,n} \twoheadrightarrow_v e_{fc} \qquad V_2 \rightsquigarrow (\emptyset,\ \emptyset,\ e_2)}{V_1^{V_3 \mapsto V_4,p,n}\ V_2 \rightsquigarrow (\emptyset,\ \emptyset,\ e_{fc}\ e_2)}$$

where the channel and process seys in the premises are empty by Lemma 8.1 and e_{fc} is the result of the translation

$$\frac{V_1 \twoheadrightarrow_v e_1 \qquad V_3 \twoheadrightarrow_v e_3 \qquad V_4 \twoheadrightarrow_v e_4 \qquad B \approx (p,n) \qquad fresh\ x}{V_1^{V_3 \mapsto V_4,p,n} \twoheadrightarrow_v \lambda x.\ \mathbf{check}\ e_4\ \mathbf{eager}\ (e_1\ (\mathbf{check}\ e_3\ \mathbf{eager}\ x\ (invert\ B)))\ B}$$

The right-hand side of the reduction proceeds as:

$$\frac{\begin{array}{lll} (V_1\ V_2^{V_3,n,p}) \notin V & V_4 \in V & (V_1\ V_2^{V_3,n,p}) \rightsquigarrow (K',\ P',\ e_1\ e_2') \qquad V_4 \twoheadrightarrow_v e_2 \qquad B \approx (p,n) \\ fresh\ \iota & fresh\ \pi & p = \langle \mathbf{write}\ \iota\ (\mathbf{catch}\ \mathbf{inl}\ \ (\mathbf{inr}\ (e_4\ (\mathbf{read}\ \iota)\ B)))\rangle_\pi \end{array}}{(V_1\ V_2^{V_3,n,p})^{V_4,p,n} \rightsquigarrow (\{\iota\} \uplus K',\ \{p\} \uplus P',\ \mathbf{seq}\ (\mathbf{write}\ \iota\ e_1\ e_2')\ (conres\ (\mathbf{read}\ \iota)))}$$

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: $V_3 = \mathbf{contract}(V_3')$

$$\frac{\begin{array}{cccc} V_3 \in V & V_3 \in V & V_3 \twoheadrightarrow_v e_2 & \mathbf{contract}(V_3) \twoheadrightarrow_v e_3 \quad B \approx (p,n) \\ fresh\ \iota & fresh\ \pi & p = \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ (\mathbf{inr}\ (e_3\ e_2\ B)))\rangle_\pi & \end{array}}{V_3^{\mathbf{contract}(V_3),p,n} \rightsquigarrow (\{\iota\},\ \{p\},\ conres\ (\mathbf{read}\ \iota))}$$

Thus it is sufficient to show

$$\begin{array}{r} K_0, P_0 + \langle \mathscr{E}[(\mathbf{check}\ e_4\ \mathbf{eager}\ (e_1\ (\mathbf{check}\ e_3\ \mathbf{eager}\ x\ (invert\ B)))\ B)\ e_2]\rangle_{\pi_0} \\ \Rightarrow^* K_0 \uplus \{\iota,\iota'\}, P_0 + \langle \mathscr{E}[\mathbf{seq}\ (\mathbf{write}\ \iota\ (e_1\ (conres\ (\mathbf{read}\ \iota'))))\ (conres\ (\mathbf{read}\ \iota))]\rangle_{\pi_0} \\ + \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ (\mathbf{inr}\ (e_4\ (\mathbf{read}\ \iota)\ B))))\rangle_\pi \\ + \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ (\mathbf{inr}\ (e_3\ e_2\ \bar{B})))\rangle_\pi \end{array}$$

which follows from our reduction semantics in $\lambda^{\Rightarrow}_{/c}$.

Subcase: $V_3 = V_{3i} \mapsto V_{3o}$
Since both V_2 and V_3 are values, we translate this as:

$$\frac{\begin{array}{ccccc} V_2 \in V & V_{3i} \in V & V_{3o} \in V & V_2 \twoheadrightarrow_v e_2 & V_{3i} \twoheadrightarrow_v e_{3i} \quad V_{3o} \twoheadrightarrow_v e_{3o} \\ & fresh\ \iota & fresh\ \pi & fresh\ x & \bar{B} \approx (n,p) \end{array}}{V_2^{V_{3i} \mapsto V_{3o},n,p} \rightsquigarrow (\emptyset,\ \emptyset,\ \lambda x.\ \mathbf{check}\ e_{3o}\ \mathbf{eager}\ (e_2\ (\mathbf{check}\ e_{3i}\ \mathbf{eager}\ x\ B))\ \bar{B})}$$

Thus it is sufficient to show

$$\begin{array}{r} K_0, P_0 + \langle \mathscr{E}[(\mathbf{check}\ e_4\ \mathbf{eager}\ (e_1\ (\mathbf{check}\ e_3\ \mathbf{eager}\ x\ (invert\ B)))\ B)\ e_2]\rangle_{\pi_0} \\ \Rightarrow^* K_0 \uplus \{\iota\}, P_0 + \langle \mathscr{E}[\mathbf{seq}\ (\mathbf{write}\ \iota\ (e_1\ e_{2v}))\ (conres\ (\mathbf{read}\ \iota))]\rangle_{\pi_0} \\ + \langle \mathbf{write}\ \iota\ (\mathbf{catch\ inl}\ (\mathbf{inr}\ (e_4\ (\mathbf{read}\ \iota)\ B))))\rangle_\pi \end{array}$$

where

$$e_{2v} = \lambda x.\ \mathbf{check}\ e_{3o}\ \mathbf{eager}\ (e_1\ (\mathbf{check}\ e_{3i}\ \mathbf{eager}\ x\ B))\ \bar{B}$$

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 8.1 (*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 **fspawn** form, and thus T remains constant. Then the proof proceeds by induction on the length of $\longrightarrow^*$ and Lemma 8.3. □

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 granu-

larity. Additional simulation proofs will be more complex, but follow this same approach: syntactic munging plus a little process management.

9 Related Works

The first language for specification and verification was Gypsy, introduced by Ambler *et al.* (1977). Later, Meyer (1992) introduced the term *software contract* with the language Eiffel, along with our moderns notions of pre- and post-condition contract enforcement, and Findler & Felleisen (2002) brought software contracts into functional languages.

9.1 Additional Software Contract Verification Techniques

After Findler & Felleisen (2002) brought software contracts into functional programming, a cottage industry of software contract verification techniques sprung up (Hinze *et al.*, 2006; Chitil, 2012; Findler *et al.*, 2008; Dimoulas *et al.*, 2009; Moore *et al.*, 2016; Dimoulas *et al.*, 2013; Degen *et al.*, 2009). Swords *et al.* (2015) provides a framework for expressing their interactions, and we extend and revise that work. 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.

Option Verification. Dimoulas *et al.* (2013) introduce *option contracts*, which allow modules to optionally enforce contracts or eschew contracts while adopting responsibility, avoiding unnecessary contract enforcement. It is possible to replicate this behavior in $\lambda^{\Rightarrow}_{/c}$, using the same pattern of encoding and structures as Dimoulas *et al.* (2013) in the presence of the **option** strategy. Such an implementation would be particularly effective in a chaperone-based system.

Random Checking. Ergün *et al.* (1998) and Dimoulas *et al.* (2013) each describe random testing for program correctness, which has since gained popularity in Clojure's `core.spec` library(Clojure, 2017), where their higher-order function contracts adopt generative checking, producing sample inputs to ensure that the provided function behaves correctly, and their flat contracts are verified eagerly. We can replicate this behavior using a **gen** strategy that accepts a generator g and ensures its input adheres to such generated values:

$$\begin{array}{l}\textbf{check}\ con\ (\textbf{gen}\ g)\ exp\ b\ \rightarrow \\ \quad \textbf{let}\ i\ =\ \textbf{chan} \\ \quad \textbf{in seq}\ \ (\textbf{spawn}\ (\textbf{write}\ i\ (\textbf{catch inl}\ (\textbf{let}\ f\ =\ (con\ (\textbf{read}\ i)\ b) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\textbf{inr}\ (\textbf{seq}\ (\textbf{check-fn}\ f\ g)\ f))))) \\ \qquad\qquad (\textbf{write}\ i\ exp) \\ \qquad\qquad (conres\ (\textbf{read}\ i))\end{array} \tag{23}$$

If f is a function, the **check-fn** procedure will use the provided generator to randomly test it before returning it; otherwise, the strategy behaves as **eager**.

Future Contracts. Dimoulas *et al.* (2011) introduce concurrent contracts via *future contracts*, which we base our **prom** contract verification on, wherein terms and their contracts are sent, as messages, to a secondary evaluator for verification. It is straightforward, however, to imagine a strategy that sends contract-expression pairs to a global, concurrent process and extending effectful operations with synchronization operations.

Security Enforcement Contracts. Moore *et al.* (2016) introduce contracts to model authorization and access control (Moore *et al.*, 2016), which provide a domain-specific language for writing security-centric contracts. It is conceivable to recreate such behavior as a strategy with customized contract inputs, wherein the strategy inspects the structure of its contract and acts accordingly. Doing so would require modifying $\lambda^{\Rightarrow}_{/c}$ to support the dynamically-bound parameter scope system this contract model requires, but we speculate that the process portion of our calculus may be utilized for this task.

Lazy Contracts. First described by Chitil *et al.* (2003) as *assertions* (without blame mechanisms), Degen *et al.* (2009) later described *lazy contracts* as allowing the user evaluator to drive the contract evaluator, wherein the contract evaluator suspends verification until the user evaluator evaluates the subject term. For example, checking a predicate on a pair will not force the pair will suspend verification until the user evaluator evaluates the pair; if the user program never does, the monitor will never verify the contract. Degen *et al.* (2012) give a sketch of implementation, suggesting that the monitor spawn a separate thread that waits until the arguments to the assertion are evaluated and then executes the contract.

This model of monitoring, when translated into $\lambda^{\Rightarrow}_{/c}$, 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 is forcing an expression drives the lazy monitor. Swords *et al.* (2015) give a sketch of a possible implementation that includes recursively parsing the input expression e. This approach closely mirrors the implementation provided by Degen *et al.* (2010), wherein individual call-by-need cells register callbacks for contract monitors. Even so, Degen *et al.* (2012) observe that such lazy verification violates basic blame consistency, and thus has questionable utility.

9.2 Contracts as Processes

Dimoulas *et al.* (2009) and Disney *et al.* (2011) 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 (Barnett *et al.*, 2005; Chen & Roşu, 2007; Havelund & Rosu, 2001). In each case, however, the work has different design goals from Swords *et al.* (2015) and our work. Dimoulas *et al.* (2009) model and explore concurrent contracts; Disney *et al.* (2011) precisely model and explain temporal contracts with non-interference and trace completeness; our work explores modeling contracts as secondary evaluators and varying evaluator interactions to encode multiple verification strategies in the same framework.

Dimoulas *et al.* (2009) introduce concurrent contracts via *future contracts*, wherein terms and their contracts are sent, as messages, to a secondary evaluator for verification. Dimoulas *et al.* (2009) 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 how and when contracts and results are communicated to provide programmers with myriad verification mechanisms that they may choose on a per-contract basis.

Disney *et al.* (2011) utilize multiple contract verification processes to monitor and validate communications as (quasi-)recursive, long-running middle-man processes that medi-

ate 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 the temporal contracts outlined by Disney *et al.* (2011) may be directly encoded in $\lambda^{\Rightarrow}_{/c}$, and that the calculus presented by Disney *et al.* (2011) might be modified to support $\lambda^{\Rightarrow}_{/c}$-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).

9.3 Contracts in Lazy Languages

Chitil *et al.* (2003) 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.* (2006) introduced full contracts with blame and a strict assertion operation for Haskell, inducing our current notion of "semi-eager" verification. Chitil (2012) also encode semi-eager verification in the Haskell language following by almost direct implementation of Findler & Felleisen (2002) into a call-by-name language.

Degen *et al.* (2009) 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 & Felleisen (2002), 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.

Reformulating our own contract interactions in a lazy variation of $\lambda^{\Rightarrow}_{/c}$ 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 stop strict evaluator positions from over-evaluation. For example, consider evaluating

$$\textbf{check}\ (\textbf{pair/c nat/c semi nat/c semi})\ \textbf{eager}\ (5,\text{-}1)B$$

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 $\lambda^{\Rightarrow}_{/c}$.

9.4 Contract Metatheory

Blume & McAllester (2004) present the first exploration of contract metatheory, introducing and exploring contract safety for eager contracts, and Findler & Blume (2006) 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 (Findler, 2014).

Guha *et al.* (2007) present the problem of parametric, polymorphic contracts as first-order values, and Ahmed *et al.* (2017) revisit this problem and prove parametricity. We appeal to this notion in our type system, relying on polymorphic contract combinators.

Finally, Keil & Thiemann (2015) provide denotational 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).

9.5 Surveys of Contract Verification

Degen *et al.* (2009) 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 & Felleisen (2011) go further, classifying different approaches through observational equivalence, based on when and how they are enforced (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.* (2009): it is only allowed to inspect parts of the program that are evaluated at runtime. Degen *et al.* (2012) revisit different contract verification systems in call-by-need languages, evaluating them for completeness and meaning preservation, classifying a number of contract systems (Findler & Felleisen, 2002; Hinze *et al.*, 2006; Chitil & Huch, 2006; Xu *et al.*, 2009; Degen *et al.*, 2009) by these properties. Conversely, our work follows Swords *et al.* (2015): instead of classifying verification strategies via secondary properties, we have presented a semantic-based comparison by directly encoding each verification strategy in $\lambda^{\Rightarrow}_{/c}$.

9.6 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.* (2004), Flanagan (2006), and Greenberg *et al.* (2010) each present a model of contract usage similar to refinement types (Freeman, 1994) tracking these contracts at the type level and enforcing (and accruing) them as values flow through these types. The work by Greenberg *et al.* (2010) in particular has recently been the subject of multiple extensions, including space efficiency (Greenberg, 2015), data-types (Sekiyama *et al.*, 2015), and stateful contracts (**?**). While manifest contracts are closely related to standard contracts (and Racket's chaperone and impersonator system (Strickland *et al.*, 2012)), our work treats contracts as values instead of type-level terms.

Xu *et al.* (2009) and Nguyen *et al.* (2014) 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 (at `https://clojure.org/guides/spec`) also supports enforcing contracts in different ways. Unlike our multi-strategy approach, however, the `spec` library allows users to determine which programming phase to check the provided spec-

ification (or "spec") at: the spec written for the program may be used 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 (2011) 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 memory changes to be observed and rolled back, supporting specification contracts alongside "framing" contracts, which ensure programs do not exhibit specific behaviors, and separation contracts, which combine these to provide separation logic-esque behavior. The roll-back capabilities also allow the program to undo effects in the event of contract violations, and integrating such a mechanism into $\lambda^{\Rightarrow}_{/c}$ may allow us to address the effect-related shortcomings of promise-based and concurrent contract verification. This integration remains as future work.

9.7 Complete Blame and Contract Monitoring

Multiple approaches to runtime verification blame assignment have been proposed (Wadler & Findler, 2009; Ahmed *et al.*, 2011; Greenberg *et al.*, 2010; Dimoulas *et al.*, 2011; 2012), 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 (Dimoulas *et al.*, 2011)).

Ensuring complete monitoring, defined by Dimoulas *et al.* (2012), in particular, introduce the notion of *complete monitoring*, a fundamental correctness criterion for contract systems that generalizes correct blame assignments by ensuring each value that moves between components is monitored. Unfortunately, proving this property for $\lambda^{\Rightarrow}_{/c}$ is particularly challenging for a number of reasons. First, their definition relies on multiple modules. We would need to first extend $\lambda^{\Rightarrow}_{/c}$ with a module model (likely as interacting processes, following Disney *et al.* (2011)). Second, Dimoulas *et al.* (2012) 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 when a delayed reference is forced after being communicated). Finally, complete monitoring is tells us little about some strategies, including **conc**: complete monitoring only requires that the program terminates with a value, diverges, or correctly detects a contract violation, and, if every contract is enforced under **conc**, a $\lambda^{\Rightarrow}_{/c}$ 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.

Gradual Typing. Gradual typing uses runtime verification in the form of casts and coercions to ensure that values have the correct types as they flow through the program (Siek & Taha, 2006). Similar to contract verification, there are a variety of ways to enforce these properties and track blame (Herman *et al.*, 2007; Wadler & Findler, 2009; Vitousek

et al., 2014). Siek *et al.* (2009), in particular, provide a discussion of different approaches in the literature, recreating each by adding optional reduction rules to the same calculus and demonstrating how different sets of these reduction rules exhibit different behaviors. Our work here similarly compares the field of contract verification strategies, but we diverge in our use of processes, our direct encoding of each strategy into an underlying core calculus, and our strategies' coexistence in the same surface language. Applying our communication-based approach to gradual typing remains as future work, but hints as the possibility of multiple error detection semantics existing together in the same language.

10 Conclusion

In this paper, we have presented a unifying framework for contract semantics, refining and extending the work presented in Swords *et al.* (2015). These extensions include a clarified semantic model that clarifies *which* process evaluates *which* part of each monitor, an explicit semantics for an **fconc** contract verification (thus demonstrating how new contract systems may use target $\lambda^{\Rightarrow}_{/c}$), extended explanations of the previous work's multi-strategy approach to verification, and an extended presentation of metatheoretical results. There is a sample implementation of this work available for Racket at `https://github.com/cgswords/ccon`.

Future Work. There is still much to explore in strategy-based contract monitoring. At the beginning of this paper, we identified three dimensions in the design space that each allow for a gradient of answers and whose answers lead to further questions about the behavior of runtime contract verification. The strategies here are not a covering of the design space, but an exposure, all expressed in a single, canonical framework that can be extended and used to implement and compare other verification patterns. This approach also hints at potential strategy combinators, or "meta-strategies", that allow us to combine and modify verification behavior. Finally, this unified approach gives us room to ask how this work applies to other runtime verification systems, explore the relation between this style of monitoring and aspect-oriented programming, and investigate what lies beyond strategies.

Acknowledgments

This research was funded by the National Science Foundation under grant numbers 1117635, 1217454, 1540276, 1518844, and the National Security Agency. We would also like to thank our JFP reviewers and editors for their excellent discussion and feedback.

References

Ahmed, Amal, Findler, Robert Bruce, Siek, Jeremy G., & Wadler, Philip. (2011). Blame for all. *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*. POPL '11. ACM.

Ahmed, Amal, Jamner, Dustin, Siek, Jeremy G., & Wadler, Philip. (2017). Theorems for free for free: Parametricity, with and without types. *Proceedings of the 22th International Conference on Functional Programming*. ICFP '17. ACM.

Ambler, A. L., Good, D. I., Browne, J. C., Burger, W. F., Choen, R. M., Hoch, C. G., & Wells, R. E. (1977). Gypsy: a language for specification and implementation of verifiable programs. *Sigplan*, 1–10. Proceedings of the ACM Conference on Language Design for Reliable Software.

Barnett, Mike, Leino, K. Rustan M., & Schulte, Wolfram. (2005). The spec# programming system: An overview. *Proceedings of the 2004 international conference on construction and analysis of safe, secure, and interoperable smart devices*. CASSIS '04. Springer-Verlag.

Blume, Matthias, & McAllester, David. (2004). A sound (and complete) model of contracts. *Proceedings of the 9th International Conference on Functional Programming*. ICFP '04. ACM.

Chen, Feng, & Roşu, Grigore. (2007). Mop: An efficient and generic runtime verification framework. *Proceedings of the 22nd annual conference on object-oriented programming systems and applications*. OOPSLA '07. ACM.

Chitil, Olaf. (2012). Practical typed lazy contracts. *Proceedings of the 17th International Conference on Functional Programming*. ICFP '12. ACM.

Chitil, Olaf, & Huch, Frank. (2006). A pattern logic for prompt lazy assertions in haskell. *Symposium on implementation and application of functional languages*. IFL '06. Springer.

Chitil, Olaf, McNeill, Dan, & Runciman, Colin. (2003). Lazy assertions. *Symposium on implementation and application of functional languages*. IFL '03. Springer-Verlag.

Clojure. (2017). *Clojure core.async*.

Degen, Markus, Thiemann, Peter, & Wehr, Stefan. (2009). True Lies: lazy contracts for lazy languages (faithfulness is better than laziness). *Arbeitstagung Programmiersprachen (ATPS)*. ATPS '09. Springer.

Degen, Markus, Thiemann, Peter, & Wehr, Stefan. (2010). Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *The Journal of Logic and Algebraic Programming*, **79**.

Degen, Markus, Thiemann, Peter, & Wehr, Stefan. (2012). The interaction of contracts and laziness. *Proceedings of the 2012 workshop on partial evaluation and program manipulation*. PEPM '12. ACM.

Dimoulas, Christos, & Felleisen, Matthias. (2011). On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems*, **33**(5).

Dimoulas, Christos, Pucella, Riccardo, & Felleisen, Matthias. (2009). Future contracts. *Proceedings of the 11th Conference on Principles and Practice of Declarative Programming*. PPDP '09. ACM.

Dimoulas, Christos, Findler, Robert Bruce, Flanagan, Cormac, & Felleisen, Matthias. (2011). Correct blame for contracts: No more scapegoating. *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*. POPL '11. ACM.

Dimoulas, Christos, Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2012). Complete monitors for behavioral contracts. *European symposium on programming*. ESOP '12. Springer-Verlag.

Dimoulas, Christos, Findler, Robert Bruce, & Felleisen, Matthias. (2013). Option contracts. *Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. ACM.

Disney, Tim, Flanagan, Cormac, & McCarthy, Jay. (2011). Temporal higher-order contracts. *Proceedings of the 16th International Conference on Functional Programming*. ICFP '11. ACM.

Ergün, Funda, Kannan, Sampath, Kumar, S. Ravi, Rubinfeld, Ronitt, & Viswanathan, Mahesh. (1998). Spot-checkers. *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*. STOC '98. ACM.

Findler, Robert Bruce. (2014). Behavioral software contracts. *Proceedings of the 19th International Conference on Functional Programming*. ICFP '14. ACM.

Findler, Robert Bruce, & Blume, Matthias. (2006). Contracts as pairs of projections. *Proceedings of the 8th International Conference on Functional and Logic Programming*. FLOPS '06. Springer-Verlag.

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for higher-order functions. *Proceedings of the 7th International Conference on Functional Programming*. ICFP '02. ACM.

Findler, Robert Bruce, Guo, Shu-Yu, & Rogers, Anne. (2008). Lazy contract checking for immutable data structures. *Implementation and Application of Functional Languages*. Springer-Verlag.

Flanagan, Cormac. (2006). Hybrid type checking. *Conference record of the 33rd symposium on principles of programming languages*. POPL '06. ACM.

Flatt, Matthew, & PLT. (2010). *Reference: Racket*. Tech. rept. PLT-TR-2010-1. PLT Inc. `http://racket-lang.org/tr1/`.

Freeman, Tim. (1994). *Refinement types for ML*. Ph.D. thesis, Carnegie Mellon University.

Friedman, Daniel, & Wise, David. (1976). *The impact of applicative programming on multiprocessing*. Tech. rept. 52. Indiana University, Computer Science Department.

Greenberg, Michael. (2015). Space-efficient manifest contracts. *Proceedings of the 42nd Symposium on Principles of Programming Languages*. POPL '15. ACM.

Greenberg, Michael, Pierce, Benjamin C., & Weirich, Stephanie. (2010). Contracts made manifest. *Proceedings of the 37th Symposium on Principles of Programming Languages*. POPL '10. ACM.

Guha, Arjun, Matthews, Jacob, Findler, Robert Bruce, & Krishnamurthi, Shriram. (2007). Relationally-parametric polymorphic contracts. *Proceedings of the 2007 Symposium on Dynamic languages*. DLS '07. ACM.

Havelund, Klaus, & Rosu, Grigore. (2001). *Monitoring java programs with java pathexplorer*. Tech. rept. NASA Ames Research Center.

Herman, David, Tomb, Aaron, & Flanagan, Cormac. 2007 (April). Space-efficient gradual typing. *Page XXVIII of: Trends in functional prog.* TFP '07.

Hinze, Ralf, Jeuring, Johan, & Löh, Andres. (2006). Typed contracts for functional programming. *Proceedings of the 8th International Conference on Functional and Logic Programming*. FLOPS '06. Springer-Verlag.

Ingerman, P. Z. (1961). Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, **4**(1).

Jeffrey, A. (1998). Semantics for core Concurrent ML using computation types. *Higher Order Operational Techniques in Semantics*. Cambridge University Press.

Keil, Matthias, & Thiemann, Peter. (2015). Blame assignment for higher-order contracts with intersection and union. *Proceedings of the 20th International Conference on Functional Programming*. ICFP '15. ACM.

Meyer, Bertrand. (1992). *Eiffel: the language*. Prentice-Hall, Inc.

Moore, Scott, Dimoulas, Christos, Findler, Robert Bruce, Flatt, Matthew, & Chong, Stephen. (2016). Extensible access control with authorization contracts. *Proceedings of the 2016 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '16. ACM.

Nguyen, Phúc C., Tobin-Hochstadt, Sam, & Van Horn, David. (2014). Soft contract verification. *Proceedings of the 19th International Conference on Functional Programming*. ICFP '14. ACM.

Ou, Xinming, Tan, Gang, Mandelbaum, Yitzhak, & Walker, David. (2004). Dynamic typing with dependent types. *Ifip tcs*.

Owens, Zachary. (2012). Contract monitoring as an effect. *Hope*.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press.

Reppy, John H. (1993). Concurrent ML: Design, application and semantics. *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*. Springer-Verlag.

Reppy, John H. (1999). *Concurrent programming in ML*. Cambridge University Press.

Sekiyama, Taro, Nishida, Yuki, & Igarashi, Atsushi. (2015). Manifest contracts for datatypes. *Proceedings of the 42nd Symposium on Principles of Programming Languages*. POPL '15. ACM.

Shinnar, Avraham. (2011). *Safe and effective contracts*. Tech. rept. Harvard University.

Siek, Jeremy G., & Taha, Walid. (2006). Gradual typing for functional languages. *Scheme and functional programming workshop*.

Siek, Jeremy G., Garcia, Ronald, & Taha, Walid. (2009). Exploring the design space of higher-order casts. *European symposium on programming*. ESOP '09. Springer-Verlag.

Strickland, T. Stephen, Tobin-Hochstadt, Sam, Findler, Robert Bruce, & Flatt, Matthew. (2012). Chaperones and impersonators: Run-time support for reasonable interposition. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. ACM.

Swords, Cameron, Sabry, Amr, & Tobin-Hochstadt, Sam. (2015). Expressing contract monitors as patterns of communication. *Proceedings of the 20th International Conference on Functional Programming*. ICFP '15. ACM.

Vitousek, Michael M., Siek, Jeremy G., Kent, Andrew, & Baker, Jim. (2014). Design and evaluation of gradual typing for Python. *Dynamic languages symposium*. DLS '14.

Wadler, Philip, & Findler, Robert Bruce. (2009). Well-typed programs can't be blamed. *Proceedings of the 18th European Symposium on Programming Languages and Systems*. ESOP '09. Springer-Verlag.

Xu, Dana N., Peyton Jones, Simon, & Claessen, Koen. (2009). Static contract checking for haskell. *Proceedings of the 36th Annual Symposium on Principles of Programming Languages*. POPL '09. ACM.