Expressing Contract Monitors as Patterns of Communication

Cameron Swords Amr Sabry Sam Tobin-Hochstadt

Indiana University
School of Informatics and Computing
Bloomington, Indiana 47401, USA
{cswords,sabry,samth}@indiana.edu

Abstract

We present a new approach to contract semantics which expresses myriad monitoring strategies using a small core of foundational communication primitives. This approach allows multiple existing contract monitoring approaches, ranging from Findler and Felleisen's original model of higher-order contracts to semi-eager, parallel, or asynchronous monitors, to be expressed in a single language built on well-understood constructs. We prove that this approach accurately simulates the original semantics of higher-order contracts. A straightforward implementation in Racket demonstrates the practicality of our approach which not only enriches existing Racket monitoring strategies, but also support a new style of monitoring in which collections of contracts collaborate to establish a global invariant.

Categories and Subject Descriptors F.3.1. [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.3. [*Programming Languages*]: Language Constructs and Features

Keywords Lazy monitoring, asynchronous monitoring, behavioral specification

1. Introduction

Since they were introduced by Meyer [25], behavioral contracts have become an integral part of modern programming practice, where they are used to express predicates as pre- and post-conditions on procedures. In a first-order setting, run-time enforcement is well-studied [25, 27, 28]: the pre-condition predicate is applied to the input; the function is run if the precondition holds; and the post-condition is checked on the result.

However, when the setting changes from first-order to higher-order functions, effectful operations, large data structures, or even lazy languages, contract checking becomes more complex. With higher-order functions, contracts must be postponed until the function is used [16]. With large data structures, programmers wish to avoid checking more than needed. With lazy languages, contracts can force excess evaluation. Fortunately, all of these issues have been tackled, with researchers proposing solid theoretical foundations [4, 15, 16], language integration accommodating existing semantics [6, 11, 19, 22], and new forms of contracts [1, 11, 18, 20, 32]. Contracts are now available in a broad range of production systems.

Even for the specific task of monitoring functions and primitive values, there are a broad number of approaches to contract monitoring presented in the literature, from eager contracts à la Findler and Felleisen [16] to future contracts as presented by Dimoulas et al. [10], and even broader shapes such as the temporal approach presented by Disney et al. [13]. There are, in fact, so many approaches to contract monitoring that Degen et al. [7, 8] and Dimoulas and Felleisen [9] have presented surveys of different approaches, discussing their similarities and differences.

Despite appearances, this cornucopia of monitoring strategies share a common core. Going back to first principles, checking that a given code fragment satisfies a contract clearly requires executing some "assertion" code, and the execution of this assertion code is conceptually distinct from the execution of the original code fragment. There is no fundamental reason why either of these modes of execution should be given supremacy. In particular, taking the view that the contract execution is somewhat subservient to the main thread of execution only restricts and complicates the language; only in the simplest cases does the interaction of the two executions follow a routine master/slave or call/return pattern. In many cases, it is conceptually clearer to view the two executions as proceeding independently, synchronizing at specific points. Prior implementations and semantic foundations for contracts have merged these non-trivial patterns of interaction into the single, fixed evaluation semantics for the host language. This not only restricts the power of contracts but obscures the powerful idea that contract checking is just a separate process of execution that interacts with the underlying program in a variety of communication patterns.

We tackle this problem head-on, giving a unifying account of multiple contract monitoring semantics while exposing their underlying communication patterns. The communication-centric account clarifies the precise differences between monitoring semantics and how they impact the overall flow of program execution.

Outline. This paper presents a new model for interpreting and expressing contract monitoring semantics as models of communication that facilitate coexistence and semantic interoperability in the presence of contracts. This paper proceeds as follows: we outline our approach through examples of contract monitoring (§2). Next we present a calculus with CML-style concurrency operations, demonstrating how these operations, combined with exceptions and delaying mechanisms, may recover varied methods of contract monitoring, including eager, semi-eager, future, and asynchronous monitoring [7, 8, 10, 16, 18] (§3–4). Then we prove that our model accurately simulates the original semantics of Findler and Felleisen [16] (§6). Next we present a thread-based implemen-

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM 978-1-4503-3669-7/15/08…$15.00.
http://dx.doi.org/10.1145/0.0

tation of our framework in Racket and use this system to create a structural contract sketched by Findler et al. [18], but with improved algorithmic bounds and performance results (§7). Finally we outline how to embed a number of additional monitoring approaches into our semantic model, including lazy, temporal, and statistical monitors (§8), discuss related work (§9), and conclude.

2. Background and Examples

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 monitors may interact with the underlying program evaluator and the precise semantics of the monitor itself. This semantic decision is a permanent fixture of the language, often lacking facilities to extend or alter the strategy.

In this section we start to address this problem, abstracting away from this "one-strategy" approach in favor of a many-strategy user language. This language abstracts over how contracts may interact with the underlying evaluator, parameterizing contract monitors with monitoring strategies which indicate precisely how and when a given contract may interface with the user program. This moves the decision of how contracts should be monitored into the programmer's hands, allowing users to naturally describe contract behavior on a program-by-program basis.

Eager Predicate Contracts. Our first example is a predicate contract that verifies its input is a natural number [1]:

$$\mathsf{nat/c} \triangleq \mathsf{pred/c}\ (\lambda x.\ x \geq 0)$$

We can now use this contract to construct a monitored subexpression inside a larger expression, using the **eager** strategy:

$$1 + \mathsf{check\ nat/c}\ \mathbf{eager}\ 5$$

When the evaluation of this expression encounters the check parameterized by **eager**, the user computation is suspended and the monitor assumes control of the evaluation. Eager monitors completely verify their contract at assertion time: if the input value is valid, the monitor terminates with that value result, and if the input does not satisfy the contract, the monitor terminates at assertion time with an exception[2]. In either case, the user evaluation is resumed with the monitor's result.

This interaction corresponds to *Eager Monitoring* in Figure 1: the check form constructs a monitoring expression (colored red) to verify the contract, pausing the user evaluation until it is complete. The bold line indicates evaluator construction, the double-arrow indicates evaluator synchronization, and the normal arrows indicate evaluation.

Eager Pair Contracts. Eager monitors produce contract violation even if the program does not rely on the value for its final result, and thus unused values may produce contract violations. To demonstrate this quality, consider a second contract combinator over pairs, written pair/c. Structural contracts require subcomponents that describe how the substructures should be monitored, and thus pair contracts accept a subcontract and strategy for each of the first and second elements of the pair:

$$\begin{array}{lll} \mathsf{nat/pcE} & \triangleq & \mathsf{pair/c} \\ & & \quad \mathsf{nat/c}\ \mathbf{eager} \\ & & \quad \mathsf{nat/c}\ \mathbf{eager} \end{array}$$

[1] By convention, we name contracts and contract combinators with a trailing /c; all of the combinators we use in this section are defined as library functions in Sec. 5.

[2] We omit blame in this section for simplicity.

This pair contract can itself be eagerly enforced, which will eagerly monitor nat/c on each of the elements of the pair, regardless of their usage in the program:

$$\begin{array}{lll} \mathsf{fst\ (check\ nat/pcE}\ \mathbf{eager}\ (5, 6)) & \Rightarrow & 5 \\ \mathsf{fst\ (check\ nat/pcE}\ \mathbf{eager}\ (5, \text{-}1)) & \Rightarrow & \mathsf{exn} \end{array}$$

The Cost of Eager Contracts. The strict enforcement strategy of eager contracts is not always a perfect-fit solution. For example, an eagerly-monitored contract that verifies its input is a prime number may require immense computational effort while the user evaluation is suspended:

$$\mathsf{check\ prime/c}\ \mathbf{eager}\ (2 * 37^{63} + 567)$$

While such enforcement may be ideal (or even necessary) in many situations, this approach may range from computationally intensive to impossible for large or infinite structures (such as streams). If this is the only monitoring strategy available, programmers will find it too expensive to validate many rich properties.

Asynchronous Contracts. Instead of suspending the user evaluator during contract verification, other approaches allow the user evaluator to proceed while the contract is concurrently enforced [10, 13]. In the simplest variant of the idea, the monitor is concurrently verified and halting the computation with an error if the contract is violated, and the user process is otherwise unimpeded.

This approach corresponds to *Asynchronous Monitoring* in Figure 1: the check form constructs a monitored expression (colored red) to verify the contract, continuing the user evaluation in parallel. If the monitor verifies the contract, it silently ends its execution. If an error is detected, however, the entire computation might halt with the error. For example, we may asynchronously enforce nat/c in a number of expressions by changing **eager** to **async**:

$$\begin{array}{lll} \mathsf{check\ nat/c}\ \mathbf{async}\ 5 & \Rightarrow & 5 \\ (\lambda x.\ x\ +\ 5)\ (\mathsf{check\ nat/c}\ \mathbf{async}\ \text{-}1) & \Rightarrow & 4\ \textit{or}\ \mathsf{exn} \end{array}$$

We use "or" in the second example because asynchronous monitors may not complete before the user evaluator. Asynchronous monitoring only guarantees "best-effort checking": if a contract is too large or complex to verify during the program's execution, and the program otherwise terminates correctly, then the remaining verification work is discarded. As a result, asynchronous contract enforcement is often "too weak", precisely because the two evaluators are completely disjoint. It may be preferable to facilitate a synchronization between the two evaluators at the user program's request.

Future Contracts. In their original presentation [10], *future contracts* track a master (user) process and a slave (monitoring) process, synchronizing during I/O events to enforce pending contracts. We take a more traditional approach to future monitors, providing the initiating process with a *promise*—a reference to the monitor result à la computational futures [3, 21]. When the initiating evaluator forces the promise, the expression retrieves the contract result, yielding either the original value or an error. This style of contract monitoring allows the user program to control *when* the program will wait for a contract result by forcing the promise (represented as a ref).

$$\begin{array}{lll} \mathsf{check\ nat/c}\ \mathbf{future}\ 5 & \Rightarrow & \mathsf{ref} \\ \mathsf{check\ nat/c}\ \mathbf{future}\ \text{-}1 & \Rightarrow & \mathsf{ref} \\ \mathsf{force\ (check\ nat/c}\ \mathbf{future}\ 5) & \Rightarrow & 5 \\ \mathsf{force\ (check\ nat/c}\ \mathbf{future}\ \text{-}1) & \Rightarrow & \mathsf{exn} \end{array}$$

This series of interactions correspond to *Future Monitoring* in Figure 1: the check form constructs a monitored expression (colored red) to verify the contract and a future-fulfillment mechanism (as another process, colored blue) before returning a reference to the future-fulfillment mechanism to the user process. This allows the user evaluator to continue computation until the value is required while a concurrent process enforces the contract, minimizing the time required in the user evaluator for contract enforcement. To

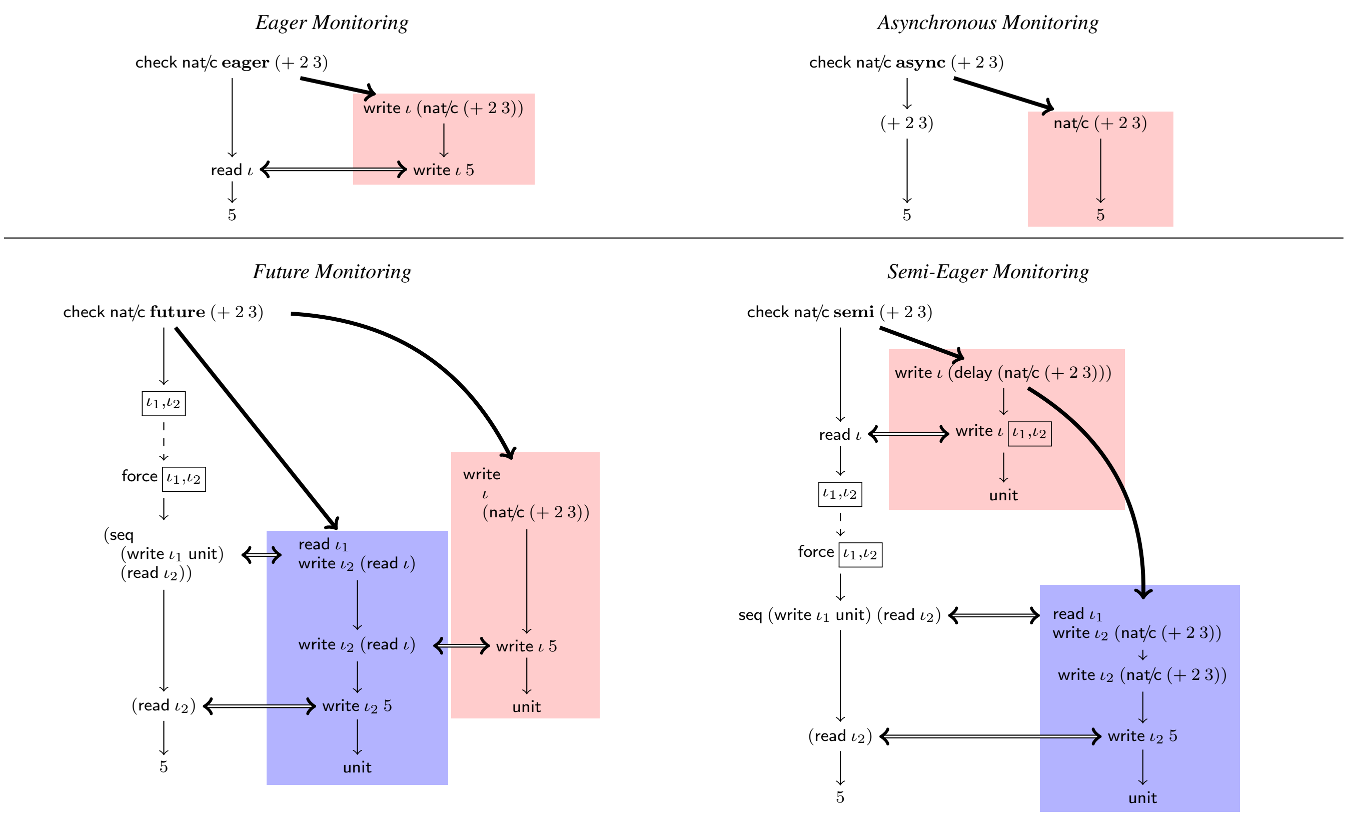


Figure 1. Communication patterns for eager, semi-eager, future, and asynchronous contract monitoring. The red regions indicate contract-checking evaluators and the blue regions indicate delay-reference evaluators.

further demonstrate this behavior, consider a revised contract over pairs that checks its sub-contracts using **future**:

$$\mathsf{nat/pcF} \triangleq \mathsf{pair/c}\ \mathsf{nat/c}\ \textbf{future}\ \mathsf{nat/c}\ \textbf{future}$$

Now we must explicitly force the subcomponents of the pair:

$$\begin{array}{rl}
\mathsf{fst}\ (\mathsf{force}\ (\mathsf{check}\ \mathsf{nat/pcF}\ \textbf{future}\ (5,\text{-}1))) & \Rightarrow\ \mathsf{ref}\\
\mathsf{force}\ (\mathsf{fst}\ (\mathsf{force}\ (\mathsf{check}\ \mathsf{nat/pcF}\ \textbf{future}\ (5,\text{-}1)))) & \Rightarrow\ 5\\
\mathsf{force}\ (\mathsf{snd}\ (\mathsf{force}\ (\mathsf{check}\ \mathsf{nat/pcF}\ \textbf{future}\ (5,\text{-}1)))) & \Rightarrow\ \mathsf{exn}\\
\mathsf{force}\ (\mathsf{fst}\ (\mathsf{check}\ \mathsf{nat/pcF}\ \textbf{eager}\ (5,\text{-}1))) & \Rightarrow\ 5\\
\mathsf{force}\ (\mathsf{check}\ \mathsf{nat/pcE}\ \textbf{future}\ (5,\text{-}1)) & \Rightarrow\ \mathsf{exn}
\end{array}$$

The last two examples are of particular interest: each pair contract requires three strategies for each of its first sub-component, second sub-component, and the overall pair. If we use **future** for all three, we receive a promise that, when forced, returns a pair of further promises. Our framework allows the free intermixing of monitoring strategies, however, so we can construct a pair contract that is enforced at assertion time while delaying each of the subcomponent contract, or that eagerly enforces its subcomponents when forced (as in the last example, which uses nat/pcE, the eager pair contract).

Semi-Eager Contracts. The presentation of future contracts presumes that the contract monitor is too computationally complex to perform while suspending the user evaluator. Delaying the actual contract evaluation, however, is often sufficient: if a structural contract may be checked in layers and pieces as the program traverses the structure, piece-wise demand-time contract enforcement is often sufficient. This suggests a fourth model of contract monitoring which delays the monitor enforcement until the user evaluator forces the expression. Findler and Felleisen first used so-called *semi-eager* evaluation to model contracts across module boundaries, enforcing contracts only when invoked by the client module [16]. Hinze et al. [22] and Degen et al. [7] model this approach by encoding the Findler-Felleisen semantics directly in Haskell, and Chitil [5] builds on this encoding, introducing a number of combinators that exploit this delaying behavior.

Future monitoring closely mirrors semi-eager monitoring: while the former immediately performs contract enforcement but delays evaluator communication, the latter delays contract enforcement and immediately performs communication afterwards. As a result, the two strategies behave identically from the user's perspective in the absence of effectful operations and computational time:

$$\begin{array}{lcl}
\mathsf{check}\ \mathsf{nat/c}\ \textbf{semi}\ 5 & \Rightarrow & \mathsf{ref}\\
\mathsf{force}\ (\mathsf{check}\ \mathsf{nat/c}\ \textbf{semi}\ \text{-}1) & \Rightarrow & \mathsf{exn}\\
(\lambda x.\ 1\ +\ \mathsf{force}\ x)\ (\mathsf{check}\ \mathsf{nat/c}\ \textbf{semi}\ 5) & \Rightarrow & 6\\
(\lambda x.\ 1\ +\ \mathsf{force}\ x)\ (\mathsf{check}\ \mathsf{nat/c}\ \textbf{semi}\ \text{-}1) & \Rightarrow & \mathsf{exn}
\end{array}$$

While the usage looks identical with respect to expected outputs, the underlying semantic differences expose an axis of variation for contract monitors: *where* the delaying mechanism occurs in the monitoring process. This difference is apparent in the pattern presented as *Semi-Eager Monitoring* in Figure 1: we move the delay mechanism creation from the user evaluator to the monitoring evaluator, and thus expose a natural variance in monitoring semantics.

Data Structural Contracts. The contracts we have presented so far omit a large part of the design space for contracts: neither predicates nor pairs readily lend themselves to recursive contracts. Binary-search trees, conversely, provide an interesting venue for exploring structural contracts in the large, presenting opportunities for contract enforcement that contrast well with the asymptotic and algorithmic obligations of the underlying structure. For example, we might construct a contract to ensure that a given tree is indeed organized in sorted order. We can do this with a dependent tree contract that takes four subcontracts and strategies for each of the leaf, left subtree, node value, and right subtree:

```
fix bst/E $lo\ hi$.
    tree/dc (pred/c null?)                              eager   leaf
            ($\lambda n.$ bst/E $lo\ n$)                eager   left
            (pred/c ($\lambda x.\ lo \leq x$ and $x \leq hi$)) eager   value
            ($\lambda n.$ bst/E $n\ hi$)                eager   right
```

This contract ensures that each leaf of the tree is a null value and each value that occurs in the tree is within the correct numeric bounds. Under eager monitoring 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 balanced tree. This style of monitoring is often preferable to ensure program safety, but such traversals become problematic as the structure increases in size. We can forgo such a strong guarantee, however, and opt for semi-eager contract enforcement:

```
fix bst/S $lo\ hi$.
    tree/dc (pred/c null?)                              eager
            ($\lambda n.$ bst/S $lo\ n$)                semi
            (pred/c ($\lambda x.\ lo \leq x$ and $x \leq hi$)) eager
            ($\lambda n.$ bst/S $n\ hi$)                semi
```

This contract will enforce the invariant on exactly the nodes we visit during the program, which will recover $O(\log n)$ complexity for insertion into a balanced tree. Even so, we have tied the user evaluator to this contract, relying on the user program's evaluation to enforce the contract. Departing from such a coupling requires a full separation of the monitoring evaluator from the user evaluator.

Future monitors provides two approaches to such separation. In the first, we construct such a monitor by replacing the two uses of **semi** with **future** in the definition of bst/S, yielding bst/F. This third implementation of the binary-search tree invariant yields a unique situation: the future monitors will traverse the entire structure of the tree, but the user evaluator only waits on the results that are relevant to the completion of the program. Thus we may complete the user evaluator's computation without enforcing the invariant over the entire tree.

Alternatively, we might enforce bst/E concurrently under the **future** strategy, demanding the promise at the end of the entire user program. This will allow us to continue with a computation, verifying the contract concurrently and retrieving the final result at the last possible moment. This last alternative for future monitoring may also be constructed with asynchronous monitoring: the monitor will traverse the entire tree and enforce the contract, reporting an error only if and when it is detected.

Function Contracts. Monitoring a function contract verifies contracts on a function's input and output values. Like a pair contract, a function contract consists of two sub-contracts and their associated strategies: the first contract, or *pre-condition*, is enforced on function inputs, and the second contract, or *post-condition*, is asserted on the result of the function application. These contracts are constructed with the function contract combinator func/c:

```
fnat/cE  $\triangleq$  func/c nat/c eager nat/c eager
fnat/cS  $\triangleq$  func/c nat/c semi nat/c semi
```

Function contracts deviate from pair contracts in one important way: while it was possible to write eager pair contracts that might produce inefficiency or diverge, the equivalent approach for function contracts is untenable. For any infinite set of valid inputs or outputs, such as the natural numbers, it is generally impossible to traverse the entire domain and codomain of a procedure to ensure that each adheres to the pre- and post-condition.

To avoid this problem, functional contract enforcement postpones the pre- and post-condition contracts until the function's precise input is available. This delay manifests as a secondary λ-abstraction around the monitored procedure, and thus while a function contract may be monitored under any strategy, that overall strategy only determines *when* the wrapped procedure is constructed [4]. (The literature suggests an alternative method for checking function contracts through statistical verification [12, 14], which we discuss in Sec. 8.) We define two monitored procedures using the contracts above:

```
sub1/mE  $\triangleq$  check fnat/cE eager ($\lambda x.\ x - 1$)
sub1/mS  $\triangleq$  check fnat/cS eager ($\lambda x.$ force $x - 1$)
```

We use **eager** in both cases to avoid delaying the construction of the wrapped procedure. Applications proceed as follows:

sub1/mE 5	$\Rightarrow$	4	(ef_1)	sub1/mS 5	$\Rightarrow$	ref	(sf_1)
sub1/mE -1	$\Rightarrow$	exn	(ef_2)	sub1/mS -1	$\Rightarrow$	exn	(sf_2)
sub1/mE 0	$\Rightarrow$	exn	(ef_3)	sub1/mS 0	$\Rightarrow$	ref	(sf_3)
				force (sub1/mS 0)	$\Rightarrow$	exn	(sf_4)

The first three examples are as expected: no contract is violated in the first, the pre-condition is violated in the second, and the post-condition is violated in the third. The semi-eager examples are more intricate: example (sf_1) behaves as expected, returning a reference that will yield 4 when forced, example (sf_2) produces an exception when the value monitored by the pre-condition is forced in the body of the function, and examples (sf_3) and (sf_4) indicate that the post-condition result must be demanded before usage.

The Many-Strategy Approach. Unfortunately, none of these approaches provide a single "silver bullet" solution to contract monitoring. While Degen et al. [7] conclude that "faithfulness is better than laziness", we assert that each strategy is an ideal fit in a number of situations, and so fixing the contract evaluator with a single strategy is a poor fit for programmer flexibility.

Furthermore, there is evidence that some useful contracts cannot be expressed with any of the strategies or combinators we have introduced so far [18]. For example, consider checking the fullness of a binary search tree, which ensures that a tree of height n has 2^n nodes. This property is easy to verify by traversing the tree, but it poses subtle issues for contract monitoring strategies. We would like to check as much of the tree as possible without forcing additional evaluation, but the *contract* requires traversal and thus we require some mechanism to back-propagate values to waiting contracts as the tree is explored.

As we will see in Sec. 7.2, asynchronous and semi-eager strategies combined with communication allow us to write this back-propagation mechanism, arranging a series of callbacks to verify the tree is full. This contract requires careful construction, but it will allow us to signal an error after we have traversed any unbalanced part of the subtree to its leaves.

The contracts we have examined thus far are all built from the same primitive effects: exceptions, concurrent processes with communication, and delaying and forcing mechanisms. In the next sections we will outline a core calculus with these operations and demonstrate how to construct general, generic, and strategy-agnostic monitoring mechanisms from them.

3. A Concurrent Calculus for Contracts

Having demonstrated the core ideas of this new approach to contract monitoring, we turn our attention toward formalizing these features in λ_{CC}, the calculus of concurrent contracts. This calculus combines several standard features, including pure λ-expressions, exceptions, delay and force primitives, and a process model inspired by Concurrent ML [24, 29, 30]. The pure λ-calculus portion of the calculus includes several common forms (the unshaded portion of Figure 2), including if, case, pairs, and recursive bindings. Each of the additional language layers provides a separate level of abstraction in λ_{CC} aimed at a distinct class of clients:

$$
\begin{array}{rcl}
e & := & x \mid v \mid e\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (e, e) \\
 & \mid & \mathit{unop}\ e \mid \mathit{binop}\ e\ e \mid \mathsf{case}\ (e;\ \mathsf{inl}\ x\ \triangleright\ e;\ \mathsf{inr}\ x\ \triangleright\ e) \\
 & \mid & \mathsf{injl}\ \ e \mid \mathsf{injr}\ \ e \\
 & \mid & \mathsf{check}\ e\ e\ e\ B \mid \mathsf{force}\ e \mid \mathsf{raise}\ e \\
 & \mid & \mathsf{read}\ e \mid \mathsf{chan}\ e \mid \mathsf{write}\ e\ e \\
 & \mid & \mathsf{spawn}\ \mathcal{T}\ e \mid \mathsf{delay}\ e \mid \mathsf{catch}\ e\ e \\
v & := & \lambda x.\, e \mid \mathsf{fix}\ f\ x.\, e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid (v, v) \\
 & \mid & \mathsf{unit} \mid n \mid \mathit{str} \mid B \mid s \\
 & \mid & \iota \mid \mathsf{dref}_{\iota_1, \iota_2} \\
B & := & (\mathit{str}, \mathit{str}, \mathit{str}) \\
s & := & \mathbf{eager} \mid \mathbf{semi} \mid \mathbf{future} \mid \mathbf{async} \mid \cdots
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} & := & \square \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (\mathcal{E}, e) \mid (v, \mathcal{E}) \\
 & \mid & \mathit{unop}\ \mathcal{E} \mid \mathit{binop}\ \mathcal{E}\ e \mid \mathit{binop}\ v\ \mathcal{E} \\
 & \mid & \mathsf{case}\ (\mathcal{E};\ \mathsf{inl}\ x\ \triangleright\ e;\ \mathsf{inr}\ x\ \triangleright\ e) \mid \mathsf{injl}\ \mathcal{E} \mid \mathsf{injr}\ \mathcal{E} \\
 & \mid & \mathsf{check}\ \mathcal{E}\ e\ e\ B \mid \mathsf{check}\ v\ \mathcal{E}\ e\ B \mid \mathsf{force}\ \mathcal{E} \mid \mathsf{raise}\ \mathcal{E} \\
 & \mid & \mathsf{write}\ \mathcal{E}\ e \mid \mathsf{write}\ v\ \mathcal{E} \mid \mathsf{chan}\ \mathcal{E} \mid \mathsf{read}\ \mathcal{E} \\
 & \mid & \mathsf{catch}\ \mathcal{E}\ e \mid \mathsf{catch}\ v\ \mathcal{E}
\end{array}
$$

$$
\begin{array}{rcl}
a & := & v \mid \mathsf{raise}\ v \\
\mathcal{P} & := & \mathcal{P} + \mathcal{P} \mid \langle n, \mathcal{T} \mid e \rangle \mid \mathsf{done}\ a \\
\mathcal{T} & := & \mathsf{U} \mid \mathsf{M} \mid \mathsf{A} \\
\mathcal{K} & := & \{\iota_0, \ldots, \iota_n\} \\
\mathcal{C} & := & \square \mid \langle n, \mathcal{T} \mid \mathcal{E} \rangle \mid \mathcal{C} + \mathcal{P}
\end{array}
$$

Figure 2. Syntax of λ_{CC}.

- The λ-calculus with check, raise, strategies s, blame B, and force (but not delay), appearing as "white" and red in Figure 2. This language fragment provides a framework for user programs that wish to enforce contracts and interact with delayed contract values. Contract writers also use this calculus to construct contracts and contract combinators that naturally adhere to the standard notions of value propagation for contracts. All of the contract combinators used in Sec. 2 and defined in Figure 6 are written in this language. Furthermore, we track blame following Dimoulas et al. [11]: each blame object contains three strings, indicating the contract, positive, and negative parties.
- The λ-calculus with the above features and synchronous process facilities chan, read, and write, including the white, red, and purple portions of Figure 2. These primitives allow contract writers to interact with the contract runtime to craft additional patterns of communication between monitoring evaluators to recover new—and sometimes improved—contract enforcement patterns. These three forms are colored purple to indicate their dual nature as contract writing facilities and underlying semantic specification. The only style of contract where we have found this necessary uses these communication facilities to propagate information between subcontracts in a recursive, dependent structural contract (as in the fullness verification described in the previous section). We show how to express this style of contract in our Racket implementation in Sec. 7.
- The full calculus, including everything in Figure 2 (the white, red, purple, and blue portions), adds the additional features to complete the semantic model of contract monitoring in terms of processes. These effects along with answers a, channel names ι, the special *delay reference* form (Sec. 3.2), process pools $\mathcal{P}$, process tags $\mathcal{T}$, channel maps $\mathcal{K}$, evaluation contexts $\mathcal{E}$, and process contexts $\mathcal{C}$, provide a basis for the implementation of check, force, and delay. Furthermore, this calculus serves as a semantic specification for the runtime evaluation of contract monitors, giving an account of the underlying operations of contracts, and it is not intended for use in user programs.

Process Evaluation Rules

$$\frac{e \mapsto e'}{\mathcal{K}; \mathcal{P} + \langle n, \mathcal{T} \mid e \rangle \Rightarrow \mathcal{K}; \mathcal{P} + \langle n, \mathcal{T} \mid e' \rangle} \quad (\textsc{Step})$$

$$\frac{n \notin \mathcal{C}[\mathsf{spawn}\ \mathcal{T}\ e]}{\mathcal{K}; \mathcal{C}[\mathsf{spawn}\ \mathcal{T}\ e] \Rightarrow \mathcal{K}; \mathcal{C}[\mathsf{unit}] + \langle n, \mathcal{T} \mid e \rangle} \quad (\textsc{Spawn})$$

$$\frac{\iota \notin \mathcal{K}}{\mathcal{K}; \mathcal{C}[\mathsf{chan}\ v] \Rightarrow \mathcal{K} \cup \{\iota\}; \mathcal{C}[v\ \iota]} \quad (\textsc{Channel})$$

$$\frac{\iota \in \mathcal{K}}{\mathcal{K}; \mathcal{C}[\mathsf{read}\ \iota] + \mathcal{C}'[\mathsf{write}\ \iota\ v] \Rightarrow \mathcal{K}; \mathcal{C}[v] + \mathcal{C}'[\mathsf{unit}]} \quad (\textsc{Sync})$$

$$\frac{}{\mathcal{K}; \langle n, \mathsf{U} \mid a \rangle + \mathcal{P}_{\mathsf{A}} \Rightarrow \mathcal{K}; \mathsf{done}\ a} \quad (\textsc{Term1})$$

$$\frac{}{\mathcal{K}; \langle n, \mathsf{M} \mid v \rangle + \mathcal{P} \Rightarrow \mathcal{K}; \mathcal{P}} \quad (\textsc{Term2})$$

$$\frac{}{\mathcal{K}; \langle n, \mathsf{A} \mid v \rangle + \mathcal{P} \Rightarrow \mathcal{K}; \mathcal{P}} \quad (\textsc{Term3})$$

$$\frac{}{\mathcal{K}; \langle n, \mathsf{A} \mid \mathsf{raise}\ v \rangle + \mathcal{P} \Rightarrow \mathcal{K}; \mathsf{done}\ (\mathsf{raise}\ v)} \quad (\textsc{Term4})$$

Exception Evaluation Rules

$$\frac{\mathsf{catch}\ h\ \square \notin \mathcal{E}}{\mathcal{E}[\mathsf{raise}\ v'] \to \mathsf{raise}\ v'} \qquad \frac{}{\mathsf{catch}\ h\ v \to v} \qquad \frac{}{\mathsf{catch}\ h\ (\mathsf{raise}\ v') \to h\ v'}$$

Figure 3. Small-step semantics for the effectful forms of λ_{CC}.

3.1 Core Features

We have already introduced the core user language (colored white and red) with canonical λ-calculus forms[3] through a series of examples, and thus we now focus on the remaining pieces of λ_{CC}: exceptions, processes, check, delay, and force. Exceptions are standard, and each of check, force, and delay are implemented in terms of processes, and thus we briefly explain exceptions and focus on the concurrent aspects of λ_{CC}, postponing delay and force until Sec. 3.2. The check form is described in Sec. 4, where we provide the semantics for each of **eager**, **async**, **future**, and **semi** monitoring in terms of the underlying process calculus and delay. In the remainder of the paper we use $\to$ for a single *local* evaluation rule and $\mapsto$ for a top-level evaluation step.

Exceptions We include the standard exception mechanisms raise and catch to report and handle contract violations. The last three rules in Figure 3 illustrate their behavior. The catch h e form installs a handler h during the evaluation of e. If that evaluation terminates with a value v, the handler is discarded and the entire form evaluates to v. Otherwise the evaluation of e may raise an exception value v', in which case the intermediate contexts are discarded until either the exception reaches the top level or encounters the closest handler. In the latter case, the handler h is invoked on v'.

Processes and Process Calculus Runtime configurations in λ_{CC}, written $\mathcal{K}; \mathcal{P}$, consist of a collection of channels $\mathcal{K}$ that scope over concurrent processes $\mathcal{P}$. The set of channels $\mathcal{K}$ grows over the

[3] The semantic relations for these forms are standard and thus omitted.

$$\mathsf{catchM} \triangleq \lambda x\ f.\ \mathsf{case}\ (x;\ \mathsf{inl}\ y \triangleright f\ y;\ \mathsf{inr}\ y \triangleright \mathsf{raise}\ y) \qquad \mathsf{rep/D} \triangleq \mathsf{fix}\ \mathsf{rep/D}\ i_1\ i_2\ x.\ \mathsf{seq}\ (\mathsf{read}\ i_1)\ (\mathsf{write}\ i_2\ x)\ (\mathsf{rep/D}\ i_1\ i_2\ x)$$

$$\frac{v \neq \mathsf{dref}_{\iota_1,\iota_2}}{\mathsf{force}\ v \rightarrow v} \qquad \frac{}{\mathsf{force}\ (\mathsf{dref}_{\iota_1,\iota_2}) \rightarrow \mathsf{seq}\ (\mathsf{write}\ \iota_1\ \mathsf{unit})\ (\mathsf{catchM}\ (\mathsf{read}\ \iota_2)\ id)}$$

$$\frac{}{\mathsf{delay}\ e \rightarrow \mathsf{chan}\ (\lambda i_1\ i_2.\ \mathsf{seq}\ (\mathsf{spawn}\ \mathsf{A}\ (\mathsf{seq}\ (\mathsf{read}\ i_1)\ (\mathsf{let}\ x = \mathsf{catch}\ \mathsf{injr}\ \ (\mathsf{injl}\ \ e)\ \mathsf{in}\ \mathsf{seq}\ (\mathsf{write}\ i_2\ x)\ (\mathsf{rep/D}\ i_1\ i_2\ x))))\ \mathsf{dref}_{i_1,i_2})}$$

Figure 4. Small-step semantics for a delaying mechanism in λ_{CC}.

course of evaluation as new channels are constructed for communication with chan. The syntax distinguishes three kinds of processes: $\langle n, \mathcal{T} \mid e\rangle$ represents an individual process made up of a unique process numerical identifier n, a process tag $\mathcal{T}$, explained below, and a λ-expression e; done a is a a halt state that indicates the computation has terminated with an answer a; and $\mathcal{P} + \mathcal{P}$ is the concurrent computation of processes (or collections of processes). Process contexts $\mathcal{C}$ allow us to decompose process configurations, analogous to decomposing expressions via evaluation contexts $\mathcal{E}$.

The remaining semantics in Figure 3 formalizes the behavior of processes. We use $\Rightarrow$ to rewrite process configurations, where $\Rightarrow$ is non-deterministic in the result depending on scheduling choices. The rules implicitly equate configurations that differ only in the names of bound variables and the order or grouping of processes.

The first rule, STEP, describes internal steps taken during process evaluation: if the process body can take a top-level step $e \ \mapsto\ e'$, then the process may take an internal step under the $\Rightarrow$ relation. The second rule, SPAWN, describes process creation. We require two arguments to spawn a process: a process tag $\mathcal{T}$, signifying the nature of the process, and an expression e to evaluate in the new process. In a well-formed configuration, there is exactly one process tagged U which represents the "main" computation. Configurations may include any number of monitoring processes, tagged M, as well as asynchronous processes, tagged A. Spawning a new process allocates a unique process identification number, adds the process to the configuration, and yields unit in the original process.

The third rule, CHANNEL, creates new channels for synchronous process communication. This form takes a one-argument procedure v as input, creates a new channel ι, adds it to $\mathcal{K}$, and provides the new channel as input to the procedure v. The fourth rule, SYNC, describes synchronous communication: if the program configuration includes a process ready to write a value across a channel and another ready read from the same channel, these two processes can simultaneously perform the communication step.

The termination behavior of a process is dictated by its *tag*, U, M, or A, described by TERM1–TERM4 in Figure 3. If a process configuration consists solely of a finished user process and asynchronous processes, the configuration may halt non-deterministically with the result of the user process. If any of the asynchronous processes has detected a contract violation, the configuration may also non-deterministically terminate with that contract violation. If a monitoring process halts with an error due to a malformed expression, the entire configuration becomes stuck and is unable to take another evaluation step.

3.2 Delay and Force

Some contract monitoring strategies (e.g., semi-eager and future) require fine-grained interaction with the user evaluator in the call-by-value calculus to delay the evaluation of code fragments. One way to express such interactions would be to require user expressions to provide synchronization hooks for the monitoring thread. Taken to its logical extreme, this approach would reduce to manually implementing the call-by-value, call-by-name, and call-by-need λ-calculi and their interactions using explicit channels and processes [31]. These encodings would allow any pattern of interaction between the user evaluator and the monitor, but they would come at a heavy price: *every* λ-expression would be expressed as a process and each application would encode the desired behavior as a pattern of communication between the procedure and argument processes. The pure λ-calculus sublanguage would evaporate, yielding a process-based calculus.

This extreme approach is incompatible with our goals of explaining, integrating, and implementing existing contract systems into existing languages with our approach. Even a less-extreme approach would require intrusive changes to add synchronous behavior in the user program. Instead we add delay and force facilities to provide "enough" control over the critical portions of evaluation without requiring substantial modification to user programs. Monitoring strategies can use delay to suspend the evaluation of λ-expressions, and user programs must explicitly use force to evaluate these suspended expressions. As we illustrate in Sec. 7, wherein we introduce our Racket implementation and use it to write several contracts, these occurrences of force are not overly intrusive. Furthermore, force may be removed if the language has sufficient support to automatically force values during runtime [21].

The need for force seems to imply greater inconvenience for programmers, compared to existing contract systems which allow the result of contract monitoring to be used directly. However, this seeming convenience relies in most cases (such as contract systems for Racket or Haskell) on only allowing strategies which fit precisely with the underlying language evaluation semantics. Instead, we allow programmers to choose between strategies without requiring changes to the underlying evaluation semantics. Only when there is a mismatch between the monitoring strategy and the evaluation, as in a call-by-value language with **semi** monitoring, is manual control over evaluation with force needed. Dimoulas et al. [10] present a call-by-value calculus with future monitoring *without* delay and force, but only ever execute flat predicate contracts in parallel. Furthermore, they manually add synchronization—equivalent to force—at effect points. They suggest that this could be implicitly performed by the runtime system, just as implicit forcing could be added.

While integral to the inner workings of **semi** and **future**, delay and force are straightforward to define with the communication and process facilities in λ_{CC} (Figure 4). The evaluation of delay e proceeds as follows: we construct a pair of new channels ι_1 and ι_2, spawn a process p_r that immediately blocks reading from ι_1, and then return a delay a *delay reference* d, a tagged pair of ι_1 and ι_2.

If d is never used, process p_r remains blocked and the expression e that was delayed is never evaluated. A process p_f having access to d may force the evaluation of e as follows: we write unit to ι_1 and then block reading from ι_2 taking care to handle the possible error value.

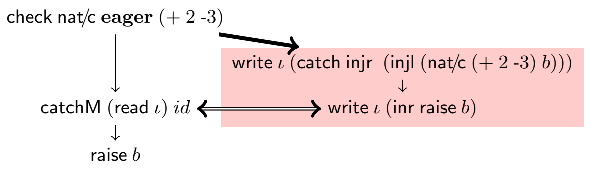
The write action of p_f on ι_1 awakens p_r. The process p_r evaluates e and writes the resulting value on ι_2. Process p_r then replicates itself via rep/D, ensuring further forces of the same delay reference produce the same value without re-evaluation. Finally, It is possible that e will raise an exception, and so the value written to ι_2 is tagged to indicate when this has occurred.

Finally, applying force to any value which is not a delay-reference returns the value unchanged. This operation is a matter of programmer convenience: a function $\lambda x.\ x\ +\ 1$ can *only* be monitored by contracts whose preconditions do *not* build delay references, e.g., preconditions that use **eager**. Using strategies like **semi** in the precondition would attempt to invoke addition on a delay reference. In contrast, the function $\lambda x.$ force $x\ +\ 1$ can be used with *any* strategy for the precondition contract whether it builds a delay reference or not.

4. Contract Monitoring with Processes

We now construct the monitoring behaviors characterized in Sec. 2, presenting the underlying semantics for check to describe the many-strategy monitoring system for λ_{CC}. This semantic model, presented in Figure 5, explicitly encodes each monitoring strategy as a pattern of process communication, illustrating their subtle and precise differences.

Eager Monitors. The eager approach to contract monitoring closely matches the CPCF calculus described by Dimoulas et al. [11], where every contract is checked at assertion time. These checks are modeled as immediate interactions between processes, as presented in Figure 1. Consider a further diagram where the monitor produces an error:


check nat/c **eager** (+ 2 -3)
write ι (catch injr (injl (nat/c (+ 2 -3) b)))
catchM (read ι) id
write ι (inr raise b)
raise b

This behavior is encoded via the process effect operators in λ_{CC} in the eager definition of check (see Figure 5). To demonstrate this interaction, consider the running example of enforcing the contract nat/c[4]:

$$\begin{aligned}
&\langle 0, \mathsf{U} \mid \mathsf{check\ nat/c}\ \mathbf{eager}\ v\ b\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota)\ id\rangle\\
&+\langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ (\mathsf{nat/c}\ v\ b)))\rangle
\end{aligned}$$

When $v := 5$, the computation proceeds as:

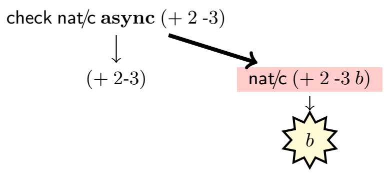
$$\begin{aligned}
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota)\ id\rangle + \langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{inl}\ 5)\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{inl}\ 5)\ id\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid 5\rangle
\end{aligned}$$

When $v :=$ -1, the computation proceeds as:

$$\begin{aligned}
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota)\ id\rangle + \langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{inr}\ b)\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{inr}\ b)\ id\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{raise}\ b\rangle
\end{aligned}$$

Here check immediately constructs the communication channel ι and the monitoring process. The initiating process performs a blocking read across ι while the monitoring process checks the contract with the monitored value. The result of the contract is handled by injecting values to the left and exceptions to the right before writing them to the initiating process. Finally, the initiating evaluator processes the result with catchM, continuing with the value or re-throwing the exception.

Asynchronous Monitors. The asynchronous approach to contract monitoring consists of stand-alone monitors that perform "best-effort" checking: if the monitor is not finished when the user program is complete, then the entire program produces the user-program answer and discards the incomplete monitor, as presented in Figure 1. Consider a further diagram where the monitor produces an error:



With explicit processes, this monitoring strategy is straightforward, as implemented in Figure 5. When invoked, check spawns an asynchronous process to perform the check; there is no additional synchronization except for possible termination of the configuration in the event of an error. When the monitor is complete, the asynchronous process terminates with either a value v or an exception raise v. This lack of synchronization can produce non-deterministic results for asynchronous monitoring:

$$\begin{aligned}
&\emptyset;\ \ \langle 0, \mathsf{U} \mid id\ (\mathsf{check\ nat/c}\ \mathbf{async}\ \text{-}1\ b)\rangle\\
\Rightarrow^*\ &\{\iota\};\ \langle 0, \mathsf{U} \mid id\ \text{-}1\rangle + \langle 1, \mathsf{A} \mid (\mathsf{nat/c}\ \text{-}1\ b)\rangle\\
\Rightarrow^*\ &\{\iota\};\ \langle 0, \mathsf{U} \mid \text{-}1\rangle\quad + \langle 1, \mathsf{A} \mid (\mathsf{nat/c}\ \text{-}1\ b)\rangle\\
\Rightarrow^*\ &\{\iota\};\ \mathsf{done}\ \text{-}1
\end{aligned}$$

$$\begin{aligned}
\Rightarrow^*\ &\{\iota\};\ \langle 0, \mathsf{U} \mid id\ \text{-}1\rangle + \langle 1, \mathsf{A} \mid (\mathsf{nat/c}\ \text{-}1\ b)\rangle\\
\Rightarrow^*\ &\{\iota\};\ \langle 0, \mathsf{U} \mid \cdots\rangle\quad + \langle 1, \mathsf{A} \mid \mathsf{raise}\ b\rangle\\
\Rightarrow^*\ &\{\iota\};\ \langle 1, \mathsf{A} \mid \mathsf{raise}\ b\rangle\\
\Rightarrow^*\ &\{\iota\};\ \mathsf{done}\ (\mathsf{raise}\ b)
\end{aligned}$$

Even with this potential non-determinism, asynchronous contracts can provide a reasonable alternative to completely verifying contracts over large structures.

Future Monitors. Originally posed by Dimoulas et al. [10], future contracts check contracts concurrently during program execution, synchronizing only when necessary for the program to continue. Dimoulas et al. [10] perform this synchronization at all program side effects; we instead synchronize when the value is demanded. The implementation of future monitoring follows eager monitoring, but delays the receipt of the resulting value until it is needed (Figure 5). The consumer of the contracted value can force the future at any time, at which point the blocking nature of read will halt the current process until contract checking is complete. Returning to our example, evaluation proceeds as follows:

$$\begin{aligned}
&\langle 0, \mathsf{U} \mid \mathsf{force}\ (\mathsf{check\ nat/c}\ \mathbf{future}\ 5\ b)\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{force}\ (\mathsf{delay}\ (\mathsf{catchM}\ (\mathsf{read}\ \iota)\ id))\rangle +\\
&\langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ (\mathsf{nat/c}\ 5\ b)))\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{force\ dref}_{\iota_2,\iota_3}\rangle +\\
&\langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ 5))\rangle + \langle 2, \mathsf{A} \mid \cdots\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota_3)\ id\rangle +\\
&\langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{inl}\ 5)\rangle + \langle 2, \mathsf{A} \mid \cdots\ \mathsf{read}\ \iota\ \cdots\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota_3)\ id\rangle + \langle 2, \mathsf{A} \mid \cdots\ \mathsf{write}\ \iota_3\ (\mathsf{inl}\ 5)\ \cdots\rangle\\
\Rightarrow^*\ &\langle 0, \mathsf{U} \mid 5\rangle + \langle 2, \mathsf{A} \mid \cdots\rangle
\end{aligned}$$

The derivation corresponds to the enforcement process presented in *Future Monitoring* in Figure 1: the check form constructs two new processes, colored red and blue, that collectively provide the monitoring future. The third, fourth, and fifth steps correspond to the synchronizations that take place in Figure 1, first between the user and delay processes, then the monitor and delay processes, and finally the user and delay processes again. The derivation ends with the monitoring process removed from the configuration and the user process continuing with the monitor result.

Semi-Eager Monitoring. Semi-eager monitoring [5–8, 22] delays contract checking until the checked value is demanded, an approach that appears naturally when Findler-Felleisen-style are recreated in lazy languages such as Haskell [5, 7, 22]. The implementation of semi-eager monitoring mirrors future monitoring, except that the contract expression is delayed instead of the user con-

[4] We elide $\mathcal{K}$ in our examples for simplicity of presentation.

$$\begin{array}{llllll}
\mathsf{check}\ c\ \mathbf{eager} & e\ b \to \mathsf{chan}\ (\lambda i. & \mathsf{seq}\ \ (\mathsf{spawn\ M}\ (\mathsf{write}\ i\ (\mathsf{catch\ injr}\ \ (\mathsf{injl} & (c\ e\ b))))) & & (\mathsf{catchM}\ (\mathsf{read}\ i)\ id)) \\
\mathsf{check}\ c\ \mathbf{future} & e\ b \to \mathsf{chan}\ (\lambda i. & \mathsf{seq}\ \ (\mathsf{spawn\ M}\ (\mathsf{write}\ i\ (\mathsf{catch\ injr}\ \ (\mathsf{injl} & (c\ e\ b))))) & (\mathsf{delay} & (\mathsf{catchM}\ (\mathsf{read}\ i)\ id))) \\
\mathsf{check}\ c\ \mathbf{semi} & e\ b \to \mathsf{chan}\ (\lambda i. & \mathsf{seq}\ \ (\mathsf{spawn\ M}\ (\mathsf{write}\ i\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ (\mathsf{delay} & (c\ e\ b)))))) & & (\mathsf{catchM}\ (\mathsf{read}\ i)\ id)) \\
\mathsf{check}\ c\ \mathbf{async} & e\ b \to & \mathsf{seq}\ \ (\mathsf{spawn\ A} & (c\ e\ b)) & & e
\end{array}$$

Figure 5. Small-step semantics for the contract monitoring mechanism in λ_{CC}.

sumption form (Figure 5). Returning again to our example, evaluation proceeds as follows:

$$\begin{array}{ll}
 & \langle 0, \mathsf{U} \mid \mathsf{force}\ (\mathsf{check}\ \mathsf{nat/c}\ \mathbf{semi}\ 5\ b)\rangle \\
\Rightarrow^* & \langle 0, \mathsf{U} \mid \mathsf{force}\ (\mathsf{catchM}\ (\mathsf{read}\ \iota)\ id)\rangle + \\
 & \langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ (\mathsf{delay}\ (\mathsf{nat/c}\ 5\ b)))))\rangle \\
\Rightarrow^* & \langle 0, \mathsf{U} \mid \mathsf{force}\ (\mathsf{catchM}\ (\mathsf{read}\ \iota)\ id)\rangle + \\
 & \langle 1, \mathsf{M} \mid \mathsf{write}\ \iota\ \mathsf{dref}_{\iota_2,\iota_3}\rangle + \langle 2, \mathsf{A} \mid \cdots\rangle \\
\Rightarrow^* & \langle 0, \mathsf{U} \mid \mathsf{force}\ \mathsf{dref}_{\iota_2,\iota_3}\rangle + \langle 2, \mathsf{A} \mid \mathsf{seq}\ (\mathsf{read}\ \iota_1)\ \ \cdots\rangle \\
\Rightarrow^* & \langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota_2)\ id\rangle + \\
 & \langle 2, \mathsf{A} \mid \mathsf{let}\ x = \mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ (\mathsf{nat/c}\ 5\ b))\ \mathsf{in}\ \ \cdots\rangle \\
\Rightarrow^* & \langle 0, \mathsf{U} \mid \mathsf{catchM}\ (\mathsf{read}\ \iota_2)\ id\rangle + \\
 & \langle 2, \mathsf{A} \mid \mathsf{seq}\ (\mathsf{write}\ \iota_2\ (\mathsf{inl}\ 5))\ (\mathsf{rep/D}\ \iota_1\ \iota_2\ (\mathsf{inl}\ 5))\rangle \\
\Rightarrow^* & \langle 0, \mathsf{U} \mid 5\rangle + \langle 1, \mathsf{A} \mid \mathsf{rep/D}\ \iota_1\ \iota_2\ (\mathsf{inl}\ 5)\rangle
\end{array}$$

When the resultant delay reference is forced, the initiating process signals the delay reference to perform the monitoring operation. The delay reference process returns the result of the contract enforcement via the correct injection, facilitating communication between the user evaluator and the delayed monitor.

5. Contract Combinators

As we have seen, a contract is a procedure that accepts a value and a blame tuple and produces a checked version of the value, raising the appropriate exception on failure. In a higher-order language, we can abstract over these contracts, producing a library of *contract combinators* written with check in λ_{CC}. Remarkably, λ_{CC} contracts look syntactically similar to existing combinator implementations, since these abstractions do not need to interact with the low-level communication semantics of check. The simplest contract combinator, pred/c, checks a predicate on the monitored value. More sophisticated contract combinators reuse the check form to enforce subcontracts on subcomponents of their input (see Figure 6).

Predicate Contracts. Predicate contracts, constructed with pred/c, follow directly from our previous description. The combinator pred/c takes a predicate c as input and produces a contract that expects a value x and blame tuple b as input. When this procedure is invoked, we evaluate the expression

$$\mathsf{if}\ c\ x\ \mathsf{then}\ x\ \mathsf{else\ raise}\ b$$

Here, x is the monitored value, c is the monitoring predicate, b is the responsible party. This expression is evaluated when the contract is enforced as $(c\ e\ b)$ in Figure 5, returning the value on success or raising the blame tuple as an exception on failure.

Pair Contracts. As discussed in Sec. 2, pairs use two subcontracts with their associated strategies for each of the left and right subcomponents. The pair combinator constructs the contract in terms of its subcontracts, monitoring each on the appropriate subcomponent of the pair with check as:

$$(\mathsf{check}\ c_1\ s_1\ (\mathsf{fst}\ p)\ b, \mathsf{check}\ c_2\ s_2\ (\mathsf{snd}\ p)\ b)$$

Here p is the monitored pair, c_1 and s_1 are the contract and strategy for the first element, and c_2 and s_2 are the contract and strategy for the second element. The contract decomposes the pair into its sub-pieces *when* the monitoring strategy specifies to do so, and construct a new pair via check, which is returned to the initiating (or demanding) location.

Function Contracts. Function contracts, constructed with func/c in Figure 6, take two contracts and strategies which indicate the pre- and post-condition and their strategies. The result is a procedure

$$\begin{array}{lll}
\mathsf{pred/c} & \triangleq & \lambda c.\ \lambda x\ b.\ \mathsf{if}\ c\ x\ \mathsf{then}\ x\ \mathsf{else\ raise}\ b \\
\mathsf{pair/c} & \triangleq & \lambda c_1\ s_1\ c_2\ s_2. \\
 & & \quad \lambda p\ b.\ (\mathsf{check}\ c_1\ s_1\ (\mathsf{fst}\ p)\ b, \mathsf{check}\ c_2\ s_2\ (\mathsf{snd}\ p)\ b) \\
\mathsf{func/c} & \triangleq & \lambda c_1\ s_1\ c_2\ s_2. \\
 & & \quad \lambda f\ b.\ \lambda x.\ \mathsf{check}\ c_2\ s_2\ (f\ (\mathsf{check}\ c_1\ s_1\ x\ (\mathsf{inv}\ b)))\ b \\
\mathsf{func/dc} & \triangleq & \lambda c_1\ s_1\ g\ s_2. \\
 & & \quad \lambda f\ b. \\
 & & \qquad \lambda x.\ \mathsf{check}\ (g\ (\mathsf{check}\ c_1\ s_1\ x\ (\mathsf{ind}\ b)))\ s_2 \\
 & & \qquad\qquad\qquad (f\ (\mathsf{check}\ c_1\ s_1\ x\ (\mathsf{inv}\ b)))\ b
\end{array}$$

Figure 6. Contract combinators for λ_{CC}.

that expects some function f and the appropriate blame labels. When invoked, the combinator constructs the function:

$$\lambda x.\ \mathsf{check}\ c_2\ s_2\ (f\ (\mathsf{check}\ c_1\ s_1\ x\ (\mathsf{inv}\ b)))\ b$$

Here f is the monitored function, x is that function's input, c_1 and s_1 are the contract and strategy for the pre-condition, and c_2 and s_2 are the contract and strategy for the post-condition. We also use the blame operator inv, which inverts the pre-condition blame labels to ensure correct blame [11]. Dependent functions follow directly[5], using ind to construct the *indy* label structure.

6. Metatheory

We briefly sketch out two metatheoretical aspects of our system: the type system and an embedding of the original contract system presented by Findler and Felleisen [16].

Typing the Contract Calculus. We sketch the type system for λ_{CC}, using the type grammar in Figure 7. The typing rules are standard: delay e and force e share the type of e. The raise form may be given any type.

We define `con` τ as an abbreviation for contract types with the shape discussed previously: The contract shorthand enforces the contract shape discussed previously: a contract takes its input and blame labels, returning the original type τ. The strategy types σ are as expected: **eager** is typed as `eager` and so forth. The check form takes a contract of type `con` τ, a strategy of type σ, a monitored expression of type τ, and a blame set of type `blame`, returning a value of type τ[6].

Communication requires types for channels, which are included in an additional type environment Δ over $\mathcal{K}; \mathcal{P}$. A process configurations is well-typed up to deadlock if each process term is made up of well-typed subterms, and each process is well-typed if the underlying λ-calculus expression is also well-typed.

Theorem 6.1. *For any well-typed configuration $\mathcal{K}; \mathcal{P}$, either: $\mathcal{K}; \mathcal{P}$ is in a well-typed halt state; there exists some well-typed configuration $\mathcal{K}'; \mathcal{P}'$ such that $\mathcal{K}; \mathcal{P} \Rightarrow \mathcal{K}'; \mathcal{P}'$; or $\mathcal{K}; \mathcal{P}$ is in a stuck state $\mathcal{S}$, which include configurations where a monitoring process has raised an exception or is otherwise blocked, or all processes are*

[5] The dependent function combinator uses the same strategy for both enforcements of c_1, but it is possible to vary this strategy between them.

[6] Explicitly typing delay references requires a type-level metafunction that uses the strategy's type to calculate the final type of a contract.

$$
\begin{array}{rcl}
\tau & := & \texttt{int} \mid \texttt{bool} \mid \texttt{str} \mid \texttt{error} \mid () \\
 & \mid & \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \texttt{chan}\ \tau \mid \texttt{con}\ \tau \mid \sigma \\
\sigma & := & \texttt{eager} \mid \texttt{seager} \mid \texttt{future} \mid \texttt{async} \\
\texttt{blame} & \triangleq & \texttt{str} \times \texttt{str} \times \texttt{str} \\
\texttt{con}\ \tau & \triangleq & \tau \rightarrow \texttt{blame} \rightarrow \tau \\
\texttt{check} & \triangleq & \texttt{con}\ \tau \rightarrow \sigma \rightarrow \tau \rightarrow \texttt{blame} \rightarrow \tau
\end{array}
$$

Figure 7. Typing grammar for λ_{CC}.

either blocked on communication that may not be further reduced by SYNC *or have delay references is usage positions.*

Correctness of Encoding. We demonstrate that our semantics faithfully recreates the original semantics provided by Findler and Felleisen's *Contracts for Higher-Order Functions* [16]. We choose this semantic model for two reasons: first, this semantics has been widely accepted and rigorously verified, and second, the subtleties of less-eager monitoring approaches vary whereas the semantics of eager monitoring is well-accepted. Furthermore, this embedding is one-way: while it may be possible to recover the exact communication structure of every collection of processes [26], it is not worthwhile or insightful. Finally, Findler and Felleisen [16] use *lax* monitoring for dependent monitors (as discussed by Dimoulas et al. [11]), and thus we omit them.

Theorem 6.2. *If c is an expression in FF, the original semantics presented by Findler and Felleisen [16], then there is some context $P \in FF$ such that $c = P[c']$ and a translation function $T(P, c')$: $FF \rightarrow \lambda_{\mathsf{CC}}$ such that either:*

1. *if $P[c'] \mapsto^* v$, then $T(P, c') \Rightarrow^* \mathsf{done}\ v$;*
2. *$\forall n$. if $P[c'] \mapsto^* \mathsf{exn}\ p$, then $T(P, c') \Rightarrow^* \mathsf{done}\ (\mathsf{raise}\ b)$ where $b = (p, n)$.*

Proof. We assume that Findler and Felleisen's I operator has already been run on the input c, which performs *obligation insertion*, or, more simply, inserts the contracts provided by the outer **val rec** binding form into the main expression. We proceed by "recoloring" the if expressions in c, indicating whether they were constructed by a contract monitor or not. We also explicitly mark contract forms containing values as value forms in the language to ease translation. Monitoring flat contracts will now produce something of the form $\mathsf{if}_c\ e_1\ e_2\ e_3$. The proof proceeds by induction on the length of $\mapsto^*$, where we we define a handful of translation functions such that $T(P, c') = F(g(P), f(c'))$ where

$$
\begin{array}{rcl}
f(c') & = & ((\mathcal{C}, \mathcal{K}), e) \\
g(P) & = & ((\mathcal{C}, \mathcal{K}), \mathcal{E})
\end{array}
$$

such that

$$F(((\mathcal{C}', \mathcal{K}'), \mathcal{E}), ((\mathcal{C}, \mathcal{K}), e)) = \mathcal{K}' \cup \mathcal{K}; \mathcal{C}'[\mathcal{C}[\mathcal{E}[e]]]$$

We only present one case in full:
<u>Case:</u> $P[V_1^{contract(V_2),p,n}] \mapsto P[\mathsf{if}_c\ (V_2\ V_1)\ V_1\ (blame\ p)]$:
We perform the following translations:

$$
\begin{array}{rcl}
f(V_1) & = & ((\mathcal{C}_{v_1}, \mathcal{K}_{v_1}), v_1) \\
f(V_2) & = & ((\mathcal{C}_{v_2}, \mathcal{K}_{v_2}), v_2) \\
c & = & \lambda x\ b.\ \mathsf{if}\ v_2\ x\ \mathsf{then}\ x\ \mathsf{else\ raise}\ b \\
f(contract(V_2)) & = & ((\mathcal{C}_{v_2}, \mathcal{K}_{v_2}), c) \\
f(p) & = & ((\mathcal{C}_p, \mathcal{K}_p), p_{cc}) \\
f(n) & = & ((\mathcal{C}_n, \mathcal{K}_n), n_{cc}) \\
f_{blame}(p) & = & b
\end{array}
$$

Then $f(V_1^{contract(V_2),p,n}) = ((\mathcal{P}_1, \mathcal{K}_1), e_1)$ where:

$$
\begin{array}{rcl}
e_1 & = & \mathsf{check}\ c\ \mathbf{eager}\ \hat{v_1}\ b \\
\mathcal{P}_1 & = & \mathcal{C}_{v1}[\mathcal{C}_{v2}[\Box]] \\
\mathcal{K}_1 & = & \mathcal{K}_{v_1} \cup \mathcal{K}_{v_2}
\end{array}
$$

Then let ι_{v_1} be a unique channel and $f(\mathsf{if}_c\ (V_2\ V_1)\ V_1\ (blame\ p)) = ((\mathcal{P}_2, \mathcal{K}_2), e_2)$ where:

$$
\begin{array}{l}
e_2 = \mathsf{catchM}\ (\mathsf{read}\ \iota_{v_1})\ id \\
\mathcal{P}_2 = \mathcal{C}_{v_1}[\mathcal{C}_{v2}[\Box + \langle n, \mathsf{M} \mid \mathsf{write}\ \iota_{v_1}\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ (c\ v_1\ b))))\rangle]] \\
\mathcal{K}_2 = \mathcal{K}_{v_1} \cup \mathcal{K}_{v_2} \cup \{\iota_{v_1}\}
\end{array}
$$

Then we compute $g(P) = (\mathcal{C}_P, \mathcal{K}_P)$.
Finally, let $\mathcal{C} = \mathcal{C}_P[\mathcal{C}_{v_1}[\mathcal{C}_{v_2}[\Box]]$ and $\mathcal{K} = \mathcal{K}_{v_1} \cup \mathcal{K}_{v_2} \cup \mathcal{K}_P$ Therefore:

$$
\begin{array}{rl}
 & F((\mathcal{C}_P, \mathcal{K}_P), ((\mathcal{P}_1, \mathcal{K}_1), e_1)) \\
= & \mathcal{K}; \mathcal{C}[\mathsf{check}\ c\ \mathbf{eager}\ \hat{v_1}\ b] \\
\Rightarrow^* & \mathcal{K} \cup \{\iota_{v_1}\}; \mathcal{C}[[\mathsf{catchM}\ (\mathsf{read}\ \iota_{v_1})\ id] + \\
 & \qquad \langle n, \mathsf{M} \mid \mathsf{write}\ \iota_{v_1}\ (\mathsf{catch\ injr}\ \ (\mathsf{injl}\ \ (c_2\ v_1\ b))))\rangle] \\
 & F((\mathcal{C}_P, \mathcal{K}_P), ((\mathcal{P}_2, \mathcal{K}_2), e_2))
\end{array}
$$

□

7. Implementation and Advanced Monitors

In a language (e.g., Racket [19]) with the right infrastructure (syntactic extension, processes, exceptions, and communication primitives), λ_{CC} can be realized as a library in less than 100 lines of code[7]. We elide the exact implementation details; they follow directly from the previous semantics. Each strategy is represented by an instance of a Racket structure, the contract combinators and blame facilities are provided as procedures, and check is provided as a syntactic transformer:

```
(define-syntax checkCon
  (syntax-rules ()
    [(_ c s e b)
     (cond [(eager-strat? s) ...]
           [(semi-strat? s) ...]
           [(future-strat? s) ...]
           [(async-strat? s)
            (begin (thread (λ () (c e b))) e)])]))
```

We now use this implementation, called ccon, to demonstrate the extensibility of our system, from simple contracts to additional contract combinators for structural contracts over trees [18]. We start with small examples:

```
> (checkCon nat/c eager 5 b)
5
> (checkCon nat/c eager -1 b)
Exception: ...
> (checkCon nat/c semi 5 b)
#<promise ...>
> (force (checkCon nat/c future 5 b))
5
```

These calls are as expected, corresponding to our earlier examples in Sec. 2 and Sec. 4. The result of the third and fourth contract demonstrate the underlying usage of Racket's `delay` for semi-eager and future strategies. (Here b is a blame structure with Server, Contract, and Client labels.)

7.1 Tree Contracts in ccon

To demonstrate the expressive power of ccon (and thus λ_{CC}), we next construct a tree data structure and define contract combinators for the structure. We define a tree structure as node in Figure 8, where a tree is either a null value, (), or a node containing a value and left and right subtrees.

We also define the tree contract combinator treerec/c, a recursive combinator which reuses itself on its left and right subtrees. This procedure determines if the current node is a leaf or internal node and then constructs the appropriate monitor. We can use this contract to enforce tree-wide invariants, such as ensuring that every value is a natural number:

```
(define nat-tree-E (tree/c nat/c eager eager nat/c eager))
> (checkCon nat-tree-E eager (make-node ...) b)
```

[7] https://github.com/cgswords/ccon

```
(struct node (left value right))

(define (treerec/c c1 s1 s2 c3 s3)
  (letrec
    ((treec
      (λ (t b)
        (match t
          [(node l v r) (let ([nv (checkCon c3 s3 v b)]
                              [nl (checkCon treec s2 l b)]
                              [nr (checkCon treec s2 r b)])
                          (node nl nv nr))]
          [v (checkCon c1 s2 v b)]))))
    treec))

(define ((tree/dc c1 s1 c2 s2 c3 s3 c4 s4) t b)
  (match t
    [(node l v r) (let* ([nv (checkCon c3 s3 v b)]
                         [nl (checkCon (c2 nv) s2 l b)]
                         [nr (checkCon (c4 nv) s4 r b)])
                    (node nl nv nr))]
    [v (checkCon c1 s2 v b)]))
```

Figure 8. Tree contracts in Racket with ccon.

We can also use this contract as a pre-condition to a procedure such as get-root-value, which returns the root value of any tree t:

```
> (get-root-value t)
5
> (checkCon (func/c nat-tree-E eager nat/c eager) eager t b)
5
```

The first invocation inspects the root node of the tree, returning the value. Conversely, the second invocation traverses the entire tree structure, enforcing the precondition contract before extracting the value from the root node. To recover the original complexity, we can use the solution presented by Findler et al. [18], rewriting nat-tree-E with semi-eager strategies. However, unlike Findler et al., we have a fine-grained decision to make: shall we eagerly or semi-eagerly enforce the value contract? Eager value checking, nat-tree-S1 below, ensures that any node we inspect will have a natural number in the value position. Conversely, full semi-eager checking, nat-tree-S2 below, will only verify that values are natural numbers when the program demands them.

```
(define nat-tree-S1 (tree/c nat/c eager semi nat/c eager))
(define nat-tree-S2 (tree/c nat/c eager semi nat/c semi))

> (get-root-val (checkCon nat-tree-S1 eager t b))
5
> (get-root-val (checkCon nat-tree-S2 eager t b))
#<promise ...>
```

Neither of these solutions is “more correct”: both encodings recover the original complexity of get-root-val, and the divergence demonstrates the precise power of explicit monitoring strategies. Furthermore, this change is syntactically insignificant: a programmer might try both out to select the most appropriate approach.

7.2 Advanced Contracts in ccon

There are a family of contracts that cannot benefit from the previous optimization, which include contracts that require *upward value propagation* through monitored structures [18], or more generally, collections of contracts that collaborate to establish global invariants. For example, one may try to ensure that a binary tree is full, i.e., that there are 2^n nodes in the tree if the height of the tree is n. Implementing this predicate is straightforward, traversing the entire tree, but this once again requires visiting every node. Alternatively, we might construct a dependent tree contract combinator wherein the left and right subtrees are passed into the value predicate, but Findler et al. [18] observe that these subtrees must be

```
(define (full/a i)
  (let ([il (make-channel)]
        [ir (make-channel)])
    (treedc
      (pred/c (λ (n) (begin (channel-put i 0) #t))) eager
      (λ (_) (full/a il)) semi
      (pred/c
        (λ (n)
          (let ([hr (sync (choice-evt ir il))]
                [hl (sync (choice-evt ir il))])
            (if (= hl hr)
                (begin (channel-put i (add1 hl)) #t)
                #f))))
      async
      (λ (_) (full/a ir)) semi)))
```

Figure 9. The implementation of full as a callback-based contract.

evaluated to verify the contract, and thus recreate the same asymptotic violations. They go on to propose an alternative solution using attributes [19] to track height information on the tree as it is explored, relying on the user evaluator to produce an invasive enforcement mechanism with amortized bounds. This approach is not currently part of the Racket contract system.

We can develop a similar solution in ccon using callbacks: we postpone checking any contract until its subcontracts have been checked by using channels within asynchronous contracts. While complex in concept, the only additional facility we require is the ability to create, propagate, and synchronize with new channels of communication, which both λ_{CC} and Racket provide as low-level operations. Thus the adventurous contract writer might implement this callback-oriented approach to fullness as in Figure 9. Furthermore, this solution lives entirely in the contract system: the user is never exposed to it beyond using force.

Each invocation of the contract is parameterized by a communication channel i, which indicates where to write the current node's height. At each leaf, the predicate writes 0 to i and succeeds. At internal nodes, we create two additional channels, il and ir, to pass to the left and right subtrees respectively. Next we assert (full/a il) and (full/a il) on the appropriate subtrees, utilizing the delaying nature of lambda abstraction to prevent divergence. These contracts are semi-eagerly monitored, and thus will not be enforced until the subtrees are demanded by the program[8]. Finally, we create an asynchronous monitor for the node's value which performs blocking reads from il and ir. The results of these reads are the heights of the left and right subtrees, so we verify they are equal, and either write the new height across i, triggering the parent node's fullness test, or signal a contract violation. This communication chain allows each contract to propagate values upward from the leaves as the tree is explored, as indicated with upward-pointing arrows in Figure 10.

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 enforcement in a custom-crafted traversal of the monitored structure.

7.3 Results

We briefly evaluate the performance of full/a against the eager fullness contract presented by Findler et al. [18] using a set of microbenchmarks. We use ccon to implement full/a as in Figure 9, and the eager fullness contract is constructed in Racket's native system as follows:

```
(define full/c (flat-named-contract 'full/c full?))
```

[8] If we used **eager** in place of **semi**, this contract would revert to an eager, $O(2^n)$ check at assertion time.

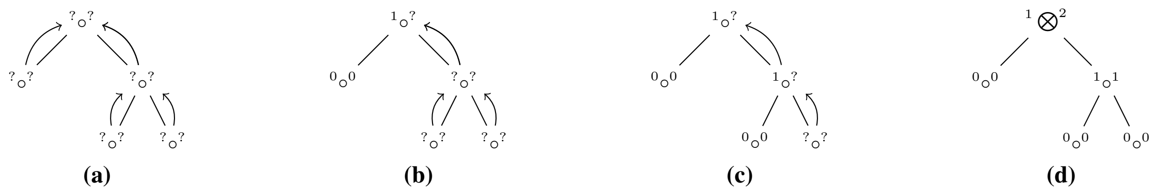


Figure 10. Evolution of contract checking during tree traversal for a full binary tree using asynchronous callbacks.

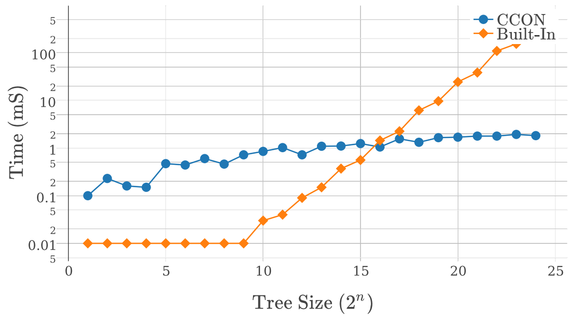


Figure 11. Timing results for full tree contract verification in Racket's built-in contract system versus our semi-eager approach. Experiments were performed on a MacBook Air with a 1.3 GHz Intel Core i5 and 4GB of RAM.

Next we set up the following experiment: for trees of size 1 to 2^{24}, we first enforce one of the contracts on the tree and then perform a single element lookup. This lookup occurs in $O(\log n)$ time, and so the bulk of the computational work in the Racket-primitive contract will consist of enforcing the fullness contract, while the bulk of the work in enforcing full/a occurs in the process and channel construction overhead incurred during execution.

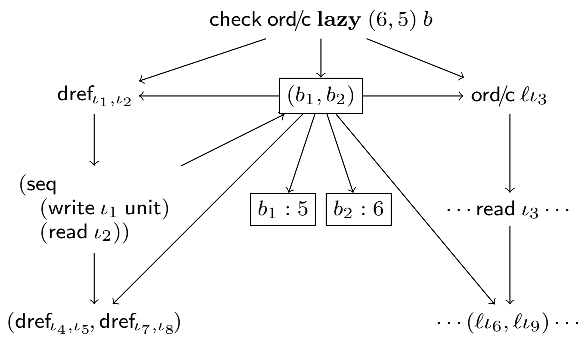
We repeat this test 100 times for each data-point and graph the arithmetic means in in Figure 11. The x-axis represents the size of the tree as a function of 2^n and the y-axis is a logarithmic scale in seconds. The orange line, *Built-In*, indicates that the eager contract run with Racket's built-in monitoring facilities provides excellent performance for small tree structures but slows down exponentially as the tree increases exponentially in size. Conversely, the blue line, *CCON*, indicates that the full/a implementation in ccon slowly only slightly as longer traversals require increasingly large numbers of processes and channels.

Our results support our initial discussion of performance recovery. Furthermore, the Racket contract system has been heavily optimized for performance [33] and the competitive results of ccon indicate that the process communication model of λ_{CC} is a viable approach to contract monitoring.

8. Sketches of Additional Monitoring Strategies

We present sketches of implementations for four strategies that appear in the literature: lazy monitors, temporal monitors, future monitors in the style of Dimoulas et al., and statistical monitors.

Lazy Contracts. Degen et al. [7, 8] introduce the idea of lazy monitors, which entirely avoid any aspect of over-evaluation. For example, checking a predicate on a pair will not force the pair; the monitoring evaluator will wait for the user evaluator to force the pair. If the pair is never used, in full, in the user program, then the monitor will never check it. Modeling lazy monitoring as commu-


check ord/c **lazy** $(6,5)$ b
dref$_{\iota_1,\iota_2}$
(b_1, b_2)
ord/c $\ell\iota_3$
(seq
(write ι_1 unit)
(read ι_2))
$b_1 : 5$
$b_2 : 6$
$\cdots$ read ι_3 $\cdots$
(dref$_{\iota_4,\iota_5}$, dref$_{\iota_7,\iota_8}$)
$\cdots (\ell\iota_6, \ell\iota_9) \cdots$
Figure 12. Lazy contract communication [8].

nication across processes 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 may force any expression under a lazy monitor and the lazy monitor will postpone contract enforcement until this forcing occurs. We accomplish this by recursively parsing the input expression e, yielding yield two new expression e_d and e_ℓ. The first, e_d, is an expression constructed from delay references wrapped around each component and sub-component in the original expression. Similarly, e_ℓ is constructed identically using *lazy monitor references* of the form $\ell\iota$. Each delay reference works in the usual way, except that it will also notify the associated lazy monitor reference when forced. When evaluated, lazy monitor references perform blocking reads, proceeding only when the associated delay reference is forced and provides the appropriate value. The structure of this communication is presented in Figure 12. This approach closely mirrors the implementation provided by Degen et al. [8], wherein individual call-by-need cells register callbacks for contract monitors.

Temporal Contracts. Disney et al. [13] introduce the concept of temporal monitors, which capture module interactions as a trace of events such as function calls and returns. They formalize this approach by treating each module in the program as a process and storing the process trace of sends and receives between these modules as the program proceeds. Finally, the monitoring system verifies that this trace conforms to certain prefix-closed predicates, constraining the behavior of each module and producing an error if this trace violates these predicates. Our approach differs by using individual processes for each monitor. We explicitly enforce predicates on values and control how processes communicate to replicate monitoring strategies, omitting the need to preserve process traces. Such traces, however, may be used to ensure that monitor strategies correctly adhere to their semantic descriptions.

This system may be constructed in λ_{CC} by recreating the original approach, using a mediator process to capture process traces. Any communication events will move through this mediator, and the process will log the events. Temporal contract assertions will register predicates with this mediator process, which will use these predicates to ensure the traces adhere to the correct shape. Such a mediator must also provide the guarantees outlined by Disney et al. [13], including information capture and correct error propagation.

Future Contracts à la Dimoulas et al. Dimoulas et al. [10] present an alternate model of future contracts that use a master user process and a slave monitoring process, delegating predicate enforcements to the slave process and synchronizing with it when the master performs effects such as reference manipulation and I/O.

This system may be constructed in λ_{CC} using a secondary global process that acts as the slave process. Any monitors with the appropriate strategy are provided to this slave process, which adds the monitor to a list of active checks. To maintain synchronous communication, it may be convenient to establish the slave as two processes: one to receive new monitoring requests and synchronize with the user process during synchronization events, and another "worker" process that polls the synchronization process for a monitor, performs it, and provides the synchronization process with the result before looping. Then, as Dimoulas et al. [10] observe, each effectful operation is wrapped in a proxy procedure that synchronizes with the slave process before executing. This synchronization mechanism would elide the need to implicitly invoke force.

Statistical Software Contracts. Dimoulas et al. [12] describe the notion of randomly checked function contracts. In lieu of iterating over the entire input space for a function, it is possible to use a spot checker [14] to obtain partial assurance of a contract by generating and testing a series of inputs. For flat and structural contracts, this approach may randomly verify its contracts: that is, the contract may (or may not) be enforced as evaluation proceeds.

Given a random-generation mechanism and a spot-checker, λ_{CC} may be extended to support this type of soft guarantee: predicate and structural contracts will use a random number generator to decide which, if any, contracts to enforce, and the full spot-checker will generate sample inputs to contracted functions. This approach induces a secondary extension to λ_{CC}: while statistical guarantees may receive their own strategy, it is more natural to introduce a *strategy combinator* that will take an existing strategy (such as **eager** or **semi**) and modify it to preserve the original enforcement strategy while performing probabilistic contract enforcement.

9. Related Work

Eiffel [25] first popularized software contracts and the idea of writing programs with pervasive contract checking. Eiffel introduced a number of theoretical and semantic concepts that have since induced a large body of research, including the underlying theoretical foundations [4, 15, 16], language integration with existing semantics [1, 6–8, 11, 19, 20, 22]. Degen et al. [7, 8] present descriptions of eager, semi-eager, and lazy monitors; we have characterized the first two precisely and sketched the third in this paper.

Dimoulas and Felleisen [9] address different approaches to monitoring through observational equivalence, relying on these observations to discuss when and how contracts affect the underlying program. They reconstruct contract satisfaction from these principles, introduce the concept of a *shy* contract (which are similar to lazy monitoring as sketched in the previous section), and propose a classification for contracts based on contract satisfaction and observational equivalence. Most of the monitors that we describe in this paper fall in a spectrum between *loose* and *shy-loose run-time checking*; the **async** strategy is an exception, since it can non-deterministically behave either like **future**, **eager**, or like no monitoring at all. We do not formally consider contract satisfaction or observational equivalence, but future work might explore reconstructing these results in our semantics.

A variety of alternative semantic models exist, primarily focusing on the original notion of eager contract monitoring [4, 15, 17]. We follow Findler and Felleisen [16] in defining a contract system rather than a model of contract satisfaction.

Multiple approaches to blame assignment, particularly in the case of dependent function contracts, have been proposed [11, 20], resulting in the definition of *complete monitoring* [11]. Since our contract combinators are library functions, we can provide any blame assignment approach (e.g. *lax*, *picky*, and *indy*). Proving complete monitoring in λ_{CC} remains future work.

10. Conclusion and Future Work

We present a unifying perspective on the semantics of contract monitoring, wherein myriad semantic approaches and contract combinators can be characterized by translation into a straightforward process calculus. This approach captures multiple existing approaches to contract monitoring and introduces several new ones. We also present a unified source language in which programmers may select strategies on a per-contract basis, rather than the current status quo where the choice is fixed by the contract system designer. Furthermore, we propose that λ_{CC} is a suitable target for new contract monitoring approaches, providing a extensible interpretation of the interactions between contracts, contract monitors, and user programs. Finally, we demonstrate how the runtime framework can be leveraged to express contracts inexpressible in existing systems, focusing on upward-propagating delayed contracts for binary trees.

In the future we hope to investigate how the formal semantics translates into existing process-oriented languages such as Erlang [2], explore the design space of strategies and strategy combinators, and investigate if proofs of complete monitoring [11] can be carried out in our system.

Acknowledgments

We thank Michael M. Vitousek, Jeremy G. Siek, Matthias Felleisen, Robert Bruce Findler, and the IU Gradual Types group for their discussion and feedback. We thank the anonymous reviewers of ICFP 2015 for their detailed reviews, which helped to improve the presentation and technical content of the paper. This material is based upon work supported by the National Science Foundation under Grant Nos. 1117635, 1217454, 1218390, 1421652, and a grant from the National Security Agency.

References

[1] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL*, 2011.

[2] J. Armstrong, S. Virding, and M. Williams. *Erlang Users Guide and Reference Manual. Version 3.2*. Ellemtel Utveklings AB, 1991.

[3] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *SAIPL*, 1977.

[4] M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 2006.

[5] O. Chitil. Practical typed lazy contracts. In *ICFP*, 2012.

[6] O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In *IFL*, 2003.

[7] M. Degen, P. Thiemann, and S. Wehr. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *ATPS*, 2009.

[8] M. Degen, P. Thiemann, and S. Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *J. Log. Algebr. Program.*, 79(7), 2010.

[9] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *TOPLAS*, 33(5), Nov. 2011.

[10] C. Dimoulas, R. Pucella, and M. Felleisen. Future contracts. In *PPDP*, 2009.

[11] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitoring for behavioral contracts. In *ESOP*, 2012.

[12] C. Dimoulas, R. B. Findler, and M. Felleisen. Option contracts. In *OOPSLA*, 2013.

[13] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *ICFP*, 2011.

[14] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. In *STOC*, 1998.

[15] R. B. Findler and M. Blume. Contracts as pairs of projections. In *FLOPS*, 2006.

[16] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[17] R. B. Findler, M. Blume, and M. Felleisen. An investigation of contracts as projections. University of Chicago Technical Report TR02-402, 2002.

[18] R. B. Findler, S.-Y. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *IFL*, 2008.

[19] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[20] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. *Journal of Functional Programming*, 2012.

[21] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. LFP, 1984.

[22] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *FLOPS*, 2006.

[23] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998.

[24] A. Jeffrey. Semantics for core concurrent ml using computation types. In *HOOTS*. Cambridge University Press, 1997.

[25] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992. ISBN 0-13-247925-7.

[26] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2, 6 1992.

[27] C. Morgan. *Programming from specifications*. 1990.

[28] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 1972.

[29] J. H. Reppy. Concurrent ML: Design, application and semantics, 1993.

[30] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. ISBN 0521480892.

[31] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[32] T. S. Strickland and M. Felleisen. Contracts for first-class classes. In *DLS*, Oct. 2010.

[33] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*, 2012.

[34] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, 1994.

A. Summary of Forms

This appendix provides a short description of the uncommon calculus operations we use in the paper, along with the figures where they are defined.

Library Definitions

- pred/c : $(\tau \rightarrow \texttt{bool}) \rightarrow \texttt{con}\ \tau$
  Constructs a predicate contract from a predicate, using raise (see Figure 6).
- pair/c : $\tau_1 \rightarrow \sigma \rightarrow \tau_2 \rightarrow \sigma \rightarrow \texttt{con}\ (\tau_1 \times \tau_2)$
  Constructs a pair contract from two subcontracts and their strategies using invocations of check (see Figure 6).
- func/c : $\tau_1 \rightarrow \sigma \rightarrow \tau_2 \rightarrow \sigma \rightarrow \texttt{con}\ (\tau_1 \rightarrow \tau_2)$
  Constructs a function contract from two subcontracts and their strategies using invocations of check (see Figure 6).

User Contract System

- check : $\texttt{con}\ \tau \rightarrow \sigma \rightarrow \tau \rightarrow \tau$
  Performs monitoring using the provided strategy; this is constructed as a "language macro" from spawn, catch, read, write, delay, and chan (see Figure 5).
- force : $\tau \rightarrow \tau$
  Forces the expression; this is constructed as a procedure that inspects its input, forcing it in the case of a delay reference and returning the value of its input otherwise (see Figure 4).
- raise : $\tau \rightarrow \tau$
  Raises its input as an exception; this operation is built into the core calculus (see Figure 3).

Communication

- chan : $(\texttt{chan}\ \tau \rightarrow \tau) \rightarrow \tau$
  Creates a new channel and passes it as the input to chan's argument; this operation is built into the core calculus (see Figure 3).
- read : $\texttt{chan}\ \tau \rightarrow \tau$
  Synchronously reads a value from the provided channel; this operation is built into the core calculus (see Figure 3).
- write : $\texttt{chan}\ \tau \rightarrow \tau \rightarrow ()$
  Synchronously writes a value to the provided channel; this operation is built into the core calculus (see Figure 3).

Runtime

- catch : $(\tau \rightarrow \tau') \rightarrow \tau' \rightarrow \tau'$
  Catches a raised exception with the provided handler, or returns the result if an exception is not raised; this operation is built into the core calculus (see Figure 3).
- delay : $\tau \rightarrow \tau$
  Delays its input, constructing a delay-cell process; this is constructed as a "language macro" from spawn, catch, read, write, and chan (see Figure 4).
- $\text{dref}_{\iota_1,\iota_2} : \tau$
  A record or tagged pair of channels that indicate a delayed reference, containing the channels necessary to interact with it via force; this is a data constructor (see Figure 4).
- spawn : $T \rightarrow \tau \rightarrow ()$
  Creates a new process using the provided process tag; this operation is built into the core calculus (see Figure 3).