Lackadaisical Contracts for Higher-Order Functions*

CAMERON MOY, University of Maryland, USA
ANFISA BOGDANENKO, PLT, Switzerland
MATTHIAS FELLEISEN, PLT, USA

Over the past twenty-five years, higher-order contracts have become a well-established mechanism for enforcing software specifications. Unlike their first-order cousins, these contracts can interact with higher-order values (e.g., functions or objects) that cross protected component boundaries. While these interactions may look benign at first, they can cause all kinds of undesirable effects including nontermination, resource consumption, reading from input devices, and writing to output devices.

In response, this paper presents lackadaisical contract checking, a semantic variant that defers interactions with higher-order values in contract code until the program itself demands the same interaction. This semantics satisfies typical metatheoretic properties (blame soundness and complete monitoring) plus strong erasure. Its implementation validates that the idea can be realized within a production-grade contract system.

CCS Concepts: • **Software and its engineering** → **Semantics**.

Additional Key Words and Phrases: software contracts, computational effects

ACM Reference Format:
Cameron Moy, Anfisa Bogdanenko, and Matthias Felleisen. 2026. Lackadaisical Contracts for Higher-Order Functions. 1, 1 (November 2026), 44 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Higher-Order Contracts and When to Check Them

When developers compose pieces of software into a large system, the correctness of the whole depends on the behavior of each piece. As long as each piece works as intended, the overall system functions properly. Inevitably, though, some piece of the system fails to live up to its specification, and maintainers must isolate the problem.

If software components come with sufficiently precise contracts [37], an error report about a failing contract narrows down the debugging search space. A contract system catches invalid exchanges of values across component boundaries, and when such an exchange occurs, the system assigns blame to one specific boundary. For first-order software, this boundary exposes the bug: it is in either one of the bordering pieces of software or a defect in the contract itself. However, contemporary languages are higher order, so developers need to deploy higher-order contracts as well [25, 26]. Compared to first-order contracts, blame is less precise in the higher-order world, and a "blame shifting" strategy might be necessary to locate bugs [36]. Even so, higher-order contracts are an effective debugging aide.

Although higher-order contracts for first-class functions and objects have been investigated for over two decades, they still pose interesting semantic and pragmatic questions. This paper asks

*Research partially conducted at Northeastern University.

Authors' Contact Information: Cameron Moy, University of Maryland, College Park, Maryland, USA, camoy@cs.umd.edu; Anfisa Bogdanenko, PLT, Zurich, Switzerland; Matthias Felleisen, PLT, Cedar Park, Texas, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

whether a higher-order contract should explore more parts of a behavioral value (e.g., a function or object) than the component that consumes it. The research literature on eager languages answers this question in the affirmative, but as the next section demonstrates, such extraneous explorations can cause serious, partially hidden problems.

```
(provide
  (contract-out
   [at-two (->i₁ ([f (-> integer? integer?)])
                 #:pre (f) (= (f 0) 0)₂
                 [res integer?])]))
```

```
(define (at-two f)
  (f 2))
```

Fig. 1. Contract for a toy example in Racket

Consider the toy example in Figure 1, written in the Racket programming language [29]. The higher-order function `at-two` consumes a function `f`. It imposes a contract on the integer function `f` and demands that `0` is a root of `f`. But, the function body of `at-two` does not apply `f` to `0`. This program raises the question whether a contract may call the given higher-order argument on more values than the function body itself. After all, `f` may perform side effects when called on `0`. Given this concrete example, the research question can be refined in two ways: whether these effects should be visible and whether the effect should be observable twice if the main program were to explore `f` at `0`, too. In either case, an "eager" contract system semantics introduces additional behavior that is not present in the program proper.

Generally speaking, the proponents of contracts tend to state that contracts add nothing to a program execution except possibly signaling contract violations. If a program does not violate any contracts, it should be possible to erase the contracts without any observable difference in behavior. Most models of such systems fail to satisfy this erasure property because a contract on a higher-order value can perform effects that change a program's behavior.

This paper presents an alternative semantics, dubbed *lackadaisical*.[1] It checks higher-order contracts only to the extent that the main program explores behavioral values. Consequently, it avoids duplicating effects and satisfies erasure. The central contribution is a novel *synchronization* mechanism that coordinates the transfer of values from main-program code to contract code. A formal model (Section 3) validates that the design satisfies traditional metatheoretic properties (blame soundness [14], complete monitoring [17]) as well as strong erasure (Section 4). An implementation (Section 5) confirms that the design can be realized within an existing contract system.

2 Lackadaisical Checking for Higher-Order Values

Before diving into a formal model, it will be helpful to see how the lackadaisical semantics works informally with two examples. Section 2.1 uses the example from Figure 1 to explain how lackadaisical checking differs from standard immediate contract checking. Section 2.2 considers a realistic example where lackadaisical checking is critical for a resource-aware evaluation of contracts, namely, for a contract that enforces the properties of a total order.

[1]The name lackadaisical is intended to capture the idea that the contract defers checks involving higher-order interactions to the main program. "Lackadaisical" avoids connotations associated with the word "lazy" because the ambient programming language remains call-by-value. See Section 6 for a discussion about contract checking in lazy languages.

2.1 Lackadaisical Checking for a Toy Example

Figure 1 displays a complete component. It defines the `at-two` function and exports it with a contract. A gray line separates the module's export interface from its body. Two pieces in the `at-two` contract deserve special attention (see highlights and indices):

(1) The function contract states that `at-two` is a function of one argument, named `f`, which maps integers to integers, and an integer result. Additionally, `at-two` must satisfy a boolean-valued precondition.[2]

(2) The precondition expression stipulates that it depends on argument `f`. Its boolean-valued expression must be satisfied on each application of `at-two`. This assertion guarantees that `f` has a root at zero. Postconditions are also supported, as **??** shows.

The ambient contract system monitors interactions with `at-two` across module boundaries. If the terms of the contract are violated, then the system raises an error blaming the component responsible for the transgression. Contract checking for higher-order functions works by wrapping functions in a proxy that monitors argument and return values [26].

Determining whether a program that imports `at-two` is in violation of the contract requires an understanding of what "satisfies the contract" actually means. Consider a client that imports the module from Figure 1 and evaluates `(at-two (λ (x) (+ x 1)))`. Racket's `->i` contract combinator implements immediate checking and thus fully evaluates the precondition *before* control enters the protected function. In this case, the precondition check fails and the contract system signals a violation. Compared to the main program, this interpretation of contract satisfaction over-eagerly explores the given function If the client were to run `(at-two (λ (x) (if (zero? x) Ω x)))`, where Ω diverges, then the program with contracts would diverge even though the program without contracts would terminate. In short, the contract adds effects by virtue of interacting with a higher-order value.

This paper proposes a *lackadaisical* interpretation of function contracts and introduces a different function contract combinator: `->l`. By replacing `->i` with `->l`, the contract system interprets contract satisfaction differently.[3] To be precise, the contract system suspends evaluation of the function contract when it encounters `(f 0)` inside the precondition. Control is then transferred to the protected function. There are two cases depending on the behavior of the exported function:

(1) If it were to apply `f` to `0`, the contract system would resume the evaluation of `(= (f 0) 0)` and complete the contract check.

(2) If it returns without ever applying `f` to `0`, as is the case with `at-two` in Figure 1, the contract system does not evaluate the suspended check.

The suspension mechanism avoids exploring the behavioral value `f` beyond what the protected function `at-two` sees. Unlike the eager exploration performed by Racket's existing contract system, the lackadaisical interpretation avoids the effect of divergence.

2.2 Enforcing Properties of Total Orders

The `at-two` example fails to illustrate where lackadaisical contract checking really matters. Consider instead a contract for sorting functions that is parameterized by the type of elements:

[2]Employing `->i`, instead of the plain `->` function contract combinator, allows specifying a relationship among the arguments and the result, including preconditions and postconditions.

[3]This example can be translated into a dependent function contract on `f`:

```
(->i ([x number?]) [result (x) (if (zero? x) (=/c 0) any/c)]).
```

Similar to a lackadaisical contract, this one ensures that `f` maps `0` to `0` without applying `f`. In contrast to the lackadaisical one, this immediate contract incurs a check on every application to `0`, even if the program has witnessed that the function maps `0` to `0`. Worse, preconditions cannot always be expressed [20] with `->i` in this way. If the precondition relates applications of a function on two different inputs, e.g., `(= (f 0) (f 1))`, expressing it with a dependent function contract is impossible.

```
(define (sort/c α) (-> (listof α) (ord/c α) (listof α)))
(define (ord/c α)  (-> α α boolean?))
```

This contract stipulates simple type-like constraints. Given some contract α, any function protected by `(sort/c α)` must take in a list of αs, a comparator of αs, and return a list of αs.[4]

Run-time-checked contracts are easy to strengthen [15]:

```
(provide
  (contract-out
    [quicksort (sort/c integer?)]))

(define (sort/c α)
  (->i ([lst (listof α)]
        [⊑  (ord/c α)])
       [res (lst ⊑) (and/c (permutation/c lst) (ascending/c ⊑))]))
```

Here the result contract *depends* on the inputs and satisfies a much stronger property than the original. It ensures that the output list is a permutation of the input list and is in ascending order according to the comparator. These additional output contracts, `(permutation/c lst)` and `(ascending/c ⊑)`, can be written as ordinary predicates.

At first glance, `sort/c` may appear to be a comprehensive specification. However, `ascending/c` assumes, but does not check, that the comparator ⊑ is a total order. If a given comparator fails this property, `sort/c` may incorrectly blame the sorting function for violating `ascending/c`, even though the caller is at fault for supplying a bogus comparator. The solution is to strengthen the `ord/c` contract and check for the properties of a total order.

```
(define (ord/c α)
  (self/c ⊑₁ ([A α])₂
    (-> A A boolean?)₃
    #| reflexivity  |# (∀ ([x A]) (⊑ x x))₄
    #| antisymmetry |# (∀ ([x A] [y A]) (implies (and (⊑ x y) (⊑ y x)) (equal? x y)))₄
    #| transitivity |# (∀ ([x A] [y A] [z A]) (implies (and (⊑ x y) (⊑ y z)) (⊑ x z)))₄
    #| totality     |# (∀ ([x A] [y A]) (or (⊑ x y) (⊑ y x)))₄))
```

Fig. 2. A full contract for comparators used with `sort`

Figure 2 shows a contract that articulates all the properties of a total order. The `self/c` contract combinator sets up a self-referential contract with a series of formulas (labeled with comments), each of which must be satisfied. The highlighted pieces of code deserve further explanation:

(1) The ⊑ identifier is a self reference that is bound to the comparator itself, for use in the formulas. Access to ⊑ allows the contract to explore its behavior, potentially more than the main program itself has observed.
(2) Set declarations introduce value collectors, which accumulate the values that each of the formulas can be quantified over. Here, `A` represents the set of αs observed as inputs to the comparator at run time.
(3) The body contract `(-> A A boolean?)` is the familiar signature of comparators. Here, `A` is bound to a contract that collects all values that flow as inputs to the comparator. These values must satisfy α.

[4] If α comes from a parametric contract [32], then the comparator can be applied only to elements present in the input list.

(4) Each boolean formula is instantiated with the variables `x`, `y`, and `z` bound to values collected by `A`. If evaluating any of these formulas yields `false`, then the contract is violated and the system raises an error.

Enforcing reflexivity, antisymmetry, and totality seems reasonably straightforward. Consider antisymmetry. If a comparator is applied to distinct `x` and `y`, the contract can check whether `(⊑ x y)` yields `true`, and if so, it evaluates `(⊑ y x)` to ensure that it returns `false`. The contract can also check reflexivity and totality in this manner. Transitivity is more interesting. Expressed as a logical formula, transitivity states that

$$\forall x, y, z \,.\, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z.$$

Note that *every* `z` such that `(⊑ y z)` is `true` introduces the possibility of a violation, whereas antisymmetry and totality require only one other application of ⊑ to witness a violation.

Constructing all these additional calls to the comparator clarifies the need to answer the research question posed in Section 1. Each of these constructed calls may cause an effect and definitely come with a run-time cost.

An immediate evaluation strategy for this `self/c` contract destroys the efficiency of the underlying sorting function. Given n comparisons at run time, checking transitivity generates $O(n^3)$ additional comparisons. Additional interactions with ⊑ inside of `self/c`, beyond what the main program itself evaluates, are responsible for the additional comparisons.

A lackadaisical evaluation strategy solves the problem. Each first-order check (i.e., an instantiation of a boolean formula) is processed separately from the main program. The two are coordinated via a *synchronization table*, which is created and maintained for every function monitored by `self/c`. The table resembles a memoization table, but it is split into two views: a read view and a write view. Contract code only reads from the synchronization table and main-program code only writes to it. Critically, when a contract attempts to interact with a higher-order value (e.g., calls to ⊑ in the formulas from Figure 2) it first tries to read the result from the synchronization table. If the main code has computed the result in the past and stored it in the synchronization table, it is returned immediately; otherwise, contract evaluation is suspended and control returns to the main program. If the desired application is eventually computed by the main program, then control returns to the contract so it can be checked.

Table 1. An example of the lackadaisical interpretation of `self/c`

application	synchronization table	contract checks generated at this stage (excerpt)

`(<= 1 2) = true`	`[1 ↦ [2 ↦ true]]`	`(<= 1 1)`
		`(<= 2 2)`
		`(implies (and (<= 1 2) (<= 2 1)) (equal? 1 2))`
		`(or (<= 1 2) (<= 2 1))` ✓

`(<= 2 3) = true`	`[1 ↦ [2 ↦ true],`
	`2 ↦ [3 ↦ true]]`
		`(<= 3 3)`
		`(implies (and (<= 2 3) (<= 3 2)) (equal? 2 3))`
		`(or (<= 2 3) (<= 3 2))` ✓
		`(implies (and (<= 1 2) (<= 2 3)) (<= 1 3))`

`(<= 1 3) = false`	`[1 ↦ [2 ↦ true, 3 ↦ false],`
	`2 ↦ [3 ↦ true]]`
		`(implies (and (<= 1 3) (<= 3 1)) (equal? 1 3))`
		`(or (<= 1 3) (<= 3 1))`
		`(implies (and (<= 1 2) (<= 2 3)) (<= 1 3)) = false` ✗

In order to understand the workings of this new contract-checking interpretation, consider a program that evaluates `(<= 1 2)`, `(<= 2 3)`, and `(<= 1 3)`, where the `<=` comparator is protected by `(ord/c integer?)`. Table 1 shows how each application modifies the synchronization table and adds to the pool of contract checks. Each row corresponds to an application of the `<=` function. The first row illustrates what happens if `(<= 1 2)` evaluates to `true`. After returning from the comparator, the synchronization table contains an entry for the inputs and output. Each of the four formulas in the `ord/c` contract can contribute contract checks to the pool. Shown in the third column of the table is an excerpt of each pool. The initial two checks concern reflexivity on the inputs 1 and 2. Because the main program has not evaluated the highlighted expressions, these checks are blocked and control is transferred to the main program. A blocked check remains in the pool until it is discharged by the corresponding application. The final check, corresponding to totality, is satisfied because `(<= 1 2)` holds and the `or` form short circuits evaluation. Once satisfied, this check can be removed from the pool.

The second row is similar to the first, except the last check listed is generated from the transitivity property. This check is blocked on `(<= 1 3)` because `(<= 1 2)` and `(<= 2 3)` have already been evaluated and their results can be read directly from the synchronization table. Since both of these expressions evaluate to `true`, the implication remains falsifiable. Indeed, the third row shows what happens if `(<= 1 3)` were to evaluate to `false`. The blocked check would also yield `false` and thus violate transitivity. At this point, the `ord/c` contract would have to signal an error.

Stepping back, the example demonstrates how immediate contract checking involving behavioral values negatively affects program execution. While Section 2.1 highlights that immediate contract checking may cause unintended divergence or duplication of observable effects, say debug prints, the shift in algorithmic complexity pointed out here has equally bad consequences. Lackadaisical contract checking avoids both of these undesirable behaviors.

3 A Semantics for Lackadaisical Contract Checking

The model of lackadaisical contracts (LCPCF) restricts and adapts the contract PCF model [12] (or CPCF for short) from immediate contract checking to lackadaisical contract checking. The extension permits predicates over first-order functions as contracts, as seen with `->1`, which is the absolute minimum needed to enable contract code to interact with higher-order values from the main program. The point of this restriction is to focus the model on the essence of lackadaisical checking. Section 5 explains how an implementation can deal with unrestricted higher-order interactions.

$$
\begin{array}{rcl}
\text{Expr} \ni e & ::= & x \mid a \mid \texttt{if}\ e\ e\ e \mid f \mid e\ e \mid \mu x.e \mid \texttt{mon}_j^{k,l}\ c\ e \\
\text{Atom} \ni a & ::= & \texttt{tt} \mid \texttt{ff} \\
\text{Fun} \ni f & ::= & \lambda x.e \\
\text{Con} \ni c & ::= & \texttt{flat}\ f \mid c \to c \mid c \to c\ \&\ f \\
\text{Var} \ni x, y, z & & \\
\text{Lab} \ni j, k, l & &
\end{array}
$$

Fig. 3. Syntax for LCPCF (CORE)

3.1 Syntax

Figure 3 presents the core syntax for LCPCF. Most of the constructs are familiar from ordinary PCF [42], which distills a functional language to its essential features (variables, booleans, conditionals, functions, applications, and recursion). CPCF adds one element, the monitor expression

$\texttt{mon}_j^{k,l}\ c\ e$, which plays two roles. (1) It protects the value of e with contract c. As such, it serves the same role as `contract-out`. (2) It names the components of the program that agree to, and supply, the contract. The contract system uses component labels (i.e., k, l, and j) to explain violations. Here, k is the label of the server party; l names the lexical context of the monitor expression, which serves as the client party; and j points to the contract-defining party, which, in a higher-order world, is code and may also cause a violation.

LCPCF removes one contract from from CPCF and adds one:

(1) $\texttt{flat}\ f$ is the common flat contract. It specifies first-order properties of atomic values.
(2) $c \to c'$ is the common basic function contract; arguments must satisfy c and results c'.
(3) $c \to c'\ \&\ f$ is the new element, a *flat-function* contract. It combines the function contract $c \to c'$ with a flat check f, which represents the precondition/postcondition component of `->1`. As such, f is a predicate over a higher-order value.

It is this last, new form that assesses a function's behavior in a lackadaisical manner and therefore deserves a thorough semantic explanation.

3.2 Semantics

The model employs a conventional reduction semantics [21]. Its presentation consists of two parts: the dynamic syntax and the reduction relation.

$$
\begin{array}{rcl}
\text{Config} \ni \varsigma & ::= & \langle \vec{t}, e, \sigma \rangle \mid \langle \texttt{err}_j^k, \sigma \rangle \\
\text{Cor} \ni t, s & ::= & \texttt{assert}_j^k\ e \\
\text{Expr} \ni e & ::= & \ldots \mid (\alpha, v) \lhd e \mid (\alpha, v) \rhd \\
\text{Val} \ni v & ::= & x \mid a \mid f \\
\text{Store} \ni \sigma & \overset{\text{def}}{=} & \text{Addr} \xrightarrow{\text{fin}} (\text{Atom} \xrightarrow{\text{fin}} \text{Atom}) \\
\text{Addr} \ni \alpha & &
\end{array}
$$

$$
\begin{array}{rcl}
\text{Ctx}^{\text{A}} \ni A^{\sigma} & ::= & \vec{s} +\!\!+ T +\!\!+ \vec{t}, e \ \ \lceil \sigma \vdash s_i \text{ blocked} \rfloor \mid \vec{s}, E \ \ \lceil \sigma \vdash s_i \text{ blocked} \rfloor \\
\text{Ctx}^{\text{T}} \ni T & ::= & \square \mid \texttt{assert}_j^k\ E \\
\text{Ctx}^{\text{E}} \ni E, F & ::= & \square \mid \texttt{if}\ E\ e\ e \mid E\ e \mid v\ E \mid \texttt{mon}_j^{k,l}\ c\ E \mid (\alpha, v) \lhd E
\end{array}
$$

$$
\sigma \vdash s \text{ blocked} \overset{\text{def}}{=} (s = T[(\alpha, v) \rhd] \wedge \sigma(\alpha)(v) \text{ undefined})
$$

Fig. 4. Syntax for LCPCF (DYNAMIC)

Figure 4 displays the extension of the core syntax needed to specify the semantics. The key element of this extension is the configuration ς, which contains an ordered sequence of *coroutines* $\vec{t}$, i.e. suspended checks; a main-program expression e; and a synchronization store σ. A coroutine is the application of a predicate to an atomic value. A configuration may also be a final error state.

The set of dynamic expressions consists of the core set and two new forms: $(\alpha, v) \lhd e$ writes the value of e to the synchronization table at α with argument v, and $(\alpha, v) \rhd$ reads the value of the synchronization table at α with argument v. The store maps an address α, which identifies a synchronization table, to a finite map between atomic values. Critically, the keys of the synchronization table are atomic so that lookups are decidable. If the keys were arbitrary values, the lookup of a function value may not be computable.

The second part of Figure 4 defines the set of evaluation contexts [23]. Evaluation contexts for coroutines (T) and expressions (E) are standard for a call-by-value language. In order to keep

evaluation deterministic, the definition of global evaluation contexts A^σ is parameterized over a store σ. Using the shape of the store, this definition concisely specifies the next coroutine to be reduced—if any. To be precise, an evaluation context selects the left-most coroutine that is not blocked according to a synchronization table in σ. If all coroutines are blocked, the main program is reduced. Concatenation of coroutine sequences is indicated by the $+\!\!+$ operator. The last part of the figure provides the natural definition for the side condition $\lceil \sigma \vdash s_i \text{ blocked} \rfloor$. In other words, the definition of A^σ determines the scheduler for coroutines, where any contract coroutine has priority over the main program.

Figures 5 to 7 display the reduction relation for LCPCF, partitioned into three groups.

$$
\begin{array}{rr}
\textsc{if-tt} & \langle A^\sigma[\texttt{if tt } e_1\ e_2], \sigma\rangle \longmapsto \langle A^\sigma[e_1], \sigma\rangle \\
\textsc{if-ff} & \langle A^\sigma[\texttt{if ff } e_1\ e_2], \sigma\rangle \longmapsto \langle A^\sigma[e_2], \sigma\rangle \\
\textsc{fun-app} & \langle A^\sigma[(\lambda x.e)\ v], \sigma\rangle \longmapsto \langle A^\sigma[[v/x]e], \sigma\rangle \\
\textsc{fix-unroll} & \langle A^\sigma[\mu x.e], \sigma\rangle \longmapsto \langle A^\sigma[[\mu x.e/x]e], \sigma\rangle
\end{array}
$$

Fig. 5. Reduction relation rules for the functional core

Figure 5 shows the first four rules, which are familiar from ordinary PCF. The use of A^σ in these rules means that the reduction can occur either in a coroutine or in the main-program expression.

$$
\begin{array}{rl}
\textsc{mon-flat} & \langle A^\sigma[\texttt{mon}_j^{k,l}\ (\texttt{flat}\ f)\ a], \sigma\rangle \longmapsto \langle \texttt{assert}_j^k\ (f\ a) +\!\!+ A^\sigma[a], \sigma\rangle \\
\textsc{mon-arr} & \langle A^\sigma[\texttt{mon}_j^{k,l}\ (c_1 \to c_2)\ v], \sigma\rangle \longmapsto \langle A^\sigma[\lambda x.\texttt{mon}_j^{k,l}\ c_2\ (v\ (\texttt{mon}_j^{l,k}\ c_1\ x))], \sigma\rangle \\
\textsc{mon-arr-flat} & \langle A^\sigma[\texttt{mon}_j^{k,l}\ (c_1 \to c_2\ \&\ f)\ v], \sigma\rangle \longmapsto \langle \texttt{assert}_j^k\ (f\ (\lambda x.(\alpha, x) \triangleright)) +\!\!+ \\
& \qquad A^\sigma[\lambda x.(\alpha, x) \triangleleft \texttt{mon}_j^{k,l}\ c_2\ (v\ (\texttt{mon}_j^{l,k}\ c_1\ x))], \\
& \qquad \sigma[\alpha \leftarrow \varnothing]\rangle \\
& \qquad \lceil \alpha \notin \text{dom}(\sigma) \rfloor
\end{array}
$$

Fig. 6. Reduction relation rules for monitoring contracts

Figure 6 displays the set of rules that explains how contract monitoring works.

(1) MON-FLAT is about flat-contract checks. For uniformity, it spawns a new coroutine that checks the predicate f on the given value. Because contract coroutines have priority, and flat contracts are given only atomic values, this contract check evaluates completely before control returns to the main program.
(2) MON-ARR reproduces the standard semantics of higher-order contract checking. A simple function-contract check wraps the given function in a proxy that checks the argument and result for conformance with the contract. Following Findler and Felleisen [26], the blame labels of the argument check are reversed because argument positions are contravariant.
(3) MON-ARR-FLAT combines the two rules into lackadaisical checks for higher-order values: it spawns a coroutine that indirectly checks the predicate f with a *handle* of the given function, and it injects a *proxy* into the program context. Put differently, the semantics splits v in half, where the contract coroutine gets a handle of v and the main program gets a proxy

of v. The handle of v takes the form of a synchronized read: $(\lambda x.(\alpha, x) \triangleright)$. The proxy of v writes to the synchronization table α every time a call to v returns a result that successfully passes its contract:

$$\lambda x.(\alpha, x) \triangleleft \mathtt{mon}_j^{k,l}\ c_2\ (v\ (\mathtt{mon}_j^{l,k}\ c_1\ x)).$$

Hence, the combination of proxy and handle is extensionally equivalent to v.

$$\begin{array}{ll}
\textsc{sync-get} & \langle A^\sigma[(\alpha, a) \triangleright], \sigma\rangle \longmapsto \langle A^\sigma[a'], \sigma\rangle \quad \lceil \sigma(\alpha)(a) = a' \rfloor \\
\textsc{sync-set} & \langle A^\sigma[(\alpha, a) \triangleleft a'], \sigma\rangle \longmapsto \langle A^\sigma[a'], \sigma(\alpha)[a \leftarrow a']\rangle \\
\textsc{assert-tt} & \langle A^\sigma[\mathtt{assert}_j^k\ \mathtt{tt}], \sigma\rangle \longmapsto \langle A^\sigma[\varepsilon], \sigma\rangle \\
\textsc{assert-ff} & \langle A^\sigma[\mathtt{assert}_j^k\ \mathtt{ff}], \sigma\rangle \longmapsto \langle \mathtt{err}_j^k, \sigma\rangle
\end{array}$$

Fig. 7. Reduction relation rules for synchronization actions and assertions

Figure 7 finishes the definition of the reduction relation. SYNC-GET and SYNC-SET realize reading from, and writing to, the synchronization table, respectively. Writing the atomic value a' to the synchronization table at address α is denoted by $\sigma(\alpha)[a \leftarrow a']$. The rules concerning assertions perform the actual checks: ASSERT-TT deletes a successful thread from the pool (indicated by ε), and ASSERT-FF terminates execution with an error that assigns blame. In the absence of flat-function contracts, the assertion-checking semantics is equivalent to the standard semantics of contracts. A flat-contract check evaluates in a new coroutine, returns a boolean value without suspending, and either resumes executing the program or terminates with an error. Coroutines can be suspended only due to the presence of a flat-function contract that interacts with its higher-order input.

In sum, the reduction relation specifies the enforcement of contracts via cooperative concurrency. The synchronization tables in the store serve to coordinate the transfer of control between contract code and main-program code. Otherwise, the semantics resembles the conventional semantics of higher-order contracts.

3.3 Types

The original PCF model is typed and so is LCPCF. Figure 8 shows its type syntax and typing rules. There is one type constructor, $\mathtt{Con}(\cdot)$, that does not exist in PCF. A value of type $\mathtt{Con}(\tau)$ is a contract for values of type τ. For example, in an extension of LCPCF with numbers, the absolute value function has type $\mathtt{Int} \to \mathtt{Int}$. This type can be refined with the contract $\mathtt{flat}\ (\lambda x.\mathtt{tt}) \to \mathtt{nat/c}$, where $\mathtt{nat/c}$ is $\mathtt{flat}\ (\lambda x.x \geq 0)$. The contract itself has type $\mathtt{Con}(\mathtt{Int} \to \mathtt{Int})$.

Concerning the typing judgment and rules, those for the core expressions come with just one contract-specific rule: the one for monitors. It delegates to a second judgment form with its own rules, namely, those for each contract form. Since the syntax comes with three kinds of contracts, there are three rules. To keep the presentation simple, the $\mathtt{flat}\ f$ and $c_1 \to c_2\ \&\ f$ contract forms place restrictions on the types of f:

(1) In the flat case, f is a predicate over atomic values. This restriction ensures that flat checks have no higher-order interactions and therefore require no synchronization.
(2) In the flat-function case, f is a predicate over first-order functions. This restriction ensures that the synchronization table contains atomic values so that retrievals remain decidable.

Highlighted in Figure 8 are two substantial departures from the usual CPCF type system. These premises ensure that flat contracts are closed (i.e., typed under the empty environment) and do

$$\begin{array}{rcl} \mathsf{Type} \ni \tau & ::= & \mathbb{A} \mid \tau \to \tau \mid \mathsf{Con}(\tau) \\ \mathsf{Atoms} \ni \mathbb{A} & ::= & \mathsf{Bool} \\ \mathsf{Env}^{\tau} \ni \Gamma & \stackrel{\text{def}}{=} & \mathsf{Var} \xrightarrow{\text{fin}} \mathsf{Type} \\ \mathsf{Expr}^{\downarrow} \ni e^{\downarrow} & ::= & x \mid a \mid \mathtt{if}\ e^{\downarrow}\ e^{\downarrow}\ e^{\downarrow} \mid f^{\downarrow} \mid e^{\downarrow}\ e^{\downarrow} \mid \mathsf{mon}_j^{k,l}\ c\ e^{\downarrow} \\ \mathsf{Fun}^{\downarrow} \ni f^{\downarrow} & ::= & \lambda x.e^{\downarrow} \end{array}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{tt} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathtt{ff} : \mathsf{Bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathtt{if}\ e_1\ e_2\ e_3 : \tau}$$

$$\frac{\Gamma[x \leftarrow \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Gamma[x \leftarrow \tau] \vdash e : \tau}{\Gamma \vdash \mu x.e : \tau} \qquad \frac{\Gamma \vdash c : \mathsf{Con}(\tau) \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{mon}_j^{k,l}\ c\ e : \tau}$$

$$\frac{\varnothing \vdash f^{\downarrow} : \mathbb{A} \to \mathsf{Bool}}{\Gamma \vdash \mathtt{flat}\ f^{\downarrow} : \mathsf{Con}(\mathbb{A})} \qquad \frac{\Gamma \vdash c_1 : \mathsf{Con}(\tau_1) \qquad \Gamma \vdash c_2 : \mathsf{Con}(\tau_2)}{\Gamma \vdash c_1 \to c_2 : \mathsf{Con}(\tau_1 \to \tau_2)}$$

$$\frac{\Gamma \vdash c_1 : \mathsf{Con}(\mathbb{A}_1) \qquad \Gamma \vdash c_2 : \mathsf{Con}(\mathbb{A}_2) \qquad \varnothing \vdash f^{\downarrow} : (\mathbb{A}_1 \to \mathbb{A}_2) \to \mathsf{Bool}}{\Gamma \vdash c_1 \to c_2\ \&\ f^{\downarrow} : \mathsf{Con}(\mathbb{A}_1 \to \mathbb{A}_2)}$$

Fig. 8. Type system for LCPCF (CORE)

not contain the fixed-point operator other than those that may flow in via an application; see the definition of $\mathsf{Expr}^{\downarrow}$.

Proposition 3.1. If $x : \mathbb{A} \to \mathbb{A} \vdash e^{\downarrow} : \tau$ then $e^{\downarrow}$ terminates if $x\ a$ also terminates for all $a \in \mathbb{A}$.

Proof. Other than x, the expression $e^{\downarrow}$ cannot refer to any variables from the surrounding context that may contain a fixed-point operator. The type system for the remaining forms is STLC extended with booleans and contracts, so the usual logical relation for termination of STLC applies. By assumption, applications of x terminate, therefore $e^{\downarrow}$ must too. □

These restrictions guarantee that contract code always terminates. If a value is monitored under the contract $c_1 \to c_2\ \&\ f^{\downarrow}$, then each application of $f^{\downarrow}$ terminates because MON-ARR-FLAT guarantees that $f^{\downarrow}$ is applied only to handles. In turn, a handle is either blocked or, according to SYNC-GET, retrieves a value from the synchronization table. Either way the application of a handle to any atomic value does not diverge. Hence, Proposition 3.1 applies.

4 The Metatheory of LCPCF

The LCPCF model satisfies four metatheoretic properties:

(1) **type soundness**, the conventional metatheorem, establishes that the semantics is well-defined and signals only contract errors (Theorem 4.1);
(2) **blame correctness**, which verifies that an error's "server" label is responsible for the value that witnesses violations of a monitored contract (Theorem 4.2);
(3) **complete monitoring**, which guarantees that each static and dynamically created channel of communication between distinct components can be monitored (Theorem 4.3); and

(4) **strong erasure**, which shows that contracts cannot interfere with program evaluation in any way other than signaling a contract violation (Theorem 4.4).

While the design proposal of Dimoulas et al. [17] also satisfies the first three metatheoretic properties, it fails to satisfy the (strong) erasure property.

Additionally, when the lackadaisical semantics discovers a contract violation, the immediate semantics does too, but formalizing the correct and rather subtle variant of this statement would demand a significant undertaking. See the end of this section for an explanation.

Type Soundness. The first property, type soundness, is entirely standard and validates that if evaluation terminates, then it yields a value or signals a contract error.

Theorem 4.1 (Type Soundness). If $\varnothing \vdash e : \mathbb{A}$ and $\langle e, \varnothing\rangle \longmapsto^* \varsigma$ then one of the following holds:

(1) $\varsigma = \langle \vec{s}, a, \sigma\rangle$ $\wr \sigma \vdash s_i$ blocked$\int$ and $\varnothing \vdash a : \mathbb{A}$,
(2) $\varsigma = \langle \mathtt{err}^k_j, \sigma\rangle$,
(3) $\exists \varsigma' . \varsigma \longmapsto \varsigma'$.

Note. The first clause also holds if the model comes with several disjoint atomic types. In that case, the answer is of the same atomic type as the program.

Blame Correctness. The second property, blame correctness, captures the idea that blame assignment should cut down the search space for bugs. This metatheoretic property is directly beneficial to programmers. As Lazarek et al. [36] show, correct blame is a prerequisite for making "blame shifting" work, a process that, through successive refinements of contracts in reaction to blame assignments, allows developers to isolate the component or contract responsible for a bug.

Key to defining blame correctness is the notion of (*component*) *responsibility*. Following Dimoulas et al. [14], responsibility can be defined via an instrumented semantics that tracks the *ownership* of values. Using these annotations, the instrumented semantics tracks values as they cross boundaries. Values accumulate a sequence of labels via these crossings. Every flat contract already comes with an obligation label that identifies the responsible component (boundary). In this context, blame correctness states that obligations and ownerships are in agreement.

Theorem 4.2 (Blame Correctness). Assume e is a well-labeled program. If

$$\langle e, \varnothing\rangle \longmapsto^* \langle A^\sigma[\mathtt{mon}^{k,l}_j\ (\mathtt{flat}^{k'}\ c)\ v], \sigma\rangle$$

or

$$\langle e, \varnothing\rangle \longmapsto^* \langle A^\sigma[\mathtt{mon}^{k,l}_j\ (c_1 \to c_2\ \&^{k'}\ f)\ v], \sigma\rangle$$

then $v = |v'|^{k''}$ (for some v') and $k = k' = k''$.

The colors of the annotation labels in the theorem are significant, explaining which roles they play. While the red k' in $\mathtt{flat}^{k'}\ c$ and $c_1 \to c_2\ \&^{k'}\ f$ are obligation labels, the blue k'' in $|v'|^{k''}$ is an ownership label. Since the three labels are identical, the theorem confirms that it is always the server that owns the value v whose obligations it checks. That is, when this check fails, the server is the legitimate owner and the to-be-blamed party.

For these colored labels to serve as inert annotations, they need to satisfy some conditions. To start with, the given program must be *well labeled*, meaning that the annotations on monitor expressions are consistent with one another. Furthermore, the underlying reduction relation may not change, that is, these colored labels cannot have any influence on evaluation.

Section E contains a complete description of the instrumented semantics together with the judgment for well-labeled expressions, plus a proof sketch.

Complete Monitoring. The complete monitoring property also concerns programming-language pragmatics. While the model makes the static channels of (value) communication explicit (via `mon`), function applications in higher-order code create such channels during execution. Unless the language enables the monitoring of such dynamically created channels, a component could "leak" a bad value and the receiving component may suffer the consequences of such leakage. The contract system should enable developers to attach contracts to such dynamic channels and properly assign blame when it discovers a mismatch between values and such contracts.

Hence the informal phrasing of the complete monitoring property states that all channels can be monitored and that blame assignment remains correct. Technically, complete monitoring strengthens the statement of blame correctness. It demands the *single-ownership policy* from the instrumented semantics for blame correctness [17]. Single-ownership stipulates that values taking part in a primitive reduction step cannot have multiple owners. Since flat-contract checks are the only construct that can eliminate an ownership annotation—by discharging an obligation—type soundness of a semantics satisfying the single-ownership policy guarantees that values flowing across a component boundary must pass through a contract check.

Theorem 4.3 (Complete Monitoring). Type soundness and blame correctness hold for well-labeled programs under the instrumented semantics following the single-ownership policy.

Proof. See Theorem E.1 for a formal restatement of complete monitoring and its proof via progress and preservation [46]. To prove preservation, the type system must be extended to the dynamic syntax (i.e., configurations) in the usual manner. □

Following tradition, the statement of complete monitoring also implies blame correctness. As such, it subsumes Theorem 4.1 and Theorem 4.2 because the single-ownership policy does not affect the semantics other than potentially introducing new stuck expressions.

Given extensive prior work [4, 10, 12, 14, 17, 24, 26, 31, 47] on the semantics of contracts and blame, it may be somewhat surprising that LCPCF satisfies complete monitoring—without having the capability of blaming a contract itself (i.e., the component labeled j). After all, Dimoulas et al. [14] resolved the "blame puzzle of contracts" with the introduction of labels for the contract-defining party. By doing so, their indy semantics could explain how a contract that probes higher-order values may itself violate contracts on these values. The point of the lackadaisical semantics, however, is to check such an interaction only if the main code performs the same one. If it does and if the interaction violates the contract on the higher-order value, the main code gets blamed, not the contract that it imposed on the higher-order value.

Strong Erasure. The fourth property, strong erasure, implies that the evaluation of an error-free program evaluates in the same way, with or without contract monitoring. Given an erasure meta-function $\mathcal{E}[\![\cdot]\!]$ that traverses a program and replaces all monitor expressions with their protected subexpression, stating the theorem is reasonably straightforward.

Theorem 4.4 (Strong Erasure). Suppose that $\varnothing \vdash e : \mathbb{A}$ and $\langle e, \varnothing\rangle \not\longmapsto^* \langle \mathtt{err}_j^k, \sigma\rangle$. In that case, $\langle e, \varnothing\rangle \longmapsto^* \langle \vec{s}, a, \sigma\rangle$ $\lfloor \sigma \vdash s_i \text{ blocked} \rfloor$ if and only if $\langle \mathcal{E}[\![e]\!], \varnothing\rangle \longmapsto^* \langle a, \varnothing\rangle$.

Proof. The proof proceeds by two stuttering simulations, one in each direction. See Section D for the complete formalization. □

Strong erasure mandates both directions, while weak erasure is just the forward direction. The strong version closely matches the intuition that contract code should not be able to introduce computational effects that are not already present in the program. In the literature, Moy et al.

[38]'s design satisfies weak erasure for their effect-handler-based contract system. Strong erasure is crucially dependent on the restriction that contract code is pure and terminating, which is enforced by the type system from Section 3.3.

Relating Lackadaisical Checking to Immediate Checking. The astute reader may now wonder about the precise relationship between LCPCF and conventional (immediate) contract-checking semantics. A formal investigation would require the presentation of the conventional model, formulated in a way that enables a direct comparison. Instead, the remainder of this section provides an informal discussion. See Section B for concrete examples that illustrate the following points.

Intuitively, lackadaisical checking should signal strictly fewer contract violations than the immediate semantics of contract checking because it delays, and potentially never finishes discharging, flat-function contract checks. Strictly speaking, this intuition is incorrect. Consider the following:

(1) Assume that a program comes with a function contract, $c_1 \to c_2 \;\&\; f$, and that this contract inspects its higher-order input (g) at some atomic value (a_1). The immediate-checking strategy may diverge during the exploration of g at a_1. But the program itself may not evaluate $g(a_1)$ before it applies g to an atomic value $a_2 \neq a_1$, where $g(a_2)$ breaks g's contract. In this case, the conventional semantics causes the program to diverge, while the lackadaisical one signals a contract violation.

(2) Consider the possibility that a curried function f consumes two functions. Its contract can explore these functions in one order at some atomic values, while f itself explores them in a different order. Suppose both explorations cause a contract violation. This setup clarifies that the two interpretations of function contracts can signal distinct contract violations.

Equipped with this background, it is now possible to clarify the relationship of the two interpretations with an informal weakening of the original claim. Suppose e is a well-typed program that terminates under both interpretations. If e evaluates to a contract error under lackadaisical checking, then e evaluates to a potentially different contract error under immediate checking.

While the informal restatement explains how lackadaisical contracts delay the discovery of contract violations, it also exposes the brittle nature of this claim. If LCPCF came with exception handlers that can observe (and react to) contract violations, even this weakened claim is too strong.

More generally still, all of these metatheorems are quite brittle, in the sense that adding internally observable effects (e.g., variable assignment, object mutation, exception handlers) can easily invalidate them. Having said that, the properties do provide guidance for the transition from a theoretical model to a practical implementation.

5 Realizing LCPCF

A programming-language model can serve as a blueprint for guiding implementation efforts. Once an implementer meets constraints that conflict with its basic assumption or metaproperties, it becomes necessary to make and justify design decisions. This section illustrates how adding lackadaisical contracts to Racket [29] raises such concerns. To start with, Racket comes without a type system, hence it is impossible to force plain-code contracts to terminate and strong erasure cannot possibly hold. Furthermore, Racket is a higher-order functional language with a rich vocabulary for expressing other effects. Consequently, an implementation effort must also answer questions concerning the interaction between the new contract forms and such linguistic features, e.g. (1) exceptions and exception handlers, (2) assignment statements and structure mutators, and (3) function arguments and results of second order or higher nature.

The choice of Racket as an implementation platform is natural. It comes with a mature higher-order contract system [25] and an easy-to-use system for metaprogramming [9, 22]. The former, combined with concurrency mechanisms, makes up the run-time support for ->1; the latter provides

the necessary compile-time support. The `self/c` contract is macro-expressible and thus does not introduce additional difficulties, i.e. this section mat focus on `->l`.

Although Racket greatly simplifies the addition of lackadaisical contract checking, almost any higher-order language, class-oriented or functional, could serve as a target. See Section A for details on how lackadaisical checking may be implemented outside of Racket.

5.1 Run-Time Support for Lackadaisical Checking

As the formal model shows, the runtime must provide mechanisms for the concurrent evaluation of flat-function contracts and their synchronization with main-program code. If the implementation language comes with threads, it enables a direct translation of the model's essential elements into code. In the case of Racket, expensive threads can be replaced with reasonably cost-effective coroutines [28, 33] based on composable and delimited continuations [30].

Concerning the synchronization of concurrent checks with the main-program code, the model demands a second run-time element, namely the synchronization table. Two pieces of code get access to this table: the proxy, which represents the monitored function inside main-program code, and the handle, which represents the same function inside of contract checks. By encapsulating this table behind a sealed interface, it becomes a separate abstraction with major benefits to the implementation. Specifically, the code for lackadaisical contract checking can be parameterized over the concurrency mechanism. Swapping in a new synchronization implementation, without changing any of the surrounding contract-system code, yields a new implementation. Section 5.5 shows that this abstraction realizes yet another advantage of modularity, namely, it can easily accommodate the implementation of lackadaisical contracts for arbitrary higher-order functions.

`type Proxy` α β `=` $\alpha \to \beta$

A proxy wraps a function and conspires to make output values available to handles. Code that is suspended, waiting on some input–output pair, is resumed once that entry is available.

`type Handle` α β `=` $\alpha \to \beta$

A handle retrieves the results of inputs that have been computed by the proxy. If the desired result is available, it is returned immediately; otherwise, the handle suspends evaluation until the result is available.

`synchronize :` $(\alpha \to \beta) \to$ `(Proxy` α β`, Handle` α β`)`

Given a function, the `synchronize` operation returns both a proxy and a handle of that function, which conspire with one another. The α type must at least support equality, but for efficiency it should also support hashing.

Fig. 9. The synchronization interface

Figure 9 displays this synchronization interface. Its parameters, α and β, indicate that it can deal with functions of arbitrary type, where α must support equality checking. The key export is the `synchronize` function, which produces two results: a `Proxy` and a `Handle`, both of which are type aliases for functions. The main code receives the `Proxy`, whose task it is to deliver the results of applications of `f` to the synchronization API; the contract code receives a `Handle` to retrieve the results. An implementation can thus define and export `synchronize` without revealing any internal details about the synchronization table itself. The rest of this subsection sketches two implementations of the interface: one that uses threads and another one that uses coroutines.

Threads. The most direct implementation of the synchronization interface assumes that `->l` instantiates `synchronize` per monitored function and evaluates each precondition clause in a separate

```
;; type [Proxy A B]  = A → B
;; type [Handle A B] = A → B
;; type [Table A B]  = [Hash A [Or [Channel B] B]]

;; (A → B) → [Proxy A B] [Handle A B]
(define (synchronize f)
  (define table (make-hash))
  (values (make-proxy table f) (make-handle table)))

;; [Table A B] (A → B) → [Proxy A B]
(define (make-proxy table f)
  (λ (x)
    (define y (f x))
    (define ch (hash-ref table x #false))
    (hash-set! table x y)
    (when (channel? ch) (channel-put ch y))
    y))

;; [Table A B] → [Handle A B]
(define (make-handle table)
  (λ (x)
    (match (hash-ref! table x make-channel)
      [(? channel? ch) (channel-get ch)]
      [(? atom? y)     y])))
```

Fig. 10. An idealized implementation via channel-synchronized threads

thread. In all such threads, each depended-upon parameter in `->l` is bound to a handle that reads from the shared table. In the main-program code, the proxy for the monitored function writes results to the shared table.

Figure 10 shows a greatly simplified implementation of thread synchronization via channels. The `synchronize` function returns a proxy and a handle, using an internal hash table to store results and (synchronous) channels. Contract-checking threads and the main program exchange values through these channels:

- When the proxy computes `(f a)`, it checks the hash table to determine whether the entry maps to a channel. If so, the checking thread is waiting for the result value. The proxy satisfies this demand by sending the value along on the channel. Otherwise, it stores the result in the hash table as an atom. This way, every input–output pair observed by the proxy becomes available for contract checks.
- When the handle computes `(f a)`, it uses `(hash-ref! table x make-channel)` to retrieve the cached value. This lookup retrieves the value of `a`, if any. Otherwise, it sets the corresponding entry to a newly created channel and returns this channel.[5]

Although the implementation in Figure 10 conveys the core idea, it makes several simplifications. For instance, more than one contract check may need the result of `(f a)`, hence a single channel does not suffice. The table would instead need to maintain a list of channels for each consumer. Additionally, the actual table of results is a nested mutable hash table because functions in Racket can take multiple arguments or a variable number of arguments. See Table 1 for an example of such a synchronization table for a two-argument function.

Coroutines. When available, cooperative coroutines are a feasible alternative to threads. In LCPCF, the transfer of control from main-program code to contract-checking code (and back) is entirely sequential and deterministic, meaning it does not rely on the preemptive behavior of threads.

As control enters a contract-protected function that monitors `f`, the contract system remembers that the contract may need the results of applications of `f`. Hence, when `(f a)` delivers a result,

[5] Racket locks a hash table during such operations to avoid race conditions.

```
;; type [Proxy A B]  = A → B
;; type [Handle A B] = A → B
;; type [Table A B]  = [Hash A [Or [FunCont B] B]]

;; The ->l contract is expected to install pt around preconditions and postconditions
(define pt (make-continuation-prompt-tag))

;; [Table A B] (A → B) → [Proxy A B]
(define (make-proxy table f)
  (λ (x)
    (define y (f x))
    (define kont (hash-ref table x #false))
    (hash-set! table x y)
    (when (continuation? kont) (kont y))
    y))

;; [Table A B] → [Handle A B]
(define (make-handle table)
  (λ (x)
    (call/comp
      (λ (kont)
        (match (hash-ref! table x kont)
          [(? continuation?) (abort/cc pt)]
          [(? atom? y)       y]))
      pt)))
```

Fig. 11. An idealized implementation via coroutines

control moves to the contract code as needed. The contract code uses the result to confirm an assertion. Once the confirmation succeeds, control resumes execution of the main program. Otherwise, the contract signals a violation. Notice how this description reads more like the cooperation between two coroutines than two communicating threads.

In contrast to the thread-based solution, a coroutine-based solution implements the synchronization API of Figure 9 with a table containing coroutines that directly evaluate contract checks, instead of channels. They are created by running the contract code up until an application of `f`. At each application, the contract creates a coroutine and stores it in the table if no entry is already available. When no more contract code can be evaluated, control transfers to the main code.

Now, if a proxy in main code writes to the synchronization table, and there are suspended coroutines at that entry, it resumes all of these coroutines (in order) before continuing evaluation. This policy faithfully replicates the priority given to contract code in the formal model (as specified by the definition of A^σ in Figure 4).

Symmetrically, if a handle attempts to look up an entry in the synchronization table that is not yet present, it captures a delimited continuation, stores it in the hash table at that entry, and returns control to the prompt that surrounds the contract. That is, instead of spawning a thread for each precondition of the monitored function, `->l` installs a fresh continuation prompt [19] that can serve as the target for capturing and aborting the continuation.

Figure 11 displays an idealized implementation of the coroutine-based solution comparable to the thread-based version of Figure 10 (the implementation of `synchronize` is identical and thus omitted). Here `make-handle` captures a functional continuation with `call/comp`, up to the designated `pt` prompt, and aborts (with `abort/cc`) to that same prompt if there is no value in the table yet. Instead of putting values on a channel, `make-proxy` applies the functional continuation to the newly computed value.

5.2 Compile-Time Support for Lackadaisical Checking

Racket's built-in contract language consists of an extensible set of combinators and a few base forms. Since the `->l` form is intended to present an interface similar to `->i`, the major compile-time concern is ensuring that all uses are syntactically well formed. The existing implementation relies

on the `syntax-parse` system [9], which enforces syntactic constraints in a straightforward manner and synthesizes useful parser-error messages when needed.

A `syntax-parse` macro for `->l` expands into code that turns the coroutine runtime from Section 5.1 into a contract combinator. Technically speaking, the macro compiles the given contract pieces into an *error projection* [24]. These error projections create the combinator and connect it to the underlying blame-assignment system [25].

5.3 Design Decision: Monitoring Functions that Signal Exceptions

In a full-fledged programming language such as Racket, a monitored function may raise an exception. Racket also provides exception handlers, meaning a contract may mask these exceptions. An implementer must make a choice of how to deal with this situation.

```
(provide
  (contract-out
   [error-sum (->l ([x integer?])
                   [res (-> integer? integer?)]
                   #:post (res)
                   (with-handlers ([exn:fail? (λ (xn) #true)])
                     (= (res 2) 0)))]))

(define (error-sum x)
  (λ (y)
    (error "bad number" (+ x y))))

---

((error-sum 0) 2)
```

Fig. 12. Contracts and exceptions

Figure 12 displays a short program that involves exceptions. The postcondition checks, under an exception handler, that `res` has a root at 2. When the snippet below the line is evaluated using Racket's conventional function contract (i.e., `->i` instead of `->l`), the contract evaluates `(res 2)` in order to compare the result to `0`, but rather than returning a value, `res` raises an exception. The exception is caught by a handler function that always returns `true`, which means the contract check succeeds. At this point, control transfers to the main-program code which applies the output function to 2. For this second use, there is no exception handler set up. Hence, the program as a whole signals an exception.

According to the lackadaisical semantics, `->l` does *not* evaluate the output function twice. Instead, the model says that the proxy communicates the result to the contract via the synchronization mechanism. Some reflection suggests at least two implementation alternatives:

(1) An implementation could enter a representation of the exception into the underlying table. Retrieving this distinct piece of data from the table would cause the contract system to re-raise the exception in the context of the check.[6]
The rationale for this choice treats the exception as another kind of output of functions. Storing it in the table leaves it to the contract system to deal with it, possibly as a way of communicating values to a remote place in the code.[7]

[6]This semantics is similar to, and inspired by, Lazy Racket [2]. In this language, thunks "cache" exceptions to be reraised whenever they are forced so that delayed computations properly interact with exception handlers.
[7]While communicating results through an exception is considered an antipattern, it is common in practice.

(2) Alternatively, an implementation could simply not record any result whatsoever in the synchronization table. The contract check would remain suspended indefinitely.
The rationale for this choice is that the precondition expression should simply witness successful input–output pairs of `f` exhibited by the main code, and signaling an exception does *not* count as successfully returning a result.

Neither the fundamental assumption of the model—contracts should not interact with higher-order values directly—nor its metatheoretic properties distinguish between these two design alternatives. Programmers, however, could argue about the pragmatics of each option [13]. For now, the implementation of `->l` realizes the second alternative, rationalizing that something went fundamentally wrong if a monitored function raises an exception, and that it should be the task of the main code to deal with the problem. Time will tell whether programmers accept this decision.

5.4 Design Decision: Monitoring Functions with Mutation Side Effects

A full-fledged programming language also supports variable assignment and mutable data, which implies that a monitored function can mutate the program's state. Hence, an implementation of `->l` must answer the question of how to deal with such side effects.

```
(provide
  (contract-out
   [at-two-twice (->l ([f (-> integer? integer?)])
                      #:pre (f) (= (f 2) 0)
                      [res integer?])]))

(define (at-two-twice f)
  (f 2)
  (f 2))
```

```
(define incr!
  (let ([count -1])
    (λ (x)
      (set! count (+ count 1))
      count)))

(at-two-twice incr!)
```

Fig. 13. Contracts and mutation

The example in Figure 13 exhibits a particularly contorted side-effect behavior. The `at-two-twice` function calls `f` twice, discarding the first result; its contract checks whether `f` has a root at 2. Below the line is a call to `at-two-twice` with a function that increments a counter. For this particular example, an evaluation using an `->i` contract would succeed, and the program would produce 2.

Once again, the model fails to guide the implementation of `->l`, other than allowing the possibility that *all* results from monitored functions are somehow communicated from the proxy to the handle. At least three implementation alternatives come to mind easily:

(1) The proxy ignores the attempt to send a different result for the same input to the handle.
(2) The proxy could communicate a second return result to the handle, and the handle could trigger an actual re-evaluation of the contract check if the values differ. An implementation using coroutines could more easily realize this behavior than a thread-based one.

(3) The proxy could signal an error if a second application of a synchronized function to the same value returns a different result.

The enumeration leaves a rationalization of these alternatives to the astute reader, because there are just too many. Indeed, an imaginative implementer could probably come up with additional alternatives involving first-class stores or transactions.

The prototype implementation makes the assumption that `->l` monitors higher-order values, ensuring they map equal inputs to equal outputs (i.e., `equal?` in Racket). The function is otherwise permitted to perform effects. These effects will not be repeated when the proxy is invoked, ensuring that no duplication of effects occurs. Concretely, the packaged `->l` enforces the third option by raising an error if it detects a violation of the equal-input-equal-output constraint. If a handle attempts to write a value into the synchronization table and notices that the entry already has a value, then the existing entry must be equal to the new one, otherwise an error is raised. Overall this choice makes it easy to explain the meaning of pre/post-conditions, including a generalization to arbitrary higher-order values.

5.5 Design Decision: Monitoring Higher-Order Functions

For simplicity, the model explains only how a contract may interact with first-*order* functions, but contemporary programming languages come with higher-order values: first-*class* functions, objects, and so on. A general implementation should lift this constraint.

At first blush, this simple statement appears to pose an insurmountable challenge given that functions are *opaque* values without a useful equality predicate. In LCPCF, the table of results relies on the first-order nature of the arguments to monitored functions so that the handle can retrieve values. Hence, extending lackadaisical contract checking to arbitrary higher-order values calls for a technique that can construct first-order representations of higher-order functions.

Given such a representation of functions, an implementation can lift the constraint by turning monitored functions automatically into *transparent* ones, where equality is defined by equality of the first-order representations.

$$
\begin{aligned}
\langle A^{\sigma}[\mathtt{mon}^{k,l}_{j}\ (c_1 \to c_2\ \&\ f)\ v], \sigma\rangle \longmapsto \langle\, & \mathtt{assert}^{k}_{j}\ (f\ (\lambda x.(\alpha, x) \triangleright)) \Vdash \\
& A^{\sigma}[\lambda x.\mathtt{let}\ x' = \texttt{transparent-procedure}\ x\ \mathtt{in} \\
& \qquad (\alpha, \texttt{dict}{\to}\texttt{list}\ x'\ ) \triangleleft \mathtt{mon}^{k,l}_{j}\ c_2\ (v\ (\mathtt{mon}^{l,k}_{j}\ c_1\ x'))], \\
& \sigma[\alpha \leftarrow \varnothing]\rangle \\
& \lceil \alpha \notin \mathrm{dom}(\sigma) \rfloor
\end{aligned}
$$

Fig. 14. Updated MON-ARR-FLAT rule (informal)

This approach demands two changes to the MON-ARR-FLAT rule from Section 3, which are shown in Figure 14. Instead of x, the proxy supplies a transparent version of the function x, called x', to v. As v runs, x' incrementally builds a first-order description of the function. Every path into this data structure represents an exploration of x in v. The `dict→list` metafunction extracts the first-order representation of x' and enables decidable lookups in the synchronization table.

An Aside on Transparent Procedures. This extension relies on an implementation of transparent procedures. Explaining how transparent procedures work requires a brief detour into the world of higher-order memoization [18, 34].

```
> (define inc (transparent-procedure (λ (x) (+ x 1))))
> inc
(procedure)
> (inc 1)
2
> (inc 2)
3
> (inc 3)
4
> inc
(procedure [1 . 2] [2 . 3] [3 . 4])
> (dict->list inc)
'([1 . 2] [2 . 3] [3 . 4])
```

Fig. 15. REPL interaction for transparent procedures

A separate Racket library provides an implementation of transparent procedures. Figure 15 displays an example REPL interaction that demonstrates how this library works. Given a procedure, `transparent-procedure` returns a transparent equivalent that builds a *partial graph* representing its extensional behavior. Printing a transparent procedure at the REPL shows its graph.

Think of `transparent-procedure` as similar to constructors of other mutable objects. Application of a transparent procedure mutates the underlying data structure based on its inputs and outputs. Applying `dict->list` to a transparent procedure returns an association list summarizing the observed extensional behavior.

```
> (define map* (transparent-procedure map))
> (map* (λ (x) (+ x 1)) '(1 2 3))
'(2 3 4)
> (map* (λ (x) (- x 1)) '(4 5 6))
'(3 4 5)
> map*
(procedure
  [(procedure [1 . 2] [2 . 3] [3 . 4]) '(1 2 3) . '(2 3 4)]
  [(procedure [4 . 3] [5 . 4] [6 . 5]) '(4 5 6) . '(3 4 5)])
```

Fig. 16. REPL interaction for higher-order transparent procedures

Higher-order functions can nest arbitrarily, meaning the memoizing data representation has deep structure. Furthermore, the creation of this representation demands that higher-order arguments and results themselves automatically become transparent too. What the interactions in Figure 16 show is that `map*`'s transparency "infects" its higher-order inputs. Both the add-one function and subtract-one function become transparent, and hence their first-order representations become part of `map*`'s representation. More precisely, the resulting first-order representation is a tree of behaviors. Each edge of the tree is either a piece of first-order data or another tree of behaviors.

Figure 17 shows an example of how `->l`, using transparent procedures, can deal with contracts that go beyond interactions with first-order functions. Consider the application of `apply-to-fn` below the line. First, the precondition is checked and `f` is applied to the add-one function (highlighted). At this point, the contract check is suspended pending an identical interaction from the main program. When the body of `apply-to-fn` is evaluated, a transparent version of `f` is applied to

```
(provide
  (contract-out
   [apply-to-fn (->l ([f (-> (-> integer? integer?) integer?)])
                     #:pre (f) (= (f (λ (x) (+ x 1))) 0)
                     [res integer?])]))

(define (apply-to-fn f)
  (f (λ (x) (+ (* x 2) 1))))

---

(apply-to-fn (λ (f) (f 0)))
```

Fig. 17. Contracts and arbitrarily higher-order interactions

a function that is extensionally equivalent to the add-one function on `0`. After the body is evaluated, the transparent version of `f` is `(procedure [(procedure [0 . 1]) . 1])`. The contract check can proceed because the add-one function also maps `0` to `1`. Finally, the contract signals an error because the returned value `1` is not zero.

An Aside on Parallel Lookup for Higher-Order Values. A subtle issue arises when mixing lackadaisical checking with higher-order values. Consider Figure 18. The `apply-to-first` function takes in two higher-order values, `f` and `g`, and then computes `f` applied to the subtract-one function plus `(f g)`.

```
(provide
  (contract-out
   [apply-to-first (->l ([f (-> (-> integer? integer?) integer?)]
                         [g (-> integer? integer?)])
                        #:pre (f g) (= (f g) 0)
                        [res integer?])]))

(define (apply-to-first f g)
  (+ (f (λ (x) (- x 1))) (f g)))

---

(apply-to-first (λ (g) (if (= (g 1) 0) (g 3) (g 2)))
                (λ (x) (+ x 1)))
```

Fig. 18. A contract that requires parallel lookup

Similar to Figure 17, the contract checks for a root of `f`. Here, however, the argument is not a constant lambda but the input `g`. Intuitively, the evaluation of `(f g)` at the contract level should resume after the function itself computes `(f g)` in its body, completing the contract check. Achieving this behavior in practice requires some care. After the body of `apply-to-first` finishes, the contract system has recorded the extensional representations of `f` and `g` as

$f =$

g	out
$3 \mapsto 2, 1 \mapsto 0$	2
$2 \mapsto 3, 1 \mapsto 2$	3

$g =$

x	out
1	2
2	3

Note that the first row for `f` corresponds to the application with the subtract-one function, while the second row corresponds to `(f g)` itself, where `g` is the add-one function. When trying to retrieve the value for `(f g)` at the contract level, there are two possible implementation strategies.

The naive strategy attempts to look up the value of `(f g)` sequentially, proceeding from top to bottom, and stops if any suspension occurs. At the first row, it attempts to determine if `g` maps 3 to 2. Because `g` has no entry for 3, it suspends at this point, waiting for a value that will never arrive in this particular example. This contract check is left unresolved although the next row matches.

A less naive strategy performs the lookup "in parallel." In other words, if the first lookup suspends, the contract system continues its search until every entry either fails or is suspended. There is no need for physical parallelism to implement this approach, though. A sophisticated implementation of coroutining evaluation ensures that the lookup behaves in this way. The actual `->l` implementation uses a coroutining strategy.

Summary. The implementation of `->l` copes with three major issues that a functional and higher-order language poses: exceptions, mutation, and higher-order functions. As the three subsections demonstrate, each solution requires decisions concerning the visibility of effects, the propagation of effects, and the use of space and time. At the moment these design decisions seem reasonable. However, they may need to be revisited in the future due to user feedback.

6 Related Work

The first presentation of higher-order contracts dates back over two decades [26]. In the intervening years, researchers have explored different aspects of the semantics of contracts. Broadly speaking, two lines of work are most relevant here: contract semantics that defer some checks (Section 6.1) and effectful contract checking (Section 6.2).

6.1 Inspirational Work on Deferred Checking

Contract semantics that defer run-time checks have a long history. In the call-by-need setting, Chitil and collaborators spent several years exploring lazy assertions for Haskell [5–8]. In the call-by-value setting, researchers from the Racket community developed deferred contract checking for immutable data structures [27] and for general-purpose languages [12]. All of these pieces of work start from the observation that effects in contract checking can interfere with program execution, but none come with a formal model satisfying all the properties from Section 4.

Lazy Assertions for Call-by-need Languages. The most closely related work, by Chitil and Huch [7], proposes *concurrent assertion checking* for Haskell. In a functional call-by-need language, assertions should preserve the termination behavior of a program, that is, if a program terminates, then it should also terminate with assertions enabled. Concurrent checking demands that interactions with higher-order values are blocked until demanded by the main program.

The primary difference between lackadaisical checking and lazy assertion semantics concerns the lack of a synchronization mechanism. Consequently, concurrent assertion checking is unstable; seemingly benign program transformations can mask assertion failures. Suppose the main program computes `(+ (f 0) (f 1))` and imposes the assertion `(and (zero? (f 0)) (zero? (f 1)))`. Here is how the two approaches work out:

(1) Under the *lazy assertion semantics*, if `(zero? (f 0))` is `true` and `(zero? (f 1))` is `false`, then this program signals an error. If, however, the main program code is `(+ (f 1) (f 0))`, then it does not signal a violation, because `(f 1)` is evaluated while the assertion check is blocked on `(zero? (f 0))`. As a result, the contract check will remain indefinitely blocked on `(zero? (f 1))` and the assertion failure goes undetected.
(2) Under the *lackadaisical contract semantics*, both variants signal an error. The second case looks up `(f 1)` via the synchronization API and raises an error, as expected.

Failure to report a violation of a specification, simply because certain expressions are evaluated in a different order, is highly problematic. A developer can no longer be certain that a program satisfies the contract if innocuous reorderings can mask errors. While additional features, such as a nondeterministic and operation, may mitigate this concern, they do not eliminate it entirely. Storing previously computed results via the synchronization API, in case the contract demands it later, is critical to ensuring that contracts are fully enforced.

Lazy Contracts for Call-by-value Languages. Findler et al. [27]'s lazy contracts for immutable data structures inspired the total-order example from Section 2.2. Their paper demonstrates how immediate contracts on data structures may cause an unreasonable consumption of resources in time and space. Generally speaking, contracts should preserve the asymptotic behavior of a program. If a program consumes some resource, then it should consume an asymptotically similar amount of that resource with contracts enabled modulo constant factors.

Racket's original contract system fails to satisfy this requirement; indeed immediate contract checking may cause asymptotically worse performance. The practical work of Findler et al. [27] fixes one important aspect of this system: contracts on structures. Consider a contract for the binary-search-tree property: it should not have to repeatedly traverse the entire tree to guarantee the property. Although the introduction of memoized data-structure contracts eliminates this particular problem, it does not solve the general concern raised in this paper; in particular, it does not address interactions between contract code and higher-order values.

Lackadaisical contract checking solves this problem not only for first-order data structures but also for higher-order values. There is, of course, no free lunch. The asymptotic gains described in Section 2.2 come at the cost of additional space consumed by the synchronization table and retrieval time for looking up entries in the table. Just as in ordinary memoization, time is traded for space. However, the consumption of time is directly proportional to the exploration of higher-order values in main-program code.

Meaning Preservation and Completeness. Degen et al. [11] describe two properties, meaning preservation and completeness (not to be confused with complete monitoring). The authors show that no contract checking strategy in a call-by-need setting can satisfy both properties, but that the usual immediate contract checking strategy does satisfy both properties in a call-by-value language. Because lackadaisical contract checking also occurs in a call-by-value setting, it satisfies both properties just as immediate checking does.

Shy Contract Satisfaction. Dimoulas and Felleisen [12] tackle the problem of what it means for a function to *satisfy* a contract. In the process, they propose a notion of *shy contract satisfaction* derived from the ideas behind observational equivalence. Like lackadaisical checking, shy checking limits what a contract can explore of its higher-order input(s). But, this specification of contract satisfaction is not computable. Indeed, it is not clear how to create a computable procedure for *checking* shy satisfaction with an algorithm.

We conjecture, though, that the lackadaisical contract semantics is a conservative approximation of shy satisfaction. To be precise, Dimoulas and Felleisen [12] describe two variants of shy contract satisfaction: a *closed* version, which limits higher-order exploration to the current main-program context, and an *open* version, which limits higher-order exploration to any possible program context. We conjecture that lackadaisical contract checking is sound with respect to the closed variant. An investigation of this relationship is left as future work.

6.2 Higher-Order Contracts and Effects

Another line of work pays close attention to the interaction between higher-order contracts and computational effects. In general, this research seeks to exploit contract-level effects in order to check some property, as opposed to restricting the semantics of contracts in order to guarantee an erasure theorem.

Effectful Software Contracts. Moy et al. [38] propose a system whose contract code and main code can perform effects without interfering with one another. Their semantics employs a two-level effect-handler system to isolate the influence of effects at each level. The goals of their work differ in that the effect-handler contract language is explicitly effectful, while the contract language for LCPCF is pure and terminating. Consequently, Theorem 4.4 is stronger than their erasure theorem.

Moreover, their semantics satisfies erasure in a rather unexpected way. Effects performed by higher-order values at the contract level are "captured" by contract effect handlers. An effect that was previously interpreted by a main-program handler would, in contract code, be serviced by a completely different contract-code handler. For example, if contract code applied a main-program function that performed mutation, it could mutate contract-local state instead of main-program state. Alternatively, if the contract code had no handler installed for mutable state, then the code would simply raise an exception and fail. The literature on lexical effect handlers [3, 48] considers this effect-capturing behavior as undesirable, making an effort to explicitly avoid it. Combining effect-handler semantics with lackadaisical checking could eliminate effect capture, meaning the two ideas are possibly synergetic and not in opposition.

Trace Contracts. Trace contracts [39] allow programmers to specify properties over multiple calls to possibly distinct functions or methods. For instance, given a function that counts up by incrementing a piece of internal mutable state, a trace contract can ensure that the function returns an increasing sequence of values over time. It does so by collecting all return values in a data structure called a trace, and then evaluating a user-defined predicate over this structure. A trace contract can also express the total-order example shown in Figure 2, but the encoding is rather cumbersome. The contract must manually intercept and organize results from comparator calls, without direct access to the comparator itself. By contrast, the contract from Figure 2 is a direct translation of the mathematical definition of a total order. For lackadaisical checking, the synchronization table itself plays the role of the trace data structure, and collects all the information needed to validate the contract. Trace contracts permit a custom representation of the trace data structure for more efficient evaluation of the contract. A similar idea could be developed for synchronization tables, reducing the cost of lackadaisical contract checking.

7 From Theory Towards Practice

Contracts play a unique role in the context of a larger software system. Traditionally, a contract is code written in the host language and executed at run time; it is therefore *part* of the system. At the same time, a contract is also a specification *of* a system; it codifies and enforces the promises and obligations that components make to one another.

This dual nature of contracts is a source of tension. As a specification, a contract should not unduly interfere with a program's evaluation, say, by diverging when the program would otherwise terminate or by duplicating observable effects. As code, contracts ordinarily have unfettered access to the larger system. Higher-order contracts exacerbate this tension by permitting the exploration of parts of the main-program code that would not otherwise be evaluated.

The lackadaisical semantics of higher-order contracts attempts to restore some balance to these elements. Without unreasonably restricting the expressive power of contracts, it guarantees

that contracts cannot perform effects beyond what the program would have performed anyway. Lackadaisical checking guarantees this property by delaying higher-order interactions in contract code until the main-program code itself demands the same interactions. Hence, evaluation bounces back and forth between contract code and the main-program code in a cooperative fashion.

Synchronizing the two parts effectively is key to the model and the prototype implementation. Its essential element is the synchronized function, which divides a memoization table in half: the handle (or reading end) is given to the contract and the proxy (or writing end) is given to the main program. These pieces coordinate the transfer of control and values between contract and program proper. The synchronization mechanism is a modular abstraction that can be modified in several ways enhancing contract checking without extensive changes to the contract system itself.

While the LCPCF model and its implementation cover the functional core of Racket, they also raise several interesting questions concerning the theory, implementation, and pragmatics of `->l`:

(1) The implementation section presents the chosen alternatives as *fait accompli*, but a thorough design exploration should assess the different alternatives. From the perspective of programming language pragmatics, the question is how each alternative affects the working developer's efforts in different situations.
(2) These alternatives come with different constant-factor overheads. And, as software developers learned over the past few decades, constant factors matter. Hence, the alternative implementations deserve a rigorous performance evaluation. Existing contract benchmarks do not exercise higher-order interactions in the ways needed for lackadaisical checking to matter. The task of finding, extracting, and describing a new benchmark suite that is relevant for lackadaisical contract checking is important, and substantial, future work.
(3) Section 4 explains, informally, the relationship between lackadaisical and immediate checking. An exhaustive theoretical analysis would require a comparable model for immediate contract checking and theorems that capture the many nuances that relate the two semantics. Determining exactly how to formalize these subtleties will be a challenge.
(4) Dependent-function contracts are an important feature of contract systems [25]. Adding dependencies to `->l` contracts poses both theoretical and practical challenges concerning how dependent evaluations interact with the synchronization mechanism.
(5) The `self/c` abstraction over `->l` is a domain-specific contract in the context of the lackadaisical semantics. As prior work illustrates [40], separating contracts from the domain-specific notation is highly beneficial. Once a clean separation is accomplished, additional abstractions often come to mind.

In sum, while the model, its metatheory, and the prototype of lackadaisical contract checking provide a solid foundation, they also sketch out a plan for significant future research.

Acknowledgments

This work was partially supported by NSF grants SHF 2116372 and 2315884. The authors would like to thank Christos Dimoulas (for insightful discussions), Jack Czenszak (for comments on early drafts); and the anonymous reviewers (for challenging feedback).

Disclosure of Use of AI in Research

The authors did not use AI in any capacity for the research or writing of this paper.

Data Availability Statement

The artifact is available at: https://zenodo.org/records/23330580.

References

[1] Joe Armstrong. 2007. A History of Erlang. In *History of Programming Languages (HOPL)*. doi:10.1145/1238844.1238850

[2] Eli Barzilay and John Clements. 2005. Laziness Without all the Hard Work: Combining Lazy and Strict Languages for Teaching. In *Functional and Declarative Programming in Education (FDPE)*. doi:10.1145/1085114.1085118

[3] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. In *Principles of Programming Languages (POPL)*. doi:10.1145/3371116

[4] Matthias Blume and David McAllester. 2006. Sound and Complete Models of Contracts. *Journal of Functional Programming (JFP)* (2006). doi:10.1017/S0956796806005971

[5] Olaf Chitil. 2011. A Semantics for Lazy Assertions. In *Partial Evaluation and Program Manipulation (PEPM)*. doi:10.1145/1929501.1929527

[6] Olaf Chitil and Frank Huch. 2006. A Pattern Logic for Prompt Lazy Assertions in Haskell. In *Implementation and Application of Functional Languages (IFL)*. doi:10.1007/978-3-540-74130-5_8

[7] Olaf Chitil and Frank Huch. 2007. Monadic, Prompt Lazy Assertions in Haskell. In *Asian Symposium on Programming Languages and Systems (APLAS)*. doi:10.1007/978-3-540-76637-7_4

[8] Olaf Chitil, Dan McNeill, and Colin Runciman. 2004. Lazy Assertions. In *Implementation and Application of Functional Languages (IFL)*. doi:10.1007/978-3-540-27861-0_1

[9] Ryan Culpepper. 2012. Fortifying Macros. *Journal of Functional Programming (JFP)* (2012). doi:10.1017/s0956796812000275

[10] Markus Degen, Peter Thiemann, and Stefan Wehr. 2009. True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). In *Arbeitstagung Programmiersprachen (ATPS)*.

[11] Markus Degen, Peter Thiemann, and Stefan Wehr. 2012. The Interaction of Contracts and Laziness. In *Partial Evaluation and Program Manipulation (PEPM)*. doi:10.1145/2103746.2103766

[12] Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-Order World. *Transactions on Programming Languages and Systems (TOPLAS)* (2011). doi:10.1145/2039346.2039348

[13] Christos Dimoulas and Matthias Felleisen. 2025. The Rational Programmer: Investigating Programming Language Pragmatics. *Communications of the ACM (CACM)* 68, 7 (June 2025), 120–130. doi:10.1145/3708981

[14] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *Principles of Programming Languages (POPL)*. doi:10.1145/1926385.1926410

[15] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). In *International Conference on Functional Programming (ICFP)*. doi:10.1145/2951913.2951930

[16] Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. 2009. Future Contracts. In *Principles and Practice of Declarative Programming (PPDP)*. doi:10.1145/1599410.1599435

[17] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*. doi:10.1007/978-3-642-28869-2_11

[18] Conal Elliott. 2008. MemoTrie: Trie-Based Memo Functions. https://hackage.haskell.org/package/MemoTrie.

[19] Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Principles of Programming Languages (POPL)*. doi:10.1145/73560.73576

[20] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming (Sci. Comput. Program.)* (1991). doi:10.1016/0167-6423(91)90036-W

[21] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

[22] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Communications of the ACM (CACM)* (2018). doi:10.1145/3127323

[23] Matthias Felleisen and Daniel P. Friedman. 1987. Control Operators, the SECD-Machine, and the λ-Calculus. In *Formal Description of Programming Concepts*.

[24] Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *Functional and Logic Programming (FLP)*. doi:10.1007/11737414_16

[25] Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for Object-Oriented Languages. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/504311.504283

[26] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *International Conference on Functional Programming (ICFP)*. doi:10.1145/581478.581484

[27] Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. 2008. Lazy Contract Checking For Immutable Data Structures. In *Implementation and Application of Functional Languages (IFL)*. doi:10.1007/978-3-540-85373-2_7

[28] Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. 2019. Rebuilding Racket on Chez Scheme (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 78 (July 2019). doi:10.1145/3341642

[29] Matthew Flatt and PLT. 2010–2026. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

[30] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding Delimited and Composable Control to a Production Programming Environment. In *International Conference on Functional Programming (ICFP).* doi:10.1145/1291151.1291178

[31] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Principles of Programming Languages (POPL).* doi:10.1145/1706299.1706341

[32] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-Parametric Polymorphic Contracts. In *Dynamic Languages Symposium (DLS).* doi:10.1145/1297081.1297089

[33] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. 1990. Representing Control in the Presence of First-class Continuations. In *Programming Language Design and Implementation (PLDI).* 66–77. doi:10.1145/93542.93554

[34] Ralf Hinze. 2000. Memo Functions, Polytypically!. In *Workshop on Generic Programming (WGP).*

[35] Oleg Kiselyov. 2012. Delimited Control in OCaml, Abstractly and Concretely. *Theoretical Computer Science (TCS)* 435 (2012), 56–76. doi:10.1016/j.tcs.2012.02.025

[36] Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. 2020. Does Blame Shifting Work?. In *Principles of Programming Languages (POPL).* doi:10.1145/3371133

[37] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* (1992). doi:10.1109/2.161279

[38] Cameron Moy, Christos Dimoulas, and Matthias Felleisen. 2024. Effectful Software Contracts. In *Principles of Programming Languages (POPL).* doi:10.1145/3632930

[39] Cameron Moy and Matthias Felleisen. 2023. Trace Contracts. *Journal of Functional Programming (JFP)* (2023). doi:10.1017/S0956796823000096

[40] Cameron Moy, Ryan Jung, and Matthias Felleisen. 2025. Contract Systems Need Domain-Specific Notations. In *European Conference on Object-Oriented Programming (ECOOP).* doi:10.4230/LIPIcs.ECOOP.2025.42

[41] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing Webassembly With Effect Handlers. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* doi:10.1145/3622814

[42] Gordon Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science (TCS)* (1977). doi:10.1016/0304-3975(77)90044-5

[43] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *International Conference on Functional Programming (ICFP).* 317–328. doi:10.1145/1596550.1596596

[44] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Programming Language Design and Implementation (PLDI).* doi:10.1145/3453483.3454039

[45] Cameron Swords. 2019. *A Unified Characterization of Runtime Verification Systems as Patterns of Communication.* Ph. D. Dissertation. Indiana University.

[46] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation (InfoComp)* (1994). doi:10.1006/inco.1994.1093

[47] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell. In *Principles of Programming Languages (POPL).* doi:10.1145/1480881.1480889

[48] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. In *Principles of Programming Languages (POPL).* doi:10.1145/3290318

Lackadaisical Contracts for Higher-Order Functions

Cameron Moy, Anfisa Bogdanenko, Matthias Felleisen

Appendices

A General Implementation Hints 29
B Comparing Lackadaisical and Immediate Contract Checking 30
C Model Details 31
D Erasure 32
D.1 Proof of Erasure: Left to Right 33
D.2 Proof of Erasure: Right to Left 35
D.3 Proof of Erasure: Common Lemmas 38
E Complete Monitoring 40

A General Implementation Hints

The adaption of lackadaisical contract checking to another programming language requires support for concurrency. As explained in Section 5.1, lackadaisical checking is compatible with both thread-based and coroutine-based concurrency.

Languages with threads either have built-in support for channels, or channels may be simulated in a user-space library.[8] One downside of this solution concerns the expense of spinning up a thread. Depending on the language, creating a new thread is typically fairly expensive.[9] Since programs can easily generate tens of thousands of contract checks, a naive thread-based implementation would incur a constant but substantial run-time overhead. Furthermore, thread-based concurrency lacks predictability. The run-time scheduler has free rein over how threads interleave, leading to the possibility that contract checks get reordered and errors are signaled in a possibly nondeterministic order.

The Racket implementation of `->l` uses coroutines based on delimited continuations. Adapting this solution to other programming languages poses a challenge. To date, most languages lack support for (delimited) continuations. Only a few, such as OCaml [35, 44] and Scala [43], implement them properly, and some, like JavaScript, implement them via global program transformations [41].

[8] The solution also readily generalizes to parallel contract checking [16, 45].

[9] Languages for the BEAM, such as Erlang or Elixir, are counterexamples. The underlying virtual machine is explicitly designed to make thread creation cheap [1].

B Comparing Lackadaisical and Immediate Contract Checking

As mentioned in Section 4, a comparison of lackadaisical and immediate contract checking defies a simple theoretical characterization. The following three examples illustrate the abstract arguments of Section 4. They hopefully bring across the difficulties of setting up a uniform table of differences.

The example programs are written in LCPCF syntax extended with numbers and basic numeric primitives. Additionally, `nat?` is a flat contract on natural numbers. In all three examples, a higher-order function is protected by a contract whose domain is specified by a flat-function contract.

For the first example, consider a client that supplies a squaring function:

```
((mon_j^{k,l} ((nat? → nat? & (λg.(== (g -1) (g 1)))) → nat?)
              (λf.(+ (f 1) (f 3))))
 (λn.(* n n)))
```

Under immediate contract checking, `λg.(== (g -1) (g 1))` is applied when the squaring function is passed to the protected function. The call `(g -1)` violates the domain contract `nat?`, so evaluation signals a contract violation that blames the contract itself. Under lackadaisical contract checking, the call `(g -1)` is suspended instead. Since the protected function never applies `f` to `-1`, the check never resumes, and the program terminates with `10` without signaling a contract violation.

The second example program uses the diverging term Ω to illustrate how a "nontermination" effect differentiates the two semantics:

```
((mon_j^{k,l} ((nat? → nat? & (λg.(== (g 0) 0))) → nat?)
              (λf.(+ (f 1) (f 3))))
 (λn.(if (== n 0) Ω (- n 2))))
```

Under immediate contract checking, the predicate eagerly evaluates `(g 0)`, which diverges. Lackadaisical contract checking suspends `(g 0)` instead, and since the program never evaluates this call, the divergence is avoided. The protected function applies `f` to `1`, which returns `-1`, a value that violates `f`'s range contract, and thus signals a contract violation that blames the client.

The third and final example deals with a client that supplies a successor function:

```
((mon_j^{k,l} ((nat? → nat? & (λg.(== (g 0) 0))) → nat?)
              (λf.(+ (f -1) (f 3))))
 (λn.(+ n 1)))
```

Under immediate contract checking, the predicate evaluates `(g 0)` to `1`. Since `0` is not a root of the successor function that the client supplied, evaluation signals a contract violation that blames the client. Under lackadaisical contract checking, `(g 0)` stays suspended, so this violation is never observed. Instead, the protected function applies `f` to `-1`, which violates the domain contract `nat?` of `f`, so evaluation signals a different contract violation that blames the server.

Table 2. A succinct summary of the behavior of the three example programs under the two semantics

	Outcome under immediate	Outcome under lackadaisical
Example 1	Violation that blames the contract	`10`
Example 2	Diverges	Violation that blames the client
Example 3	Violation that blames the client	Violation that blames the server

Table 2 presents a comparison table for the three example programs. A glance at the table demonstrates that the relationship between the two contract semantics is non-uniform.

C Model Details

$\boxed{\text{CONTRACT (TYPE)}}$ extends CONTRACT (DYNAMIC)

$\text{Store}^{\tau} \ni \Psi \stackrel{\text{def}}{=} \text{Addr} \xrightarrow{\text{fin}} \text{Type}$

Fig. 19. Type Syntax for LCPCF (DYNAMIC)

$\boxed{\Psi \vdash \langle \vec{t}, e, \sigma \rangle}$ for LCPCF (DYNAMIC)

$$\frac{\Psi \vdash t_i \quad \ldots \quad \varnothing, \Psi \vdash e : \tau \quad \Psi \vdash \sigma}{\Psi \vdash \langle \vec{t}, e, \sigma \rangle}$$

$\boxed{\Psi \vdash t}$ for LCPCF (DYNAMIC)

$$\frac{\varnothing, \Psi \vdash e : \texttt{Bool}}{\Psi \vdash \texttt{assert}^k_j\ e}$$

$\boxed{\Gamma, \Psi \vdash e : \tau}$ for LCPCF (DYNAMIC) extends $\Gamma \vdash e : \tau$

$$\frac{\Gamma, \Psi \vdash v : \mathbb{A} \qquad \Gamma, \Psi \vdash e : \mathbb{A}' \qquad \Psi(\alpha) = \mathbb{A} \to \mathbb{A}'}{\Gamma, \Psi \vdash (\alpha, v) \lhd e : \mathbb{A}'} \qquad \frac{\Gamma, \Psi \vdash v : \mathbb{A} \qquad \Psi(\alpha) = \mathbb{A} \to \mathbb{A}'}{\Gamma, \Psi \vdash (\alpha, v) \rhd : \mathbb{A}'} \qquad \frac{}{\Gamma, \Psi \vdash \texttt{err}^k_j : \tau}$$

$\boxed{\Psi \vdash \sigma}$ for LCPCF (DYNAMIC)

$$\frac{\forall (\alpha, \mathbb{A} \to \mathbb{A}') \in \sigma, \forall (v, v') \in \sigma(\alpha)\ .\ \varnothing, \Psi \vdash v : \mathbb{A} \wedge \varnothing, \Psi \vdash v' : \mathbb{A}'}{\Psi \vdash \sigma}$$

Fig. 20. Typing Rules for LCPCF (DYNAMIC)

D Erasure

Lackadaisical contracts do not interfere with program evaluation except when signaling contract errors. The LCPCF model satisfies strong erasure (Theorem 4.4): When a program runs to completion without error, erasing the contracts does not change the outcome and vice versa.

$\boxed{\mathcal{E}[\![\cdot]\!] : \text{Expr} \to \text{Expr}}$ for LCPCF (CORE)

$$\begin{aligned}
\mathcal{E}[\![x]\!] &\stackrel{\text{def}}{=} x \\
\mathcal{E}[\![a]\!] &\stackrel{\text{def}}{=} a \\
\mathcal{E}[\![\texttt{if } e_g\ e_t\ e_f]\!] &\stackrel{\text{def}}{=} \texttt{if } \mathcal{E}[\![e_g]\!]\ \mathcal{E}[\![e_t]\!]\ \mathcal{E}[\![e_f]\!] \\
\mathcal{E}[\![\lambda x.e]\!] &\stackrel{\text{def}}{=} \lambda x.\mathcal{E}[\![e]\!] \\
\mathcal{E}[\![\mu x.e]\!] &\stackrel{\text{def}}{=} \mu x.\mathcal{E}[\![e]\!] \\
\mathcal{E}[\![e_f\ e_a]\!] &\stackrel{\text{def}}{=} \mathcal{E}[\![e_f]\!]\ \mathcal{E}[\![e_a]\!] \\
\mathcal{E}[\![\texttt{mon}_j^{k,l}\ c\ e]\!] &\stackrel{\text{def}}{=} \mathcal{E}[\![e]\!]
\end{aligned}$$

Fig. 21. Erasure metafunction

The proof proceeds by two stuttering simulations (Figure 22), one in each direction. The auxiliary lemmas for the forward direction, which prove the standard weak erasure property, are collected in Section D.1; the backward direction is addressed in Section D.2. The lemmas common to the two directions can be found in Section D.3. By convention, metavariables with a tilde are in simulation, e.g., $\widetilde{e}$ implicitly means that $e \sim \widetilde{e}$.

$\boxed{\varsigma \sim \varsigma}$ for LCPCF (DYNAMIC)

$$\langle \vec{t}, e, \sigma \rangle \sim \langle \widetilde{e}, \varnothing \rangle$$

$\boxed{A^\sigma \sim A^\sigma}$ for LCPCF (DYNAMIC)

$$\vec{s}, E \sim \widetilde{E} \ \lceil \sigma \vdash s_i \text{ blocked} \rfloor$$

$\boxed{e \sim e}$ for LCPCF (DYNAMIC)

$$\begin{aligned}
x &\sim x \\
a &\sim a \\
\texttt{if } e_g\ e_t\ e_f &\sim \texttt{if } \widetilde{e_g}\ \widetilde{e_t}\ \widetilde{e_f} \\
\lambda x.e &\sim g \ \lceil e \sim g\ x,\ x \notin \text{FV}(g) \rfloor \\
\lambda x.e &\sim \lambda x.\widetilde{e} \\
\mu x.e &\sim \mu x.\widetilde{e} \\
e_f\ e_a &\sim \widetilde{e_f}\ \widetilde{e_a} \\
\texttt{mon}_j^{k,l}\ c\ e &\sim \widetilde{e} \\
(\alpha, v) \lhd e &\sim \widetilde{e}
\end{aligned}$$

$\boxed{E \sim E}$ for LCPCF (DYNAMIC)

$$\begin{aligned}
\square &\sim \square \\
\texttt{if } E\ e_t\ e_f &\sim \texttt{if } \widetilde{E}\ \widetilde{e_t}\ \widetilde{e_f} \\
E\ e &\sim \widetilde{E}\ \widetilde{e} \\
v\ E &\sim \widetilde{v}\ \widetilde{E} \\
\texttt{mon}_j^{k,l}\ c\ E &\sim \widetilde{E} \\
(\alpha, v) \lhd E &\sim \widetilde{E}
\end{aligned}$$

Fig. 22. Simulation Relation for LCPCF (DYNAMIC)

Theorem 4.4 (Strong Erasure). Suppose that $\varnothing \vdash e : \mathbb{A}$ and $\langle e, \varnothing\rangle \not\longmapsto^* \langle \mathtt{err}_j^k, \sigma\rangle$. In that case, $\langle e, \varnothing\rangle \longmapsto^* \langle \vec{s}, a, \sigma\rangle$ $\wr \sigma \vdash s_i$ blocked$\int$ if and only if $\langle \mathcal{E}[\![e]\!], \varnothing\rangle \longmapsto^* \langle a, \varnothing\rangle$.

Definition (Non-erroring). A configuration ς is *non-erroring* if $\varsigma \not\longmapsto^* \langle \mathtt{err}_j^k, \sigma\rangle$ for any σ.

Proof. Let non-erroring $\varsigma = \langle e, \varnothing\rangle$ and $\widetilde{\varsigma} = \langle \mathcal{E}[\![e]\!], \varnothing\rangle$. By Lemma D.2 and the definition of simulation in Figure 22, $\varsigma \sim \widetilde{\varsigma}$. By Lemma D.1, we know that $\widetilde{\varsigma}$ is well-typed.

We verify the forward and backward directions separately.

Case Forward Erasure.

Let $\varsigma_v = \langle \vec{s}, a, \sigma\rangle$ $\wr \sigma \vdash s_i$ blocked$\int$.

By Lemma D.3, there exists some final configuration $\widetilde{\varsigma}_v$ such that $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$ and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Let $\widetilde{v}$ be the value with which $\widetilde{\varsigma}_v$ stopped. Since the final configurations are in simulation and simulation preserves atoms, then $\widetilde{v} = a$. We conclude that $\widetilde{\varsigma}_v = \langle a, \varnothing\rangle$.

Case Backward Erasure.

Let $\widetilde{\varsigma}_v = \langle a, \varnothing\rangle$.

By Lemma D.5, there exists some final configuration ς_v such that $\varsigma \longmapsto^* \varsigma_v$ and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Let v be the value with which ς_v stopped. Since the final configurations are in simulation and simulation preserves atoms, then $v = a$.

We conclude that $\varsigma_v = \langle \vec{s}, a, \sigma\rangle$ $\wr \sigma \vdash s_i$ blocked$\int$. □

Lemma D.1 (Erasure Preserves Type). For all well-typed expressions $\Gamma \vdash e : \tau$, $\Gamma \vdash \mathcal{E}[\![e]\!] : \tau$.

Proof. By induction on $\Gamma \vdash e : \tau$. □

Lemma D.2 (Erasure Inclusion). For all $e \in$ Expr of LCPCF (CORE), $e \sim \mathcal{E}[\![e]\!]$.

Proof. By induction on e. □

D.1 Proof of Erasure: Left to Right

Lemma D.3 (Forward Erasure). If ς is non-erroring, $\varsigma \longmapsto^* \varsigma_v$ for some final configuration ς_v, and $\varsigma \sim \widetilde{\varsigma}$ for some $\widetilde{\varsigma}$, then there exists some final configuration $\widetilde{\varsigma}_v$ such that $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$ and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Proof. Proceed by induction on $\varsigma \longmapsto^* \varsigma_v$.

Case $\varsigma = \varsigma_v$.

Let $\widetilde{\varsigma}_v = \widetilde{\varsigma}$. Then, $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$ in zero steps, and $\varsigma_v \sim \widetilde{\varsigma}_v$ because $\varsigma \sim \widetilde{\varsigma}$.

Case $\varsigma \longmapsto^+ \varsigma_v$.

By definition of $\longmapsto^+$, there exists some ς' such that $\varsigma \longmapsto \varsigma' \longmapsto^* \varsigma_v$.

Then we can apply Lemma D.4. It follows that there exists $\widetilde{\varsigma}'$ with $\varsigma' \sim \widetilde{\varsigma}'$ and $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$.

Since ς is non-erroring, so is ς'.

The inductive hypothesis states that there exists some $\widetilde{\varsigma}_v$ such that $\widetilde{\varsigma}' \longmapsto^* \widetilde{\varsigma}_v$ and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Thus, $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}' \longmapsto^* \widetilde{\varsigma}_v$ or $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$. □

Lemma D.4 (Forward Simulation). If ς is non-erroring, $\varsigma \longmapsto \varsigma'$, and $\varsigma \sim \widetilde{\varsigma}$, then there exists $\widetilde{\varsigma}'$ with $\varsigma' \sim \widetilde{\varsigma}'$ and $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$.

Proof. By Lemma D.15, ς uniquely decomposes as $\langle A^\sigma[r], \sigma\rangle$ for some redex $r \in$ Redex.

Now consider A^σ. Suppose $A^\sigma = \vec{s}, T, \vec{t}, e$ $\wr \sigma \vdash s_i$ blocked$\int$. Then r occurs in the coroutine context T. Because ς is non-erroring, ς' must be of the form $\langle \vec{t'}, e, \sigma'\rangle$. Let $\widetilde{\varsigma} = \widetilde{\varsigma}' = \langle \widetilde{e}, \varnothing\rangle$ where $e \sim \widetilde{e}$. By the definition of the simulation relation, $\varsigma' \sim \widetilde{\varsigma}'$. Also, since $\widetilde{\varsigma} = \widetilde{\varsigma}'$, $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in zero steps.

LCPCF (PROOF) extends LCPCF (DYNAMIC)

$$\begin{aligned} \text{CoreRedex} \ni r_c &::= \texttt{if}\ v\ e\ e \mid v\ v \mid \mu x.e \\ \text{BodyRedex} \ni r_b &::= r_c \mid \texttt{mon}_j^{k,l}\ c\ v \mid (\alpha, v) \lhd v \\ \text{Redex} \ni r &::= r_b \mid \texttt{assert}_j^k\ v \mid (\alpha, v) \rhd \end{aligned}$$

Fig. 23. Syntax for LCPCF (PROOF).

Otherwise, $A^\sigma = \vec{s}, E\ \wr\sigma \vdash s_i \text{ blocked}\int$. By Lemma D.16, the body of the configuration uniquely decomposes as $\varsigma = \langle \vec{s}, E[r_b], \sigma \rangle$ for some body redex $r_b \in \text{BodyRedex}$.

Proceed by case analysis on r_b, each case instantiating the unique step $\varsigma \longmapsto \varsigma'$.

Case $\langle \vec{s}, E[\texttt{if tt}\ e_1\ e_2], \sigma \rangle \longmapsto \langle \vec{s}, E[e_1], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[\texttt{if tt}\ \widetilde{e}_1\ \widetilde{e}_2]$ such that $E \sim \widetilde{E}$ and $e_1 \sim \widetilde{e}_1$.

By the reduction relation, $\langle \widetilde{E}[\texttt{if tt}\ \widetilde{e}_1\ \widetilde{e}_2], \varnothing \rangle \longmapsto \langle \widetilde{E}[\widetilde{e}_1], \varnothing \rangle$. Let $\widetilde{\varsigma}' = \langle \widetilde{E}[\widetilde{e}_1], \varnothing \rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in one step.

Finally, by Lemma D.12, $\langle \vec{s}, E[e_1], \sigma \rangle \sim \langle \widetilde{E}[\widetilde{e}_1], \varnothing \rangle$.

The case for when the guard is `ff` is symmetric.

Case $\langle \vec{s}, E[(\lambda x.e_b)\ v], \sigma \rangle \longmapsto \langle \vec{s}, E[[v/x]e_b], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[g\ \widetilde{v}]$ such that $E \sim \widetilde{E}$, $\lambda x.e_b \sim g$ and $v \sim \widetilde{v}$. First, consider the η-expansion case, where $e_b \sim g\ x$ with $x \notin \text{FV}(g)$.

By Lemma D.13, $[v/x]e_b \sim g\ \widetilde{v}$. Then by Lemma D.12, $\langle \vec{s}, E[[v/x]e_b], \sigma \rangle \sim \langle \widetilde{E}[g\ \widetilde{v}], \varnothing \rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[g\ \widetilde{v}], \varnothing \rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in zero steps.

If this is not an η-expansion, then it must be the case that $g = \lambda x.\widetilde{e}_b$ with $e_b \sim \widetilde{e}_b$. By the reduction relation, $\langle \widetilde{E}[(\lambda x.\widetilde{e}_b)\ \widetilde{v}], \varnothing \rangle \longmapsto \langle \widetilde{E}[[\widetilde{v}/x]\widetilde{e}_b], \varnothing \rangle$. By Lemma D.13 and Lemma D.12, $\langle \vec{s}, E[[v/x]e_b], \sigma \rangle \sim \langle \widetilde{E}[[\widetilde{v}/x]\widetilde{e}_b], \varnothing \rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[[\widetilde{v}/x]\widetilde{e}_b], \varnothing \rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in one step.

Case $\langle \vec{s}, E[\mu x.e_b], \sigma \rangle \longmapsto \langle \vec{s}, E[[\mu x.e_b/x]e_b], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[\mu x.\widetilde{e}_b]$ such that $E \sim \widetilde{E}$ and $e_b \sim \widetilde{e}_b$.

By the reduction relation, $\langle \widetilde{E}[\mu x.\widetilde{e}_b], \varnothing \rangle \longmapsto \langle \widetilde{E}[[\mu x.\widetilde{e}_b/x]\widetilde{e}_b], \varnothing \rangle$. Also, Lemma D.13 and Lemma D.12 give us $\langle \vec{s}, E[[\mu x.e_b/x]e_b], \sigma \rangle \sim \langle \widetilde{E}[[\mu x.\widetilde{e}_b/x]\widetilde{e}_b], \varnothing \rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[[\mu x.\widetilde{e}_b/x]\widetilde{e}_b], \varnothing \rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in one step.

Case $\langle \vec{s}, E[\texttt{mon}_j^{k,l}\ (\texttt{flat}\ f)\ a], \sigma \rangle \longmapsto \langle \texttt{assert}_j^k\ (f\ a), \vec{s}, E[a], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[\widetilde{a}]$ such that $E \sim \widetilde{E}$ and $a \sim \widetilde{a}$.

Also, by Lemma D.12, $\langle \texttt{assert}_j^k\ (f\ a), \vec{s}, E[a], \sigma \rangle \sim \langle \widetilde{E}[\widetilde{a}], \varnothing \rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[\widetilde{a}], \varnothing \rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in zero steps.

Case $\langle \vec{s}, E[\texttt{mon}_j^{k,l}\ (c_1 \to c_2)\ v], \sigma \rangle \longmapsto \langle \vec{s}, E[\lambda x.\texttt{mon}_j^{k,l}\ c_2\ (v\ (\texttt{mon}_j^{l,k}\ c_1\ x))], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[\widetilde{v}]$ such that $E \sim \widetilde{E}$ and $v \sim \widetilde{v}$.

Since $\lambda x.e \sim g$ when $e \sim g\ x$, the right-hand side of this reduction step is simulated by $\widetilde{v}$:

$$\texttt{mon}_j^{k,l}\ c_2\ (v\ (\texttt{mon}_j^{l,k}\ c_1\ x)) \sim \widetilde{v}\ x$$

Then, by Lemma D.12, $\langle \vec{s}, E[\lambda x.\mathsf{mon}_j^{k,l}\, c_2\ (v\ (\mathsf{mon}_j^{l,k}\, c_1\ x))], \sigma\rangle \sim \langle \widetilde{E}[\widetilde{v}], \varnothing\rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[\widetilde{v}], \varnothing\rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in zero steps.

Case $\langle \vec{s}, E[\mathsf{mon}_j^{k,l}\, (c_1 \to c_2\ \&\ f)\ v], \sigma\rangle \longmapsto \langle \mathsf{assert}_j^k\ (f\ (\lambda x.(\alpha, x) \triangleright)) \Vdash \vec{s} \Vdash$
$E[\lambda x.(\alpha, x) \triangleleft \mathsf{mon}_j^{k,l}\, c_2\ (v\ (\mathsf{mon}_j^{l,k}\, c_1\ x))],$
$\sigma[\alpha \leftarrow \varnothing]\rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing\rangle$ decomposes into $\widetilde{E}[\widetilde{v}]$ such that $E \sim \widetilde{E}$ and $v \sim \widetilde{v}$.

Since $\lambda x.e \sim g$ when $e \sim g\ x$, the right-hand side of this reduction step is simulated by $\widetilde{v}$:

$$(\alpha, x) \triangleleft \mathsf{mon}_j^{k,l}\, c_2\ (v\ (\mathsf{mon}_j^{l,k}\, c_1\ x)) \sim \widetilde{v}\ x$$

Then, by Lemma D.12, $\varsigma' \sim \langle \widetilde{E}[\widetilde{v}], \varnothing\rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[\widetilde{v}], \varnothing\rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in zero steps.

Case $\langle \vec{s}, E[(\alpha, a) \triangleleft a'], \sigma\rangle \longmapsto \langle \vec{s}, E[a'], \sigma(\alpha)[a \leftarrow a']\rangle$**.**

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing\rangle$ decomposes into $\widetilde{E}[\widetilde{a}']$ such that $E \sim \widetilde{E}$ and $a' \sim \widetilde{a}'$.

Also, by Lemma D.12, $\langle \vec{s}, E[a'], \sigma(\alpha)[a \leftarrow a']\rangle \sim \langle \widetilde{E}[\widetilde{a}'], \varnothing\rangle$.

Let $\widetilde{\varsigma}' = \langle \widetilde{E}[\widetilde{a}'], \varnothing\rangle$. Then $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}'$ in zero steps. □

D.2 Proof of Erasure: Right to Left

Lemma D.5 (Backward Erasure). Let $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$ where $\widetilde{\varsigma}_v = \langle \widetilde{v}, \varnothing\rangle$ is a final configuration that holds the result value. If $\varsigma \sim \widetilde{\varsigma}$ for some well-typed and non-erroring ς, then there exists some final configuration ς_v such that $\varsigma \longmapsto^* \varsigma_v$ and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Proof. Proceed by induction on $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$.

Case $\widetilde{\varsigma} = \widetilde{\varsigma}_v$**.**

Here, $\widetilde{\varsigma} \longmapsto^* \widetilde{\varsigma}_v$ in zero steps, so it must be the case that $\varsigma \sim \widetilde{\varsigma}_v$. However, by definition of the simulation relation, ς may still contain contract redexes and might not itself be final.

By Lemma D.7, we know there exists $\varsigma_v = \langle \vec{s}', e', \sigma'\rangle$ $\wr \sigma' \vdash s_i' \text{ blocked} \int$ such that (1) $\varsigma \longmapsto^* \varsigma_v$ (2) $\varsigma_v \sim \widetilde{\varsigma}_v$, and (3) $e' \in \mathrm{Val}$, since no core redex simulates the value $\widetilde{v}$. It follows that the configuration ς_v is final, and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Case $\widetilde{\varsigma} \longmapsto^+ \widetilde{\varsigma}_v$**.**

By definition of $\longmapsto^+$, there exists some $\widetilde{\varsigma}'$ such that $\widetilde{\varsigma} \longmapsto \widetilde{\varsigma}' \longmapsto^* \widetilde{\varsigma}_v$.

Then we can apply Lemma D.6. It follows that there exists ς' with $\varsigma' \sim \widetilde{\varsigma}'$ such that $\varsigma \longmapsto^+ \varsigma'$.

Since ς is non-erroring, so is ς', and ς' is well-typed by preservation.

The inductive hypothesis states that there exists some ς_v such that $\varsigma' \longmapsto^* \varsigma_v$ and $\varsigma_v \sim \widetilde{\varsigma}_v$.

Thus, $\varsigma \longmapsto^+ \varsigma' \longmapsto^* \varsigma_v$ or $\varsigma \longmapsto^* \varsigma_v$ in one or more steps. □

Definition (Aligned). A configuration $\langle \vec{s}, E[r_c], \sigma_s\rangle$ with blocked coroutines and $r_c \in \mathrm{CoreRedex}$ is *aligned* with $\langle \widetilde{E}[\widetilde{r_c}], \varnothing\rangle$ when $E \sim \widetilde{E}$ and $r_c \sim \widetilde{r_c}$ without relating r_c by η-expansion.

Lemma D.6 (Backward Simulation). If $\widetilde{\varsigma} \longmapsto \widetilde{\varsigma}''$ and $\varsigma \sim \widetilde{\varsigma}$ for a well-typed and non-erroring ς, then there exists ς'' with $\varsigma'' \sim \widetilde{\varsigma}''$ and $\varsigma \longmapsto^+ \varsigma''$.

Proof. Let $\varsigma = \langle \vec{t}, e, \sigma\rangle$ and $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing\rangle$. Since $\varsigma \sim \widetilde{\varsigma}$, we know that $e \sim \widetilde{e}$ by definition of the simulation relation. Given that $\widetilde{\varsigma}$ takes a step, $\widetilde{e} \notin \mathrm{Val}$. Then, by Lemma D.7, we know there exists $\varsigma' = \langle \vec{s}', e', \sigma'\rangle$ $\wr \sigma' \vdash s_i' \text{ blocked} \int$ such that (1) $\varsigma \longmapsto^* \varsigma'$, (2) $\varsigma' \sim \widetilde{\varsigma}$, and (3) there exists unique E, r_c such that $e' = E[r_c]$ where $r_c \in \mathrm{CoreRedex}$ and ς' is aligned with $\widetilde{\varsigma}$. Furthermore, by

Lemma D.17, there also exists unique $\widetilde{E}, \widetilde{r_c}$ such that $\widetilde{\varsigma} = \langle \widetilde{E}[\widetilde{r_c}], \varnothing \rangle$. By Lemma D.11, $E \sim \widetilde{E}$ and $r_c \sim \widetilde{r_c}$.

Proceed by cases on $\widetilde{\varsigma} \longmapsto \widetilde{\varsigma}''$.

Case $\langle \widetilde{E}[\texttt{if tt } \widetilde{e_1}\ \widetilde{e_2}], \varnothing \rangle \longmapsto \langle \widetilde{E}[\widetilde{e_1}], \varnothing \rangle$**.**

If $r_c \sim \widetilde{r_c}$, it must be the case that $r_c = \texttt{if tt } e_1\ e_2$ with $e_1 \sim \widetilde{e_1}$ and $e_2 \sim \widetilde{e_2}$.
Here, $\varsigma' = \langle \vec{s}, E[\texttt{if tt } e_1\ e_2], \sigma \rangle$ and thus $\langle \vec{s}, E[\texttt{if tt } e_1\ e_2], \sigma \rangle \longmapsto \langle \vec{s}, E[e_1], \sigma \rangle$. Pick $\varsigma'' = \langle \vec{s}, E[e_1], \sigma \rangle$.
By Lemma D.12, $\langle \vec{s}, E[e_1], \sigma \rangle \sim \langle \widetilde{E}[\widetilde{e_1}], \varnothing \rangle$. Then $\varsigma'' \sim \widetilde{\varsigma}''$ and $\varsigma' \longmapsto \varsigma''$.
The case for when guard is `ff` is symmetric.

Case $\langle \widetilde{E}[(\lambda x.\widetilde{e_b})\ \widetilde{v}], \varnothing \rangle \longmapsto \langle \widetilde{E}[[\widetilde{v}/x]\widetilde{e_b}], \varnothing \rangle$**.**

Let $g = \lambda x.\widetilde{e_b}$. Then, if $r_c \sim \widetilde{r_c}$, it must be the case that $r_c = f\ v$ with $f \sim g$ and $v \sim \widetilde{v}$.
Since ς' is aligned with $\widetilde{\varsigma}$, f is not related to g by an η-expansion, so $f = \lambda x.e_b$ with $e_b \sim \widetilde{e_b}$.
Here, $\varsigma' = \langle \vec{s}, E[(\lambda x.e_b)\ v], \sigma \rangle$. Then we can apply a reduction step, $\langle \vec{s}, E[(\lambda x.e_b)\ v], \sigma \rangle \longmapsto \langle \vec{s}, E[[v/x]e_b], \sigma \rangle$. Pick $\varsigma'' = \langle \vec{s}, E[[v/x]e_b], \sigma \rangle$.
By Lemma D.13 and Lemma D.12, $\langle \vec{s}, E[[v/x]e_b], \sigma \rangle \sim \langle \widetilde{E}[[\widetilde{v}/x]\widetilde{e_b}], \varnothing \rangle$. Then $\varsigma'' \sim \widetilde{\varsigma}''$ and $\varsigma' \longmapsto \varsigma''$.

Case $\langle \widetilde{E}[\mu x.\widetilde{e_b}], \varnothing \rangle \longmapsto \langle \widetilde{E}[[\mu x.\widetilde{e_b}/x]\widetilde{e_b}], \varnothing \rangle$**.**

If $r_c \sim \widetilde{r_c}$, it must be the case that $r_c = \mu x.e_b$ with $e_b \sim \widetilde{e_b}$.
Here, $\varsigma' = \langle \vec{s}, E[\mu x.e_b], \sigma \rangle$ and thus $\langle \vec{s}, E[\mu x.e_b], \sigma \rangle \longmapsto \langle \vec{s}, E[[\mu x.e_b/x]e_b], \sigma \rangle$. Pick $\varsigma'' = \langle \vec{s}, E[[\mu x.e_b/x]e_b], \sigma \rangle$.
By Lemma D.13 and Lemma D.12, $\langle \vec{s}, E[[\mu x.e_b/x]e_b], \sigma \rangle \sim \langle \widetilde{E}[[\mu x.\widetilde{e_b}/x]\widetilde{e_b}], \varnothing \rangle$. Then $\varsigma'' \sim \widetilde{\varsigma}''$ and $\varsigma' \longmapsto \varsigma''$.

In every case, $\varsigma' \longmapsto \varsigma''$ in exactly one step. Since $\varsigma \longmapsto^* \varsigma'$ by Lemma D.7, it follows that $\varsigma \longmapsto^+ \varsigma''$. □

Definition (Contract Weight). The *contract weight* of a configuration body is the lexicographic pair (M, N), where M is the total size of the contracts in its monitors and N is the number of its functions and synchronized writes.

Lemma D.7 (Simulation Up To Value or Core Redex). If $\varsigma \sim \widetilde{\varsigma}$ for some well-typed and non-erroring ς, then there exists some well-typed and non-erroring $\varsigma' = \langle \vec{s}', e', \sigma' \rangle$ $\lbag \sigma' \vdash s_i' \text{ blocked} \rbag$ such that (1) $\varsigma \longmapsto^* \varsigma'$, (2) $\varsigma' \sim \widetilde{\varsigma}$, and (3) either $e' \in \text{Val}$ or there exists unique F, r_c such that $e' = F[r_c]$ where $r_c \in \text{CoreRedex}$ and ς' is aligned with $\widetilde{\varsigma}$.

Proof. Let $\varsigma = \langle \vec{t}, e, \sigma \rangle$ and $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$. Since $\varsigma \sim \widetilde{\varsigma}$, we know that $e \sim \widetilde{e}$ by definition of the simulation relation. Proceed by induction on the contract weight (M, N) of e.

By Lemma D.10, there exists some non-erroring and well-typed $\varsigma_s = \langle \vec{s}, e, \sigma_s \rangle$ $\lbag \sigma_s \vdash s_i \text{ blocked} \rbag$ such that $\varsigma \longmapsto^* \varsigma_s$ and $\varsigma_s \sim \widetilde{\varsigma}$. Coroutine steps do not modify the body expression, so ς_s has the same body e as ς.

If $e \in \text{Val}$, we simply pick $\varsigma' = \varsigma_s$. This configuration immediately satisfies all three conditions. Otherwise, by Lemma D.16, there exists unique E, r_b such that $\varsigma_s = \langle \vec{s}, E[r_b], \sigma_s \rangle$ where $r_b \in \text{BodyRedex}$.

Case $r_b \in \text{CoreRedex}$ and ς_s is aligned with $\widetilde{\varsigma}$**.**

Pick unique context $F = E$, unique core redex $r_c = r_b$, and $\varsigma' = \langle \vec{s}, F[r_c], \sigma_s \rangle$. This case immediately satisfies all three conditions.

Case $r_b \in$ CoreRedex and ς_s is not aligned with $\widetilde{\varsigma}$.

A core redex that is not aligned with the simulation must be an η-reduction, so let $r_b = (\lambda y.e_\eta)\ v$. Additionally, by Lemma D.11 and Lemma D.17, it must be the case that $\widetilde{e} = \widetilde{E}[g\ \widetilde{v}]$ with $E \sim \widetilde{E}$, $v \sim \widetilde{v}$, $e_\eta \sim g\ y$ and $y \notin \mathrm{FV}(g)$.

By Lemma D.9, there exists some ς_s' such that $\varsigma_s \longmapsto \varsigma_s'$ and $\varsigma_s' \sim \widetilde{\varsigma}$. This step substitutes v for y in e_η.

Since $e_\eta \sim g\ y$ and $y \notin \mathrm{FV}(g)$, the variable y occurs in e_η once as an application argument and, possibly, as a key in a synchronized write. If it occurs as a key, well-typedness ensures that v is an atom. Otherwise, it is substituted exactly once, so the substitution does not grow the size of the body expression. Therefore, after this step M stays the same and N decreases.

The configuration ς_s' is non-erroring because ς is, and it is well-typed by preservation. Applying the inductive hypothesis to $\varsigma_s' \sim \widetilde{\varsigma}$ yields some ς' with $\varsigma_s' \longmapsto^* \varsigma'$ that satisfies (2) and (3). It also satisfies (1), since $\varsigma \longmapsto^* \varsigma_s \longmapsto \varsigma_s' \longmapsto^* \varsigma'$.

Case $r_b \notin$ CoreRedex.

By Lemma D.8, there exists some ς_s' such that $\varsigma_s \longmapsto \varsigma_s'$ and $\varsigma_s' \sim \widetilde{\varsigma}$.

By inspection of the stepping relation, this step decreases the contract weight of the body:

- Rules MON-FLAT, MON-ARR, and MON-ARR-FLAT remove one contract constructor, so M decreases.
- Rule SYNC-SET leaves M the same and removes a synchronized write, so N decreases.

Applying the inductive hypothesis to $\varsigma_s' \sim \widetilde{\varsigma}$ yields the desired ς'. □

Lemma D.8 (Non-Core Reductions Preserve Simulation). Let

$$\varsigma = \langle \vec{s}, E[r_b], \sigma \rangle \quad \wr \sigma \vdash s_i \text{ blocked} \int$$

be a well-typed and non-erroring configuration with $\varsigma \sim \widetilde{\varsigma}$, where E and r_b are the unique decomposition of the body and $r_b \in$ BodyRedex but $r_b \notin$ CoreRedex. Then there exists some ς' such that $\varsigma \longmapsto \varsigma'$ and $\varsigma' \sim \widetilde{\varsigma}$.

Proof. Proceed by cases on the stepping relation for non-core redexes.

Case $\langle \vec{s}, E[\mathtt{mon}_j^{k,l}\ (\mathtt{flat}\ f)\ a], \sigma \rangle \longmapsto \langle \mathtt{assert}_j^k\ (f\ a), \vec{s}, E[a], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[\widetilde{a}]$ such that $E \sim \widetilde{E}$ and $\mathtt{mon}_j^{k,l}\ (\mathtt{flat}\ f)\ a \sim \widetilde{a}$. Also, by definition of simulation, $a \sim \widetilde{a}$.

Then, by Lemma D.12, $\langle \mathtt{assert}_j^k\ (f\ a), \vec{s}, E[a], \sigma \rangle \sim \langle \widetilde{E}[\widetilde{a}], \varnothing \rangle$, so $\varsigma' \sim \widetilde{\varsigma}$.

Case $\langle \vec{s}, E[\mathtt{mon}_j^{k,l}\ (c_1 \to c_2)\ v], \sigma \rangle \longmapsto \langle \vec{s}, E[\lambda x.\mathtt{mon}_j^{k,l}\ c_2\ (v\ (\mathtt{mon}_j^{l,k}\ c_1\ x))], \sigma \rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ decomposes into $\widetilde{E}[\widetilde{v}]$ such that $E \sim \widetilde{E}$ and $\mathtt{mon}_j^{k,l}\ (c_1 \to c_2)\ v \sim \widetilde{v}$.

Since the simulation relation relates $\lambda x.e$ to g whenever $e \sim g\ x$, the right-hand side of this reduction step is also simulated by $\widetilde{v}$:

$$\mathtt{mon}_j^{k,l}\ c_2\ (v\ (\mathtt{mon}_j^{l,k}\ c_1\ x)) \sim \widetilde{v}\ x$$

Then, by Lemma D.12, $\langle \vec{s}, E[\lambda x.\mathtt{mon}_j^{k,l}\ c_2\ (v\ (\mathtt{mon}_j^{l,k}\ c_1\ x))], \sigma \rangle \sim \langle \widetilde{E}[\widetilde{v}], \varnothing \rangle$, so $\varsigma' \sim \widetilde{\varsigma}$.

Case $\langle \vec{s}, E[\mathtt{mon}_j^{k,l}\ (c_1 \to c_2\ \&\ f)\ v], \sigma\rangle \longmapsto \langle \mathtt{assert}_j^k\ (f\ (\lambda x.(\alpha, x) \triangleright)) +\!\!+ \vec{s} +\!\!+$
$E[\lambda x.(\alpha, x) \triangleleft \mathtt{mon}_j^{k,l}\ c_2\ (v\ (\mathtt{mon}_j^{l,k}\ c_1\ x))],$
$\sigma[\alpha \leftarrow \varnothing]\rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing\rangle$ decomposes into $\widetilde{E}[\widetilde{v}]$ such that $E \sim \widetilde{E}$ and $\mathtt{mon}_j^{k,l}\ (c_1 \to c_2\ \&\ f)\ v \sim \widetilde{v}$.

Since the simulation relation relates $\lambda x.e$ to g whenever $e \sim g\ x$, the right-hand side of this reduction step is also simulated by $\widetilde{v}$:

$$(\alpha, x) \triangleleft \mathtt{mon}_j^{k,l}\ c_2\ (v\ (\mathtt{mon}_j^{l,k}\ c_1\ x)) \sim \widetilde{v}\ x$$

Then, by Lemma D.12, $\varsigma' \sim \widetilde{\varsigma}$.

Case $\langle \vec{s}, E[(\alpha, a) \triangleleft a'], \sigma\rangle \longmapsto \langle \vec{s}, E[a'], \sigma(\alpha)[a \leftarrow a']\rangle$.

By Lemma D.11, the body expression of $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing\rangle$ decomposes into $\widetilde{E}[\widetilde{a}']$ such that $E \sim \widetilde{E}$ and $(\alpha, a) \triangleleft a' \sim \widetilde{a}'$. Also, by definition of simulation, $a' \sim \widetilde{a}'$.

Then, by Lemma D.12, $\langle \vec{s}, E[a'], \sigma(\alpha)[a \leftarrow a']\rangle \sim \langle \widetilde{E}[\widetilde{a}'], \varnothing\rangle$, so $\varsigma' \sim \widetilde{\varsigma}$. □

Lemma D.9 (η-Reductions Preserve Simulation). Let $\varsigma = \langle \vec{s}, E[(\lambda y.e_\eta)\ v], \sigma\rangle$ $\wr\sigma \vdash s_i$ blocked$\int$ be well-typed and non-erroring, with $E \sim \widetilde{E}$, $v \sim \widetilde{v}$, $e_\eta \sim g\ y$, and $y \notin \mathrm{FV}(g)$. Then there exists some ς' such that $\varsigma \longmapsto \varsigma'$ and $\varsigma' \sim \langle \widetilde{E}[g\ \widetilde{v}], \varnothing\rangle$.

Proof. Following FUN-APP, pick $\varsigma' = \langle \vec{s}, E[[v/y]e_\eta], \sigma\rangle$ so that $\varsigma \longmapsto \varsigma'$.
Since $y \notin \mathrm{FV}(g)$, Lemma D.13 gives $[v/y]e_\eta \sim g\ \widetilde{v}$, and by Lemma D.12, $\varsigma' \sim \langle \widetilde{E}[g\ \widetilde{v}], \varnothing\rangle$. □

Lemma D.10 (Coroutine Irrelevance). For any well-typed and non-erroring $\varsigma = \langle \vec{t}, e, \sigma\rangle$ with $\varsigma \sim \widetilde{\varsigma}$ there exists some well-typed and non-erroring $\varsigma_s = \langle \vec{s}, e, \sigma_s\rangle$ $\wr\sigma_s \vdash s_i$ blocked$\int$ such that $\varsigma \longmapsto^* \varsigma_s$ and $\varsigma_s \sim \widetilde{\varsigma}$.

Proof. By Proposition 3.1, coroutine code terminates. Because ς is non-erroring, ASSERT-FF never applies and each coroutine will either get discarded by ASSERT-TT or blocked on a synchronized read. The simulation is preserved because Figure 22 constrains only the body expression, which is unaffected by coroutine steps. Finally, ς_s is non-erroring because it is reachable from ς, and it is well-typed by preservation. □

D.3 Proof of Erasure: Common Lemmas

Lemma D.11 (Simulation Decomposition). If $\langle \vec{t}, e, \sigma\rangle \sim \langle \widetilde{e}, \varnothing\rangle$ and $e = E[e_s]$, then there exist $\widetilde{E}, \widetilde{e_s}$ with $E \sim \widetilde{E}, e_s \sim \widetilde{e_s}$ such that $\widetilde{e} = \widetilde{E}[\widetilde{e_s}]$.

Proof. By induction on $e \sim \widetilde{e}$. □

Lemma D.12 (Simulation Composition). If $E \sim \widetilde{E}, e \sim \widetilde{e}$, then for all $\langle \vec{t}, E[e], \sigma\rangle$ it is the case that $\langle \vec{t}, E[e], \sigma\rangle \sim \langle \widetilde{E}[\widetilde{e}], \varnothing\rangle$

Proof. By induction on $E \sim \widetilde{E}$. □

Lemma D.13 (Simulation Substitution). If $e \sim \widetilde{e}, v \sim \widetilde{v}$, then $[v/x]e \sim [\widetilde{v}/x]\widetilde{e}$

Proof. By induction on $e \sim \widetilde{e}$. □

Lemma D.14 (Unique Decomposition of Expressions). If $e \notin \mathrm{Val}$, then there exists unique E, r such that $e = E[r]$.

Proof. By induction on e. □

Lemma D.15 (Unique Decomposition of Configurations). If $\varsigma = \langle \vec{t}, e, \sigma \rangle$ and $e \notin \mathsf{Val}$, then there exists unique A^σ, r such that $\varsigma = \langle A^\sigma[r], \sigma \rangle$.

Proof. Proceed by cases on whether all coroutines are blocked.

Case For all $i < |\vec{t}|$ such that $\sigma \vdash t_i$ blocked**.**

By Lemma D.14, there exists unique $E, r \,.\, e = E[r]$. Pick $A^\sigma = \vec{t}, E$.

Otherwise.

By well ordering, let $i \in \mathbb{N}$ be the smallest number such that $\sigma \vdash t_i$ blocked does not hold. By the grammar, $t_i = \mathtt{assert}^k_j\ e'$. If $e' = v$, then pick $r = \mathtt{assert}^k_j\ v$ and $A^\sigma = t_0, \ldots, t_{i-1}, \square, t_{i+1}, \ldots, t_n, e$. Otherwise, by Lemma D.14, there exists unique E, r' such that $e' = E[r']$. We then pick $r = r'$ and $A^\sigma = t_0, \ldots, t_{i-1}, \mathtt{assert}^k_j\ E, t_{i+1}, \ldots, t_n, e$. □

Lemma D.16 (Unique Decomposition of Body Expression). For every configuration $\varsigma = \langle \vec{s}, e, \sigma_s \rangle$ $\wr \sigma_s \vdash s_i$ blocked$\int$ with $e \notin \mathsf{Val}$, there exists unique E, r_b such that $\varsigma = \langle \vec{s}, E[r_b], \sigma_s \rangle$ where $r_b \in \mathsf{BodyRedex}$.

Proof. From Lemma D.15, for the case where all coroutines are blocked, there exists unique E, r such that $\varsigma = \langle \vec{s}, E[r], \sigma_s \rangle$.

It remains to rule out the two redexes of Figure 23 that are not body redexes. $\mathtt{assert}^k_j\ e'$ cannot occur in e because it is a coroutine rather than an expression. $\lambda x.(\alpha, x) \triangleright$ cannot occur in e either: this expression can only be introduced dynamically by MON-ARR-FLAT, which will be placed in a coroutine and not the configuration body. It follows that $r \in \mathsf{BodyRedex}$. □

Lemma D.17 (Unique Decomposition of Configuration Simulation). For every $\varsigma \sim \widetilde{\varsigma}$ with $\widetilde{\varsigma} = \langle \widetilde{e}, \varnothing \rangle$ and $\widetilde{e} \notin \mathsf{Val}$, there exists unique E, r_c such that $\widetilde{\varsigma} = \langle E[r_c], \varnothing \rangle$ where $r_c \in \mathsf{CoreRedex}$.

Proof. From Lemma D.15, for the case where there are no coroutines, there exists unique E, r such that $\widetilde{\varsigma} = \langle E[r], \varnothing \rangle$.

We can show that $r \in \mathsf{CoreRedex}$ by induction on simulation relation. □

E Complete Monitoring

$$
\begin{array}{l}
\boxed{\textsc{lcpcf (owner)}}\ \text{extends } \textsc{lcpcf (dynamic)} \\
\quad \mathsf{Expr} \ni e ::= \ldots \mid |e|^{l} \\
\quad \mathsf{Con} \ni c ::= \texttt{flat}\ |f|^{l} \mid c \to c \mid c \to c\ \&\ |f|^{l} \\
\quad \mathsf{Val} \ni v ::= \ldots \mid |v|^{l} \\
\boxed{\textsc{lcpcf (owner)}}\ \text{extends } \textsc{lcpcf (dynamic)} \\
\quad \mathsf{Ctx}^{\mathsf{A}} \ni A^{\sigma,l} ::= \vec{s}, T^{l}, \vec{t}, e \mid \vec{s}, E^{l} \mid \lfloor \sigma \vdash s_i \text{ blocked} \rfloor \\
\quad \mathsf{Ctx}^{\mathsf{T}} \ni T^{l} ::= \square \mid \texttt{assert}^{k}_{l}\ E^{l_o} \mid \texttt{assert}^{k}_{l'}\ E^{l} \\
\quad \mathsf{Ctx}^{\mathsf{E}} \ni E^{l}, F^{l} ::= \mid \texttt{if}\ E^{l}\ e\ e \mid E^{l}\ e \mid v\ E^{l} \mid (\alpha, v) \lhd E^{l} \mid \texttt{mon}^{l,k}_{j}\ c\ E^{l_o} \mid \texttt{mon}^{l',k}_{j}\ c\ E^{l} \mid |E^{l_o}|^{l} \\
\qquad\qquad \mid |E^{l}|^{l'} \\
\quad \mathsf{Ctx}^{\mathsf{E}} \ni E^{l_o}, F^{l_o} ::= \square \mid \texttt{if}\ E^{l_o}\ e\ e \mid E^{l_o}\ e \mid v\ E^{l_o} \mid (\alpha, v) \lhd E^{l_o}
\end{array}
$$

Fig. 24. Syntax for LCPCF (OWNER)

$$
\begin{array}{rl}
\textsc{if-tt} & \langle A^{\sigma,l}[\texttt{if}\ ||\texttt{tt}||^{l}\ e_1\ e_2], \sigma\rangle \longmapsto \langle A^{\sigma,l}[e_1], \sigma\rangle \\
\textsc{if-ff} & \langle A^{\sigma,l}[\texttt{if}\ ||\texttt{ff}||^{l}\ e_1\ e_2], \sigma\rangle \longmapsto \langle A^{\sigma,l}[e_2], \sigma\rangle \\
\textsc{fun-app} & \langle A^{\sigma,l}[||\lambda x.e||^{l}\ ||v||^{l}], \sigma\rangle \longmapsto \langle A^{\sigma,l}[|[|v|^{l}/x]e|^{l}], \sigma\rangle \\
\textsc{fix-unroll} & \langle A^{\sigma,l}[\mu x.e], \sigma\rangle \longmapsto \langle A^{\sigma,l}[|[|\mu x.e|^{l}/x]e|], \sigma\rangle \\
\textsc{mon-flat} & \langle A^{\sigma,l}[\texttt{mon}^{k,l}_{j}\ (\texttt{flat}^{l'}\ |f|^{j})\ ||a||^{l}], \sigma\rangle \longmapsto \langle \texttt{assert}^{k}_{j}\ (|f|^{j}\ a), A^{\sigma,l}[a], \sigma\rangle \\
\textsc{mon-arr-ho} & \langle A^{\sigma}[\texttt{mon}^{k,l}_{j}\ (c_1 \to c_2)\ v], \sigma\rangle \longmapsto \langle A^{\sigma,l}[\lambda x.\texttt{mon}^{k,l}_{j}\ c_2\ |v\ (\texttt{mon}^{l,k}_{j}\ c_1\ |x|^{l})|^{k}], \sigma\rangle \\
\textsc{mon-arr-fo} & \langle A^{\sigma,l}[\texttt{mon}^{k,l}_{j}\ (c_1 \to c_2\ \&^{l'}\ |f|^{j})\ ||v||^{l}], \sigma\rangle \longmapsto \langle \texttt{assert}^{k}_{j}\ (|f|^{j}\ (\lambda x.(\alpha, x) \rhd)) \Rightarrow \\
 & \qquad A^{\sigma,l}[\lambda x.(\alpha, x) \lhd \texttt{mon}^{k,l}_{j}\ c_2\ |v\ (\texttt{mon}^{l,k}_{j}\ c_1\ |x|^{l})|^{k}], \\
 & \qquad \sigma[\alpha \leftarrow \varnothing]\rangle \\
 & \qquad \lfloor \alpha \notin \mathrm{dom}(\sigma) \rfloor \\
\textsc{match-tt} & \langle A^{\sigma,l}[\texttt{assert}^{k}_{j}\ ||\texttt{tt}||^{j}], \sigma\rangle \longmapsto \langle A^{\sigma,l}[\varepsilon], \sigma\rangle \\
\textsc{match-ff} & \langle A^{\sigma,l}[\texttt{assert}^{k}_{j}\ ||\texttt{ff}||^{j}], \sigma\rangle \longmapsto \langle \texttt{err}^{k}_{j}, \sigma\rangle \\
\textsc{sync-get} & \langle A^{\sigma,l}[(\alpha, ||a||^{l}) \rhd], \sigma\rangle \longmapsto \langle A^{\sigma,l}[a'], \sigma\rangle \ \lfloor \sigma(\alpha)(a) = a' \rfloor \\
\textsc{sync-set} & \langle A^{\sigma,l}[(\alpha, ||a||^{l}) \lhd a'], \sigma\rangle \longmapsto \langle A^{\sigma,l}[a'], \sigma(\alpha)[a \leftarrow a']\rangle
\end{array}
$$

Fig. 25. Reduction Relation for LCPCF (OWNER)

The ownership semantics in Figure 25 follows Dimoulas et al. [17] fairly closely, except in two situations. First, the semantics avoids so-called "loosely well-formed terms" by injecting additional ownership labels into the contractum. Doing so is safe because additional ownership labels can only cause more expressions to become blocked. Second, to make lookups in the synchronization table ignorant of ownership labels, keys are stripped of their owner when placed into the table. Doing so is safe because there is no way to extract the ownerless key from the synchronization table. Values *can* be extracted from the synchronization table and therefore the semantics does not strip off their ownership label. (Though, because writes to the synchronization table happen only after a flat-contract check, these values never have an ownership label anyway.)

$\boxed{\Psi \vdash \langle \vec{t}, e, \sigma \rangle}$ for LCPCF (OWNER) extends $\Psi \vdash \langle \vec{t}, e, \sigma \rangle$

$$\frac{\Psi \vdash t_i \qquad \ldots \qquad \varnothing, \Psi, \varnothing, l_o \vdash e : \tau \qquad \Psi \vdash \sigma}{\Psi \vdash \langle \vec{t}, e, \sigma \rangle}$$

$\boxed{\Psi \vdash t}$ for LCPCF (OWNER) extends $\Psi \vdash t$

$$\frac{\varnothing, \Psi, \varnothing, j \vdash e : \mathsf{Bool}}{\Psi \vdash \mathsf{assert}_j^k \ |e|^j}$$

$\boxed{\Gamma, \Psi, \Xi, l \vdash e : \tau}$ for LCPCF (OWNER) extends $\Gamma, \Psi \vdash e : \tau$

$$\frac{\Gamma(x) = \tau \qquad \Xi(x) = l}{\Gamma, \Psi, \Xi, l \vdash x : \tau} \qquad \frac{\Gamma[x \leftarrow \tau], \Psi, \Xi[x \leftarrow l], l \vdash e : \tau'}{\Gamma, \Psi, \Xi, l \vdash \lambda x.e : \tau \to \tau'} \qquad \frac{\Gamma[x \leftarrow \tau], \Psi, \Xi[x \leftarrow l], l \vdash e : \tau}{\Gamma, \Psi, \Xi, l \vdash \mu x.e : \tau}$$

$$\frac{\Gamma, \Psi, \Xi, k, l, j \vdash c : \mathsf{Con}(\tau) \qquad \Gamma, \Psi, \Xi, k \vdash e : \tau \qquad k \neq l}{\Gamma, \Psi, \Xi, l \vdash \mathsf{mon}_j^{k,l} \ c \ |e|^k : \tau} \qquad \frac{\Gamma, \Psi, \Xi, l \vdash e : \tau}{\Gamma, \Psi, \Xi, l \vdash |e|^l : \tau}$$

$\boxed{\Gamma, \Psi, \Xi, k, l, j \vdash c : \tau}$ for LCPCF (OWNER) extends $\Gamma, \Psi \vdash c : \tau$

$$\frac{\Gamma, \Psi, \Xi, j \vdash f : \mathbb{A} \to \mathsf{Bool}}{\Gamma, \Psi, \Xi, k, l, j \vdash \mathsf{flat}^k \ |f|^j : \mathsf{Con}(\mathbb{A})} \qquad \frac{\Gamma, \Psi, \Xi, l, k, j \vdash c_1 : \mathsf{Con}(\tau_1) \qquad \Gamma, \Psi, \Xi, k, l, j \vdash c_2 : \mathsf{Con}(\tau_2)}{\Gamma, \Psi, \Xi, k, l, j \vdash c_1 \to c_2 : \mathsf{Con}(\tau_1 \to \tau_2)}$$

$$\frac{\Gamma, \Psi, \Xi, l, k, j \vdash c_1 : \mathsf{Con}(\mathbb{A}) \qquad \Gamma, \Psi, \Xi, k, l, j \vdash c_2 : \mathsf{Con}(\mathbb{A}) \qquad \Gamma, \Psi, \Xi, j \vdash f : (\mathbb{A} \to \mathbb{A}) \to \mathsf{Bool}}{\Gamma, \Psi, \Xi, k, l, j \vdash c_1 \to c_2 \ \& \ |f|^j : \mathsf{Con}(\mathbb{A} \to \mathbb{A})}$$

$\boxed{\Psi \vdash \sigma}$ for LCPCF (OWNER) extends $\Psi \vdash \sigma$

$$\frac{\forall (\alpha, \tau \to \tau'), \forall (v, v') \in \sigma(\alpha) \ . \ \varnothing, \Psi, \varnothing, l_o \vdash v : \tau \wedge \varnothing, \Psi, \varnothing, l_o \vdash v' : \tau'}{\Psi \vdash \sigma}$$

Fig. 26. Typing Rules for LCPCF (OWNER)

$\boxed{\text{LCPCF (PROOF)}}$ extends LCPCF (OWNER)

$$\begin{aligned} \text{Done} \ni d &::= v \mid \mathsf{err}_j^k \\ \text{Redex} \ni r &::= \mathsf{if} \ v \ e \ e \mid v \ v \mid \mu x.e \mid \mathsf{mon}_j^{k,l} \ c \ |v|^l \mid \mathsf{assert}_j^k \ v \mid (\alpha, v) \triangleright \mid (\alpha, v) \triangleleft v \end{aligned}$$

Fig. 27. Syntax for LCPCF (PROOF)

Theorem E.1 (Complete monitoring). If $\varnothing \vdash e : \tau$ and $\langle e, \varnothing \rangle \longmapsto^* \varsigma$ then either $\varsigma = \langle \vec{t}, d, \sigma \rangle$ or $\exists \varsigma' . \ \varsigma \longmapsto \varsigma'$. Moreover, if $\varsigma = \langle A^{\sigma,l}[\mathsf{mon}_j^{k,l} \ (\mathsf{flat}^{k'} \ f) \ v], \sigma \rangle$ or $\varsigma = \langle A^{\sigma,l}[\mathsf{mon}_j^{k,l} \ (c_1 \to c_2 \ \&^{k'} \ f) \ v], \sigma \rangle$ then it must be that $k' = k$ and $v = |v'|^k$.

Proof. Proceed by induction on the relation $\longmapsto^*$ and Lemma E.2. Intermediate expressions must be well typed. Therefore, Lemma E.3 and Lemma E.5 imply that every value guarded by a flat contract or a flat-function contract must satisfy the desired constraints. □

This statement of complete monitoring, which contains within it the exact statement of blame correctness, is slightly stronger than the traditional formulation of complete monitoring.

Lemma E.2 (Progress and preservation). For all configurations $\varsigma = \langle \vec{t}, e, \sigma \rangle$ if $\Psi \vdash \varsigma$ then either $e \in \text{Done}$ or $\exists \varsigma', \Psi' . \ \varsigma \longmapsto \varsigma' \wedge \Psi' \vdash \varsigma'$.

Proof. If $e \in$ Done there is nothing to show. Otherwise, by Lemma E.3, $\varsigma = \langle A^{\sigma,l}[r], \sigma\rangle$. By Lemma E.5, $\varnothing, \Psi, \varnothing, l \vdash r : \tau$. Proceed by cases on r.

Case $r = \texttt{if}\ v\ e_1\ e_2$.

By inversion, $\varnothing, \Psi, \varnothing, l \vdash v : \texttt{Bool}$, $\varnothing, \Psi, \varnothing, l \vdash e_1 : \tau$, and $\varnothing, \Psi, \varnothing, l \vdash e_2 : \tau$. By Lemma E.9, $v = ||\texttt{tt}||^l$ or $v = ||\texttt{ff}||^l$. In the first case, pick $\varsigma' = \langle A^{\sigma,l}[e_1], \sigma\rangle$, $\Psi' = \Psi$. In the second case, pick $\varsigma' = \langle A^{\sigma,l}[e_2], \sigma\rangle$, $\Psi' = \Psi$. These configurations are well typed by Lemma E.8.

Case $r = v_1\ v_2$.

By inversion, $\varnothing, \Psi, \varnothing, l \vdash v_1 : \tau \to \tau'$ and $\varnothing, \Psi, \varnothing, l \vdash v_2 : \tau$. By Lemma E.9, $v_1 = ||\lambda x.e||^l$ and $v_2 = ||v||^l$. Pick $\varsigma' = \langle A^{\sigma,l}[|[|v|^l/x]e|^l], \sigma\rangle$, $\Psi' = \Psi$. This configuration is well-typed by Lemma E.10 and Lemma E.8.

Case $r = \mu x.e$.

By inversion, $\varnothing, \Psi, \varnothing, l \vdash \mu x.e : \tau$. Pick $\varsigma' = \langle A^{\sigma,l}[[|\mu x.e|^l/x]e], \sigma\rangle$, $\Psi' = \Psi$. This configuration is well-typed by Lemma E.10 and Lemma E.8.

Case $r = \texttt{mon}^{k,l}_j\ c\ |v|^k$.

By inversion, $\varnothing, \Psi, \varnothing, k, l, j \vdash c : \texttt{Con}(\tau)$ and $\varnothing, \Psi, \varnothing, k \vdash v : \tau$. Proceed by cases on c.

Case $c = \texttt{flat}^k\ |f|^j$.

It must be that $\tau = \mathbb{A}$. By inversion, $\varnothing, \Psi, \varnothing, j \vdash f : \mathbb{A} \to \texttt{Bool}$. By Lemma E.9, $v = ||a||^k$. Pick $\varsigma' = \langle \texttt{assert}^k_j\ (|f|^j\ a) \Vdash A^{\sigma,l}[a], \sigma\rangle$, $\Psi' = \Psi$.

$$\frac{\dfrac{\varnothing, \Psi, \varnothing, l \vdash |f|^j\ a : \texttt{Bool}}{\Psi \vdash \texttt{assert}^k_j\ (|f|^j\ a)} \qquad \dfrac{\varnothing, \Psi, \varnothing, l \vdash a : \mathbb{A}}{\varnothing, \Psi, \varnothing, l \vdash A^{\sigma,l}[a] : \mathbb{A}} \qquad \Psi \vdash \sigma}{\Psi \vdash \langle \texttt{assert}^k_j\ (|f|^j\ a), A^{\sigma,l}[a], \sigma\rangle}$$

Case $c = c_1 \to c_2$.

It must be that $\tau = \tau_1 \to \tau_2$. By inversion, it follows that $\varnothing, \Psi, \varnothing, l, k, j \vdash c_1 : \texttt{Con}(\tau_1)$ and $\varnothing, \Psi, \varnothing, k, l, j \vdash c_2 : \texttt{Con}(\tau_2)$. Pick $\varsigma' = \langle A^{\sigma,l}[\lambda x.\texttt{mon}^{k,l}_j\ c_2\ |v\ (\texttt{mon}^{l,k}_j\ c_1\ |x|^l)|^k], \sigma\rangle$ and $\Psi' = \Psi$.

$$\frac{\dfrac{\varnothing, \Psi, \varnothing, l \vdash \lambda x.\texttt{mon}^{k,l}_j\ c_2\ |v\ (\texttt{mon}^{l,k}_j\ c_1\ |x|^l)|^k : \tau_1 \to \tau_2}{\varnothing, \Psi, \varnothing, l \vdash A^{\sigma,l}[\lambda x.\texttt{mon}^{k,l}_j\ c_2\ |v\ (\texttt{mon}^{l,k}_j\ c_1\ |x|^l)|^k] : \tau_1 \to \tau_2} \qquad \Psi \vdash \sigma}{\Psi \vdash \langle A^{\sigma,l}[\lambda x.\texttt{mon}^{k,l}_j\ c_2\ |v\ (\texttt{mon}^{l,k}_j\ c_1\ |x|^l)|^k], \sigma\rangle}$$

Case $c = c_1 \to c_2\ \&^k\ |f|^j$.

It must be that $\tau = \mathbb{A} \to \mathbb{A}$. By inversion,

$$\varnothing, \Psi, \varnothing, l, k, j \vdash c_1 : \texttt{Con}(\mathbb{A})$$

and $\varnothing, \Psi, \varnothing, k, l, j \vdash c_2 : \texttt{Con}(\mathbb{A})$. Pick

$$\begin{aligned}\varsigma' = \langle &\texttt{assert}^k_j\ (|f|^j\ (\lambda x.(\alpha, x) \triangleright)),\\ &A^{\sigma,l}[\lambda x.(\alpha, x) \triangleleft \texttt{mon}^{k,l}_j\ c_2\ |v\ (\texttt{mon}^{l,k}_j\ c_1\ |x|^l)|^k],\\ &\sigma[\alpha \leftarrow \varnothing]\rangle\end{aligned}$$

and $\Psi' = \Psi[\alpha \leftarrow \mathbb{A} \to \mathbb{A}]$. Showing $\Psi' \vdash \varsigma'$ is similar to combining the previous two cases.

Case $r = \texttt{assert}^k_j\ v$.

By inversion, $\varnothing, \Psi, \varnothing, j \vdash v : \texttt{Bool}$. By Lemma E.9, $v = ||\texttt{tt}||^j$ or $v = ||\texttt{ff}||^j$. If $v = ||\texttt{tt}||^j$ then pick $\varsigma' = \langle A^{\sigma,l}[\varepsilon], \sigma\rangle$, $\Psi' = \Psi$. If $v = ||\texttt{ff}||^j$ then pick $\varsigma' = \langle \texttt{err}^k_j, \sigma\rangle$, $\Psi' = \Psi$.

Case $r = (\alpha, v) \rhd$.

By inversion, $\varnothing, \Psi, \varnothing, l \vdash v : \mathbb{A}$. By Lemma E.9, $v = ||a||^l$. Pick $\langle A^{\sigma,l}[a'], \sigma\rangle$ where $\sigma(\alpha)(a) = a'$. It must be that $\sigma(\alpha)(a)$ is defined, otherwise that would contradict the definition of $A^{\sigma,l}$.

Case $r = (\alpha, v) \lhd v'$.

By inversion, $\varnothing, \Psi, \varnothing, l \vdash v : \mathbb{A}$ and $\varnothing, \Psi, \varnothing, l \vdash v' : \mathbb{A}$. By Lemma E.9, $v = ||a||^l$ and $v' = ||a'||^l$. Pick $\varsigma' = \langle A^{\sigma,l}[a'], \sigma(\alpha)[a \leftarrow a']\rangle, \Psi' = \Psi$. □

Lemma E.3 (Unique decomposition of configurations). If $\varsigma = \langle \vec{t}, e, \sigma\rangle$ and $e \notin \text{Done}$, then $\exists! A^{\sigma,l}, r \,.\, \varsigma = \langle A^{\sigma,l}[r], \sigma\rangle$.

Proof. Proceed by cases on whether all threads are blocked.

Case $\forall i < |\vec{t}| \,.\, \sigma \vdash t_i$ blocked.

By Lemma E.4, $\exists! E^l, r \,.\, e = E^l[r]$. Pick $A^{\sigma,l} = \vec{t}, E^l$.

Otherwise.

By well ordering, let $i \in \mathbb{N}$ be the smallest number such that $\sigma \vdash t_i$ blocked does not hold. It must be that $t_i = \texttt{assert}^k_j\ e'$. If $e' = v$ then pick $A^{\sigma,l_o} = t_0, \ldots, t_{i-1}, \square, t_{i+1}, \ldots, t_n, e$. It cannot be that $e' = \texttt{err}^k_j$ because errors occur only at the top level. Suppose $e' \notin \text{Val}$ and $e' \neq \texttt{err}^k_j$. By Lemma E.4, $\exists! E^l, r \,.\, e' = E^l[r]$. Pick $A^{\sigma,l} = t_0, \ldots, t_{i-1}, \texttt{assert}^k_j\ E^l, t_{i+1}, \ldots, t_n, e$. □

Lemma E.4 (Unique decomposition of expressions). If $e \notin \text{Done}$, then $\exists! E^l, r \,.\, e = E^l[r]$.

Proof. Proceed by induction on e.

Case $e = |e'|^k$.

By induction, $e' = F^l[r]$. Pick $E^l = |F^l|^k$.

Case $e = x \vee e = a$.

Contradicts $e \notin \text{Val}$.

Case $e = \texttt{if}\ e_1\ e_2\ e_3$.

If $e_1 = v_1$ then pick $E^{l_o} = \square, r = \texttt{if}\ v_1\ e_2\ e_3$. It cannot be that $e_1 = \texttt{err}^k_j$ because errors occur only at the top level. Otherwise, by induction $e_1 = F^l[r]$. Pick $E^l = \texttt{if}\ F^l\ e_2\ e_3$.

Otherwise.

Similar to one of the above cases. □

Lemma E.5 (Well-typed expression in a configuration). If $\Psi \vdash \langle A^{\sigma,l}[e], \sigma\rangle$ then $\varnothing, \Psi, \varnothing, l \vdash e : \tau$.

Proof. Proceed by cases on $A^{\sigma,l}$.

Case $A^{\sigma,l} = \vec{s}, \texttt{assert}^k_j\ E^l, \vec{T}, e \wedge \sigma \vdash s_i$ blocked.

Because the configuration is well typed, $\Psi \vdash \texttt{assert}^k_j\ E^l[e]$ and hence $\varnothing, \Psi, \varnothing, j \vdash E^l[e] : \texttt{Bool}$. Apply Lemma E.6.

Case $A^{\sigma,l} = \vec{s}, E^l \wedge \sigma \vdash s_i$ blocked.

Because the configuration is well typed, $\varnothing, \Psi, \varnothing, l_o \vdash E^l[e] : \tau$. Apply Lemma E.7. □

Lemma E.6 (Well-typed expression in an owned evaluation context). If $\varnothing, \Psi, \varnothing, k \vdash E^l[e] : \tau$ then $\varnothing, \Psi, \varnothing, l \vdash e : \tau'$.

Proof. Proceed by induction on E^l.

Case $E^l = \mathtt{mon}_j^{l,k}\ c\ F^{l_o}$**.**

By the typing rule, $\varnothing, \Psi, \varnothing, l \vdash F^{l_o}[e] : \tau$. Apply Lemma E.7.

Case $E^l = \mathtt{mon}_j^{l',k}\ c\ F^{l}$**.**

By the typing rule, $\varnothing, \Psi, \varnothing, l' \vdash F^{l}[e] : \tau$. Apply the inductive hypothesis.

Case $E^l = |F^{l_o}|^{l}$**.**

By the typing rule, $\varnothing, \Psi, \varnothing, l \vdash F^{l_o}[e] : \tau$ and $l = k$. Apply Lemma E.7.

Case $E^l = |F^{l}|^{l'}$**.**

By the typing rule, $\varnothing, \Psi, \varnothing, l' \vdash F^{l}[e] : \tau$ and $l' = k$. Apply the inductive hypothesis.

Otherwise.

Similar to one of the cases above or one of the cases from Lemma E.7. □

Lemma E.7 (Well-typed expression in an unowned evaluation context). If

$$\varnothing, \Psi, \varnothing, l \vdash E^{l_o}[e] : \tau$$

then $\varnothing, \Psi, \varnothing, l \vdash e : \tau'$.

Proof. Proceed by induction on E^{l_o}.

Case $E^{l_o} = \square$**.**

Follows directly by assumption.

Case $E^{l_o} = \mathtt{if}\ F^{l_o}\ e_2\ e_3$**.**

By the typing rule $\varnothing, \Psi, \varnothing, l \vdash F^{l_o}[e] : \mathtt{Bool}$. Apply the inductive hypothesis.

Otherwise.

Similar to one of the cases above. □

Lemma E.8 (Replace a well-typed expression in a configuration). If $\Psi \vdash \langle A^{\sigma,l}[e], \sigma\rangle$ and $\varnothing, \Psi, \varnothing, l \vdash e : \tau$ and $\varnothing, \Psi, \varnothing, l \vdash e' : \tau$, then $\Psi \vdash \langle A^{\sigma,l}[e'], \sigma\rangle$.

Proof. Similar to Lemma E.5. □

Lemma E.9 (Canonical forms). If $\varnothing, \Psi, \varnothing, l \vdash v : \tau$ then $v = ||v'||^{l}$. Additionally, the following hold:

(1) $\tau = \mathtt{Bool}$ implies $v' = \mathtt{tt}$ or $v' = \mathtt{ff}$,
(2) $\tau = \tau_1 \to \tau_2$ implies $v' = \lambda x.e$.

Proof. Proceed by inspection of the typing rules. □

Lemma E.10 (Substitution preserving typing). If $\Gamma[x \leftarrow \tau'], \Psi, \Xi[x \leftarrow k], l \vdash e : \tau$ and $\Gamma, \Psi, \Xi, k \vdash e' : \tau'$, then $\Gamma, \Psi, \Xi, l \vdash [e'/x]e : \tau$.

Proof. Proceed by induction on e. □