Contract System Metatheories à la Carte

A Transition-System View of Contracts

SHU-HUNG YOU, Northwestern University, USA
CHRISTOS DIMOULAS, Northwestern University, USA
ROBERT BRUCE FINDLER, Northwestern University, USA

Over the last 15 years, researchers have studied a wide variety of important aspects of contract systems, ranging from internal consistency (complete monitoring and correct blame) to subtle details about the semantics of contracts combinators (dependency) to the difficulty of efficient checking (avoiding asymptotically bad redundant checking). Although each paper offers essential insights about contract systems, they also differ in inessential ways, making it hard to know how their metatheories combine. Even worse, the metatheories share tremendous tedium in their definitions and proofs, occupying researchers' time with no benefit.

In this paper, we present the idea that higher-order contract systems can be viewed as transition systems and show that this perspective offers an important opportunity for reuse in their metatheories. We demonstrate the value of this perspective by proving representative properties from the literature, and by contributing a new proof establishing that elimination of redundant contract checks can eliminate asymptotic slowdowns. To confirm our claims and encourage the adoption of our ideas, we provide a mechanized development in Agda.

CCS Concepts: • **Theory of computation → Program semantics**; *Proof theory*.

Additional Key Words and Phrases: Higher-order contract, modular metatheory, transition system

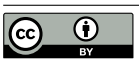
ACM Reference Format:
Shu-Hung You, Christos Dimoulas, and Robert Bruce Findler. 2025. Contract System Metatheories à la Carte: A Transition-System View of Contracts. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 419 (October 2025), 29 pages. https://doi.org/10.1145/3764861

1 Introduction

Contract system metatheory has a long history with many papers establishing a wide range of properties of various contract systems. Although we must not neglect the experience gained by building and using contract system in real-world software, the theoretical investigation of contract systems has had a direct positive influence on the development of our understanding of contracts. The most notable example is the discovery of the correct semantics of dependent contracts. Findler and Felleisen [23]'s original higher-order contract paper included an incorrect semantics for dependency, and it took significant theoretical developments [4, 8, 9, 11, 15, 22, 30, 83] to eventually lead to the correct semantics [15, 19]. Furthermore, this fundamental theory work provides intellectual tools to understand how to generalize the ideas in higher-order contracts, leading to a wide range of innovative contract designs [14, 16, 17, 46–50, 68, 81].

Unfortunately, contract system metatheory is not a panacea. Indeed, the labor involved when proving properties about higher-order contract systems leads researchers to either leave out some features or expend unreasonable amounts of energy repeating similar proofs over and over.

Authors' Contact Information: Shu-Hung You, Northwestern University, Evanston, Illinois, USA, shu-hung.you@eecs.northwestern.edu; Christos Dimoulas, Northwestern University, Evanston, Illinois, USA, chrdimo@northwestern.edu; Robert Bruce Findler, Northwestern University, Evanston, Illinois, USA, robby@cs.northwestern.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART419
https://doi.org/10.1145/3764861

Greenberg [28] articulates the situation well and provides the inspiration for our work, writing that "changing our calculus to have a more interesting notion of blame, like indy semantics [15] or involutive blame labels [80, 79], would be a matter of *pushing a shallow change* in the semantics through the proofs" (emphasis ours).

The crux of the issue is that reasoning about contract systems is both brittle and tedious. Specifically, stating contract system properties relies on an ad hoc collection of information about its operational semantics and proving them is a tedious case analysis on the semantics's reduction rules. As an example, for a minimal formal model like CPCF [13], stating and proving a staple property called correct blame requires: the definition of an annotated semantics; the definition of four well-formedness judgments with thirty-six inference rules that describe an invariant about the semantics; and a lengthy, but straightforward, progress-and-preservation proof for the invariant [15].

To make matters worse, it is unclear how pieces of this process can be reused for different properties or different contract systems. For instance, consider correct blame combined with complete monitoring [19], another staple property of contract systems. The straightforward approach for stating and proving these two properties for one formal model requires repeating all of the above steps in nearly the same way with only a few (but important) differences. To avoid the redundancy, Dimoulas et al. [19] fuse the two properties into one. While the fusion avoids redundancy, it is not always possible. Indeed, for comparing different checking regimes for gradual typing (an application of contracts), Greenman et al. [31] end up defining seven variants of annotated reduction rules and well-formedness judgments, and repeat essentially the same proof three times.

With this paper, we show that the view of *contract systems as transition systems* alleviates the unnecessary labor. Of course, this is not the first paper that points out connections between transition systems and contract checking, or more generally, monitoring program behavior. Indeed, for more than two decades the runtime verification community has been studying the design and implementation of program monitors based on transition systems [41, 7, 42, 40]. Closer to home, Dimoulas et al. [18, 17], and Swords et al. [70, 71] have explored how the ideas from runtime verification translate to the world of (higher-order) contract systems and can underpin contract systems' formal semantics, design, and implementation. This paper, however, is the first one that develops a reusable, modular metatheory for contract systems, where the key enabling step is the view of contract systems as transitions systems.

1.1 The Challenge of Reuse and Our Approach

There is a major challenge to overcome, however: the evaluation of a program with higher-order contracts is tightly coupled with the way that the contract system monitors the evaluation of the program and performs its checks. Because of this tight coupling, the operational semantics of higher-order contract systems weaves the reductions that are related to evaluating the program together with those that are related to monitoring contracts in a complex three-part pattern: some monitor-related reductions set up *proxies* that attach contracts to program values, these proxies act as anchors for other monitor-related reductions that inspect how the rest of the program uses proxied values, and finally, the monitor-related reductions rely on program-related ones both for the evaluation of the predicates that are part of contracts and for the propagation of proxied values. In general, which monitor-related rule fires when, and which other rules it cooperates with (and how) depends on the specifics of the contract system. As a result, augmenting the monitor-related rules to propagate extra, property-specific information needed for reasoning about the contract system seems to leave little space for reuse. To unlock reuse, the paper combines a series of insights:

Transition systems for higher-order contract systems. The reduction semantics of a contract system induces a transition system that captures the distinction between monitor-related and

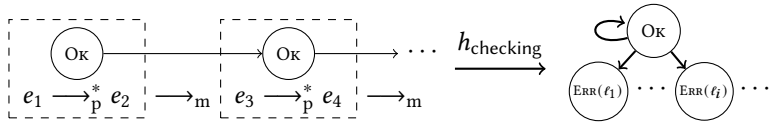


Fig. 1. A view of the reduction semantics of contract systems as transition systems

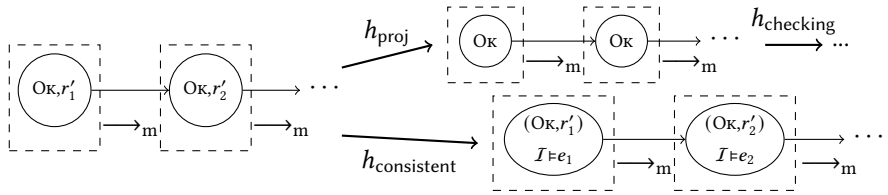


Fig. 2. A view of contract systems as transition systems enables decomposition

program-related reductions. The diagram on the left in Figure 1 illustrates how. Each state of the transition system (each dotted box) contains a set of program terms, namely all of the ones that are reachable only via program-related reduction steps, $e_1 \longrightarrow_p e_2$. Monitor-related reduction steps, in contrast, trigger transitions in the enclosing transition system. Intuitively, the states represent observations that the contract system makes about the evaluation of a program. Importantly, the monitor calculus can simplify the parts of the proof that concern the terms in a single state, as they do not depend on the details of the specific contract system.

Homomorphisms between transition systems are proofs. Proofs of a wide range of properties of contract systems boil down to mapping the information represented by the states of the induced transition system to states of a transition system that satisfies a target property in an obvious manner, by construction. For example, the homomorphism h_{checking} in Figure 1 demonstrates that the contract system on the left does not mask errors, as the transition system on the right has no transition from the Err states to the Ok state. Homomorphism composition also enables proof reuse, as suggested by the arrows in Figure 2. Indeed, we return to these figures, formally defining these transition systems and homomorphisms to establish composite properties of contract systems.

Abstract annotations can represent both contracts and property-related information. Existing proof techniques for properties of contract systems rely on property-specific annotations. These annotations represent meta-information about the workings of the contract system that annotated semantics collect and propagate through evaluation. However, contract systems already come with the "plumbing" needed to associate information with program values and propagate it through evaluation; they have monitor-related reduction rules that attach contracts to values for contract checking. Hence, abstracting over the annotations that these rules manipulate can kill two birds with one stone. For instance, a set of annotations can represent contracts and another proof-related meta-information, such as Dimoulas et al.'s notion of ownership. Similarly, monitor-reduction rules that are abstracted over the way they handle annotations can be specialized to perform contract checking or to propagate ownership. Finally, the two resulting systems can be composed for the modular definition of a contract system that does both.

Working with contract systems: a design recipe. The combination of transition systems, homomorphisms, and abstract annotations suggests a recipe for proving a property about a contract system: (i) organize the contract system into layers of abstract annotations on top of contract-agnostic proxies; (ii) enrich the annotations with information that is needed for proving the property, modifying the monitor-specific reduction rules accordingly; (iii) abstract away unnecessary noise

by constructing a simpler transition system whose behavior captures the desired property; and (iv) prove that the property is preserved by the monitor-specific rules, thereby establishing a homomorphism that transfers the property from the simpler transition system to the original contract system. Even better, properties proven in this manner are amenable to reuse.

1.2 An Overview of the Monitor Calculus

The paper demonstrates the above insights in the context of a formal framework based on the *monitor calculus*. The monitor calculus comes with a fixed set of language features that are commonly found in the literature, but the insights of the paper extend beyond the specifics of these features and we hope this paper provides a roadmap for future contract system metatheories.

The reduction rules and the syntax of the monitor calculus are also fixed but parameterized. Independent of the parameterization, there is only a single reduction sequence starting from any term and thus only a single result it can produce. However, filling in the holes-parameters in the definition of the syntax and reduction rules produces a specific instance of the monitor calculus; each such instance corresponds to a different contract system that records information for stating and proving its specific properties. The holes in the syntax of the monitor calculus have two flavors: global and local. Global holes occupy a global register of the calculus' abstract machine; local holes decorate the contract-specific syntactic forms of the calculus. Global holes record information about the reduction sequence of a term as a whole; local holes record information local to a specific (sub)term. For both forms of holes, each instantiation of the monitor calculus must fill in holes in the reduction rules of the calculus to specify how the rules propagate the corresponding information as the term reduces. The monitor calculus allows this propagation to take place only when the term reduces via a monitor-specific reduction, but neither the global nor the local information can change the reduction sequence of a term except by causing it to terminate early.

To help with proving properties of contract systems, the metatheory of the monitor calculus comes with key lemmas that themselves have holes. Filling in these holes amounts to specifying desired evaluation invariants of the global and local information of an instance of the calculus. The role of the key lemmas of the calculus is to facilitate establishing that the invariants hold, which in turn serve as the bedrock for establishing properties of the instance. Overall, the holes-parameters of our calculus can be filled in to state and prove properties that cover a range of contract systems and their properties, from well-established properties like correct blame and complete monitoring for Findler and Felleisen [23]'s classic contract system to facts about the time complexity of Greenberg [29]'s space-efficient contract checks that have not been proven before.

To make these parameters and their interaction with the monitor calculus precise, we develop the definitions and key lemmas of the monitor calculus by exploring a series of contract systems, showing how their common parts correspond to the fixed pieces of the monitor calculus itself and how their differences correspond to different instantiations of the calculus. Of course, to truly have a monitor calculus that delivers on its promise of enabling proof reuse, the development must be spelled out formally and in full. We have done so, in Agda. You [84] contains Agda code with all of our definitions and proofs of all of the theorems and lemmas.

1.3 Roadmap

Section 2 describes variants of a simple contract system and uses them to introduce the monitor calculus. Sections 3 and 4 explains how homomorphisms provide the backbone of proofs and how the monitor calculus can automate some of the work. Sections 5 and 6 demonstrate the power of the paper's framework through two contract systems with the same language features but different semantics; one checks every contract, while the other collapses redundant checks. Most importantly, these two sections also show how the view of contracts as transition systems enables stating and

$$\begin{array}{rcl} \kappa & ::= & \mathsf{unit}/\mathsf{c} \mid \mathsf{flat}^{\ell}(x.\,e) \mid \kappa_1 \times\!/\mathsf{c}\ \kappa_2 \mid \kappa_1 +\!/\mathsf{c}\ \kappa_2 \mid \mathsf{box}/\mathsf{c}\ \kappa \mid \kappa_a \to\!/\mathsf{c}\ \kappa_r \mid t \mid \mu/\mathsf{c}\ t.\kappa \\ A & ::= & [\kappa_1, \ldots, \kappa_m] \qquad r ::= \textsc{Ok} \mid \textsc{Err}(\ell) \qquad \ell \in \mathsf{Label} \\ \tau & ::= & \mathsf{unit} \mid \mathsf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathsf{Box}\,\tau \mid \tau_a \to \tau_r \mid t \mid \mu t.\tau \end{array}$$

$$\begin{array}{rcl} e & ::= & \mathbf{B}\#A\,\{\,e\,\} \mid \mathsf{proxy}(A,\ e^m) \mid () \mid \mathsf{zero} \mid \mathsf{suc}(e) \mid \mathsf{fold}_{\mathsf{nat}}(e,\ e_z,\ x\,y.\,e_s) \\ & \mid & \langle e_1, e_2\rangle \mid \pi_1(e) \mid \pi_2(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}\,e\,\mathsf{of}\,\{x.e_1 \mid y.e_2\} \\ & \mid & \mathsf{box}(e) \mid \mathsf{unbox}(e) \mid x \mid \lambda x.e \mid e_1\,e_2 \mid \mathsf{unroll}(e) \mid \mathsf{roll}_{\tau}(e) \mid \mathsf{fix}\,x.e \\ v & ::= & () \mid n \mid \langle v_1, v_2\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid \mathsf{roll}_{\tau}(v) \mid \mathsf{box}(v) \mid \lambda x.e \mid \mathsf{proxy}(A,\ e^m) \end{array} \qquad \begin{array}{rcl} e^m & ::= & \mathsf{box}(e) \\ & \mid & \lambda x.e \\ n & ::= & \mathsf{zero} \\ & \mid & \mathsf{suc}(n) \end{array}$$

Fig. 3. The syntax of the monitor calculus instantiated for extended CPCF

proving key properties of the two semantics in a reusable and modular manner, including reuse across semantics. The paper concludes with related work and closing remarks about the significance and future of this work.

2 The Monitor Calculus

We introduce the monitor calculus through the formal models of two variants of Findler and Felleisen [23]'s contracts for higher-order functions. Section 2.1 presents the step-by-step construction of a model for higher-order contracts without blame as an instance of the calculus, and along the way, explains how the monitor calculus's parameters have been instantiated to enable this construction. Section 2.2 composes this first instance with another that equips the model with *blame objects*, and hence, illustrates how the parameters of the calculus make it possible to enrich a model with extra local information in a modular manner.

Throughout the remainder of the paper, we leave out unsurprising parts of definitions, including them only in You [84], our Agda development.

2.1 Syntax and Operational Semantics

As a first example of an instance of the monitor calculus, we consider Dimoulas and Felleisen [13]'s CPCF, which is an extension of call-by-value PCF [54] with features for checking contracts. Because the base monitor calculus is larger than CPCF, our instance will be larger than CPCF. Concretely, the monitor calculus has natural numbers, pairs, disjoint sums, immutable references, and recursive data structures, and so will our extended CPCF.

The calculus has two special features for monitoring the behavior of a program: *boundaries* and *proxies*. Boundaries and proxies both connect a contract to a specific part of the program, explicating which contract should be checked against which part of the program. Syntactically a boundary is written $\mathbf{B}\#A\,\{e\}$, and is a term with a nested term e. In conventional contract systems, boundaries evaluate by evaluating e and then determining if its value satisfies a contract; the monitor calculus generalizes this idea as we shall see presently. A proxy is the corresponding value form, used to indicate that the immediate properties of the value have been checked against its contract but that there may be other properties of the value remaining to be checked; it is written $\mathsf{proxy}(A,\ v)$. Typically contract systems check first-order properties (function arity, properties of base values, etc) as they reduce boundaries. If the checks pass and there are more properties as yet unchecked, the boundaries reduce to proxies (properties of the domain or range of a function, properties of the elements of vectors, etc); if there are no remaining checks, the boundaries simply evaporate.

In the monitor calculus, both boundaries and proxies have a second piece, an annotation A, that sits where the contract would sit in a conventional contract system, which brings us to the first holes-parameters of the monitor calculus. In general, to fill in the holes in the monitor calculus's syntax, each instance must specify the local *annotations* A and the global *register* r. For the CPCF

$$\boxed{\Gamma ::= x_1 : \tau_1, \ldots, x_m : \tau_m} \quad \boxed{\Gamma \vdash e : \tau} \quad \boxed{\Delta :\equiv \{t_1, \ldots, t_n\}} \quad \boxed{\Delta \vdash_{\mathrm{c}} \kappa : \mathrm{Ctc}\ \tau}$$

$$\frac{\vdash A : \mathrm{Ann}_\tau \qquad \vdash e : \tau}{\Gamma \vdash \mathbf{B}\#A\,\{e\} : \tau} \qquad \frac{\vdash A : \mathrm{Ann}_\tau \qquad \vdash e^m : \tau}{\Gamma \vdash \mathrm{proxy}(A,\ e^m) : \tau} \qquad \frac{\vdash_{\mathrm{c}} \kappa_1 : \mathrm{Ctc}\ \tau \quad \cdots \quad \vdash_{\mathrm{c}} \kappa_m : \mathrm{Ctc}\ \tau}{\vdash [\kappa_1, \ldots, \kappa_m] : \mathrm{Ann}_\tau}$$

$$\frac{x : \mathrm{nat} \vdash e : \mathrm{nat}}{\Delta \vdash_{\mathrm{c}} \mathrm{flat}^{\ell}(x.\,e) : \mathrm{Ctc\ nat}} \qquad \frac{\Delta \vdash_{\mathrm{c}} \kappa_a : \mathrm{Ctc}\ \tau_a \qquad \Delta \vdash_{\mathrm{c}} \kappa_r : \mathrm{Ctc}\ \tau_r}{\Delta \vdash_{\mathrm{c}} \kappa_a \to\!\!\!/\mathrm{c}\ \kappa_r : \mathrm{Ctc}\ (\tau_a \to \tau_r)} \qquad \frac{t \in \Delta}{\Delta \vdash_{\mathrm{c}} t : \mathrm{Ctc}\ t}$$

$$\frac{\Delta \vdash_{\mathrm{c}} \kappa_1 : \mathrm{Ctc}\ \tau_1 \qquad \Delta \vdash_{\mathrm{c}} \kappa_2 : \mathrm{Ctc}\ \tau_2}{\Delta \vdash_{\mathrm{c}} \kappa_1 \times\!\!\!/\mathrm{c}\ \kappa_2 : \mathrm{Ctc}\ (\tau_1 \times \tau_2)} \qquad \frac{\Delta, t \vdash_{\mathrm{c}} \kappa : \mathrm{Ctc}\ \tau}{\Delta \vdash_{\mathrm{c}} \mu\!/\mathrm{c}\ t.\kappa : \mathrm{Ctc}\ (\mu t.\tau)}$$

Fig. 4. The contract-related typing rules of the monitor calculus instantiated for extended CPCF

instance, A is a list of contracts, and r is either Ok, indicating there have been no contract violations or $\mathrm{Err}(\ell)$ for some ℓ, indicating there has been a violation of a flat contract the label ℓ. Figure 3 depicts the full syntax definitions for CPCF as an instance of the monitor calculus. The nonterminal τ ranges over types, κ over contracts, e over terms, and e^m over proxy-able terms. Parts of CPCF that are specific to this instance of the monitor calculus are blue; portions of the definitions in black are the same for all instances.

And, already at this point, we can see some of the flexibility of the monitor calculus because the contracts themselves are written as part of the instantiation of the parameter of the monitor calculus, and not directly in the monitor calculus itself. Therefore, we can easily vary the syntax of contracts, and the precise interaction between the contracts and the rest of the calculus.

Beyond boundaries and proxies, the base monitor calculus contains some standard features and, for brevity, we discuss only two of them that play a special role in the metatheory of contract systems. First, isorecursive types enable the expression of commonly used datatypes whose contracts pose interesting challenges when reasoning about blame and space-efficient contracts; we return to these issues in Sections 5 and 6. Second, immutable references are particularly useful for modeling lazy contracts [24]. For example, when a boundary encounters a pair $\langle v_1, v_2 \rangle$, both of its components are immediately checked (cf. the [R-CrCons] rule in Figure 5). In contrast, when a pair stores two references, i.e. $\langle \mathrm{box}(v_1), \mathrm{box}(v_2) \rangle$, a pair of proxies is created after crossing the boundary, delaying the contract checks of the individual components of the pairs until they are accessed via unbox.

The first row of Figure 4 presents a key subset of the monitor calculus's type rules, those for boundaries, proxies and annotations. The rules for boundaries and proxies recursively type the term e and e^m, but in an empty context. The annotation portion and the conclusion of the rules are part of the base monitor calculus and, hence, are generic. They reference the generic annotation A, enabling each instance of the monitor calculus to determine which annotations are appropriate for a particular τ. For this CPCF instance of the monitor calculus, the rightmost rule in that first line says that all of the contracts in a particular A must have the same type and they must match the type in the annotation. The premise of the rule is blue to indicate that it is part of the instantiation of the parameter of the monitor calculus. The A in the conclusion is also blue (written out as a sequence of contracts), as it too can change in different instances.

The remainder of Figure 4 gives the rules for well-typed contracts and follows the design of CPCF with one contract form for each type. For values of type τ, $\mathrm{Ctc}\ \tau$ denotes the type of their contracts. A *flat contract*, i.e., a contract for a natural number, consists of a label ℓ and a predicate e that may use only the identifier x, that binds the value being checked. And, although we write e in the definition of A, we do not actually allow proxies or boundaries, to simplify the formal development. Including boundaries and proxies does not require a fundamental change to the

monitor calculus itself, but does require a complex stratification to ensure that the semantics is well-defined. We include that stratification in the Agda development [84].

Still, the premise of the rule for flat contracts can simply use the typing judgment for terms. Note that the premise of the typing rules for flat contracts is blue, as it is the instance's decision to type the terms with the typing judgment for terms of the monitor calculus. However, the instance is not allowed to change the definition of the typing judgment for terms.

A contract for a pair, $\kappa_1 \times\!\!/\mathsf{c}\ \kappa_2$, is well-typed when both pieces of the contract are well-typed and, similarly, a function contract is well-typed when its domain contract and range contract are well-typed. We omit the typing rules for contracts for disjoint sums and immutable reference cells—they follow the same pattern. Finally, μ/c builds an isorecursive contract, $\mu/\mathsf{c}\ t.\kappa$, which is well-typed when the body κ is well-typed in a context extended with the additional type variable t, mirroring how the recursive type $\mu t.\tau$ binds the type variable t in τ.

To give a faithful semantics to CPCF in our instance, when the term e in $\mathbf{B}\#A\,\{e\}$ evaluates to a flat value like a number, we must check the sequence of contracts inside A. When instead the term e evaluates to a higher-order value, such as a function or a box, a proxy is created and wrapped around the value. The proxy inherits the same contracts A that were on the boundary. All subsequent operations applied to the value are intercepted by the monitor calculus to allow for the latent checks in the annotated list of contracts.

As an example, consider $\mathsf{proxy}([\mathit{isOdd} \to\!\!/\mathsf{c}\ \mathit{isEven}],\ \lambda x.\mathsf{suc}(\mathsf{suc}(x)))$, a function that adds two to its argument with a contract. The function's contract is an arrow contract whose domain and range are two contracts on natural numbers, *isOdd* and *isEven*; we're using those names to stand for $\mathsf{flat}^{\ell_\mathsf{O}}(z.\,\mathsf{fold}_\mathsf{nat}(z,\ 0,\ x\,y.\,\mathit{neg}\,y))$ and $\mathsf{flat}^{\ell_\mathsf{E}}(z.\,\mathsf{fold}_\mathsf{nat}(z,\ 1,\ x\,y.\,\mathit{neg}\,y))$, respectively, where *neg* is $\lambda z.\mathsf{fold}_\mathsf{nat}(z,\ 1,\ x\,y.\,0)$. These two contracts are flat contracts, meaning they represent a predicate on z; flat contracts treat 0 as false and all other naturals as true. To compute with naturals, a term of the form $\mathsf{fold}_\mathsf{nat}(n,\ e_1,\ x\,y.\,e_2)$ iterates over n, returning e_1 if n is zero and e_2 with x bound to the predecessor of n and y bound to the recursive result of $\mathsf{fold}_\mathsf{nat}$ when n is not zero.

Monitor calculus evaluation pairs the global *register* with a term. To get a sense of the semantics, the following sequence of reduction steps shows how the CPCF instance of the monitor calculus reduces the application of the proxied function from above to 13.

$$\begin{aligned}
&\textsc{Ok}, \mathsf{proxy}([\mathit{isOdd} \to\!\!/\mathsf{c}\ \mathit{isEven}],\ \lambda x.\mathsf{suc}(\mathsf{suc}(x)))\ 13 \longrightarrow\\
&\textsc{Ok}, \mathbf{B}\#[\mathit{isEven}]\ \{\ (\lambda x.\mathsf{suc}(\mathsf{suc}(x)))\ (\mathbf{B}\#[\mathit{isOdd}]\ \{\ 13\ \})\ \} \longrightarrow\\
&\textsc{Ok}, \mathbf{B}\#[\mathit{isEven}]\ \{\ (\lambda x.\mathsf{suc}(\mathsf{suc}(x)))\ 13\ \} \longrightarrow^{*} \textsc{Ok}, \mathbf{B}\#[\mathit{isEven}]\ \{\ 15\ \} \longrightarrow \textsc{Err}(\ell_\mathsf{E}), 15
\end{aligned}$$

Because of the proxy, the contract system intervenes at the application and creates two *boundaries*, one around the entire call and another around the argument. CPCF distributes the contract from the proxy to the new boundaries to ensure that the function adheres to the behavior prescribed by its contract. In the last step, since the result of the application does not satisfy the range of $\mathit{isOdd} \to\!\!/\mathsf{c}\ \mathit{isEven}$, CPCF sets the register to $\textsc{Err}(\ell_\mathsf{E})$.

Figure 5 shows the selected definition of the monitor-related reduction relation of the monitor calculus. As foreshadowed in Section 1, the main reduction relation is split into a program-related reduction relation $\longrightarrow_\mathrm{p}$ and the monitor-related reduction relation $\longrightarrow_\mathrm{m}$. The $\longrightarrow_\mathrm{p}$ relation is common for all instances of the calculus and contains standard rules such as the application of functions and the destruction of pairs, hence we focus on $\longrightarrow_\mathrm{m}$ in Figure 5.

The $\longrightarrow_\mathrm{m}$ relation is the part of the semantics of the monitor calculus that can be tailored by an instance. There are a fixed set of rules in $\longrightarrow_\mathrm{m}$; each specific instance tailors them by filling in holes to specify how the As and rs on the left- and right-hand sides of the rule are related to each other. Overall, the rules with holes act as scaffolding for defining different instances of the monitor calculus with the goal of balancing the expressiveness of the instances with the opportunities of

[R-CrNat] $r, \mathbf{B}\#A\,\{\ n\ \} \longrightarrow_{\mathrm{m}} r', n$ where $A = [\kappa_1, \ldots, \kappa_m]$, $(r, r') \in \mathit{checkCtcs}([\kappa_1, \ldots, \kappa_m],\ n)$

[R-CrCons] $r, \mathbf{B}\#A\,\{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} r', \langle\ \mathbf{B}\#A_1\ \{v_1\}\ ,\ \mathbf{B}\#A_2\ \{v_2\}\ \rangle$ where
$A = [\kappa_1 \mathbin{\not\times_{\mathrm{c}}} \kappa_1', \ldots, \kappa_m \mathbin{\not\times_{\mathrm{c}}} \kappa_m']$, $A_1 = [\kappa_1, \ldots, \kappa_m]$, $A_2 = [\kappa_1', \ldots, \kappa_m']$, and $(r, r') \in \{(\textsc{Ok}, \textsc{Ok})\}$

[R-CrRoll] $r, \mathbf{B}\#A\,\{\ \mathrm{roll}_\tau(v)\ \} \longrightarrow_{\mathrm{m}} r', \mathrm{roll}_\tau(\mathbf{B}\#A'\ \{\ v\ \})$ where
$A = [\mu/\mathrm{c}\ t.\kappa_1, \ldots, \mu/\mathrm{c}\ t.\kappa_m]$, $A' = [\kappa_1[(\mu/\mathrm{c}\ t.\kappa_1)/t], \ldots, \kappa_m[(\mu/\mathrm{c}\ t.\kappa_m)/t]]$, and $(r, r') \in \{(\textsc{Ok}, \textsc{Ok})\}$

[R-CrMon] $r, \mathbf{B}\#A\,\{\ v\ \} \longrightarrow_{\mathrm{m}} r', \mathrm{proxy}(A',\ v)$ where
$v = \lambda x.e$ or $v = \mathrm{box}(v')$, $A = [\kappa_1, \ldots, \kappa_m]$, $A' = [\kappa_1, \ldots, \kappa_m]$, and $(r, r') \in \{(\textsc{Ok}, \textsc{Ok})\}$

[R-Prxβ] $r, \mathrm{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} r', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#A_a\ \{v\})\ \}$ where
$A = [\kappa_1 \to_{\mathrm{c}} \kappa_1', \ldots, \kappa_m \to_{\mathrm{c}} \kappa_m']$, $A_r = [\kappa_1', \ldots, \kappa_m']$, $A_a = [\kappa_m, \ldots, \kappa_1]$, and $(r, r') \in \{(\textsc{Ok}, \textsc{Ok})\}$

[R-MrgPrx] $r, \mathbf{B}\#A\,\{\ \mathrm{proxy}(A',\ e^m)\ \} \longrightarrow_{\mathrm{m}} r', \mathrm{proxy}(A'',\ e^m)$ where
$A = [\kappa_1, \ldots, \kappa_l]$, $A' = [\kappa_1', \ldots, \kappa_m']$, $A'' = [\kappa_1', \ldots, \kappa_m', \kappa_1, \ldots, \kappa_l]$, and $(r, r') \in \{(\textsc{Ok}, \textsc{Ok})\}$

$\mathit{checkCtcs}([\kappa_1, \ldots, \kappa_m],\ n)$ is the binary relation over Status such that

(1) $(\textsc{Ok}, \textsc{Ok}) \in \mathit{checkCtcs}([\mathrm{flat}^{\ell_1}(x.\,e_1), \ldots, \mathrm{flat}^{\ell_m}(x.\,e_m)],\ n)$ if for all $1 \le i \le m$, there exists n_i' such that $e_i[n\,/\,x] \longrightarrow_{\mathrm{p}}^* \mathrm{suc}(n_i')$.

(2) $(\textsc{Ok}, \textsc{Err}(\ell_k)) \in \mathit{checkCtcs}([\mathrm{flat}^{\ell_1}(x.\,e_1), \ldots, \mathrm{flat}^{\ell_m}(x.\,e_m)],\ n)$ if $e_k[n\,/\,x] \longrightarrow_{\mathrm{p}}^* \mathrm{zero}$ and for all $1 \le i \le k-1$, there exists n_i' such that $e_i[n\,/\,x] \longrightarrow_{\mathrm{p}}^* \mathrm{suc}(n_i')$.

Fig. 5. The monitor-related reduction relation of the monitor calculus instantiated for CPCF

proof reuse across properties and instances. For example, the bottom portion of Figure 5 shows the monitor-specific rules for CPCF where the parts in blue correspond to the relations that fill in the holes of the rules. These relations capture how CPCF checks contracts and updates the register r and the annotations A as it evaluates a term.

To better understand the instantiation process, first take a look at the second rule, [R-CrCons]. The first line of the rule is black, indicating that no instance can change it. It says that when there is a boundary that contains a pair of values, it reduces to a pair of boundaries that contain the values. The instance, however, controls the second and third lines, effectively specifying a relation that relates the As and rs that appear on the left and right sides of the rule. For the CPCF instance, the contracts in the A on the left of the rule must be pairs (because of the contract type rules in blue from Figure 4) and the A_1, which appears on the right, must be the contracts from the first components of those pair contracts; similarly for the second component in A_2. The annotation portion of this relation has not limited reduction but, in general, there may be reduction sequences that, for specific instances, get stuck because the relation that the instance specifies does not relate certain As or rs. The [R-CrCons] also illustrates this situation with its use of r and r', as they must both be Ok. If an earlier step discovers a contract violation then r becomes $\textsc{Err}(\ell)$ (for some ℓ) and thus this rule does not apply, nor does any other.

The [R-CrNat] rule checks flat contracts using a more complex relation, but one that still can be expressed as a relation on the As and rs that appear in the rule. When a number n crosses a boundary with the contracts $[\kappa_1, \ldots, \kappa_m]$, the *checkCtcs* relation determines the new content r' of the register depending on whether a contract violation has occurred. The relation *checkCtcs*, defined at the bottom of Figure 5, takes a number and a list of flat contracts, and runs the contracts in order using the program-related reduction relation ($\longrightarrow_{\mathrm{p}}$). If the number does not satisfy contract $\mathrm{flat}^{\ell_k}(x.\,e_k)$, *checkCtcs* returns $(\textsc{Ok}, \textsc{Err}(\ell_k))$. Otherwise, it returns $(\textsc{Ok}, \textsc{Ok})$. And, since the terms in the flat contracts do not have boundaries or proxies, the process of checking a contract cannot itself signal a contract violation (but see You [84] for a version that lifts this restriction).

$$A \;::=\; [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle] \qquad b ::= \{\mathsf{pos} = \ell_p;\ \mathsf{neg} = \ell_n\}$$

$$r \;::=\; \textsc{Ok} \mid \textsc{Err}(\ell) \qquad \mathit{blameSwap}(b) :\equiv \{\mathsf{pos} = b.\mathsf{neg};\ \mathsf{neg} = b.\mathsf{pos}\}$$

$$[\text{R-CrNat}]\ r, \mathbf{B}\#A\ \{\ n\ \} \longrightarrow_{\mathrm{m}} r', n \quad A = [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle], (r, r') \in \mathit{checkCtcs}([\kappa_1, \ldots, \kappa_m],\ n)$$

$$[\text{R-Prx}\beta]\ r, \mathsf{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} r', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ \ (\mathbf{B}\#A_a\ \{v\})\ \}\ \text{ where}$$
$$A = [\langle b_1, (\kappa_1 \rightarrow\!/\!c\ \kappa_1')\rangle, \ldots, \langle b_m, (\kappa_m \rightarrow\!/\!c\ \kappa_m')\rangle], A_r = [\langle b_1, \kappa_1'\rangle, \ldots, \langle b_m, \kappa_m'\rangle],$$
$$A_a = [\langle \mathit{blameSwap}(b_m), \kappa_m\rangle, \ldots, \langle \mathit{blameSwap}(b_1), \kappa_1\rangle], \text{ and } (r, r') \in \{(\textsc{Ok}, \textsc{Ok})\}$$

Fig. 6. The instance of the monitor calculus that adds blame

There are two more rules that are worth paying attention to here. The first one is [R-Prxβ]. As discussed through the example reduction sequence earlier, when applying a proxied function annotated with the contracts $[(\kappa_1 \rightarrow\!/\!c\ \kappa_1'), \ldots, (\kappa_m \rightarrow\!/\!c\ \kappa_m')]$ to an argument v, the [R-Prxβ] rule creates two new boundaries: one around the function application and the other around the argument v. The [R-Prxβ] rule further annotates the boundary around the function application with the list of range contracts, $[\kappa_1', \ldots, \kappa_m']$, and the boundary around v with list of the domain contracts, $[\kappa_m, \ldots, \kappa_1]$. Notably, the CPCF instance reverses the order of the domain contracts: since the contracts closer to the end of the list of contracts of the proxied function are the ones ascribed to the function last, their domains are checked first.

The second interesting rule is [R-MrgPrx], which merges annotations when a proxy crosses a boundary that imposes on it a list of new annotations, as it illustrates an important design choice in the monitor calculus, namely that proxies never proxy other proxies. Instead, each instance has an opportunity to merge the contracts in some fashion or possibly even do some checking to see if the merged contracts make sense. In the case of CPCF, we simply concatenate the two lists of contracts, as shown in the figure. As before, the order of concatenation is reversed so that contracts that are attached to the function first are checked first.

The remaining rules decompose and propagate annotations on boundaries and proxies in accordance with the types of the proxies. They follow the same basic ideas, and are omitted here.

2.2 Tracking Errors with Blame Objects

As a second instance of the monitor calculus, this section demonstrates how to add blame to CPCF, capturing the blame assignment mechanism of Racket's contract system [23]. To do so, we instantiate the calculus a second time, this time pairing contracts with new information: blame objects. As before, the labels on flat contracts pinpoint which predicate fails when contract checking detects a violation. The additional blame objects, meanwhile, capture Findler and Felleisen [23]'s blame assignment algorithm. In Section 3, to demonstrate a proof using transition systems, we establish a relationship between labels on flat contracts and blame objects, following Dimoulas et al. [15], whose work first provides a semantic justification for Findler and Felleisen [23]'s blame assignment algorithm. In this section, we simply organize and present the necessary definitions.

To encode both blame assignment and contract checking, we pair each contract with a record that consists of two labels that name a *positive party* and a *negative party*. These two parties correspond to the producer and the consumer, respectively, for any value that crosses a boundary.

The syntax of the new instance is given in the top of Figure 6 showing only the instantiation of the parameters of the monitor calculus' syntax (A and r) — the rest of the calculus' syntax does not change. The new annotations are lists of blame-contract pairs, where the contract is the same as our first instance. The blame object is a record with a pos field and a neg field that each store a label. We use notation $\{\mathsf{pos} = \ell_p;\ \mathsf{neg} = \ell_n\}$ to construct a record and dot to access fields.

$A \in \text{Ann} \quad r \in \text{Reg}$

$e ::= \mathbf{B}\#A\,\{e\} \mid \text{proxy}(A,\ e^m) \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \langle e_1, e_2\rangle \mid \pi_1(e) \mid \pi_2(e) \mid \cdots$

$$\frac{e \longrightarrow_{\text{p}} e'}{r, e \longrightarrow r, e'}\ [\text{R-Prg}] \qquad \frac{r, e \longrightarrow_{\text{m}} r', e'}{r, E[e] \longrightarrow r', E[e']}\ [\text{R-Mon}]$$

$$E[\pi_1(\langle v_1, v_2\rangle)] \longrightarrow_{\text{p}} E[v_1] \qquad E[\pi_2(\langle v_1, v_2\rangle)] \longrightarrow_{\text{p}} E[v_2] \qquad E[(\lambda x.e)\,v] \longrightarrow_{\text{p}} E[e[v\,/\,x]]$$

[R-CrNat] $r, \mathbf{B}\#A\,\{\ n\ \} \longrightarrow_{\text{m}} r', n$ $\quad (A, r, r') \in R_{\text{crnat}}$

[R-CrCons] $r, \mathbf{B}\#A\,\{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\text{m}} r', \langle\ \mathbf{B}\#A_1\,\{v_1\}\,,\ \mathbf{B}\#A_2\,\{v_2\}\ \rangle$ $\quad (A, A_1, A_2, r, r') \in R_{\text{crcon}}$

[R-CrMon] $r, \mathbf{B}\#A\,\{\ v\ \} \longrightarrow_{\text{m}} r', \text{proxy}(A',\ v)\,,\ v$ is $\lambda x.e$ or $\text{box}(v')$ $\quad (A, A', r, r') \in R_{\text{crmon}}$

[R-Prxβ] $r, \text{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\text{m}} r', \mathbf{B}\#A_r\,\{\ (\lambda x.e)\ (\mathbf{B}\#A_a\,\{v\})\ \}$ $\quad (A, A_a, A_r, r, r') \in R_{\text{pxb}}$

[R-MrgPrx] $r, \mathbf{B}\#A\,\{\ \text{proxy}(A',\ e^m)\ \} \longrightarrow_{\text{m}} r', \text{proxy}(A'',\ e^m)$ $\quad (A, A', A'', r, r') \in R_{\text{mrg}}$

The parameters of the calculus are $\mathscr{A} :\equiv (\text{Ann}, \text{Reg})$ and $\mathscr{T} :\equiv (R_{\text{crnat}}, R_{\text{crcon}}, R_{\text{crmon}}, \ldots)$.

Fig. 7. Selected rules of the monitor calculus, $\lambda_{\text{m}}[\mathscr{A};\mathscr{T}]$.

The remainder of Figure 6 shows the new instance's relations for two of the monitor-related reduction rules. Importantly, the blame objects do not influence contract checking; they are merely propagated during evaluation. The [R-CrNat] rule shows this clearly, as the same *checkCtcs* function from Figure 5 is used to check the contracts $\kappa_1, \ldots, \kappa_m$, ignoring the blame objects.

The [R-Prxβ] rule shows how blame objects are propagated; each b_i is paired with the contract $\kappa_i \rightarrow\!\!\!\!/_{\text{c}}\ \kappa_i'$ on the left-hand side. Thus, b_i is paired with κ_i and κ_i', respectively, on the right-hand side. However, since the role of provider and the role of consumer are reversed for function arguments, the blame objects on the boundary around the argument (i.e., $\mathbf{B}\#A_a\,\{v\}$) have to be flipped using $\mathit{blameSwap}(b_i)$ to match the reversal of the roles.

The rest of the rules follow the same pattern and are omitted.

3 Proofs For Monitor Calculus Instances Via Transition Systems

Instances of the monitor calculus give rise to transition systems which in turn, serve as the basis for proving properties of the instances. In general, our proof recipe involves designing a transition system that satisfies a property by construction, and exhibiting a homomorphism to this transition system from the one induced by the instance. To explain this process in precise terms, Section 3.1 describes instances formally, Section 3.2 defines induced transition systems, and Section 3.3 brings in homomorphisms and offers a first proof. Then, in Section 3.4, we discuss a second, more complex proof about the property involving blame objects and labels on flat contracts mentioned briefly in Section 2.2. Section 3.5 demonstrates, through a proof, how transition systems enable a modular metatheory of contract systems; a composite contract system can be projected to simpler ones, and proofs of properties of the projected contract systems can be composed into a proof of a property of the composite contract system. Even better, pieces of these proofs are reusable: one of the projected contract systems is the instance from Section 2.1, and the proof that it does not mask errors, which we describe in Section 3.3, transfers to the composite system.

3.1 Annotation Languages and Instances

Abstracting from the two instances in Section 2.1 and Section 2.2, Figure 7 provides a formal overview of the unistantiated monitor calculus. Specifically, it depicts key elements of its syntax, presents a selection from its reduction rules, and highlights its parameters in blue. Notably, the main reduction relation, $\longrightarrow$, is formed by joining $\longrightarrow_{\text{p}}$ and $\longrightarrow_{\text{m}}$. The missing rules of $\longrightarrow_{\text{p}}$ are standard, as are the evaluation contexts, E.

An instance of the parameters, which we notate as a pair $(\mathscr{A}, \mathscr{T})$, is called an *annotation language*. Each $\mathscr{A}$ consists of a definition of the annotations (A) and the register (r), and each $\mathscr{T}$ consists of a set of named relations, one filling in the side conditions in each of the rules in the calculus's $\longrightarrow_{\mathrm{m}}$.

For the example annotation language in Section 2.1, we say that $\mathscr{A}ctc$ is the blue definitions of A and r in Figure 3, and that $\mathscr{T}c$ is the set of relations written in blue in Figure 5. Similarly, for the annotation language from Section 2.2, we say that $\mathscr{A}bctc$ is the brown definitions in the upper part of Figure 6, and that $\mathscr{T}bc$ is the set of relations in the lower part of the same figure. For better readability, we define the relations in $\mathscr{T}$ implicitly with the assertions "$A =$" in Figures 5 and 6.

In principle, the blue premise in the upper-right rule in Figure 4 also deserves a place in each annotation language $(\mathscr{A}, \mathscr{T})$. However, all other annotations in the paper either have trivial types or ones similar to Figure 4, so we omit them hereafter. Also, from this point forward, we work exclusively with well-typed terms without explicitly naming the typing derivations. Finally, given an annotation language, we use λ_{m} to *instantiate* the monitor calculus, and obtain an instance. The two instances from Section 2 are written $\lambda_{\mathrm{m}}[\mathscr{A}ctc; \mathscr{T}c]$ and $\lambda_{\mathrm{m}}[\mathscr{A}bctc; \mathscr{T}bc]$.

3.2 Induced Transition Systems

Each instance $\lambda_{\mathrm{m}}[\mathscr{A}; \mathscr{T}]$ induces a transition system. Its states are pairs consisting of the register r of the instance, and a set of (closed) terms of the monitor calculus that all reduce to each other only via program-related reductions, $\longrightarrow_{\mathrm{p}}$. Two states are connected with a transition when any of the terms in one state can reduce to a term in the other via the monitor-related reduction relation, $\longrightarrow_{\mathrm{m}}$. To formalize induced transition systems, we define the equivalence relation $\sim_{\mathrm{p}}$ as the reflexive, symmetric, and transitive closure of $\longrightarrow_{\mathrm{p}}$ and we notate the equivalence class of e as $[e]_{\mathrm{p}}$.

Definition 3.1. We write $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ to denote the *induced transition system* of an instance $\lambda_{\mathrm{m}}[\mathscr{A}; \mathscr{T}]$. Its set of states is $\{\ (r, [e]_{\mathrm{p}}) \mid \text{any term } e \text{ and register } r\ \}$. For any two states $(r_1, [e_1]_{\mathrm{p}})$ and $(r_2, [e_2]_{\mathrm{p}})$, there is a transition if and only if there exists E, e_1', and e_2' such that $E\left[e_1'\right] \sim_{\mathrm{p}} e_1$, $E\left[e_2'\right] \sim_{\mathrm{p}} e_2$ and $r_1, e_1' \longrightarrow_{\mathrm{m}} r_2, e_2'$.

As a concrete example of an induced transition system, the diagram on the left of Figure 1 visualizes $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc; \mathscr{T}c]$, which is induced by $\lambda_{\mathrm{m}}[\mathscr{A}ctc; \mathscr{T}c]$ from Section 2.1. In particular, the states of the transition system pair the status Ok or $\textsc{Err}(\ell_k)$ with the equivalence class $[e]_{\mathrm{p}}$.

3.3 Using Homomorphisms from Induced Transition Systems, a First Proof

Our primary approach to proving facts about contract systems is via establishing homomorphisms between transition systems. Typically, we construct a transition system that obviously has some desirable property, just by the way it is defined. Then, we exhibit a homomorphism from the induced transition systems of an instance to the one with the desirable property, proving that the instance also possesses the property.

As a first example, we prove that CPCF never masks errors, as promised in Section 1. Although this is a simple property, it serves to illustrate the key ideas of proofs done with the monitor calculus. Specifically, we show that the register r of $\lambda_{\mathrm{m}}[\mathscr{A}ctc; \mathscr{T}c]$ never goes from $\textsc{Err}(\ell)$ to Ok by demonstrating a homomorphism from the induced transition system $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc; \mathscr{T}c]$ to the transition system on the right-hand side of Figure 1.

To start, we state the formal definition of transition system homomorphisms from Rutten [57]. A homomorphism is a mapping h from the states of a source transition system to the states of a destination transition system that satisfies two properties. First, it must preserve transitions, i.e., if there is a transition between s_1 and s_2, then there must be a transition between $h(s_1)$ and $h(s_2)$. Second, it must reflect transitions, i.e., if there is a transition between $h(s_1)$ and some state t_2 in

$$\boxed{b \vdash \kappa \textbf{ consistent}} \quad \boxed{p ::= + \mid -} \quad \boxed{\delta :\equiv \{t_1 : p_1, \ldots, t_m : p_m\}} \quad \boxed{\delta \vdash^p \kappa \textbf{ sgn}}$$

$$\frac{b.\text{pos} = \ell}{b \vdash \text{flat}^\ell(x.\, e) \textbf{ consistent}}\ [\text{C-Nat}] \qquad \frac{\mathit{blameSwap}(b) \vdash \kappa_a \textbf{ consistent} \quad b \vdash \kappa_r \textbf{ consistent}}{b \vdash (\kappa_a \rightarrow\!\!/_c\, \kappa_r) \textbf{ consistent}}\ [\text{C-App}]$$

$$\frac{\delta(t) = p}{\delta \vdash^p t \textbf{ sgn}} \qquad \frac{\delta, t : p \vdash^p \kappa \textbf{ sgn}}{\delta \vdash^p (\mu/_c\, t.\kappa) \textbf{ sgn}} \qquad \frac{\delta \vdash^- \kappa_a \textbf{ sgn} \quad \delta \vdash^+ \kappa_r \textbf{ sgn}}{\delta \vdash^+ (\kappa_a \rightarrow\!\!/_c\, \kappa_r) \textbf{ sgn}} \qquad \frac{\delta \vdash^+ \kappa_a \textbf{ sgn} \quad \delta \vdash^- \kappa_r \textbf{ sgn}}{\delta \vdash^- (\kappa_a \rightarrow\!\!/_c\, \kappa_r) \textbf{ sgn}}$$

$$\frac{\vdash e \textbf{ econsistent} \quad b_1 \vdash \kappa_1 \textbf{ consistent} \quad \cdots \quad b_m \vdash \kappa_m \textbf{ consistent} \quad \delta \vdash^p \kappa_1 \textbf{ sgn} \quad \cdots \quad \delta \vdash^p \kappa_m \textbf{ sgn}}{\vdash \mathbf{B}\#[\langle b_1, \kappa_1 \rangle, \ldots, \langle b_m, \kappa_m \rangle]\ \{e\} \textbf{ econsistent}}\ [\text{E-B}]$$

Fig. 8. Consistency of blame objects and the labels on contracts

the destination transition system, then there must be some state s_2 in the source transition system such that $h(s_2) = t_2$ and a transition in the source transition system from s_1 to s_2.

When the destination transition system is explicitly designed to demonstrate that certain bad transition can never happen (as in our non-masking example) then we need only transition preservation, not reflection. Accordingly, we define a *weak homomorphism* as a mapping h between the states of the transition systems that satisfies preservation, but might not satisfy reflection.

Next, we formally define the destination transition system through $\preccurlyeq$, a relation on the set $\{\text{Ok}\} \cup \{\text{Err}(\ell) \mid \text{any label } \ell\}$. $\preccurlyeq$ is the smallest reflexive relation such that $\text{Ok} \preccurlyeq \text{Err}(\ell)$. Then, our transition system that, prevents masking by construction, has a transition between any two states r_1, r_2, iff $r_1 \preccurlyeq r_2$ and r_1, r_2 are not both $\text{Err}(\ell)$.

With the two transition systems defined and the notion of a homomorphism in hand, establishing that CPCF never masks errors is straightforward. We define the function h_{checking} as $(r, [e]_{\text{p}}) \longmapsto r$. Then, we complete the proof by verifying that h_{checking} is well-defined and a weak homomorphism.

It is worth noting that despite this being a simple property with a proof that is easily obtained otherwise, our approach still manages to save work by automating away the traditional induction over the evaluation contexts and limiting the tedious inspection of the rules to those in the $\longrightarrow_{\text{m}}$ relation. Of course these may seem like small benefits for a pretty simple property and contract system, but, as we discuss further on, our approach unlocks further savings by enabling the reuse of the proof of non-masking as-is when proving composite properties of the same or even different contract systems. In comparison, the standard approach requires defining new formal models that support proving all the properties together through either the usual subject reduction avenue, or through ad-hoc (bi)simulation arguments.

3.4 Using the Annotations of a Monitor Calculus Instance, a Second Proof

Beyond global information, i.e., what the register of an instance records, most properties of contract systems also concern *local information*, that is, information recorded in the annotations A of an instance. As an example, we return to $\lambda_{\text{m}}[\mathscr{A}bctc; \mathscr{T}bc]$, the instance described in Section 2.2, and discuss a consistency property between labels on flat contracts and blame objects. Consistency dictates that when blame objects and labels on flat contracts are appropriately matched up for a term, they remain in sync during its evaluation.

Naturally, we formalize consistency as a judgment that imposes constraints on the annotations of $\lambda_{\text{m}}[\mathscr{A}bctc; \mathscr{T}bc]$. Intuitively, a blame object κ is associated with the term it protects. Hence, for flat contracts, which check values *produced* by the protected term, the positive part of the corresponding blame object, $b.\text{pos}$, should match the label on the contract. After all, if the check fails, it is the producer of the value that should be blamed. The top row of Figure 8 turns this intuition into a

pair of inference rules. For a flat contract, rule [C-Nat] checks that b.pos equals the label on the contracts . For a function contract, rule [C-App] recursively checks the domain contract and the range contract, after swapping the fields of the blame object for the domain contract to reflect that the argument of a function is produced by the function's consumer.

There is an issue for consistency, having to do with recursive contracts. Intuitively, in a recursive contract $\mu/\mathrm{c}\, t.\kappa$, the t can appear in a negative position, which means there is no assignment of labels to flat contracts that stay in sync with blame objects as the μ unfolds.

Concretely, consider the recursive contract $\mu/\mathrm{c}\, t.(t \to/\mathrm{c}\ \mathit{isEven})$, where *isEven* is as defined in Section 2.1. Importantly, *isEven* has label ℓ_{E}. Assume that $b \vdash \mu/\mathrm{c}\, t.(t \to/\mathrm{c}\ \mathit{isEven})$ **consistent** holds for some blame object b whose two components, b.pos and b.neg, are distinct. Thus, we know that $b.\mathrm{pos} = \ell_{\mathsf{E}}$ must hold. During evaluation, due to [R-CrRoll] from Figure 5, consistency preservation means that we need to prove $b \vdash ((\mu/\mathrm{c}\, t.(t \to/\mathrm{c}\ \mathit{isEven})) \to/\mathrm{c}\ \mathit{isEven})$ **consistent**. In this contract, however, the *domain contract* is again $\mu/\mathrm{c}\, t.(t \to/\mathrm{c}\ \mathit{isEven})$. Hence, we need to show $b.\mathrm{neg} = \ell_{\mathsf{E}}$, which would be a contradiction.[1]

Accordingly, we define $\delta \vdash^{p} \kappa$ **sgn** that limits all contract variables in recursive contracts to positive positions, similar to the standard subtyping restriction for recursive types [6, 3, 53, Chapter 21].

To complete the definition of consistency, we also have to lift $b \vdash \kappa$ **consistent** and $\delta \vdash^{p} \kappa$ **sgn** to all term forms of the instance. The rule for the boundary form, rule [E-B], is the only noteworthy rule as it connects consistency for contracts (in blue) with consistency for the terms of the monitor calculus (in black). Looking ahead, the rest of our development eliminates altogether the need for the black portions to be defined explicitly.

With these definitions in hand, we wish to prove that consistency is preserved. To do so, we first define a transition system that captures consistency by construction. As before, the states of the transition system are pairs, but where the set of terms in the pair consists only of consistent ones. That is, the states are $(r, [e]_c)$ where $[e]_c$ are elements of the partition induced by $\sim_p$, but with only terms that are consistent. Clearly, this transition system preserves consistency.

To establish that consistency holds for $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{bctc}; \mathscr{T}\mathit{bc}]$, we need to construct a homomorphism from the induced transition system of the instance to the one that satisfies consistency by construction. There is one wrinkle, however, as there is no homomorphism from the first to the second because the induced transition system contains inconsistent terms. Of course, this is not a problem, as we are not interested in such terms.

Rather than directly using $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{bctc}; \mathscr{T}\mathit{bc}]$, we resort to its subsystem whose states are a subset of the states in $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{bctc}; \mathscr{T}\mathit{bc}]$ that are closed under the original transition relation. Then to complete the proof we show that there is a homomorphism from the subsystem to the transition system that satisfies consistency by construction.

At this point, the definitions required to complete the proofs may not seem like a net win for our approach, as showing that the subsystem is closed under the original transition relation and that the homomorphism exists amounts to the same work that would have to be done with a standard subject reduction approach. There are two ways, however, that these additional definitions save us work. The first comes from couching these definitions in homomorphisms which enables proof reuse as we show in the next subsection. The other comes from our framework that provides scaffolding for simplifying the proofs and is the subject of the section after that.

[1] Studying this issue led the first author to uncover a bug in the Racket contract system, which is now fixed.

3.5 Transition Systems Enable Proof Reuse

Beyond proving specific results, homomorphisms between transition system also offer an opportunity for proof reuse. Specifically, we can prove composite properties of an instance by putting together existing proofs, including proofs of properties of different instances.

As a concrete example, consider $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$ and the proof that it both preserves consistency and does not mask errors. Clearly, we can prove the two properties independently as discussed in Sections 3.3 and 3.4 and then put them together. However, for the non-masking property, Section 3.3 actually presents its proof for $\lambda_m[\mathscr{A}ctc; \mathscr{T}c]$, not $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$. Thankfully, the view of contracts as transition systems helps alleviate the labor of repeating the same proof for $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$.

As mentioned in Section 1, we can connect the induced transition systems $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc; \mathscr{T}c]$ and $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}bctc; \mathscr{T}bc]$, and transfer the non-masking property from the first to the second. Figure 2 demonstrates how, graphically. Specifically, h_{proj} is a weak homomorphism that turns $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}bctc; \mathscr{T}bc]$ to $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc; \mathscr{T}c]$, while h_{checking} is the homomorphism that proves the non-masking property of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc; \mathscr{T}c]$. The composition of the two (weak) homomorphisms $h_{\mathsf{checking}} \circ h_{\mathsf{proj}}$ sends the induced transition system, $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}bctc; \mathscr{T}bc]$, to the simple transition system on the right of Figure 1, and, switching back to instances, we see that $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$ does not mask errors.

The key to the construction of h_{proj} and similar weak homomorphisms is the concept of instance projection. Intuitively, we can transform $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$ to $\lambda_m[\mathscr{A}ctc; \mathscr{T}c]$ by simply erasing blame objects. Formally, instance projections relate a composite instance to its constituents.

Definition 3.2 (Instance Projection). Let $\lambda_m[\mathscr{A}; \mathscr{T}]$ and $\lambda_m[\mathscr{A}'; \mathscr{T}']$ be two instances where $\mathscr{A}$ is $(\mathsf{Ann}, \mathsf{Reg})$ and $\mathscr{A}'$ is $(\mathsf{Ann}', \mathsf{Reg}')$. An *instance projection* from $\lambda_m[\mathscr{A}; \mathscr{T}]$ to $\lambda_m[\mathscr{A}'; \mathscr{T}']$ is a pair of functions $\pi_A : \mathsf{Ann} \to \mathsf{Ann}'$ and $\pi_R : \mathsf{Reg} \to \mathsf{Reg}'$ that map annotations and register contents of $\mathscr{A}$ to that of $\mathscr{A}'$ such that for any relation R in $\mathscr{T}$, and for any annotations and registers related by R, their images under $\pi_A(-)$ and $\pi_R(-)$ are also related by the corresponding relation R' in $\mathscr{T}'$.

The last condition entails that, given an instance projection (π_A, π_R) from $\lambda_m[\mathscr{A}; \mathscr{T}]$ to $\lambda_m[\mathscr{A}'; \mathscr{T}']$, when $r_1, e_1 \longrightarrow_m r_2, e_2$ in $\lambda_m[\mathscr{A}; \mathscr{T}]$ then $\pi_R(r_1), \pi_{\mathsf{expr}}(e_1) \longrightarrow_m \pi_R(r_2), \pi_{\mathsf{expr}}(e_2)$ in $\lambda_m[\mathscr{A}'; \mathscr{T}']$ where $\pi_{\mathsf{expr}}(-)$ recursively applies $\pi_A(-)$ to the given term. As an example, the instance projection from $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$ to $\lambda_m[\mathscr{A}ctc; \mathscr{T}c]$ consists of the π_A that erases blame objects from the annotations of $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$, and an identity π_R for the register.

The important property of instance projections is that they give rise to weak homomorphisms from their source to their target instance, exactly like h_{proj} from Figure 2. It is worth explaining here though, the significance of the adjective "weak" in the context of the monitor calculus. An instance of the monitor calculus can have an arbitrary relation in each of the monitor-related reduction rules, e.g., the blue text in Figure 5. In many situations, these relations merely move information from one annotation to another, but the relations may also block reduction, in the simplest case by being empty. Accordingly, in the general case, the second condition in the definition of a homomorphism, about reflecting transitions, may not hold. Intuitively, this means that there may be transitions in the destination transition system that are absent in the source transition system. In all of the properties we have established using the monitor calculus, e.g., the non-masking property of $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$ (but also properties we discuss in following sections), the theorems we are interested in seek to show that certain bad transitions never happen, and thus weak homomorphisms are sufficient.

Theorem 3.3. *Let $(\mathscr{A}, \mathscr{T})$ and $(\mathscr{A}', \mathscr{T}')$ be two annotation languages such that there is an instance projection (π_A, π_R) from $\lambda_m[\mathscr{A}; \mathscr{T}]$ to $\lambda_m[\mathscr{A}; \mathscr{T}']$, then $h : (r, [e]_p) \longmapsto (\pi_R(r), [\pi_{\mathsf{expr}}(e)]_p)$ mapping $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ to $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}'; \mathscr{T}']$ is a weak homomorphism. Here, $\pi_{\mathsf{expr}}(-)$ is the function that recursively applies $\pi_A(-)$ to all annotations in a given term.*

Thus, due to Theorem 3.3, just by constructing the instance projection that erases blame objects in $\lambda_m[\mathscr{A}bctc; \mathscr{T}bc]$, we obtain a proof of its non-masking property, by reusing the proof for $\lambda_m[\mathscr{A}ctc; \mathscr{T}c]$.

4 Simplifying Proofs about Monitor Calculus Instances via Invariants

Building on the monitor calculus, we provide scaffolding that simplifies proofs based on transition systems like the ones in Section 3.3, Section 3.4, and Section 3.5. First, in Section 4.1, we introduce the notion of an *invariant* that captures the details of a specific property of a contract system. The invariant specifies the property in terms of the annotations and the register of an instance so, in Section 4.2, we introduce the satisfaction relation that lifts an invariant to the instance's terms. With the satisfaction relation in hand, Section 4.3 presents a systematic way for constructing a transition system that satisfies an invariant by construction. Given such an *invariant-satisfying* transition system, the metatheory of the monitor calculus provides additional help for transferring an invariant to the induced transition system of an instance, thereby establishing that the corresponding contract system has the desired property.

4.1 Abstracting out Properties of Monitor Calculus Instances as Invariants

Properties of contract systems typically concern three aspects of the information encoded in the semantics: global information (facts about the register r of an instance), local information (facts about its annotations A), and contextual information (facts about the terms around the boundaries and proxies that contain specific annotations). The invariant is a five tuple whose first two components cover the global information and whose last three cover both the local and contextual information.

Definition 4.1 (Invariant). An *invariant* $\mathcal{I}$ for an instance $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ consists of five components:

$\mathbb{R}$	A predicate over the register.
$\preccurlyeq$	A preorder over the set $\{\, r \mid \mathbb{R}(r) \text{ holds} \,\}$.
$\mathscr{J}$	Indices, an arbitrary (countable) set.
$\lhd_A$	A binary relation over $\mathscr{J}$ for each A.
$\mathbb{A}[\![\, A^{j,j'} \,]\!]$	A function interpreting each $A^{j,j'}$ as a proposition. [2]

The first component of the invariant restricts the shape of registers; the second dictates how they can change. As an example, the non-masking property corresponds to an invariant $\mathcal{I}_{nonmsk}$ where $\mathbb{R}$ imposes no restrictions, and $\preccurlyeq$ is the preorder from the first step of the proof in Section 3.3.

The $\mathbb{A}$ function describes properties of the annotations of boundaries and proxies that hold for all states, capturing invariants of the local information encoded by the annotations. It takes three arguments, an annotation and two js. The j's capture contextual information, so let's ignore them to start, as not all properties require them. For example, the blue parts of the consistency in Figure 8 is wired into an invariant $\mathcal{I}_{cons}$ via a definition of $\mathbb{A}$ that returns the consistency judgment.

Besides global and local information, in order to accommodate the varied requirements of different properties, invariants also need access to the third kind of information: *contextual information*. Looking ahead, Section 5 proves complete monitoring, which requires establishing a well-formedness property similar to consistency, but with a twist: the blame objects on a boundary or proxy term also need to match with blame objects in surrounding boundaries and proxies. The first component for context information is an index set $\mathscr{J}$ that represents the relevant information about the contexts. The second one is a family of binary relations $\lhd_A$ that, given an annotation A, relates pairs of elements j and j' of $\mathscr{J}$. Intuitively, when $j \lhd_A j'$, which we indicate with $A^{j,j'}$, j represents a surrounding context of a boundary or proxy with annotation A, and index j' represents a new context for its sub-term. Put differently, $A^{j,j'}$ describes a constraint between annotations and the contexts they can appear in. And, thus, the $\mathbb{A}$ actually receives these triples.

[2] For example, in our Agda mechanization [84], $\mathbb{A}$ returns a type, i.e. a value of type Set_0.

$\boxed{\mathcal{I} \vDash^j e}$

$\mathcal{I} \vDash^j \mathbf{B}\#A\,\{e\}$	iff	there exists j' such that $j \lhd_A j'$, $\mathbb{A}[\![A^{j,j'}]\!]$ and $\mathcal{I} \vDash^{j'} e$
$\mathcal{I} \vDash^j \text{proxy}(A,\ e^m)$	iff	there exists j' such that $j \lhd_A j'$, $\mathbb{A}[\![A^{j,j'}]\!]$ and $\mathcal{I} \vDash^{j'} e^m$
$\mathcal{I} \vDash^j ()$		
$\mathcal{I} \vDash^j \langle e_1, e_2 \rangle$	iff	$\mathcal{I} \vDash^j e_1$ and $\mathcal{I} \vDash^j e_2$
$\mathcal{I} \vDash^j e\ e_a$	iff	$\mathcal{I} \vDash^j e$ and $\mathcal{I} \vDash^j e_a$

Fig. 9. Selected rules of the satisfaction relation where $\mathcal{I} :\equiv (\mathscr{J}, \lhd_A, \mathbb{R}, \preccurlyeq, \mathbb{A})$.

Given the formal definition of invariants, we revisit the definition of the non-masking and consistency properties as the invariants $\mathcal{I}\mathit{nonmsk}$ and $\mathcal{I}\mathit{con}$, respectively.

Definition 4.2. The invariant $\mathcal{I}\mathit{nonmsk}$ for $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{ctc}; \mathscr{T}c]$ equals $(\mathscr{J},\ \{\lhd_A\}_A,\ \mathbb{R},\ \preccurlyeq,\ \mathbb{A})$ where $\mathscr{J} :\equiv \top$, $\lhd_A$ relates all indices, $\preccurlyeq$ is the smallest reflexive relation including $\textsc{Ok} \preccurlyeq \textsc{Err}(\ell)$ for all ℓ, and both $\mathbb{R}(\cdot)$ and $\mathbb{A}$ always hold.

Definition 4.3. The invariant $\mathcal{I}\mathit{con}$ for $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{bctc}; \mathscr{T}\mathit{bc}]$ equals $(\mathscr{J},\ \{\lhd_A\}_A,\ \mathbb{R},\ \preccurlyeq,\ \mathbb{A})$ where $\mathscr{J} :\equiv \top$, $\lhd_A$ relates all indices, $\preccurlyeq$ and $\mathbb{R}(\cdot)$ always hold, and

$$\mathbb{A}[\![\ [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]^{j,j'}\ ]\!] :\equiv\ (b_1 \vdash \kappa_1\ \mathbf{consistent}) \wedge \cdots \wedge (b_m \vdash \kappa_m\ \mathbf{consistent}) \\ \wedge\ (\vdash^+ \kappa_1\ \mathbf{sgn}) \wedge \cdots \wedge (\vdash^+ \kappa_m\ \mathbf{sgn})$$

4.2 The Satisfaction Relation

The *satisfaction relation* lifts an invariant $\mathcal{I}$ to the terms of an instance, and hence, it bridges the gap between the proof-specific $\mathcal{I}$ and the generic pieces of the monitor calculus. Formally, the satisfaction relation is an indexed binary relation $\mathcal{I} \vDash^j e$ that determines whether all annotations appearing in e satisfy $\mathcal{I}$ in context j. As an example, and to connect to the proofs in Section 3, we can replace $\vdash e\ \mathbf{econsistent}$ from Figure 8 with $\mathcal{I}\mathit{con} \vDash^{()} e$.

Figure 9 presents the interesting cases of $\mathcal{I} \vDash^j e$. The boundary case (and the proxy case, respectively) is the workhorse of the satisfaction relation. Ignoring j and $j \lhd_A j'$ for the moment, for $\mathcal{I} \vDash^j \mathbf{B}\#A\,\{e\}$ to hold, the satisfaction relation extracts the $\mathbb{A}$ function from $\mathcal{I}$ and demands $\mathbb{A}[\![A^{j,j'}]\!]$ to hold, in addition to the requirement that $\mathcal{I} \vDash^{j'} e$ recursively holds. The rest of the cases simply recur over the sub-terms of the given term, e.

Returning to the indices, their goal is to thread contextual information through the pieces of a term. For instance, consider the boundary case. $\mathcal{I} \vDash^j \mathbf{B}\#A\,\{e\}$ asks for the existence of $j' \in \mathscr{J}$ such that $j \lhd_A j'$ and $\mathcal{I} \vDash^{j'} e$, and therefore, it asks for two contexts related by $\lhd_A$ such that j describes the context of the boundary term and j' describes the context the boundary "creates" for its sub-terms. Hence, the satisfaction relation pipes context information through terms so that $\mathcal{I}$ can use it to check contextual properties through the appropriate design of $\mathscr{J}$ and $\lhd_A$. In Section 5 we put this idea to work to establish complete monitoring. As a simpler example, if we design an invariant with $\mathscr{J} :\equiv \{n \mid 0 \le n < k\}$ and $n + 1 \lhd_A n$ for all n, the satisfaction relation entails that all boundaries and proxies in a term can appear in at most k enclosing boundaries or proxies.

4.3 Systematic Construction of Invariant-Satisfying Transition Systems

Given an invariant $\mathcal{I}$ for an instance $\lambda_{\mathsf{m}}[\mathscr{A}; \mathscr{T}]$, the satisfaction relation $\mathcal{I} \vDash^j e$ is sufficient for constructing an invariant-satisfying transition system $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$. $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$ is similar to $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ except that its states are restricted based on $\mathcal{I}$. Formally, the definition of $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$ follows that of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ except that it only uses registers that satisfy $\mathbb{R}(-)$ from $\mathcal{I}$, and refers to

alternative equivalence classes which we notate as $[e]_{\mathcal{I},j}$. Specifically, for any (closed) term e such that $\mathcal{I} \vDash^j e$ holds, $e' \in [e]_{\mathcal{I},j}$ iff $e' \sim_{\mathrm{p}} e$ *and* $\mathcal{I} \vDash^j e'$ where $\sim_{\mathrm{p}}$ is as defined in Section 3.2.

Definition 4.4. For any $(\mathscr{A}, \mathscr{T})$, any $\mathcal{I} :\equiv (\mathscr{J}, \{\lhd_A\}_A, \mathbb{R}, \preccurlyeq, \mathbb{A})$, and any index $j \in \mathscr{J}$, we write $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$ to denote the invariant-satisfying transition system constructed using $\mathcal{I}$. Its set of states is $\{\ (r, [e]_{\mathcal{I},j}) \mid \text{any } e \text{ and } r \text{ such that } \mathbb{R}(r) \text{ and } \mathcal{I} \vDash^j e\ \}$. For any two states $(r_1, [e_1]_{\mathcal{I},j})$ and $(r_2, [e_2]_{\mathcal{I},j})$, there is a transition if and only if there exists E, e_1', and e_2' such that $E\left[e_1'\right] \in [e_1]_{\mathcal{I},j}$, $E\left[e_2'\right] \in [e_2]_{\mathcal{I},j}$ and $r_1, e_1' \longrightarrow_{\mathrm{m}} r_2, e_2'$.

The construction of $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$ points to a way to transfer $\mathcal{I}$ to $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ by simply mapping states generated by e and an r in $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ to the ones generated by the same e and r in $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$, ignoring the states where $\mathcal{I}$ does not hold. To do so, however, we need two requirements for invariants: monotonicity and soundness which, essentially, ask that the transitions of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ "preserve" $\mathcal{I}$.

Definition 4.5 (Monotonic Invariant). $\mathcal{I} :\equiv (\mathscr{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{R}, \preccurlyeq, \mathbb{A})$ is *monotonic* if for any r_1,r_2,e_1, e_2, if $\mathbb{R}(r_1)$ holds, $r_1, e_1 \longrightarrow_{\mathrm{m}} r_2, e_2$ and $\mathcal{I} \vDash^j e_1$ then $\mathbb{R}(r_2)$ holds and $r_1 \preccurlyeq r_2$.

Definition 4.6 (Sound Invariant). $\mathcal{I} :\equiv (\mathscr{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{R}, \preccurlyeq, \mathbb{A})$ is *sound* if for any r_1, r_2, e_1, e_2 such that $\mathbb{R}(r_1)$ and $\mathbb{R}(r_2)$ both hold, if $r_1 \preccurlyeq r_2$, $r_1, e_1 \longrightarrow_{\mathrm{m}} r_2, e_2$ and $\mathcal{I} \vDash^j e_1$ then $\mathcal{I} \vDash^j e_2$.

Equipped with the precise definitions of monotonicity and soundness, we construct a homomorphism from a subsystem of the induced transition system to the invariant-satisfying system.

Theorem 4.7. *Let $\mathcal{I} :\equiv (\mathscr{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{R}, \preccurlyeq, \mathbb{A})$ be a monotonic and sound invariant. Assume that for some r_0, e_0 and j, both of $\mathbb{R}(r_0)$ and $\mathcal{I} \vDash^j e_0$ hold. Let $\mathcal{T}$ be the transition system obtained by restricting the states of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ to those reachable from $(r_0, [e_0]_p)$, i.e. $\mathcal{T}$ is the smallest subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ containing $(r_0, [e_0]_p)$. Then, the map $(r, [e]_p) \longmapsto (r, [e]_{\mathcal{I},j})$ is a well-defined function from $\mathcal{T}$ to $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$, and it is a homomorphism.*

As an illustration of how Theorem 4.7 simplifies proofs about contract systems, we follow up on the non-masking property from Section 3.3 and the consistency property from Section 3.4. For the non-masking property, by Theorem 4.7, we only need to prove that $\mathcal{I}\mathit{nonmsk}$ is monotonic and sound. But $\mathbb{R}$ and $\mathbb{A}$ in $\mathcal{I}\mathit{nonmsk}$ always hold, so it boils down to showing that $r_1 \preccurlyeq r_2$ whenever $r_1, e_1 \longrightarrow_{\mathrm{m}} r_2, e_2$. This is easily verified by a case analysis on $\mathscr{T}c$. Similarly, for consistency, we prove that $\mathcal{I}\mathit{con}$ is monotonic and sound, and use Theorem 4.7.

As a bonus to the above definitions of monotonicity and soundness, our Agda development streamlines their presentation. For instance, the [R-CrCons] case of soundness becomes:

Rule	[R-CrCons] $r_1, \mathbf{B}\#A\ \{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} r_2, \langle\ \mathbf{B}\#A_1\ \{v_1\}\,,\ \mathbf{B}\#A_2\ \{v_2\}\ \rangle$
Assumption	$\mathbb{R}(r_1)$, $\mathbb{R}(r_2)$, $r_1 \preccurlyeq r_2$, $j \lhd_A j'$ and $\mathbb{A}[\![\ A^{j,j'}\ ]\!]$
Obligation	$j \lhd_{A_1} j'$, $j \lhd_{A_2} j'$, $\mathbb{A}[\![\ A_1^{j,j'}\ ]\!]$, and $\mathbb{A}[\![\ A_2^{j,j'}\ ]\!]$

5 Complete Monitoring

As a first application of our proof framework, we revisit complete monitoring [15, 19]. Complete monitoring builds on the notions of *ownership* and *obligations*. Intuitively, a region of code, that is, a term that resides in a boundary, owns the values it creates until it hands them off to another region. If the hand-off is through a boundary, the original owner relinquishes the value to the receiving region since the contract system can inspect it and "bless" its migration. Otherwise, the originating and the receiving region co-own the value, and hence, are both responsible for its behavior. In this context, ownership indicates which region of code is responsible for which value during the evaluation of a program; when a contract system is a complete monitor, a value is never co-owned

$$A ::= \langle \ell_n, \ell_p \rangle \ \ (\ell_n, \ell_p \text{ are labels}) \qquad r ::= ()$$

[R-CrNat] $r, \mathbf{B}\#A\ \{\ n\ \} \longrightarrow_{\mathrm{m}} r', n$ where $A = \langle \ell_n, \ell_p \rangle$ and $(r, r') \in \{((), ())\}$

[R-CrCons] $r, \mathbf{B}\#A\ \{\ \langle v_1, v_2 \rangle\ \} \longrightarrow_{\mathrm{m}} r', \langle\ \mathbf{B}\#A_1\ \{v_1\}\ ,\ \mathbf{B}\#A_2\ \{v_2\}\ \rangle$ where
$A = \langle \ell_n, \ell_p \rangle, A_1 = \langle \ell_n, \ell_p \rangle, A_2 = \langle \ell_n, \ell_p \rangle$, and $(r, r') \in \{((), ())\}$

[R-CrMon] $r, \mathbf{B}\#A\ \{\ v\ \} \longrightarrow_{\mathrm{m}} r', \mathrm{proxy}(A',\ v)$ where
$v = \lambda x.e$ or $v = \mathrm{box}(v'), A = \langle \ell_n, \ell_p \rangle, A' = \langle \ell_n, \ell_p \rangle$, and $(r, r') \in \{((), ())\}$

[R-PrxB] $r, \mathrm{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} r', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ \ (\mathbf{B}\#A_a\ \{v\})\ \}$ where
$A = \langle \ell_n, \ell_p \rangle, A_r = \langle \ell_n, \ell_p \rangle, A_a = \langle \ell_p, \ell_n \rangle$, and $(r, r') \in \{((), ())\}$

[R-MrgPrx] $r, \mathbf{B}\#A\ \{\ \mathrm{proxy}(A',\ e^m)\ \} \longrightarrow_{\mathrm{m}} r', \mathrm{proxy}(A'',\ e^m)$ where
$A = \langle \ell_n, \ell_s \rangle, A' = \langle \ell_s, \ell_p \rangle, A'' = \langle \ell_n, \ell_p \rangle$, and $(r, r') \in \{\ ((), ())\ \}$

Fig. 10. The annotation language ($\mathscr{A}owner, \mathscr{T}o$).

by two or more regions. Obligations denote which region of code is responsible for which flat contract; when a contract system is a complete monitor, it blames a region only when there is a violation of one of its obligations by one of the values it owns.

In the original formulation of complete monitoring, ownership and obligations are encoded as annotations that are propagated during evaluation by an annotated semantics. Given these annotations, complete monitoring becomes preservation and progress for a convoluted notion of well-formedness for annotated terms and contracts, and its proof is a tedious subject reduction.

In our reformulation of complete monitoring, we split it into three orthogonal conditions, each of which we prove in isolation and without unnecessary boilerplate, and then we simply compose the proofs. In our framework, complete monitoring comprises three invariants: *obligation consistency*, which says that blame objects stay in sync with obligations; the *single-owner policy*, which matches the corresponding restriction of the original formulation; and *owner consistency*, which says that blame objects stay in sync with ownership. For obligation consistency, since labels on flat contracts are essentially Dimoulas et al. [15, 19]'s obligations, we reuse the proof of consistency ($\mathscr{I}con$) for $\lambda_{\mathrm{m}}[\mathscr{A}bctc;\mathscr{T}bc]$ from Section 4.3. For the single-owner policy, we prove it for a new instance, $\lambda_{\mathrm{m}}[\mathscr{A}owner;\mathscr{T}o]$, with just ownership annotations. For owner consistency, we prove it for a third instance, $\lambda_{\mathrm{m}}[\mathscr{A}obctc;\mathscr{T}obc]$, that combines ownership annotations, blame objects and contracts. Since there are instance projections from $\lambda_{\mathrm{m}}[\mathscr{A}obctc;\mathscr{T}obc]$ to $\lambda_{\mathrm{m}}[\mathscr{A}bctc;\mathscr{T}bc]$ and $\lambda_{\mathrm{m}}[\mathscr{A}owner;\mathscr{T}o]$, by Theorem 3.3, $\lambda_{\mathrm{m}}[\mathscr{A}obctc;\mathscr{T}obc]$ meets all three conditions. To complete the proof that extended CPCF is a complete monitor, we also have to show that the ownership annotations do not cause the evaluation to get stuck. For $\lambda_{\mathrm{m}}[\mathscr{A}owner; \mathscr{T}o]$, the proof is straightforward, but for $\lambda_{\mathrm{m}}[\mathscr{A}obctc; \mathscr{T}obc]$, contract checking can complicate matters. Hence, we also prove a progress theorem for the instance $\lambda_{\mathrm{m}}[\mathscr{A}obctc; \mathscr{T}obc]$; its corresponding Agda proof is in You [84].

5.1 The Single-Owner Policy

Figure 10 shows the syntax of $\mathscr{A}owner$ and the monitor-related reduction rules $\mathscr{T}o$. As usual, blue indicates the instance-specific parts of the definitions. In $\mathscr{A}owner$, annotations are pairs of labels, and the register is unused. An annotated boundary $\mathbf{B}\#\langle \ell_n, \ell_p \rangle\ \{e\}$ marks that ℓ_n is the owner of the boundary term itself, while ℓ_p is the owner of the region e. As an example, consider the term $\mathbf{B}\#\langle \ell_{\mathrm{C}}, \ell_{\mathrm{S}} \rangle\ \{\ (\lambda x.x)\ \ (\mathbf{B}\#\langle \ell_{\mathrm{S}}, \ell_{\mathrm{Q}} \rangle\ \{\ \lambda y.y\ \})\}$. It is divided into three regions: the outer boundary, the application inside the outer boundary, including the function but excluding its argument, and the argument. ℓ_{C} owns the outer boundary, ℓ_{S} owns the application, ℓ_{Q} owns the argument. This

$$A ::= \langle A^{own}, A^{bctc}\rangle, \quad A^{own} ::= \langle \ell_n, \ell_p\rangle, \quad A^{bctc} ::= [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle], \quad r ::= \textsc{Ok} \mid \textsc{Err}(\ell)$$

$$[\text{R-MrgPrx}] \quad r, \mathbf{B}\#A\,\{\,\text{proxy}(A',\ e^m)\,\} \longrightarrow_{\text{m}} r', \text{proxy}(A'',\ e^m) \text{ where}$$
$$A = \langle\langle \ell_n, \ell_s\rangle, [\langle b_1, \kappa_1\rangle, \ldots, \langle b_l, \kappa_l\rangle]\rangle, A' = \langle\langle \ell_s, \ell_p\rangle, [\langle b'_1, \kappa'_1\rangle, \ldots, \langle b'_m, \kappa'_m\rangle]\rangle,$$
$$A'' = \langle\langle \ell_n, \ell_p\rangle, [\langle b'_1, \kappa'_1\rangle, \ldots, \langle b'_m, \kappa'_m\rangle, \langle b_1, \kappa_1\rangle, \ldots, \langle b_l, \kappa_l\rangle]\rangle, \text{ and } (r, r') \in \{\,(\textsc{Ok}, \textsc{Ok})\,\}$$

Fig. 11. The combined annotation language ($\mathscr{A}obctc, \mathscr{T}obc$).

example satisfies the single-owner policy as each sub-term has a single owner, because the two ℓ_Ss match. But, not all terms satisfy single ownership. For instance, if the inner boundary's annotation were $\langle \ell_P, \ell_Q\rangle$, then both ℓ_P and ℓ_S own the function, violating the policy. In general, a term adheres to the single-owner policy when the owner of each boundary (or a proxy) term matches the owner of the region it appears immediately inside.

The reduction rules in Figure 10 adjust ownership annotations as values migrate from one region to another, and halt the evaluation when they detect a violation of the single-owner policy. [R-CrNat] removes the boundary form around n to mark the hand-off from ℓ_p to $\ell_n - \ell_n$, the owner of the surrounding region obtains ownership of n implicitly. [R-CrCons] pushes the boundary inside the pair as the pair crosses the boundary — ownership for the elements of the pair becomes the same as the pair before the reduction as those do not migrate to the surrounding region until a projection on the pair. Most of the remaining rules are similar, with two exceptions. [R-Prxβ] inserts a boundary around the argument v to keep track of its migration from the region that owns the application to the region that owns the function $\lambda x.e$ — the new boundary around v has a *flipped* pair of labels since v retains its owner but the owner of the function gets ahold of the boundary term. [R-MrgPrx] ensures that when a proxy crosses a boundary it respects the single-owner policy — the two ℓ_s labels equate the owner of the proxy with the owner of the region that the proxy resides in. If the owners do not match, [R-MrgPrx] would not fire.

The single-owner policy is formalized as the invariant $\mathcal{I}single$ of $\lambda_{\text{m}}[\mathscr{A}owner;\mathscr{T}o]$. It relies on local information, i.e., ownership annotations, and contextual information, i.e. indices, to express that ownership of a boundary (and proxy) must match ownership of the region the boundary (or proxy) appears in. Specifically, the satisfaction relation takes care of threading ownership so that for every boundary (and proxy) j is the owner of the surrounding context and j' is the owner of the new region the boundary (or proxy) delimits.

Definition 5.1. $\mathcal{I}single$ is $(\text{Label}, \{\lhd_A\}_A, \mathbb{R}_{single}, \preccurlyeq_{single}, \mathbb{A})$ where $\mathbb{R}_{single}$ and $\preccurlyeq_{single}$ impose no constraints and

$$\lhd_A \quad :\equiv \quad \{(\ell_n, \ell_p)\}, \text{ for } A = \langle \ell_n, \ell_p\rangle$$
$$\mathbb{A}[\![\ \langle \ell_n, \ell_p\rangle^{j,j'}\ ]\!] \quad :\equiv \quad j = \ell_n \wedge j' = \ell_p$$

Because $\mathcal{I}single$ insists that the labels match, all invariant-satisfying terms either take a step or are values. Thus, the preservation of $\mathcal{I}single$ entails that $\lambda_{\text{m}}[\mathscr{A}owner;\mathscr{T}o]$ never detects a violation of the single-owner policy as it evaluates a term that satisfies $\mathcal{I}single$. Based on Theorem 4.7, in order to establish preservation of $\mathcal{I}single$, it is sufficient to show that $\mathcal{I}single$ is monotonic and sound, which follows the same simple proof pattern as that for consistency ($\mathcal{I}con$) in Section 4.3.

Theorem 5.2. *$\mathcal{I}single$ is monotonic and sound.*

5.2 Ownership Consistency

To capture the relationship between ownership and blame objects, we combine the annotation languages ($\mathscr{A}owner, \mathscr{T}o$) and ($\mathscr{A}bctc, \mathscr{T}bc$) to form ($\mathscr{A}obctc, \mathscr{T}obc$). Figure 11 gives the relevant definitions

and one example transition rule, where the components from $\mathscr{A}owner$ are in blue and those from $\mathscr{A}bctc$ are in brown. To distinguish the annotations of $(\mathscr{A}obctc, \mathscr{T}obc)$ from those of $(\mathscr{A}owner, \mathscr{T}o)$ and $(\mathscr{A}bctc, \mathscr{T}bc)$, we decorate the metavariable As as A^{own} and A^{bctc}, respectively.

Similar to the invariant $\mathscr{I}cons$ from Section 4.1, we define a judgment BlameSeq as part of characterizing ownership consistency in an invariant. To be concrete, for any annotation $\langle A^{own}, A^{bctc}\rangle$ where $A^{own} = \langle \ell_n, \ell_p\rangle$ and $A^{bctc} = [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]$, BlameSeq$(\ell_p,\ \ell_n,\ [b_1, \ldots, b_m])$ holds iff:

(1) $\ell_p = b_1.\text{pos}$ and $b_m.\text{neg} = \ell_n$, and
(2) $b_1.\text{neg} = b_2.\text{pos}, b_2.\text{neg} = b_3.\text{pos}, \ldots,$ and $b_{n-1}.\text{neg} = b_n.\text{pos}$.

Condition (1) relates ownership to blame parties, making sure that the the owner of the region in the boundary (or proxy) matches the positive party of the blame object at the front, and similarly the owner of the boundary (or proxy) itself matches the negative party of the blame object at the end. Condition (2) makes sure that there are no dangling blame parties in the middle. In other words, BlameSeq enforces that ownership and blame objects are in sync, which together with consistency of blame objects, entails that the book keeping of extended CPCF for assigning blame is in sync with ownership.

Theorem 5.3 defines the invariant for ownership consistency, $\mathscr{I}ocons$. The indices of $\mathscr{I}ocons$ are labels, and two indices j, j' are related at annotation $A = \langle A^{own}, [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\rangle$ if BlameSeq determines that j' and j are consistent with the blame objects. Moreover, the $\mathbb{A}$ function of $\mathscr{I}ocons$ asks that ownership labels and indices match appropriately.

Definition 5.3. $\mathscr{I}ocons$ is the six tuple $(\text{Label},\ \{\lhd_A\}_{A:\text{Ann}_\tau},\ \mathbb{R}_{ocons},\ \preccurlyeq_{ocons},\ \mathbb{A})$ where $\mathbb{R}_{ocons}$ and $\preccurlyeq_{ocons}$ always hold, and

$$\lhd_{\langle A^{own}, [\langle b_1,\kappa_1\rangle,\ldots,\langle b_m,\kappa_m\rangle]\rangle} \quad :\equiv \quad \{(j, j') \mid \text{BlameSeq}(j',\ j,\ [b_1, \ldots, b_m])\}$$
$$\mathbb{A}[\![\ \langle\langle \ell_n, \ell_p\rangle, A^{bctc}\rangle^{j,j'}\ ]\!] \quad :\equiv \quad (j = \ell_n \wedge j' = \ell_p)$$

Similar to Section 5.1, preservation of $\mathscr{I}ocons$ is a consequence of its monotonicity and soundness.

Theorem 5.4. *The invariant $\mathscr{I}ocons$ is monotonic and sound.*

6 Space-Efficient Contracts

Due to the accumulation of redundant proxies around values, contract checking can result in unnecessary space and time cost. Indeed, the issue is bad both in theory [72, bubble example] and in practice [73]. In response, Feltey et al. [21] take inspiration from Greenberg [29]'s space-efficient latent contracts and develop collapsible contracts as a practical way to mitigate certain pathologies in the performance of gradual typing due to duplicate checks [73].

Greenberg [29]'s space-efficient contracts capture the essential, general principles for avoiding redundancy by merging proxies. In Section 6.1, we introduce $\lambda_{\text{m}}[\mathscr{A}se;\mathscr{T}s]$, an instance with space-efficient contracts, and prove its space-efficiency. $\lambda_{\text{m}}[\mathscr{A}se;\mathscr{T}s]$ relies on a decidable predicate *isStronger* to determine whether one flat contract rejects a superset of the values another one rejects, and hence, avoid space explosion. The proof of space-efficiency proceeds by establishing the invariant that, as long as there are no recursive contracts in the program, the size of each contract is in $O(|\mathcal{K}| \cdot 2^H)$, where $\mathcal{K}$ is the set of all distinct flat contracts and H is the height of the tallest contract in the initial program.

Section 6.2 tackles time complexity. It does so in two steps via the instance $\lambda_{\text{m}}[\mathscr{A}ccs;\mathscr{T}ccs]$ that records the number of different operations in its register. First, since time complexity depends on the number of contract checks, we prove that the total number of flat contract checks performed by $\lambda_{\text{m}}[\mathscr{A}ccs;\mathscr{T}ccs]$ is bounded linearly by the number of monitor-related reductions. Since the monitor calculus takes a single step when checking an entire list of flat contracts, the bound implies that space-efficient contracts avoid an asymptotic increase in the number of contract checks. Second,

$$
\begin{array}{rcl@{\qquad}rcl}
A & ::= & {}^{\text{se}}\kappa & {}^{\text{se}}\kappa & ::= & \text{unit}/\text{se} \mid [\text{flat}^{\ell_1}(x.\, e_1), \ldots, \text{flat}^{\ell_m}(x.\, e_m)] \mid {}^{\text{se}}\kappa_1 \times/\text{se}\ {}^{\text{se}}\kappa_2 \\
r & ::= & \text{Ok} \mid \text{Err}(\ell) & & \mid & {}^{\text{se}}\kappa_1 +/\text{se}\ {}^{\text{se}}\kappa_2 \mid \text{box}/\text{se}\ {}^{\text{se}}\kappa \mid {}^{\text{se}}\kappa_a \to/\text{se}\ {}^{\text{se}}\kappa_r \mid t \mid \mu/\text{se}\, t.{}^{\text{se}}\kappa
\end{array}
$$

$$
\begin{array}{rl}
\text{[R-CrNat]} & r, \mathbf{B}\#A\,\{\ n\ \} \longrightarrow_{\text{m}} r', n \text{ where} \\
A = [\text{flat}^{\ell_1}(x.\, e_1), \ldots, \text{flat}^{\ell_m}(x.\, e_m)], & (r, r') \in \mathit{checkCtcs}([\text{flat}^{\ell_1}(x.\, e_1), \ldots, \text{flat}^{\ell_m}(x.\, e_m)],\ n) \\
\text{[R-Prx}\beta\text{]} & r, \text{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\text{m}} r', \mathbf{B}\#A_r\,\{\ (\lambda x.e)\ (\mathbf{B}\#A_a\,\{v\})\ \} \text{ where} \\
& A = {}^{\text{se}}\kappa_a \to/\text{se}\ {}^{\text{se}}\kappa_r,\ A_r = {}^{\text{se}}\kappa_r,\ A_a = {}^{\text{se}}\kappa_a, \text{and } (r, r') \in \{(\text{Ok}, \text{Ok})\} \\
\text{[R-MrgPrx]} & r, \mathbf{B}\#A\,\{\ \text{proxy}(A',\ e^m)\ \} \longrightarrow_{\text{m}} r', \text{proxy}(A'',\ e^m) \text{ where} \\
& A = {}^{\text{se}}\kappa_1,\ A' = {}^{\text{se}}\kappa_2,\ A'' = \mathit{join}({}^{\text{se}}\kappa_2, {}^{\text{se}}\kappa_1), \text{ and } (r, r') \in \{(\text{Ok}, \text{Ok})\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{join}([\kappa'_1, \ldots, \kappa'_k],\ [\kappa_1, \ldots, \kappa_m]) & \equiv & \mathit{joinp}([\kappa'_1, \ldots, \kappa'_k], [\kappa_1, \ldots, \kappa_m]) \\
\mathit{join}(({}^{\text{se}}\kappa_1 \to/\text{se}\ {}^{\text{se}}\kappa_3), ({}^{\text{se}}\kappa_2 \to/\text{se}\ {}^{\text{se}}\kappa_4)) & \equiv & \mathit{join}({}^{\text{se}}\kappa_2, {}^{\text{se}}\kappa_1) \to/\text{se}\ \mathit{join}({}^{\text{se}}\kappa_3, {}^{\text{se}}\kappa_4) \\
\mathit{joinp}([\,],\ [\kappa_1, \ldots, \kappa_m]) \equiv [\kappa_1, \ldots, \kappa_m] & & \\
\mathit{joinp}([\text{flat}^{\ell}(x.\, e), \kappa'_2, \ldots, \kappa'_k], [\kappa_1, \ldots, \kappa_m]) \equiv \text{flat}^{\ell}(x.\, e) :: \mathit{drop}(\mathit{joinp}([\kappa'_2, \ldots, \kappa'_k], [\kappa_1, \ldots, \kappa_m]), e) & & \\
\mathit{drop}([\,],\ e) & \equiv & [\,] \\
\mathit{drop}([\text{flat}^{\ell}(x.\, e'), \kappa_2, \ldots, \kappa_m], e) & \equiv & \textbf{if } \mathit{isStronger}(e,\ e') \textbf{ then } \mathit{drop}([\kappa_2, \ldots, \kappa_m], e) \\
& & \textbf{else } \text{flat}^{\ell}(x.\, e') :: \mathit{drop}([\kappa_2, \ldots, \kappa_m], e)
\end{array}
$$

Fig. 12. The space-efficient annotation language ($\mathscr{A}se, \mathscr{T}s$)

since time complexity also depends on merging contracts, we prove that the number of operations needed for merging is in $O(k \cdot |\mathcal{K}|^2 \cdot 2^H)$, where k is the number of monitor-related reductions. Hence, maintaining space-efficiency does not cause asymptotic slowdowns.

Finally, Section 6.3 shows that space-efficient contracts signal the same errors as extended CPCF through the instance $\lambda_{\text{m}}[\mathscr{A}scctc; \mathscr{T}sc]$ that combines $\lambda_{\text{m}}[\mathscr{A}se; \mathscr{T}s]$ and $\lambda_{\text{m}}[\mathscr{A}ctc; \mathscr{T}c]$ into one instance.

6.1 Space Efficiency

Figure 12 presents the syntax and the monitor-related rules of ($\mathscr{A}se, \mathscr{T}s$). We use ${}^{\text{se}}\kappa$ to denote space-efficient contracts. Similar to ordinary contracts, ${}^{\text{se}}\kappa$ also has one constructor for each type, but for the nat type, ${}^{\text{se}}\kappa$ contains a list of flat contracts instead of just one. Accordingly, each $\mathscr{A}se$ annotation contains only a single contract, unlike $\mathscr{A}ctc$, which has a list of contracts. Intuitively, a space-efficient contract ${}^{\text{se}}\kappa$ represents a contract without redundancy as long as the reduction rules eliminate the redundant checks in the leaves of ${}^{\text{se}}\kappa$.

We parameterize the annotation language ($\mathscr{A}se, \mathscr{T}s$) by a decision procedure *isStronger* that takes two flat contracts and determines whether the first flat contract subsumes the second. Specifically, for any e, e' that have only one free variable x, if $\mathit{isStronger}(e,\ e')$ is true then it should be the case that for all n, if $e[n\,/\,x]$ terminates with a positive integer, $e'[n\,/\,x]$ also terminates with a positive integer. If $\mathit{isStronger}(e,\ e')$ is false, the decision procedure makes no claims about the relationship between the flat contracts and this is always allowed, unless e and e' are the same term. When e and e' are the same, $\mathit{isStronger}(e,\ e')$ must be true. This decision procedure, and the interpretation of its results, matches the Racket contract system's `contract-stronger?` operation.

The reduction rules $\mathscr{T}s$ are similar to $\mathscr{T}c$ but simpler, because some complexity moves into *join*, used in [R-MrgPrx] to combine the contract on the boundary and the contract on the proxy.

The *join* function is adapted from Greenberg [29]'s work. It takes two space-efficient contracts and merges the lists of flat contracts in the inputs. To combine the flat contracts, *join* calls *joinp*, which removes redundant checks. Specifically, *joinp* takes two lists of flat contracts and, for each in the first list, calls *drop* to filter out those in the second list that *isStronger* reports as redundant.

$$A ::= {}^{\mathfrak{se}}\kappa \qquad r ::= \langle s, c, w, k\rangle \qquad s \in \text{Status} ::= \textsc{Ok} \mid \textsc{Err}(\ell) \qquad c, w, k \in \mathbb{N}$$

[R-CrNat] $r, \mathbf{B}\#A\,\{\ n\ \} \longrightarrow_{\mathrm{m}} r', n$ where
$A = [\mathrm{flat}^{\ell_1}(x.\,e_1), \ldots, \mathrm{flat}^{\ell_m}(x.\,e_m)]$, $(r, r') \in \{(\langle s, c, w, k\rangle, \langle s', c', w, k+1\rangle) \mid$
$c' = c + m$ and $(s, s') \in \mathit{checkCtcs}([\mathrm{flat}^{\ell_1}(x.\,e_1), \ldots, \mathrm{flat}^{\ell_m}(x.\,e_m)],\ n)\}$

[R-MrgPrx] $r, \mathbf{B}\#A\,\{\,\mathrm{proxy}(A',\ e^m)\,\} \longrightarrow_{\mathrm{m}} r', \mathrm{proxy}(A'',\ e^m)$ where
$A = {}^{\mathfrak{se}}\kappa_1,\ A' = {}^{\mathfrak{se}}\kappa_2,\ A'' = \mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)$, and
$(r, r') \in \{\langle\langle \textsc{Ok}, c, w, k\rangle, \langle \textsc{Ok}, c, w', k+1\rangle\rangle \mid w' = w + \mathrm{tickCount}(\mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1))\}$

Fig. 13. The annotation language ($\mathscr{A}ccs$, $\mathscr{T}ccs$) that counts contract checks.

To present the formal definition of the invariant $\mathscr{I}size$ in Theorem 6.1, we need to introduce a few more definitions to capture the prerequisites. First, the $\mathsf{ht}(\cdot)$ and $\mathsf{sz}(\cdot)$ functions compute the height and the size of a contract in the usual way; here are two of their cases:

$$\mathsf{sz}\big([\mathrm{flat}^{\ell_1}(x.\,e_1), \ldots, \mathrm{flat}^{\ell_m}(x.\,e_m)]\big) = m \qquad \mathsf{ht}\big([\mathrm{flat}^{\ell_1}(x.\,e_1), \ldots, \mathrm{flat}^{\ell_m}(x.\,e_m)]\big) = 0$$
$$\mathsf{sz}({}^{\mathfrak{se}}\kappa_a \to_{\mathfrak{se}} {}^{\mathfrak{se}}\kappa_r) = 1 + \mathsf{sz}({}^{\mathfrak{se}}\kappa_a) + \mathsf{sz}({}^{\mathfrak{se}}\kappa_r) \qquad \mathsf{ht}({}^{\mathfrak{se}}\kappa_a \to_{\mathfrak{se}} {}^{\mathfrak{se}}\kappa_r) = 1 + \max(\mathsf{ht}({}^{\mathfrak{se}}\kappa_a), \mathsf{ht}({}^{\mathfrak{se}}\kappa_r))$$

Second, the $\mathsf{NonRec}({}^{\mathfrak{se}}\kappa)$ predicate holds iff ${}^{\mathfrak{se}}\kappa$ contains no recursive contracts. Third, $\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa)$ takes a judgment, a space-efficient contract, and asserts that all lists of flat predicates in ${}^{\mathfrak{se}}\kappa$ satisfy J. We introduce two judgments for use in conjunction with AllFlats: $\mathsf{NonEmpty}(xs)$ takes a list, xs, and asserts that it is non-empty. $\mathsf{UniqSub}(xs, ys)$ takes two lists and asserts that xs contains only the distinct elements of ys. Finally, the constant $c_0 > 0$ captures the constant factor in the bound.

Definition 6.1. $\mathscr{I}size$ is the six tuple $(\mathscr{J},\ \{\lhd_A\}_A,\ \mathbb{R}_{size},\ \preccurlyeq_{size},\ \mathbb{A})$ where $\mathscr{J} :\equiv \top, \mathbb{R}_{size}, \lhd_A$, and $\preccurlyeq_{size}$ always hold, and $\mathbb{A}[\![\ {}^{\mathfrak{se}}\kappa^{j,j'}\ ]\!] :\equiv \mathsf{NonRec}({}^{\mathfrak{se}}\kappa) \wedge \mathsf{AllFlats}(\mathsf{NonEmpty}, {}^{\mathfrak{se}}\kappa) \wedge \mathsf{AllFlats}(\mathsf{UniqSub}(-, \mathcal{K}), {}^{\mathfrak{se}}\kappa) \wedge \mathsf{ht}({}^{\mathfrak{se}}\kappa) \le H \wedge \mathsf{sz}({}^{\mathfrak{se}}\kappa) \le c_0 \cdot 2^H \cdot |\mathcal{K}|$.

Theorem 6.2. *There exists $c_0 > 0$ such that $\mathscr{I}size$ is monotonic and sound.*

6.2 Asymptotic Time Complexity

Figure 13 displays the instance $\lambda_{\mathrm{m}}[\mathscr{A}ccs; \mathscr{T}ccs]$ we use for proving time complexity bounds for space-efficient contracts. To reason about time complexity, we introduce three counters in the register: c tracks the number of flat contract checks, w represents how many primitive operations *join* has performed, and k counts the total number of monitor-related reduction steps.

In rule [R-CrNat], we increment c by m as m more flat contracts have been checked. In rule [R-MrgPrx], we presuppose that $\mathrm{tickCount}(\mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1))$ returns the number of primitive operations $\mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)$ performs. In our Agda proof [84], we follow Danielsson [10] and define $\mathit{join}(\cdot,\cdot)$ in a monadic style to track the number of primitive operations. Also, we follow Guéneau et al. [36] and Guéneau [35] to work with asymptotic complexity in a precise and correct manner.

Definition 6.3. $\mathscr{I}chkbnd$ is the six tuple $(\mathscr{J},\ \{\lhd_A\}_A,\ \mathbb{R}_{chk},\ \preccurlyeq_{chk},\ \mathbb{A})$ where $\mathscr{J} :\equiv \top$, $\lhd_A$, and $\preccurlyeq_{chk}$ always hold, $\mathbb{A}[\![\ {}^{\mathfrak{se}}\kappa^{j,j'}\ ]\!] :\equiv \mathsf{AllFlats}(\mathsf{UniqSub}(-, \mathcal{K}), {}^{\mathfrak{se}}\kappa)$, and $\mathbb{R}_{chk}(\langle s, c, w, k\rangle)$ holds iff $c \le k \cdot |\mathcal{K}|$.

Theorem 6.4. *$\mathscr{I}chkbnd$ is monotonic and sound.*

To give an upper bound of w, we develop the asymptotic complexity of *join*, *joinp* and *drop*. In particular, in Theorem 6.5, the factor $|\mathcal{K}|^2$ comes from *joinp*, and the factor 2^H, the maximum size of contracts of height H, comes from *join*. The constant $c_0 > 0$ is existentially quantified across Theorem 6.5 and Theorem 6.6.

$$A ::= \langle {}^{\mathfrak{se}}\kappa, [\kappa_1, \ldots, \kappa_m]\rangle \qquad r ::= \langle s, s\rangle \qquad s \in \mathsf{Status}$$

[R-CrNat] $r, \mathbf{B}\#A\,\{\ n\ \} \longrightarrow_{\mathrm{m}} r', n$ where
$A = \langle [\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)], [\kappa_1, \ldots, \kappa_m]\rangle$,
$(r, r') \in \{(\langle s_1, s_2\rangle, \langle s_1', s_2'\rangle) \mid (s_2, s_2') \in \mathit{checkCtcs}([\kappa_1, \ldots, \kappa_m],\ n),$ and
$(s_1, s_1') \in \mathit{checkCtcs}([\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)],\ n)\}$

[R-Prxβ] $r, \mathsf{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} r', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#A_a\ \{v\})\ \}$ where
$A = \langle {}^{\mathfrak{se}}\kappa_a \rightarrow\!\!/_{\mathfrak{se}}\ {}^{\mathfrak{se}}\kappa_r, [(\kappa_1 \rightarrow\!\!/_{\mathfrak{c}}\ \kappa_1'), \ldots, (\kappa_m \rightarrow\!\!/_{\mathfrak{c}}\ \kappa_m')]\rangle$, $A_r = \langle {}^{\mathfrak{se}}\kappa_r, [\kappa_1', \ldots, \kappa_m']\rangle$,
$A_a = \langle {}^{\mathfrak{se}}\kappa_a, [\kappa_m, \ldots, \kappa_1]\rangle$, and $(r, r') \in \{(\langle \mathrm{Ok}, \mathrm{Ok}\rangle, \langle \mathrm{Ok}, \mathrm{Ok}\rangle)\}$

[R-MrgPrx] $r, \mathbf{B}\#A\,\{\,\mathsf{proxy}(A',\ e^m)\,\} \longrightarrow_{\mathrm{m}} r', \mathsf{proxy}(A'',\ e^m)$ where
$A'' = \langle \mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1), [\kappa_1', \ldots, \kappa_m', \kappa_1, \ldots, \kappa_l]\rangle$, $A = \langle {}^{\mathfrak{se}}\kappa_1, [\kappa_1, \ldots, \kappa_l]\rangle$,
$A' = \langle {}^{\mathfrak{se}}\kappa_2, [\kappa_1', \ldots, \kappa_m']\rangle$, and $(r, r') \in \{(\langle \mathrm{Ok}, \mathrm{Ok}\rangle, \langle \mathrm{Ok}, \mathrm{Ok}\rangle)\}$

Fig. 14. The annotation language ($\mathscr{A}\mathit{sctc}, \mathscr{T}\mathit{sc}$) for establishing the correctness of space-efficient contracts.

Definition 6.5. $\mathscr{I}\mathit{sebnd}$ is the six tuple $(\mathscr{J},\ \{\lhd_A\}_A,\ \mathbb{R}_{\mathit{sebnd}},\ \preccurlyeq_{\mathit{sebnd}},\ \mathbb{A})$ where $\mathscr{J} :\equiv \top$, $\lhd_A$, and $\preccurlyeq_{\mathit{sebnd}}$ always hold, $\mathbb{A}[\![\ {}^{\mathfrak{se}}\kappa^{j,j'}\ ]\!] :\equiv \mathsf{NonRec}({}^{\mathfrak{se}}\kappa) \wedge \mathsf{AllFlats}(\mathsf{NonEmpty}, {}^{\mathfrak{se}}\kappa) \wedge \mathsf{AllFlats}(\mathsf{UniqSub}(-, \mathcal{K}), {}^{\mathfrak{se}}\kappa) \wedge \mathsf{ht}({}^{\mathfrak{se}}\kappa) \le H$, and $\mathbb{R}_{\mathit{sebnd}}(\langle s, c, w, k\rangle)$ holds iff $w \le c_0 \cdot k \cdot |\mathcal{K}|^2 \cdot 2^H$.

Theorem 6.6. *$\mathscr{I}\mathit{sebnd}$ is monotonic and sound.*

6.3 Equivalence of Space-Efficient Contracts and Finder-Felleisen Contracts

Since space-efficient contracts remove checks at run time, we need to ensure that the remaining checks are sufficient to detect contract violations correctly. To prove the equivalence, we follow the ideas from Pottier and Simonet [55, 56]'s non-interference proof and create ($\mathscr{A}\mathit{scctc}, \mathscr{T}\mathit{sc}$), an annotation language that runs two contract systems from ($\mathscr{A}\mathit{ctc}, \mathscr{T}\mathit{c}$) and ($\mathscr{A}\mathit{se}, \mathscr{T}\mathit{s}$) simultaneously. As the monitor calculus separates monitor-related rules from program-related rules, the two contract systems, ($\mathscr{A}\mathit{ctc}, \mathscr{T}\mathit{c}$) and ($\mathscr{A}\mathit{se}, \mathscr{T}\mathit{s}$), are completely encapsulated in the annotations and the registers of ($\mathscr{A}\mathit{scctc}, \mathscr{T}\mathit{sc}$). Consequently, their equivalence boils down to proving that the register of ($\mathscr{A}\mathit{scctc}, \mathscr{T}\mathit{sc}$) always has equal components.

Figure 14 shows the syntax and the monitor-related rules of ($\mathscr{A}\mathit{scctc}, \mathscr{T}\mathit{sc}$). The annotations pair a space-efficient contract with a list of ordinary contracts, and the register keeps two distinct copies of contract-checking status. As one would expect, the monitor-related rules defined in $\mathscr{T}\mathit{sc}$ apply the rules from both $\mathscr{T}\mathit{s}$ and $\mathscr{T}\mathit{c}$ to propagate the contracts and separately manage the register.

To prove that $\lambda_{\mathrm{m}}[\mathscr{A}\mathit{ctc}; \mathscr{T}\mathit{c}]$ and $\lambda_{\mathrm{m}}[\mathscr{A}\mathit{se}; \mathscr{T}\mathit{s}]$ always produce the same contract-checking results, we define an invariant $\mathscr{I}\mathit{sim}$, which asks that the register of ($\mathscr{A}\mathit{scctc}, \mathscr{T}\mathit{sc}$) always contains equal components. Since, in $\mathscr{T}\mathit{sc}$, the register is updated by [R-CrNat] through two independent uses of *checkCtcs*, we must ensure they return identical results. We relate their inputs by $\sim$, which becomes a restriction captured by $\mathbb{A}$ in $\mathscr{I}\mathit{sim}$. The definition of $\sim$ requires that the list of flat contracts in the $\mathscr{A}\mathit{se}$ annotation is a sublist of the flat contracts in the $\mathscr{A}\mathit{ctc}$ annotation, and that each removed flat contract is subsumed by a preceding contract in the $\mathscr{A}\mathit{se}$ annotation. Formally, for flat contracts κ_1 through κ_m, $[\kappa_{a_1}, \ldots, \kappa_{a_l}] \sim [\kappa_1, \ldots, \kappa_m]$ if and only if $1 = a_1 < \cdots < a_l \le m$ and, for each j and each $a_j < i < a_{j+1}$, at least one of $\kappa_{a_1}, \ldots, \kappa_{a_j}$ subsumes κ_i.

There is one caveat, however: $\sim$ is preserved by $\longrightarrow_{\mathrm{m}}$ only when all recursive contracts appearing in the program are covariant. Hence, we add the condition $\vdash^{+}\ {}^{\mathfrak{se}}\kappa\ \mathbf{sgn}$ in $\mathbb{A}$. Without this prerequisite, a recursive contract that is referenced in negative positions can cause space-efficient contracts to blame the wrong party, as we saw with blame consistency in Section 3.4.

Definition 6.7. $\mathcal{I}sim$ equals $(\mathcal{J}, \{\lhd_A\}_A, \mathbb{R}, \preccurlyeq, \mathbb{A})$ where $\mathcal{J} :\equiv \top$, $\lhd_A$ are relations that always hold, $\preccurlyeq$ is the smallest reflexive relation that includes $\langle \textsc{Ok}, \textsc{Ok}\rangle \preccurlyeq \langle \textsc{Err}(\ell), \textsc{Err}(\ell)\rangle$ for all labels ℓ, $\mathbb{R}(\langle s, s'\rangle) \Leftrightarrow s = s'$, and $\mathbb{A}[\![\ \langle \overset{\text{ᵒ}}{\kappa}, [\kappa_1, \ldots, \kappa_m]\rangle^{j,j'}\ ]\!] :\equiv (\vdash^+ \overset{\text{ᵒ}}{\kappa}\ \mathbf{sgn}) \wedge (\overset{\text{ᵒ}}{\kappa} \sim [\kappa_1, \ldots, \kappa_m])$.

Theorem 6.8. *The invariant $\mathcal{I}sim$ is monotonic and sound.*

Crucial to the proof is that the $\sim$ relation forces the set of all flat contracts appearing in the program to be partially ordered in accordance with the set of values each flat contract accepts, and that *isStronger* approximates this partial order.

7 Related Work

Our work lives in the context of the vast literature on the foundations of (higher-order) contracts [4, 12, 13, 15, 16, 18–20, 22, 23, 28–30, 38, 39, 45, 58, 68, 70, 71, 75, 77, 82, 83] and those of contracts' most prominent application, gradual typing [1, 2, 21, 25–27, 31–34, 37, 43, 51, 59–67, 76, 74, 78–80].

This vast literature is ample with formal models, and their theorems and proofs. Often, new models build on previous ones by extending them with new features, or by modifying their semantics to introduce new mechanisms for enforcing contracts and gradual types. But in most cases, new models, which inevitably share features with previous ones, are similar but subtly different from their predecessors. This variability trickles down to theorems and proofs; analogous properties "feel" the same but look formally different from one model to another, and their proofs have to be repeated. Even in publications, such as those of Greenman et al. [33, 31], that claim to present a unifying framework for a spectrum of models, the shared part between the models is the definition of their source syntax; semantics, theorems, and proofs are redone for every point in the spectrum.

In fact, Greenman et al.'s work offers an opportunity for examining how well our framework can support the comparison of different models as well as revealing general limitations of our approach and our specific monitor calculus. Overall, we believe that Greenman et al.'s work would benefit significantly from our framework. Specifically, we conjecture that Greenman et al.'s Natural, Amnesic, and Forgetful semantics can be defined as instances of the same monitor calculus with different annotation languages that reflect the variation of the type-enforcing wrappers. For example, one annotation language can encode Natural's "solid" wrappers while another can encode the "fluid" wrappers of Amnesic and Forgetful that become inactive as they accumulate around values. Moreover, our framework can model Co-natural with the same instance as that for Natural but with the restriction that pairs are always pairs of boxes so that they encode Co-natural's lazy pairs. This way, Co-natural trivially inherits all properties of Natural without the need for additional proofs. As for properties and their proofs, complete monitoring, blame soundness, blame completeness and error preorder for Natural, Co-natural, Amnesic, and Forgetful can be expressed in a similar manner as the properties in Section 5 and Section 6. For instance, in Section 5 we combine the annotation languages $(\mathscr{A}owner, \mathscr{T}o)$ and $(\mathscr{A}bctc, \mathscr{T}bc)$ to obtain the composite instance $\lambda_{\mathsf{m}}[\mathscr{A}obctc; \mathscr{T}obc]$. We expect the same to apply to Greenman et al.'s work allowing for ownership to be defined separately and reused across Natural, Co-natural, Amnesic, and Forgetful.

That said, our framework cannot handle all of Greenman et al.'s work. One central design decision in our approach is that the notions of boundaries and proxies are required, as they provide the basic structure on which the monitor calculus is built. As a result, the Transient semantics, which utilizes a check-injection compilation pass rather than proxies, is out of reach. Furthermore, due to the critical role of boundaries and proxies, properties that require annotating arbitrary subexpressions are challenging to state and prove with our approach. For instance, since ownership annotations can only be associated with boundary and proxy expressions, establishing blame soundness for a Transient-like semantics is not possible in our framework even if it were possible to capture the semantics as an instance of the calculus. Finally, the present monitor calculus has a standard type system, meaning that certain tag errors are impossible. These errors might be important in

applications of contract systems to mixed-type systems, which would require a variation of the monitor calculus with a different underlying type system in order to reflect them faithfully.

Of course, we are not the first to observe that the repetition of contract systems and gradual typing metatheory. Siek and Chen [62] develop a parameterized cast calculus that abstracts certain parts of the enforcement mechanism for gradual types away, allowing the blame-subtyping theorem and the dynamic gradual guarantee to be reused across different instantiations of their cast calculus. Gierczak et al. [27] also introduce a parameterized model and use it to streamline the design of the logical relations that underpin their vigilance property. But, despite a uniform presentation of the property there is little reuse in the proofs. Swords [69] and Swords et al. [70, 71] build a model based on Concurrent ML that uniformly represents contract enforcement mechanisms as communicating processes, essentially explicating their workings as programs. They do not abstract over properties and proofs, however. Our work is the first that aims for a general unifying framework for the metatheory of contracts that puts an emphasis on proof reuse.

Beyond the prior work on contracts, an important, direct source of inspiration for our work is the foundations of safe language interoperability [5, 44, 52]. Specifically, we owe the idea of boundaries and proxies as the building blocks for a unifying, parameterized framework to that work.

8 Conclusion

Every researcher with an interest in the metatheory of contract systems has experienced the banality of proving their properties. Whether the target is complete monitoring or the correctness of space-efficient contracts, the only creative step in the proof is constructing an invariant of evaluation that implies the target property. After that, and despite the variability of properties, the proofs devolve into tedious inductive arguments and painful case analyses that repeat endlessly across contract systems and properties. The central offering of this paper is how to separate the creative from the routine; researchers should focus on the information needed for distilling the target property to an invariant — given a few facts about the invariant, the rest can be abstracted away. Our hope with this work is not that it facilitates work that might be labeled "theory for theory's sake", but rather that it enables researchers to focus on designing novel, practical contract systems, and to answer any truly challenging metatheoretical questions that arise through such design work without getting distracted or bogged down by insignificant details.

While the paper provides evidence in favor of its approach, it also points out next steps for realizing its vision in full. First, Section 5 shows that the preservation of invariants often needs to be combined with progress, which currently lives outside our framework. However, since homomorphisms reflect transitions, some form of progress property should be transferable across transition systems. Second, the paper just scratches the surface of structured information in registers. Beyond recording information about different events, such as different operations, systematically structured registers open the way for supporting expressive, stateful contracts [17, 46, 49, 48]. Third, the framework does not easily accommodate the addition of new language features, a common activity in the literature of contract systems. Currently, such extensions require reproving the monitor calculus metatheory even though they are typically orthogonal. Therefore, in principle, they should be as amenable to proof reuse as the monitor calculus's composite instances are.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. You was partially supported during this work by the National Science Foundation under Awards No. 2237984 and No. 2421308. Dimoulas was partially supported by NSF under Awards No. 2237984 and No. 2412400. Findler was partially supported by NSF under Award No. 2421308. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and without Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP, Article 39 (Aug 2017), 28 pages. doi:10.1145/3110283

[2] Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast Insertion Strategies for Gradually-Typed Objects. In *Proceedings of the 9th Symposium on Dynamic Languages* (Indianapolis, Indiana, USA) *(DLS '13)*. 27–36. doi:10.1145/2508168.2508171

[3] Roberto M. Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (Sep 1993), 575–631. doi:10.1145/155183.155231

[4] Matthias Blume and David McAllester. 2004. A Sound (and Complete) Model of Contracts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP '04)*. 189–200. doi:10.1145/1016850.1016876

[5] Samuele Buro and Isabella Mastroeni. 2019. On the Multi-Language Construction. In *Programming Languages and Systems (ESOP'19)*. 293–321. doi:10.1007/978-3-030-17184-1_11

[6] Luca Cardelli. 1986. Amber. In *Combinators and Functional Programming Languages (LITP'85)*. 21–47. doi:10.1007/3-540-17184-3_38

[7] Feng Chen and Grigore Roşu. 2007. Mop: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. 569–588. doi:10.1145/1297027.1297069

[8] Olaf Chitil. 2012. Practical Typed Lazy Contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. 67–76. doi:10.1145/2364527.2364539

[9] Olaf Chitil, Dan McNeill, and Colin Runciman. 2003. Lazy Assertions. In *Implementation of Functional Languages (IFL '03)*. 1–19. doi:10.1007/978-3-540-27861-0_1

[10] Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. 133–144. doi:10.1145/1328438.1328457

[11] Markus Degen, Peter Thiemann, and Stefan Wehr. 2009. True Lies: Lazy Contracts for Lazy Languages (Faithfulness is Better than Laziness). In *4. Arbeitstagung Programmiersprachen (ATPS'09)*. Lübeck, Germany, 14. http://www.stefanwehr.de/publications/DegenThiemannWehr2009.html

[12] Markus Degen, Peter Thiemann, and Stefan Wehr. 2012. The Interaction of Contracts and Laziness. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM '12)*. 97–106. doi:10.1145/2103746.2103766

[13] Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems* 33, 5, Article 16 (Nov 2011), 29 pages. doi:10.1145/2039346.2039348

[14] Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. 2013. Option Contracts. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[15] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. 215–226. doi:10.1145/1926385.1926410

[16] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *2014 IEEE 27th Computer Security Foundations Symposium*. 3–17. doi:10.1109/CSF.2014.9

[17] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 117–131. doi:10.1145/2951913.2951930

[18] Christos Dimoulas, Riccardo Pucella, and Matthias Felleisen. 2009. Future Contracts. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (Coimbra, Portugal) *(PPDP '09)*. 195–206. doi:10.1145/1599410.1599435

[19] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Programming Languages and Systems (ESOP '12)*. 214–233. doi:10.1007/978-3-642-28869-2_11

[20] Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal Higher-Order Contracts. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. 176–188. doi:10.1145/2034773.2034800

[21] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA, Article 133 (Oct 2018), 27 pages. doi:10.1145/3276503

[22] Robert Bruce Findler and Matthias Blume. 2006. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming (FLOPS '06)*. Springer, 226–241.

[23] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. 48–59. doi:10.1145/581478.581484

[24] 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 '07)*. 111–128.
[25] Ronald Garcia. 2013. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. 417–428. doi:10.1145/2500365.2500603
[26] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. 429–442. doi:10.1145/2837614.2837670
[27] Olek Gierczak, Lucy Menon, Christos Dimoulas, and Amal Ahmed. 2024. Gradually Typed Languages Should Be Vigilant! *Proceedings of the ACM on Programming Languages (PACMPL)* 8, OOPSLA1, Article 125 (April 2024), 29 pages. doi:10.1145/3649842
[28] Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 181–194. doi:10.1145/2676726.2676967
[29] Michael Greenberg. 2016. Space-Efficient Latent Contracts. In *Trends in Functional Programming (TFP)*. 3–23.
[30] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. 353–364. doi:10.1145/1706299.1706341
[31] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–Untyped Interactions: A Comparative Analysis. *ACM Transactions on Programming Languages and Systems* 45, 1, Article 4 (Mar 2023), 54 pages. doi:10.1145/3579833
[32] Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, ICFP, Article 71 (Jul 2018), 32 pages. doi:10.1145/3236766
[33] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. Complete Monitors for Gradual Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, OOPSLA, Article 122 (Oct 2019), 29 pages. doi:10.1145/3360548
[34] Jessica Gronski and Cormac Flanagan. 2007. Unifying Hybrid Types and Contracts. In *Trends in Functional Programming (TFP'07)*. 54–70.
[35] Armaël Guéneau. 2019. *Mechanized verification of the correctness and asymptotic complexity of programs : the right answer at the right time.* Thèse de doctorat. Université Paris Cité. https://theses.hal.science/tel-03071720 Logic in Computer Science [cs.LO]. NNT : 2019UNIP7110. tel-03071720.
[36] Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems (ESOP '18)*. 533–560.
[37] David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189.
[38] Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed Contracts for Functional Programming. In *Functional and Logic Programming (FLOPS '06)*. 208–225.
[39] Matthias Keil and Peter Thiemann. 2015. Blame Assignment for Higher-Order Contracts with Intersection and Union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. 375–386. doi:10.1145/2784731.2784737
[40] Shiyi Kong, Minyan Lu, Luyi Li, and Lihua Gao. 2020. Runtime Monitoring of Software Execution Trace: Method and Tools. *IEEE Access* 8 (2020), 114020–114036. doi:10.1109/ACCESS.2020.3003087
[41] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. 1999. Runtime Assurance Based On Formal Specifications. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*.
[42] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. doi:10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).
[43] Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!. In *Programming Languages and Systems (ESOP '08)*. 16–31.
[44] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. 3–10. doi:10.1145/1190216.1190220
[45] Hernán Melgratti and Luca Padovani. 2017. Chaperone Contracts for Higher-Order Sessions. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP, Article 35 (Aug 2017), 29 pages. doi:10.1145/3110279
[46] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible Access Control with Authorization Contracts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. 214–233. doi:10.1145/2983990.2984021
[47] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. SHILL: A Secure Shell Scripting Language. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. 183–199.

[48] Cameron Moy, Christos Dimoulas, and Matthias Felleisen. 2024. Effectful Software Contracts. *Proceedings of the ACM on Programming Languages (PACMPL)* 8, POPL, Article 88 (Jan 2024), 28 pages. doi:10.1145/3632930

[49] Cameron Moy and Matthias Felleisen. 2023. Trace contracts. *Journal of Functional Programming* 33 (2023), e14. doi:10.1017/S0956796823000096

[50] Cameron Moy, Ryan Jung, and Matthias Felleisen. 2025. Contract Systems Need Domain-Specific Notations. In *39th European Conference on Object-Oriented Programming (ECOOP 2025)*. 42:1–42:24.

[51] Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual type theory. *Proceedings of the ACM on Programming Languages (PACMPL)* 3, POPL, Article 15 (Jan. 2019), 31 pages. doi:10.1145/3290328

[52] Daniel Patterson. 2022. *Interoperability Through Realizability: Expressing High-Level Abstractions using Low-Level Code*. Ph. D. Dissertation. Northeastern University. https://dbp.io/pubs/2022/dbp-dissertation.pdf

[53] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[54] G.D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255. doi:10.1016/0304-3975(77)90044-5

[55] François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. 319–330. doi:10.1145/503272.503302

[56] François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems* 25, 1 (Jan. 2003), 117–158. doi:10.1145/596980.596983

[57] J.J.M.M. Rutten. 2000. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249, 1 (2000), 3–80. doi:10.1016/S0304-3975(00)00056-6 Modern Algebra.

[58] Taro Sekiyama and Atsushi Igarashi. 2017. Stateful Manifest Contracts. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 530–544. doi:10.1145/3009837.3009875

[59] Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *Programming Languages and Systems (ESOP '09)*. 17–31. doi:10.1007/978-3-642-00590-9_2

[60] Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015. Blame and coercion: together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 425–435. doi:10.1145/2737924.2737968

[61] Jeremy Siek, Peter Thiemann, and Philip Wadler. 2021. Blame and coercion: Together again for the first time. *Journal of Functional Programming* 31 (2021), e20. doi:10.1017/S0956796821000101

[62] Jeremy G. Siek and Tianyu Chen. 2021. Parameterized cast calculi and reusable meta-theory for gradually typed lambda calculi. *Journal of Functional Programming* 31 (2021), e30. doi:10.1017/S0956796821000241

[63] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming 2006*. 12 pages. doi:10.1145/1163566.1163568 University of Chicago Technical Report TR-2006-06 http://scheme2006.cs.uchicago.edu/13-siek.pdf.

[64] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*. 2–27. doi:10.1007/978-3-540-73589-2_2

[65] Jeremy G. Siek and Sam Tobin-Hochstadt. 2016. The Recursive Union of Some Gradual Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 388–410. doi:10.1007/978-3-319-30936-1_21

[66] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*. 274–293. doi:10.4230/LIPIcs.SNAPL.2015.274

[67] Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without Blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. 365–376. doi:10.1145/1706299.1706342

[68] T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. 2013. Contracts for First-Class Classes. *ACM Transactions on Programming Languages and Systems* 35, 3, Article 11 (Nov 2013), 58 pages. doi:10.1145/2518189

[69] Cameron Swords. 2019. *A Unified Characterization of Runtime Verification Systems as Patterns of Communication*. Ph. D. Dissertation. Indiana University. http://cswords.com/paper/cswords.thesis.pdf

[70] Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2015. Expressing Contract Monitors as Patterns of Communication. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. 387–399. doi:10.1145/2784731.2784742

[71] Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2018. An extended account of contract monitoring strategies as patterns of communication. *Journal of Functional Programming* 28 (2018), e4. doi:10.1017/S0956796818000047

[72] Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. 2015. Towards Practical Gradual Typing. In *European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2015.4

[73] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. 456–468. doi:10.1145/2837614.2837630

[74] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. 793–810. doi:10.1145/2384616.2384674

[75] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining Delimited Control with Contracts. In *Programming Languages and Systems (ESOP '13)*, Matthias Felleisen and Philippa Gardner (Eds.). 229–248. doi:10.1007/978-3-642-37036-6_14

[76] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Dynamic Languages Symposium (DLS '06)*. 964–974. doi:10.1145/1176617.1176755

[77] Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Programming Languages and Systems (ESOP '10)*. 550–569. doi:10.1007/978-3-642-11957-6_29

[78] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. 762–774. doi:10.1145/3009837.3009849

[79] Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*. 309–320. doi:10.4230/LIPIcs.SNAPL.2015.309

[80] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems (ESOP '09)*. 1–16. doi:10.1007/978-3-642-00590-9_1

[81] Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: Higher-Order Contracts for Modern Services. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP, Article 36 (Aug 2017), 28 pages. doi:10.1145/3110280

[82] Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. The Root Cause of Blame: Contracts for Intersection and Union Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA, Article 134 (Oct 2018), 29 pages. doi:10.1145/3276504

[83] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. 41–52. doi:10.1145/1480881.1480889

[84] Shu-Hung You. 2025. Contract System Metatheories à la Carte: Supplementary Material. https://github.com/shhyou/monitor-calculus/tree/oopsla25-formalization. Retrieved on Aug 19, 2025.

Received 2025-03-24; accepted 2025-08-12