NORTHWESTERN UNIVERSITY

The Monitor Calculus: Modular Metatheories for Contract Systems

Flexible Specification with Annotations and Reusable Metatheories with Transition Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Shu-Hung You

EVANSTON, ILLINOIS

June 2025

Abstract

Modern contract systems generalize run-time assertions on boolean expressions to higher-order functions and beyond, equipping programmers with a lightweight tool for specifying and monitoring the behavior of programs. To date, researchers have studied the theory of contract systems in every little detail, from the design of expressive notations for contracts to the correct enforcement of contracts for complex programming language constructs, and even to the asymptotic behavior of the contract system.

Although each paper contributes unique insight to the study of contract systems, researchers are bound to define custom contract calculi in order to formulate their desired metatheoretic properties. Each new contract calculus may include additional annotations that track necessary invariants for proving the metatheoretic property and custom evaluation rules of the contracts. As a result, the literature contains a large number of similar but subtly different contract calculi which require a considerable amount of repeated labor.

I address the problem with a novel transition-system-based representation of higher-order contracts. The new representation allows for building connections between a contract system and its proof invariants, and further permits the reuse of metatheories. Specifically, the transition-system representation is based on a parameterized calculus which monitors higher-order values with proxies and generates parameter-determined transition events. The calculus has a fixed base language and a predetermined monitoring strategy, but the annotations that the proxies carry are parameterized. Consequently, extra invariants needed for proving metaproperties can be encoded as custom transition events generated by the proxies, and the collection of all events

naturally forms a transition system. With such transition systems, the metaproperties manifest themselves as various transition structures and hence metatheories can be reused by relating different transition systems with homomorphisms.

In this dissertation, I make three contributions based on the transition-system representation of contracts. First, I present *the monitor calculus* and its transition-system-based metatheory. Through the monitor calculus, I formulate properties of contract systems as particular transition structures of transition systems. Second, I apply the new representation to contract systems in the literature to construct a collection of reusable metatheories for contracts, including Findler and Felleisen's higher-order contract system, its blame tracking mechanism, and Greenberg's space-efficient contracts. Last, I mechanize the monitor calculus together with its metatheory and applications in Agda for the modular development of the metatheory of contracts.

Glossary of Notations

Notation	Terminology	Page	Location
$(\mathscr{A}, \mathscr{T})$	annotation language	page 57	Section 4.2
The parameters of the monitor calculus, including the definition of A and s, and the relations defining how annotations propagate and and how the global state changes in $\longrightarrow_m$.			
A	annotation	page 59	Definition 4.7, Section 4.2
s	global states	page 59	Definition 4.7, Section 4.2
$\longrightarrow$	the reduction relation	page 60	Definition 4.8, Section 4.2
$\longrightarrow_{\mathrm{p}}$	program-related reduction relation	page 55	Figure 4.3, Section 4.1
The reduction rules of non-boundary and non-proxy forms.			
$\longrightarrow_{\mathrm{m}}$	monitor-related reduction relation	page 61	Figure 4.4, Section 4.2
The reduction rules of boundary forms and proxy forms.			
$\lambda_{\mathsf{m}}[\mathscr{A}; \mathscr{T}]$	an instantiation	page 57	Definition 4.7, Section 4.2
An instantiation of the calculus where the parameters are instantiated with $\mathscr{A}$ *and* $\mathscr{T}$.			
$\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$	the induced transition system	page 68	Definition 5.3, Section 5.1
The transition system induced by the instance $\lambda_m[\mathscr{A}; \mathscr{T}]$.			
$\mathcal{I}$	interpretation of an annotation language	page 70	Definition 5.4, Section 5.2,
A property of the annotations and the global states.			
$\mathcal{I} \vDash e$	the satisfaction relation	page 72	Figure 5.1, Section 5.2
$\mathcal{I} \vDash^j e$		page 118	Figure 7.16, Section 7.4
The relation that determines whether a term e satisfies $\mathcal{I}$ *(at j).*			
$\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}]$ $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}, j]$	the interpretation-satisfying transition system	page 75	Definition 5.14, Section 5.3
The transition system that satisfies $\mathcal{I}$ *by construction.*			
$(\pi_{\mathsf{A}}, \pi_{\mathsf{S}})$	projection of annotation languages	page 65	Definition 4.11, Section 4.3
The pair of functions that projects an annotation language to another annotation language.			

Contents

List of Figures

Part I

Introduction

Chapter 1

The Problem with Metatheory Reuse

Formal specification is an essential tool in the development of reliable software [Hoare 1969; Parnas 1972; Lamport 1994, 2002]. Such specifications document the expected behavior of software in a formal language, thereby opening the door to automatic enforcement of specifications in programming languages. Among many embodiments of the idea, Meyer [1991a, 1992]'s *contracts* in Eiffel [Meyer 2005] systematically reorganize isolated run-time assertions into *preconditions* and *postconditions* at function boundaries to specify the behavior of functions. The preconditions describe the expectations of the function's input so the clients of the functions understand what to supply and the postconditions describe expectations of the function's output so the author of the function understands what to produce.

Although modern contract systems generalize run-time assertions on boolean expressions to higher-order functions and beyond, they follow the same architecture as Eiffel contracts. More concretely, Eiffel enforces the conditions prescribed by the contracts via monitoring each pre- and post-condition at every function call. Whenever a value that passed through a function boundary fails to comply with the contracts, the contract system signals an error to protect the body of the function from unwanted inputs. Thus, Eiffel guarantees that the runtime behavior of all functions meet the specifications coded in the contracts, effectively providing a tool for the programmers to document the otherwise implicit behavior of their programs.

$$\begin{array}{l}
\|\mathsf{mon}^{\ell_p,\ell_n}(\lfloor \mathsf{even} \rfloor^{\ell_n} \to\!\!\!\!/\mathsf{c} \lfloor \mathsf{odd} \rfloor^{\ell_p}, \|\lambda x.x\|^{\ell_p})\|^{\ell_n}\ \|5\|^{\ell_n} \longrightarrow \\
\|\mathsf{mon}^{\ell_p,\ell_n}(\lfloor \mathsf{odd} \rfloor^{\ell_p},\ \|\lambda x.x\|^{\ell_p}\ \|\mathsf{mon}^{\ell_n,\ell_p}(\lfloor \mathsf{even} \rfloor^{\ell_n},\ \|5\|^{\ell_n})\|^{\ell_p})\|^{\ell_n} \longrightarrow \\
\textsc{Err}(\ell_n)
\end{array}$$

Figure 1.1: An example reduction sequence from CPCF

Contracts occupy a unique position in the landscape of formal specifications. They not only enable the expression of custom and complex conditions [Dimoulas et al. 2016], their dynamic nature also offers a low barrier when scaling to advanced language features [Strickland et al. 2013; Strickland and Felleisen 2009; Takikawa et al. 2013; Moy et al. 2024]. Both the expressiveness and the scalability of contracts contributed to their popularization as witnessed by the Racket [Felleisen et al. 2015, 2018] programming language.

When establishing the metatheory of a new contract calculus, however, researchers often need to introduce new syntactic constructs to track invariants needed by the metatheory, and porting theorems from a previous contract calculus demands a considerable amount of tedious routine work. As Greenberg [2016] writes in his study of space-efficient contracts, "changing our calculus to have a more interesting notion of blame, like indy semantics [Dimoulas et al. 2011] or involutive blame labels [Wadler and Findler 2009; Wadler 2015], would be a matter of pushing a shallow change in the semantics through the proofs" (p. 31).

Figure 1.1 displays an concrete example reduction sequence for the contract calculus developed by Dimoulas et al. [2012]. In addition to the contract, $\lfloor \mathsf{even} \rfloor^{\ell_n} \to\!\!\!\!/\mathsf{c} \lfloor \mathsf{odd} \rfloor^{\ell_p}$, the expressions in the figure include additional information such as the obligation labels on the contracts, $\lfloor - \rfloor^{\ell}$, and the ownership annotations, $\|-\|^{\ell}$. Consequently, Dimoulas et al. [2012] need to augment their language to handle the evaluation of these additional constructs and repeat proofs that have been done for the original, unannotated language.

My work aims to solve this problem, in a very general sense. Specifically, I cast the problem of reusing metatheories of contracts as the problem of *composing* contract calculi and reusing the metatheories of the individual calculus. In this dissertation, I present a novel framework that unifies the representation of contracts. The new representation supports building calculi that

capture extra artifacts needed for the metatheory of a contract system, proving properties about the artifacts within the same framework, composing all the artifacts together on top of a fixed base calculus, and transferring properties about the artifacts to the composite calculus. Furthermore, the artifacts are reusable across different calculi, thus forming reusable metatheories of contract systems. In the rest of the introduction, I shall analyze contract systems in the literature and introduce the key idea of my work.

1.1 Proxies in Finder-Felleisen Higher-Order Contracts

While there are many different formalizations of contract systems, they share a common basis. In Findler and Felleisen [2002]'s original design, monitors wrap around contracted functions to intercept their applications and insert contract checks. Similarly, Siek and Taha [2006] design the cast calculus to serve as an intermediate language for gradually type languages, with higher-order casts guarding against functions to insert dynamic type checks. Subsequent research either attach more information to monitors and casts such as Dimoulas et al. [2011, 2012]'s proof of the Correct Blame Theorem for CPCF and Wadler and Findler [2009]'s addition of blame to the cast calculus, or they enrich the computation of monitors with a merging operation to achieve space-efficiency [Herman et al. 2010; Greenberg 2014, 2015, 2016]. Siek and Wadler [2010] particularly study how a general class of merge operation should handle blame to achieve both space efficiency and correctness of blame.

From this observation, I single out monitors — or casts in gradual typing — as the primary syntactic construct for constructing a new representation of contracts. In my dissertation, I call this construct *proxies*. Proxies replace contracts and blame labels on monitors by an abstract set of annotations to reify proof invariants of contract systems merely as different kinds of annotations. The merging of space-efficient contracts can be abstractly represented as computation on the annotations. Consequently, the introduction of annotation-carrying proxies transforms the combination of contracts and the invariants on them into the composition of annotations.

(a) Ok, $\mathsf{proxy}(\mathsf{even} \not\rightarrow_{\mathsf{c}} \mathsf{odd},\ \lambda x.x)\ 5 \longrightarrow_{\mathrm{m}}$
Ok, $\mathbf{B}\#\mathsf{odd}\ \{\ (\lambda x.x)\ (\mathbf{B}\#\mathsf{even}\ \{\ 5\ \})\ \} \longrightarrow_{\mathrm{m}}$
Err, $\mathbf{B}\#\mathsf{odd}\ \{(\lambda x.x)\ 5\} \longrightarrow_{\mathrm{p}}$
Err, $\mathbf{B}\#\mathsf{odd}\ \{\ 5\ \}$

(b) $(), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.x\big)\ 5 \longrightarrow_{\mathrm{m}}$
$(), \mathbf{B}\#\langle \ell_n, \ell_p \rangle\ \big\{\ (\lambda x.x)\ (\mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{\ 5\ \})\ \big\} \longrightarrow_{\mathrm{m}}$
$(), \mathbf{B}\#\langle \ell_n, \ell_p \rangle\ \{(\lambda x.x)\ 5\} \longrightarrow_{\mathrm{p}}$
$(), \mathbf{B}\#\langle \ell_n, \ell_p \rangle\ \{\ 5\ \} \longrightarrow_{\mathrm{m}} (), 5$

Figure 1.2: (a) An example contracted function and (b) labeling the owner of the expressions

To allow modular and reusable development of contract metatheories, I develop the *monitor calculus*, a novel foundation for Findler and Felleisen [2002]'s higher-order contracts. The monitor calculus extends the lambda calculus with proxies that monitor the application of higher-order functions. Although I choose a fixed set of language features and a predetermined monitoring strategy for the monitor calculus, the proxies are augmented with annotations and each use of the proxy can be associated with a custom action on the annotation.

To be more concrete, Figure 1.2 shows an example reduction sequence in my calculus. Its constructs include a *proxy*, written as $\mathsf{proxy}(A,\ v)$, that monitors a higher-order value, carrying a custom annotation A, and intervenes on the evaluation when the higher-order value is accessed. In the example on the left, the annotation $\mathsf{even} \not\rightarrow_{\mathsf{c}} \mathsf{odd}$ is an actual contract that specifies the behavior of the monitored value. The *boundaries*, written as $\mathbf{B}\#A\ \{e\}$, are similar to proxies except that they wrap around arbitrary expressions and only takes action when the nested expression evaluates to a value.

In the monitor calculus, although the evaluation rules of proxies and boundaries are not customizable, the annotations and their computation are. Beyond contracts, proof artifacts such as the ownership labels [Dimoulas et al. 2011, 2012] are also expressible as annotations as displayed on the right of Figure 1.2. In other words, the monitor calculus is capable of expressing not only the base contract system, but also extra proof information needed for building the metatheory, and all of these use cases are unified in a single calculus.

I defer the details of the monitor calculus to Chapter 2. Beyond these examples, the annotations and their computation can be customized to approximate more contract systems in the literature, and I have also used it to model Greenberg [2016]'s space-efficient contracts together

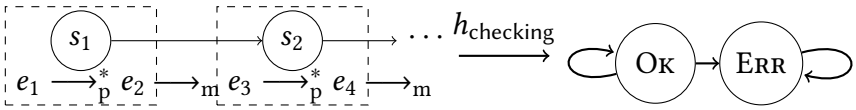

Figure 1.3: Proving properties for the transition system derived from the monitor calculus

with its two completely distinct types of proof artifacts for proving the time complexity and the functional correctness of space-efficient contracts.

1.2 Contracts as Transition Systems

In the literature, proofs about contract systems often involve constructing an invariant of a calculus that is annotated with additional proof artifacts, with the correct blame property [Dimoulas et al. 2011, 2012; Takikawa et al. 2012] being a prominent example. While some other examples such as Greenberg [2016]'s space-efficient contract do not introduce additional annotations on contracts, its proof also establishes invariants on the contracts. This pattern—associating extra structure on contracts, either in the calculus or in the proofs—can be abstracted and unified by switching to a *transition-system view* of contract systems.

Concretely, the reduction rules of the monitor calculus can be divided into two sets: those that reduce boundaries and proxies, and those that reduce common language constructs like functions. Figure 1.2 specifically marks monitor-related reductions using $\longmapsto_{\mathrm{m}}$ and other reductions using $\longrightarrow_{\mathrm{p}}$. The $\longmapsto_{\mathrm{m}}$ reductions capture the instantiation-specific actions associated with proxies whereas the $\longrightarrow_{\mathrm{p}}$ reductions are identical for all instantiations. This separation of reductions naturally gives a transition system that is the basis of a reusable metatheory.

As a visualization of the idea, the left of Figure 1.3 illustrates the transition system induced by an instantiation of the monitor calculus. The states are formed by pairing the global states with the set of expressions that are equivalent under $\longrightarrow_{\mathrm{p}}$ as visualized by dashed rectangles in the figure. In addition, one state of the transition system can transit to another if there is a boundary-related reduction ($\longmapsto_{\mathrm{m}}$) between any of the expressions in the two states.

The induced transition systems enable one to establish properties of the instantiations of the

$$\langle (), \textsc{Ok} \rangle,\ \mathsf{proxy}\big(\langle \langle \ell_n, \ell_p \rangle, \mathsf{even} \rightarrow\!\!\!\!/_{\mathsf{c}}\ \mathsf{odd} \rangle,\ \lambda x.x\big)\ 5 \longrightarrow_{\mathrm{m}}$$
$$\langle (), \textsc{Ok} \rangle,\ \mathbf{B}\#\langle \langle \ell_n, \ell_p \rangle, \mathsf{odd} \rangle \left\{ (\lambda x.x)\ (\mathbf{B}\#\langle \langle \ell_p, \ell_n \rangle, \mathsf{even} \rangle \{ 5 \}) \right\} \longrightarrow_{\mathrm{m}}$$
$$\langle (), \textsc{Err} \rangle, \mathbf{B}\#\langle \langle \ell_n, \ell_p \rangle, \mathsf{odd} \rangle \{(\lambda x.x)\ 5\} \longrightarrow_{\mathrm{p}} \cdots$$

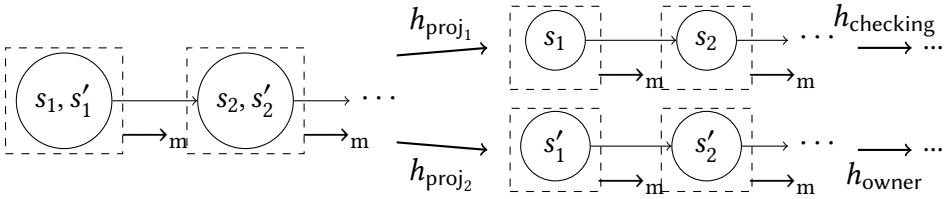

Figure 1.4: Combining contracts with invariant.

monitor calculus by studying its behavior. For example, normally a contract violation immediately terminates the evaluation of programs. This means that in Figure 1.2 (b), once the global state is changed from Ok to Err, it should never change back to Ok. Speaking in term of transition systems, this property about the global state can be proven by establishing the homomorphisn h_{checking} from the induced transition system at the left of Figure 1.3 to the one at the right, since it has no edge from the state Err to the state Ok. In other words, the proof is the construction of the transition system at the right of the figure and the fact that h_{checking} is a homomorphism.

This approach for proving properties generalizes to other instantiations of the monitor calculus as well and I have applied it to re-establish Dimoulas et al. [2011, 2012]'s Correct Blame Theorem for the instantiation of the monitor calculus in Figure 1.2 (c) using the same method. In fact, the framework I develop includes a mechanized recipe for systematically constructing such homomorphisms and transition systems.

1.3 Reusable Metatheory

In the general case, proving properties of the instantiated monitor calculus by establishing homomorphisms is just as easy or difficult as proving them using a bespoke calculus, but the key benefit of the transition system representation of contracts is metatheory reuse. Once properties about the individual proof artifact have been proven, they can be easily carried to the complete contract system by pairing annotations and global states.

Figure 1.4 illustrates this approach in pictures. The top half of the figure pairs up contracts and ownership labels [Dimoulas et al. 2011] together in the instantiation. Similar to the visualization in Figure 1.3, the bottom left shows the transition system corresponding to the reduction sequence at the top of the figure, and proving properties of the combined instantiation amounts to studying the behavior of this transition system.

Since the combined instantiation simply pairs the annotations and the global states, there are homomorphisms h_{proj_1} and h_{proj_2}[1] from the induced transition system of the combined instantiation to that of the individual instantiations from Figure 1.2. Furthermore, since the composition of homomorphisms is again a homomorphism, properties of the individual proof artifact automatically hold for the combined one. In Figure 1.4, the bottom right illustrates this composition in action. The homomorphism h_{checking} is just the same one from Figure 1.3 and it corresponds to the proof that the global state never changes from Ok to Err.

1.4 Thesis Statement

Combining the monitor calculus, which is capable of capturing contract systems, and proof artifacts needed for their metatheory, with the idea that behaviors of the instantiations of the monitor calculus can be studied through its induced transition system, my dissertation defends the following statement.

> **Thesis Statement.** Transition-system-based theory offers a foundation for constructing **modular metatheories** for contract systems.

To support the thesis, I develop the idea presented in Section 1.1 into a full-fledged and parameterized calculus that abstract over Findler and Felleisen [2002]'s contract system to capture its variants. I further develop the metatheory of the this calculus and demonstrate that properties of contract systems can be established through studying the behavior of the corresponding transition systems. The transition-system-based approach from Sections 1.2 and 1.3 also enables me to

[1] Technically, these are weak homomorphisms.

automate Greenberg [2016]'s "shallow change." Specifically, the dissertation demonstrates how to combine contracts and ownership labels [Dimoulas et al. 2011], just as Figure 1.4 illustrates.

Scope and Goals. The calculus presented in this dissertation fixes the base language, the set of intervention points of the contract system, and the combinator language of contracts. However, the calculus permits customizing the checking strategies of contracts and the additional artifacts for tracking invariants of the contracts.

Validation. To validate the thesis statement, I have used my framework to prove Dimoulas et al. [2011]'s blame correctness theorem for my model of CPCF, as well as the correctness of Greenberg [2016]'s space-efficient contract in my calculus. The proofs and the framework are mechanized in about 15k lines of Agda code and, more over, the instantiations are modular—the mechanization of the annotations can be freely combined with new annotations, and the proofs can be carried to the combined annotations without undue work. The metatheory of my calculus and the applications presented in the dissertation are mechanized in Agda, available at

https://plt.cs.northwestern.edu/shuhung-phd/

1.5 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2237984 and Grant 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.

1.6 Related Work

My work lives in the context of the vast literature on the foundations of (higher-order) contracts [Blume and McAllester 2004; Degen et al. 2012; Dimoulas and Felleisen 2011; Dimoulas et al. 2011, 2014, 2009, 2012; Disney et al. 2011; Findler and Blume 2006; Findler and Felleisen

2002; Greenberg 2015, 2016; Greenberg et al. 2010; Hinze et al. 2006; Keil and Thiemann 2015; Melgratti and Padovani 2017; Sekiyama and Igarashi 2017; Strickland et al. 2013; Swords et al. 2015, 2018; Takikawa et al. 2013; Tov and Pucella 2010; Williams et al. 2018; Xu et al. 2009] and those of contracts' most prominent application, gradual typing [Ahmed et al. 2017; Allende et al. 2013; Feltey et al. 2018; Garcia 2013; Garcia et al. 2016; Gierczak et al. 2024; Greenman et al. 2023; Greenman and Felleisen 2018; Greenman et al. 2019; Gronski and Flanagan 2007; Herman et al. 2010; Matthews and Ahmed 2008; New et al. 2019; Siek et al. 2009, 2015a, 2021; Siek and Chen 2021; Siek and Taha 2006, 2007; Siek and Tobin-Hochstadt 2016; Siek et al. 2015b; Siek and Wadler 2010; Tobin-Hochstadt and Felleisen 2006; Takikawa et al. 2012; Vitousek et al. 2017; Wadler 2015; Wadler and Findler 2009].

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. [2019, 2023], 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.

Of course, I am not the first to observe that the metatheory of contract systems (and gradual typing) is rife with repetition. Siek and Chen [2021] 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. [2024] 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 terms of its proofs for the different

instances of the model. Swords [2019] and Swords et al. [2015, 2018] build a model based on Concurrent ML that uniformly represents contract enforcement mechanisms as communicating processes, essentially explicating their workings as programs. However, they stop short of taking advantage of their model to abstract over properties and proofs. My work is the first that aims for a general unifying framework for the metatheory of contracts that puts an emphasis on proof reuse.

As a final note, besides the prior work on contracts that I revisit in the dissertation, an important, direct source of inspiration for my endeavor is the work on the foundations of safe language interoperability [Matthews and Findler 2007; Buro and Mastroeni 2019; Patterson 2022]. Specifically, the design of the monitor calculus owes to that line of work the idea of boundaries and proxies as the building block for a unifying, parameterized framework for the study of contract systems.

Chapter 2

The Monitor Calculus

A Parameterized Contract Calculus

As a prelude to the full formal definition, I give a brief overview of *the monitor calculus*, the core device in my dissertation for building the theory. To be more concrete, the monitor calculus is a parameterized calculus that can be instantiated to capture various monitoring systems in the literature including Findler and Felleisen [2002]'s contract system and Greenberg [2016]'s variant of space-efficient latent contracts. In Section 2.1, I introduce the monitor calculus, which is inspired by Dimoulas and Felleisen [2011]'s CPCF. Section 2.2 summarizes the parameters of the calculus together with the definition of certain terminology that I use throughout the dissertation. Finally, Section 2.3 discusses an approach for building contract systems that can be composed with other proof artifacts.

2.1 Intercepting Monitor-Related Events

The monitor calculus is a generalization of Dimoulas and Felleisen [2011]'s CPCF, a core calculus for contracts. To capture contract systems, the monitor calculus adds two language constructs to the conventional call-by-value λ-calculus: a *boundary*, $\mathbf{B}\#\kappa\,\{e\}$, that represents a component e protected by the contract κ and a *proxy*, $\mathrm{proxy}(\kappa,\ v)$, a value v carrying the contract κ.

Proxies can be attached to higher-order values such as functions and boxes. When a higher-order value is wrapped by a proxy, all operations applied on it are intercepted by the proxy to allow the intervention of the monitoring system. For example, consider the evaluation sequence about the application of a contracted function in Figure 2.1.

$$
\begin{aligned}
& \textsc{Ok},\ \mathsf{proxy}(\mathsf{isOdd} \rightarrow\!\!\!\!/\, \mathsf{isEven},\ \lambda x.x + 2)\ 5 \\
\longrightarrow\ & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ (\lambda x.x + 2)\ (\mathbf{B}\#\mathsf{isOdd}\ \{\ 5\ \})\ \} \\
\longrightarrow\ & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ (\lambda x.x + 2)\ 5\ \} \\
\longrightarrow\ & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ 5 + 2\ \} \\
\longrightarrow\ & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ 7\ \} \\
\longrightarrow\ & \textsc{Err}, 7
\end{aligned}
$$

Figure 2.1: An example of applying a contracted function

In Figure 2.1, the subject of the evaluation sequence is a pair that includes the global state of the monitoring system and the program being evaluated. In this example, the global status is either Ok or Err, indicating whether there have been any contract violations or not. The initial program contains a function $\lambda x.x + 2$ that has the contract $\mathsf{isOdd} \rightarrow\!\!\!\!/\, \mathsf{isEven}$. This is denoted by a proxy that carries both the contract and the function being monitored.

The example applies the contracted function to 5. After the application step, the program reduces into two boundary terms where one guards the application's result using the range contract and the other guards the argument using the domain contract. In the last step, since the result does not satisfy the range of $\mathsf{isOdd} \rightarrow\!\!\!\!/\, \mathsf{isEven}$, the monitoring system sets the global state to Err.

The sharp reader may notice that contract violations only affect the global state of the monitoring system rather than terminating the entire program with error like what CPCF does. In fact, this is the first step towards representing contracts as transition systems. I shall discuss this issue more in Chapter 3.

The monitor calculus can also be used to model monitoring systems other than contract checkers. For example, it can model the system that tracks the origin of each source component. In Figure 2.2, I show an instance of this idea using the ownership information that Dimoulas et al.

[2011, 2012] designed in their study of the blame correctness property. In their work, boundaries divide the program into multiple parts where each part is assigned an ownership label. Conceptually, the ownership labels represent standalone source components in the original program.

Figure 2.2 repurposes the program from Figure 2.1 to demonstrate how the monitor calculus can track the ownership labels. The program itself remains unchanged, but the boundaries and proxies do not carry contracts anymore. Instead, they are annotated with a pair of labels naming the owners outside and inside the boundaries.

The colors on the program distinguish the role of each subexpression and are synthesized from the annotations. Black represents boundary or proxy terms dividing the program into different components. Brown represents the caller $\lambda x.x + 2$ and blue represents the callee. This time, since the ownership information is fully captured by the annotations, the global state of the monitor calculus is the unit value, $()$, instead of Ok and Err.

$$
\begin{aligned}
& (), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.x + 2\big)\ 5 \\
\longrightarrow\ & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{\ (\lambda x.x + 2)\ (\mathbf{B}\#\langle \ell_p, \ell_n \rangle \left\{\ 5\ \right\})\ \right\} \\
\longrightarrow\ & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{\ (\lambda x.x + 2)\ 5\ \right\} \\
\longrightarrow\ & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{\ 5 + 2\ \right\} \\
\longrightarrow\ & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{\ 7\ \right\} \\
\longrightarrow\ & (), 7
\end{aligned}
$$

Figure 2.2: Labeling the owner of the subexpressions

At the beginning of the evaluation sequence in Figure 2.2, the proxy carries the ownership labels $\langle \ell_n, \ell_p \rangle$. The label ℓ_n is the name of the caller, $[]\ 5$, and ℓ_p is the name of the callee, $\lambda x.x + 2$. In the second step, the proxy intercepts the function call and pushes the argument 5 into the boundary. Since the argument belongs to the callee, it is colored brown. The boundary around 5 flipped the labels to be $\langle \ell_p, \ell_n \rangle$ as the context outside is part of the callee.

In the third step, 5 passes thought the boundary around it. Therefore, its color becomes blue, signifying that it is part of ℓ_p. The evaluation continues inside ℓ_p until the last step where the result of the function call, 7, passes through the boundary again to return to the caller. Thus, its color changes from blue to brown in the last step.

$$
\begin{array}{ll}
 & \textsc{Ok},\ \mathsf{proxy}(\mathsf{isOdd} \not\rightarrow_{\mathsf{c}} \mathsf{isEven},\ \lambda x.x+2)\ 5 \\
\longrightarrow_{\mathrm{m}} & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ (\lambda x.x+2)\ (\mathbf{B}\#\mathsf{isOdd}\ \{\ 5\ \})\ \} \\
\longrightarrow_{\mathrm{m}} & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ (\lambda x.x+2)\ 5\ \} \\
\longrightarrow_{\mathrm{p}} & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ 5+2\ \} \\
\longrightarrow_{\mathrm{p}} & \textsc{Ok},\ \mathbf{B}\#\mathsf{isEven}\ \{\ 7\ \} \\
\longrightarrow_{\mathrm{m}} & \textsc{Err}, 7
\end{array}
$$

Figure 2.3: Classifying reduction steps into $\longrightarrow_{\mathrm{p}}$ and $\longrightarrow_{\mathrm{m}}$

2.2 A Unified Representation of Contracts and Proof Invariants

The structural similarity of the programs in Figure 2.1 and Figure 2.2 is a hint at an abstraction opportunity. In fact, the monitor calculus provides an unified account of extensions over contract systems by abstracting the contracts on boundaries and proxies into *annotations* that can be instantiated in different ways. Accompanying this change, evaluation rules that propagate contracts are also generalized to include arbitrary computation over the annotations; the status of monitoring systems is also replaced by arbitrary states. Reflecting changes into the notation, I will use A to denote the annotations, s to denote the global states and use the $\longrightarrow_{\mathrm{m}}$ relation to capture the so called monitor-related reductions. I will name other kinds of reductions by $\longrightarrow_{\mathrm{p}}$, the program-related reductions.

The $\longrightarrow_{\mathrm{m}}$ relation involves boundaries and proxies that carry annotations. Whereas the computation of the annotations varies from monitoring system to monitoring system, the $\longrightarrow_{\mathrm{m}}$ relation determines the creation, the elimination and the propagation of boundaries and proxies, but leaves open the computation rules of the annotations and the states. Since the $\longrightarrow_{\mathrm{p}}$ relation does not reduce boundaries and proxies, the reduction steps are the same for all annotations. Figure 2.3 illustrates this classification of the reduction steps by labeling each evaluation step in Figure 2.1 as either $\longrightarrow_{\mathrm{m}}$ or $\longrightarrow_{\mathrm{p}}$. To give a sense of the overall changes to the monitor calculus, here are

two reduction rules in the $\longrightarrow_{\mathrm{m}}$ relation relevant to the evaluation sequence in Figure 2.3.

$$\text{[R-Cross-Nat]} \quad s, \mathbf{B}\#A\,\{\,n\,\} \longrightarrow_{\mathrm{m}} s', n$$

$$\text{[R-Proxy-}\beta\text{]} \quad s, \mathsf{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} s', \mathbf{B}\#A_r\,\{\ (\lambda x.e)\ (\mathbf{B}\#A_a\,\{\,v\,\})\ \}$$

The [R-Cross-Nat] rule specifies how a number migrates through a boundary in the monitor calculus. In this rule, it is entirely up to the individual instantiation of the monitor calculus to specify the change of the global state from s to s'. Next, the [R-Proxy-β] rule specifies the general pattern of the application of a proxied function. The [R-Proxy-β] rule also allows potential changes to the global state as the metavariables s and s' suggest. In addition, the rule shows that the annotation A turns into two new annotations A_a and A_r after a step. As with the case for the global state, the relationship between A, A_a and A_r is up to the specific instantiation.

Annotation Languages, and Instantiations. Summarizing the changes up to this point, any instantiation of the monitor calculus is determined by *two parameters*: the set of the annotations, $\mathsf{Ann} \ni A$, the set of the global states, $\mathsf{State} \ni s$, and their computation rules in the $\longrightarrow_{\mathrm{m}}$ relation. Hereafter, I shall use $\mathscr{A} :\equiv (\mathsf{Ann}, \mathsf{State})$ to refer to the set of the annotations and the global states. Similarly, I refer to $\mathscr{T}$ as the *transition steps* and use it to denote the collection of annotation computation rules in the $\longrightarrow_{\mathrm{m}}$ relation. Putting $\mathscr{A}$ and $\mathscr{T}$ together, an *annotation language* is a pair $(\mathscr{A}, \mathscr{T})$. Given a specific annotation language $(\mathscr{A}, \mathscr{T})$, I shall denote the instantiated monitor calculus, i.e. the *instance*, by $\lambda_{\mathrm{m}}[\mathscr{A};\mathscr{T}]$.

Figure 2.4 shows the two annotation languages corresponding to the contract example from Figure 2.1 and the ownership example from Figure 2.2. In the contract example, $\mathscr{A}ctc$ is the set of contracts and the contract checking status. The transition steps, $\mathscr{T}c$, describes the contract checking process. In the ownership example, $\mathscr{A}owner$ gives the annotations, the ownership labels, and the global states, $\{()\}$. The transition steps, $\mathscr{T}o$, describes how the ownership labels propagate in an $\longrightarrow_{\mathrm{m}}$ step.

$\mathscr{A}ctc = (\{\kappa \mid \kappa \text{ is a contract}\}, \text{Status}) \qquad \kappa ::= \cdots \qquad \text{Status} \ni st ::= \{\textsc{Ok}, \textsc{Err}\}$
$\mathscr{T}c$ is:

[R-Cross-Nat]	$\textsc{Ok}, \mathbf{B}\#\kappa\,\{\,n\,\}$	$\longrightarrow_{\mathrm{m}}$	$\textsc{Ok}, n$ if n satisfies κ
	$\textsc{Ok}, \mathbf{B}\#\kappa\,\{\,n\,\}$	$\longrightarrow_{\mathrm{m}}$	$\textsc{Err}, n$ if n does not satisfy κ
	where κ is a predicate		
[R-Cross-Lam]	$\textsc{Ok}, \mathbf{B}\#(\kappa_a \to\!\!/\mathrm{c}\ \kappa_r)\,\{\,\lambda x.e\,\}$	$\longrightarrow_{\mathrm{m}}$	$\textsc{Ok}, \mathsf{proxy}(\kappa_a \to\!\!/\mathrm{c}\ \kappa_r,\ \lambda x.e)$
[R-Proxy-β]	$\textsc{Ok}, \mathsf{proxy}(\kappa_a \to\!\!/\mathrm{c}\ \kappa_r,\ \lambda x.e)\ v$	$\longrightarrow_{\mathrm{m}}$	$\textsc{Ok}, \mathbf{B}\#\kappa_r\,\{\ (\lambda x.e)\ (\mathbf{B}\#\kappa_a\,\{\,v\,\})\ \}$

$\mathscr{A}owner = (\{\langle \ell_n, \ell_p\rangle \mid \ell_p, \ell_n \in \mathsf{Label}\}, \{()\})$
$\mathscr{T}o$ is:

[R-Cross-Nat]	$(), \mathbf{B}\#\langle \ell_n, \ell_p\rangle\,\{\,n\,\}$	$\longrightarrow_{\mathrm{m}}$	$(), n$
[R-Cross-Lam]	$(), \mathbf{B}\#\langle \ell_n, \ell_p\rangle\,\{\,\lambda x.e\,\}$	$\longrightarrow_{\mathrm{m}}$	$(), \mathsf{proxy}\big(\langle \ell_n, \ell_p\rangle,\ \lambda x.e\big)$
[R-Proxy-β]	$(), \mathsf{proxy}\big(\langle \ell_n, \ell_p\rangle,\ \lambda x.e\big)\ v$	$\longrightarrow_{\mathrm{m}}$	$(), \mathbf{B}\#\langle \ell_n, \ell_p\rangle\,\big\{\ (\lambda x.e)\ \big(\mathbf{B}\#\langle \ell_p, \ell_n\rangle\,\{\,v\,\}\big)\ \big\}$

Figure 2.4: Selected rules of two instantiations of the monitor calculus

$\mathscr{A}octc = (\{\ \langle\langle \ell_n, \ell_p\rangle, \kappa\rangle \mid \ell_p, \ell_n \in \mathsf{Label},\ \kappa \text{ is a contract}\ \},\ \{((), \textsc{Ok}), ((), \textsc{Err})\})$
$\mathscr{T}oc$ is:

[R-Cross-Nat]	$\langle (), \textsc{Ok}\rangle, \mathbf{B}\#\langle\langle \ell_n, \ell_p\rangle, \kappa\rangle\,\{\,n\,\}$	$\longrightarrow_{\mathrm{m}}$	
	$\langle (), \textsc{Ok}\rangle, n$ if n satisfies κ		
	$\langle (), \textsc{Ok}\rangle, \mathbf{B}\#\langle\langle \ell_n, \ell_p\rangle, \kappa\rangle\,\{\,n\,\}$	$\longrightarrow_{\mathrm{m}}$	
	$\langle (), \textsc{Err}\rangle, n$ if n does not satisfy κ		
	where κ is a predicate		
[R-Cross-Lam]	$\langle (), \textsc{Ok}\rangle, \mathbf{B}\#\langle\langle \ell_n, \ell_p\rangle, \kappa_a \to\!\!/\mathrm{c}\ \kappa_r\rangle\,\{\,\lambda x.e\,\}$	$\longrightarrow_{\mathrm{m}}$	
	$\langle (), \textsc{Ok}\rangle, \mathsf{proxy}\big(\langle\langle \ell_n, \ell_p\rangle, \kappa_a \to\!\!/\mathrm{c}\ \kappa_r\rangle,\ \lambda x.e\big)$		
[R-Proxy-β]	$\langle (), \textsc{Ok}\rangle, \mathsf{proxy}\big(\langle\langle \ell_n, \ell_p\rangle, \kappa_a \to\!\!/\mathrm{c}\ \kappa_r\rangle,\ \lambda x.e\big)\ v$	$\longrightarrow_{\mathrm{m}}$	
	$\langle (), \textsc{Ok}\rangle, \mathbf{B}\#\langle\langle \ell_n, \ell_p\rangle, \kappa_r\rangle\,\big\{\ (\lambda x.e)\ \big(\mathbf{B}\#\langle\langle \ell_p, \ell_n\rangle, \kappa_a\rangle\,\{\,v\,\}\big)\ \big\}$		

Figure 2.5: A composite language of annotations

2.3 Building Composite Instantiations

A full-scale model for proving the correct blame property of a contract system should not only enforce contracts but also tracks the ownership information to facilitate identification of the correct blame. To the contrary, an annotation language often handles only one part of the calculus in a proof. The $\mathscr{A}ctc$ annotation language in Figure 2.4, for example, handles the contract checking aspect separately from the $\mathscr{A}owner$ annotation language that handles the ownership tracking aspect.

The annotation language that corresponds to a full-scale contract model should then combine the contract part together with its proof artifact, the ownership tracking part. In Figure 2.5, $\mathscr{A}octc$ is one such instance that includes both contracts and ownership labels. The annotations of $\mathscr{A}octc$ are pairs of contracts and ownership labels, and the transition steps propagates them simultaneously.

To see how $\mathscr{A}octc$ works, here is an example evaluation sequence of the combined monitoring system that applies the same contracted function to 5.

$$
\begin{aligned}
& \langle (), \textsc{Ok} \rangle, \mathsf{proxy}\big(\langle \langle \ell_n, \ell_p \rangle, \mathsf{isOdd} \rightarrow\!\!\!\!/_c\, \mathsf{isEven} \rangle,\ \lambda x.x + 2\big)\ 5 \\
\longrightarrow\ & \langle (), \textsc{Ok} \rangle, \mathbf{B}\#\langle \langle \ell_n, \ell_p \rangle, \mathsf{isEven} \rangle \left\{\ (\lambda x.x + 2)\ (\mathbf{B}\#\langle \langle \ell_p, \ell_n \rangle, \mathsf{isOdd} \rangle \{\ 5\ \})\ \right\} \\
\longrightarrow\ & \langle (), \textsc{Ok} \rangle, \mathbf{B}\#\langle \langle \ell_n, \ell_p \rangle, \mathsf{isEven} \rangle \{\ (\lambda x.x + 2)\ 5\ \} \\
\longrightarrow\ & \cdots
\end{aligned}
$$

From this evaluation sequence, it is clear that the composite instantiation propagates both the ownership labels $\langle \ell_n, \ell_p \rangle$ and the contracts $\mathsf{isOdd} \rightarrow\!\!\!\!/_c\, \mathsf{isEven}$ simultaneously without interfering with each other. This example may not be particularly interesting, but a practical use case of composite instantiations is to augment the transition steps and the global states of $\mathscr{A}ctc$ to track contract checking costs without modifying the original transition steps $\mathscr{T}c$. In that case, using a composite instantiation offers more confidence that the addition of cost tracking does not alter the semantics of contracts.

Although Figure 2.5 lists the syntax of the annotations and the transition steps, the composi-

$\mathscr{A} = (\text{Ann}, \text{State})$ $\quad \mathscr{A}ctc = (\{\kappa \mid \kappa \text{ is a contract}\}, \text{Status})$ $\quad \kappa ::= \cdots$ $\quad \text{Status} \ni st ::= \{\text{Ok}, \text{Err}\}$
$\mathscr{T}c'$ is:

$$
\begin{array}{lll}
\text{[R-Cross-Nat]} & s, \mathbf{B}\#A\,\{\,n\,\} & \longrightarrow_{\text{m}} \quad s, n \\
& \quad \text{if } n \text{ satisfies } \kappa \text{ and } \pi_{\text{S}}(s) = \text{Ok} & \\
& s, \mathbf{B}\#A\,\{\,n\,\} & \longrightarrow_{\text{m}} \quad s', n \\
& \quad \text{if } n \text{ does not satisfy } \kappa,\ \pi_{\text{S}}(s) = \text{Ok}, \text{ and } s' = \mathit{put}(s, \text{Err}) & \\
& \text{where } \pi_{\text{A}}(A) = \kappa \text{ and } \kappa \text{ is a predicate} & \\
\text{[R-Cross-Lam]} & s, \mathbf{B}\#A\,\{\,\lambda x.e\,\} & \longrightarrow_{\text{m}} \quad s, \text{proxy}(A',\ \lambda x.e) \\
& \text{where } \pi_{\text{S}}(s) = \text{Ok} \text{ and } \pi_{\text{A}}(A) = \pi_{\text{A}}(A') = \kappa_a \not\rightarrow_{\text{c}} \kappa_r & \\
\text{[R-Proxy-}\beta\text{]} & s, \text{proxy}(A,\ \lambda x.e)\ v & \longrightarrow_{\text{m}} \quad s, \mathbf{B}\#A_r\,\{\ (\lambda x.e)\ (\mathbf{B}\#A_a\,\{\,v\,\})\ \} \\
& \text{where } \pi_{\text{S}}(s) = \text{Ok}, \pi_{\text{A}}(A) = \kappa_a \not\rightarrow_{\text{c}} \kappa_r, \pi_{\text{A}}(A_r) = \kappa_r, \text{ and } \pi_{\text{A}}(A_a) = \kappa_a, &
\end{array}
$$

Figure 2.6: Extensible specifications of an instantiation

tion can in fact partially reuse the annotation languages of the individual instantiations. The key to enable this automation is to define the annotation languages in an extensible style.

Figure 2.6 rewrites ($\mathscr{A}ctc$,$\mathscr{T}c$) from Figure 2.4 in such extensible style. In the figure, the transition steps $\mathscr{T}c'$ is defined for an unspecified annotation language $\mathscr{A}$ such that there is a function π_{A} for extracting contracts from the annotation, another function π_{S} for extracting the contract checking status from the global state, and a third function *put* for updating the status in the global state.

Note that in the figure, the syntax of contracts is still defined in the same manner as $\mathscr{A}ctc$, but the transition steps $\mathscr{T}c'$ only specifies the contract-related part in the annotations and the global states. As such, the annotation language in Figure 2.6 can be instantiated with $\mathscr{A}octc$, the composite annotation language that pairs contracts with ownership labels, by supplying the appropriate π_{A}, π_{S}, and *put* functions that extract or update the corresponding components in the annotations and the global states, e.g. by

$$
\begin{array}{lcl}
\pi_{\text{A}}(\langle\langle \ell_n, \ell_p \rangle, \kappa\rangle) & = & \kappa \\
\pi_{\text{S}}(\langle (), st\rangle) & = & st \\
\mathit{put}(\langle (), st\rangle, st') & = & \langle (), st'\rangle
\end{array}
$$

In fact, this is the approach taken in my Agda implementation of the framework. For the purpose

of readability, however, in the rest of the dissertation I omit the reference to the functions π_A, π_S, and *put* since they make it difficult to read the rules for propagating the annotations.

Chapter 3

Proving Properties of Contract Systems via Transition Systems

In this chapter, I show how I view contract systems as transition systems. Specifically, for each instantiation of the monitor calculus, there is a naturally associated transition system whose transition behavior reflects properties of the instantiation in a natural way. As a result, studying the behavior of the associated transition system with homomorphisms leads to the understanding of the instantiation.

Starting from Section 3.1, I give a brief recap of transition systems and discuss how meta-theoretic properties of contract systems can be reframed as structures of the corresponding transition system that is associated with the instantiations of the monitor calculus. Section 3.2 demonstrates the idea through two specific properties of contracts and the corresponding transition systems. Moreover, as Section 3.3 explains, the proven properties can be reused when constructing composite monitoring systems through the composition of homomorphisms. Finally, Section 3.4 explores the applications of transition systems from a different perspective: can the study of transition systems offer a systematic method for proving properties of monitoring systems? Section 3.4 answers this question with a "yes" with the interpretation of annotations.

3.1 A Quick Recap of Transition Systems

Instantiations of the monitor calculus naturally induce transition systems that help study the behavior of the monitoring system and create reusable metatheory across instantiations. In my setup, this is done by mapping the *induced transition systems* into ones that are either better understood or contain more information. In this section, I briefly go over the definition of transition systems and their homomorphisms to prepare for the rest of the discussion.

Definition 3.1. A *transition system* $\mathcal{T} :\equiv (T, \rightarrow_t)$ is a set T and a binary relation $\rightarrow_t \subseteq T \times T$.

The theory of transition systems is the formal foundation of many areas [Sangiorgi 2009], including automata theory [Ginzburg 1968], the theory of concurrent processes [Hennessy and Milner 1985], model checking [Clarke et al. 1986], and more. In the early study of the theory of transition systems, homomorphisms are used to study the behavior of transition systems and their equivalences [Sifakis 1983; Arnold and Dicky 1989; Arnold and Castellani 1996]. In my work, homomorphisms are used for understanding the behavior of the transition system induced by instantiations of the monitor calculus. Below, I recap several related concepts about transition system homomorphisms needed in the dissertation.

Definition 3.2 (Homomorphism)**.** Let $\mathcal{S} :\equiv (S, \rightarrow_s)$ and $\mathcal{T} :\equiv (T, \rightarrow_t)$ be two transition systems. A *homomorphism* from $\mathcal{S}$ to $\mathcal{T}$ is a function $h : S \rightarrow T$ such that (i) for all $s, s' \in S$, if $s \rightarrow_s s'$ then $h(s) \rightarrow_t h(s')$, and (ii) for all $s \in S$ and $t' \in T$, if $h(s) \rightarrow_t t'$ then there exists $s' \in S$ such that $h(s') = t'$ and $s \rightarrow_s s'$.

Let me explain through an example how transition system homomorphisms help unravel specific behavior of a transition system that one may be interested in. Let $\preccurlyeq$ be the smallest preorder generated by Ok $\preccurlyeq$ Err and consider the system $\mathcal{T}_{err} :\equiv (\{\text{Ok}, \text{Err}\}, \preccurlyeq)$. For any transition system $\mathcal{S} :\equiv (S, \rightarrow_s)$, if we can construct a homomorphism $h : S \rightarrow \{\text{Ok}, \text{Err}\}$ from $\mathcal{S}$ to $\mathcal{T}_{err}$, it follows that the states in the set $h^{-1}(\{\text{Ok}\})$ can stay within the same group of states, or they can transit to one of the states in $h^{-1}(\{\text{Err}\})$. However, $\mathcal{S}$ has no transition from any state in $h^{-1}(\{\text{Err}\})$ to a state in $h^{-1}(\{\text{Ok}\})$.

Instantiations of the monitor calculus can be regarded as transition systems, so their behavior can be studied by mapping to other transition systems such as $\mathcal{T}_{err}$ as well. For example, both instantiations of the monitor calculus in Section 2.2 are transition systems where the states are the configuration, (s, e), and the transitions are the respective $\longrightarrow$ relation. In the example about contracts, if it can be proved that the map projecting the first component of the configuration is a homomorphism to $\mathcal{T}_{err}$, one learns that the monitor calculus never changes the s part of the configuration from Err to Ok.

Apart from the standard definition of homomorphisms, two more concepts are useful for studying the instantiations of the monitor calculus. First, in $\mathscr{A}owner$, the ownership labels example in Figure 2.4, not all combinations of label annotations on boundaries are reasonable. Therefore, only a *subsystem* of the transition system from the monitor calculus should be considered. Second, when composing multiple instantiations of the monitor calculus into one, the resulting instantiation may have a strictly "smaller" $\longrightarrow_{\mathrm{m}}$ relation because it may be the intersection of all $\longrightarrow_{\mathrm{m}}$ relations of the individual instantiation. Consequently, a weaker notion of homomorphism is needed to relate the transition system of the composite instantiation and the transition systems of the individual instantiation.

Definition 3.3. Given a transition system $\mathcal{T} :\equiv (T, \rightarrow_t)$, a *subsystem* of $\mathcal{T}$ consists of a subset $S \subseteq T$ that is closed under $\rightarrow_t$ together with the restriction of $\rightarrow_t$ to S. Formally, it is the system $(S, \{(t, t') \mid t, t' \in S \text{ and } t \rightarrow_t t'\})$.

Example 3.4. Given a transition system $\mathcal{T} :\equiv (T, \rightarrow_t)$ and a state $t \in T$, the minimum subsystem of $\mathcal{T}$ containing t has as states the intersection of all $T' \subseteq T$ such that $t \in T'$ and T' is closed under $\rightarrow_t$. It also equals the subsystem obtained by restricting T to those reachable from t.

Definition 3.5 (Weak Homomorphism)**.** Let $\mathcal{S} :\equiv (S, \rightarrow_s)$ and $\mathcal{T} :\equiv (T, \rightarrow_t)$ be two transition systems. A *weak homomorphism* from $\mathcal{S}$ to $\mathcal{T}$ is a function $h : S \rightarrow T$ such that for all $s, s' \in S$, if $s \rightarrow_s s'$ then $h(s) \rightarrow_t h(s')$.

Definition 3.5 removes condition (ii) from Definition 3.2, the definition of homomorphisms.

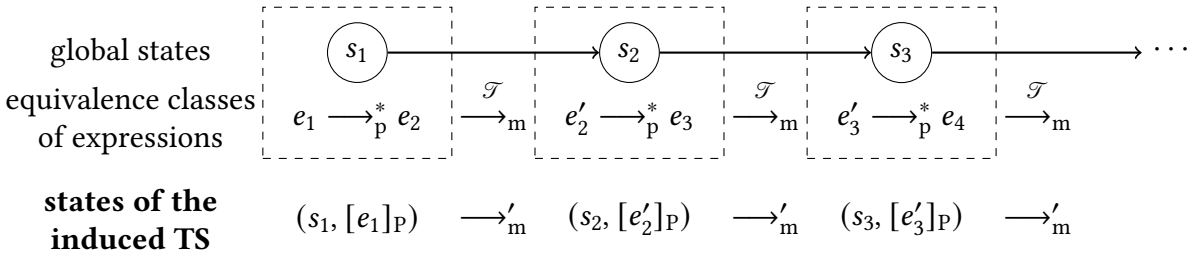

Figure 3.1: $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$, the transition system induced by the instantiation $\lambda_{\mathrm{m}}[\mathscr{A};\mathscr{T}]$

Conditions (i) and (ii) are also known as *preservation* and *reflection* of the transitions, respectively. As I will discuss in Section 3.3, when using (weak) homomorphisms to back-transport properties from all individual instantiations of the monitor calculus to the composite instantiation, the transitions of the composite instantiation are preserved but not all transitions from the individual instantiation are reflected.

3.2 From the Monitor Calculus to Transition Systems

In this section, I explain how an instantiation of the monitor calculus gives rise to a transition system and how properties of the instantiated calculus manifest as specific behaviors of the induced transition system. Consequently, this system can be used to prove properties of the instantiation. To illustrate this idea, I discuss two example properties about the contract annotation languages and the ownership label annotation language and examine how the respective induced transition system behaves.

Given an annotation language $(\mathscr{A}, \mathscr{T})$ where $\mathscr{A} :\equiv (\mathsf{Ann}, \mathsf{State})$, the notation $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ represents the *induced transition system* of the instantiation[1] $\lambda_{\mathrm{m}}[\mathscr{A};\mathscr{T}]$. Its states are the pairs formed by one s from State and one equivalence class of expressions partitioned by the $\longrightarrow_{\mathrm{p}}$ relation. The transition between the states, $\longrightarrow'_{\mathrm{m}}$, behaves like the $\longrightarrow_{\mathrm{m}}$ relation except that it is appropriately lifted to work on the sets of equivalence classes. More precisely, let Expr be the set of all

[1]The notations $\mathscr{A}$, $\mathscr{T}$, and $\lambda_{\mathrm{m}}[\mathscr{A};\mathscr{T}]$ are introduced in page 29 at the end of Section 2.2.

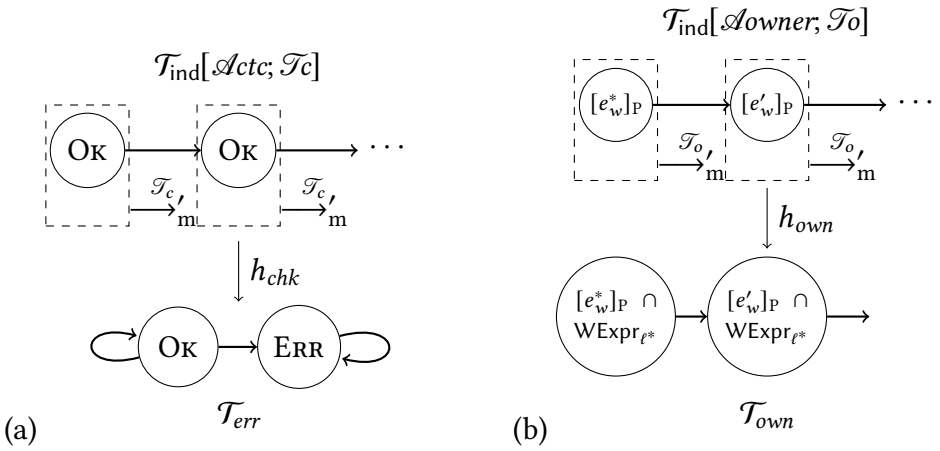

Figure 3.2: Proving properties by mapping to other transition systems: (a) the state changes only from Ok to Err, and (b) the ownership labels comply with the single-owner policy.

expressions of the monitor calculus, the induced transition system is

$$\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}] :\equiv (\{\, (s, [e]_{\mathrm{P}}) \mid s \in \mathsf{State} \wedge e \in \mathsf{Expr} \,\}, \longrightarrow'_{\mathrm{m}}).$$

Here, $[e]_{\mathrm{P}}$ denotes the equivalence class of e, i.e., it is the set $\{e' \mid e' \in \mathsf{Expr} \wedge e' \sim_{\mathrm{P}} e\}$ where $\sim_{\mathrm{P}}$ is the reflexive, symmetric and transitive closure of $\longrightarrow_{\mathrm{p}}$. The relation $\longrightarrow'_{\mathrm{m}}$ is defined by lifting $\longrightarrow_{\mathrm{m}}$ to the equivalence classes: $s, [e_1]_{\mathrm{P}} \longrightarrow'_{\mathrm{m}} s, [e_2]_{\mathrm{P}}$ iff $s, e'_1 \longrightarrow_{\mathrm{m}} s', e'_2$ for some $e'_1 \in [e_1]_{\mathrm{P}}$ and $e'_2 \in [e_2]_{\mathrm{P}}$.

Figure 3.1 partially visualizes this induced transition system. In the figure, each dashed rectangle encloses one state of of the transition system and $\longrightarrow'_{\mathrm{m}}$ draws the transitions between the states. For simplicity, the visualization only shows one $\longrightarrow_{\mathrm{p}}$ reduction sequence for each state instead of drawing the entire equivalence class. In the actual transition system, multiple states can transit to the same target.

The Non-masking Property of $\mathscr{A}ctc$. With the induced transition system of a given annotation language in hand, let me demonstrate how properties of instantiations of the monitor calculus can be reified as additional structures of the induced transition system using the contract annotation language, $\lambda_{\mathrm{m}}[\mathscr{A}ctc;\mathscr{T}c]$, as the first example. Concretely, a contract violation in CPCF immediately terminates the evaluation of the program. When modeling contract systems with $(\mathscr{A}ctc, \mathscr{T}c)$, however, the contract monitoring result is separately recorded in s as either Ok or Err. Consequently,

the evaluation of the program may continue even after a contract violation has been detected. To better match the behavior of CPCF, $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$ should possess the *non-masking* property of errors: if s ever becomes Err, it will never change back to Ok.

The non-masking property of the instantiated calculus, $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$, naturally transcribes to a property about the behavior of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc;\mathscr{T}c]$, i.e. its induced transition system: the s part in states of the induced transition system admits a preorder. Let $(s_1, [e_1]_{\mathrm{P}}) \preccurlyeq' (s_2, [e_2]_{\mathrm{P}})$ if and only if $s_1 = \textsc{Ok}$ or $s_2 = \textsc{Err}$, it can be seen that $\preccurlyeq'$ equips the induced transition system of $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$ with an additional preorder structure on its states. This preorder $\preccurlyeq'$ formally expresses that the induced transition system can transit from $(\textsc{Ok}, [e_1]_{\mathrm{P}})$ to $(\textsc{Err}, [e_2]_{\mathrm{P}})$ for some e_1, e_2 or stay unchanged, but the state never changes from $(\textsc{Err}, [e_2]_{\mathrm{P}})$ to $(\textsc{Ok}, [e_1]_{\mathrm{P}})$ for any e_1, e_2.

While it is straightforward to verify that the transition of the states via $\longrightarrow'_{\mathrm{m}}$ is monotonic with respect to $\preccurlyeq'$, an alternative and more systematic approach is to relate the induced transition system to a simpler transition system whose state transitions are obviously correct. As an example, recall that the transition system $\mathcal{T}_{err}$ is defined as $(\{\textsc{Ok}, \textsc{Err}\}, \preccurlyeq)$ where $\preccurlyeq$ is the smallest preorder generated by $\textsc{Ok} \preccurlyeq \textsc{Err}$. By showing that $h_{chk} : (s, [e]_{\mathrm{P}}) \longmapsto s$ is a homomorphism from $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc;\mathscr{T}c]$ to $\mathcal{T}_{err}$, the monotonicity of the states is proved using the argument in Section 3.1.

Figure 3.2 (a) visualizes the proof of the non-masking property that uses h_{chk}. The top of Figure 3.2 (a) is the induced transition system of $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$ and the bottom is $\mathcal{T}_{err}$. The map h_{chk} ignores the expressions of the induced transition system, leaving only the s part of the states. By inspecting $\mathcal{T}_{err}$, it is apparent that Err in $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$ cannot be changed to Ok because there is no corresponding transition in $\mathcal{T}_{err}$.

The Single-Owner Property of $\mathscr{A}owner$. More generally, properties of the instantiations of the monitor calculus can be rephrased as certain transition behavior of its induced transition system and subsequently proved by establishing homomorphisms from the induced transition system into a better understood one. This time, other than contracts, I show that the preservation of the single-owner property of $(\mathscr{A}owner, \mathscr{T}o)$, the language of ownership labels from Figure 2.4, can be addressed with the same idea.

$$\boxed{\ell \Vdash e}$$

$$\frac{\ell_p \Vdash e}{\ell_p \Vdash \lambda x.e} \qquad \frac{\ell_p \Vdash e_1 \qquad \ell_p \Vdash e_2}{\ell_p \Vdash e_1\ e_2} \qquad \frac{\ell_p = \ell_q \qquad \ell_r \Vdash e}{\ell_p \Vdash \mathbf{B}\#\langle \ell_q, \ell_r \rangle\ \{e\}} \qquad \frac{\ell_p = \ell_q \qquad \ell_r \Vdash e}{\ell_p \Vdash \mathsf{proxy}\big(\langle \ell_q, \ell_r \rangle,\ \lambda x.e\big)}$$

Figure 3.3: Well-formedness of ownership labels. Adapted from Dimoulas et al. [2012].

When labeling the owners of subexpressions in the ownership annotation language, the labels on nested boundaries or proxies must match each other. This corresponds to the *single-owner policy* from Dimoulas et al. [2011, 2012]. As a concrete example, it ought to be the case that $\ell_p = \ell_q$ in the following expression in order to sensibly assign an owner to the region containing the application of the function $\lambda x.\, x + 2$.

$$\mathbf{B}\#\langle \ell_n, \ell_p \rangle\ \{\ (\lambda x.\, x + 2)\ (\mathbf{B}\#\langle \ell_q, \ell_r \rangle\ \{\ 5\ \})\ \}$$

Dimoulas et al. [2012] characterize the single-owner policy using the well-formedness judgment $\ell \Vdash e$. Figure 3.3 displays some of the inference rules adapted from their work. In the judgement, the label to the left of $\Vdash$ indicates the owner of the current expression. Therefore, in the boundary case, the premise $\ell_p = \ell_q$ requires that the outer label on the boundary matches the label of the current owner. In the other premise of the boundary case, the owner changes to the inner label on the boundary, signifying that the owner of e is ℓ_r. Similarly, the premises of the proxy case match the label to the left of $\Vdash$ with the labels on the proxy.

The compliance of the single-owner policy or, more formally, the preservation of the well-formedness for all expressions, can be proved in terms of transition system homomorphisms, too. Let WExpr_ℓ be the set of expressions that are well formed under label ℓ, i.e.,

$$\mathsf{WExpr}_\ell :\equiv \{e_w \mid \ell \Vdash e_w\}.$$

If an initial expression e_w^* is a member of WExpr_{ℓ^*} for some specific label ℓ^*, I shall prove that the monitor calculus preserves this membership relationship when taking reduction steps with the homomorphism h_{own} from a subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{owner}; \mathscr{T}o]$ containing e_w^* to $\mathcal{T}_{own}$, the transition system at the bottom of Figure 3.2 (b).

The transition system $\mathcal{T}_{own}$ in particular satisfies the single-owner policy **by construction**: it is defined in a manner similar to the induced transition system of $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{ctc}; \mathscr{T}c]$ except that the

states only include the equivalence classes of expressions that satisfy the well-formedness judgement. As Figure 3.2 (b) depicts, for each state $[e_w]_{\mathrm{P}}$ at the top, the corresponding state at the bottom intersects the equivalence class with WExpr_{ℓ^*}. The transitions are adjusted accordingly and, moreover, states whose intersection is empty are removed. Formally, the transition system at the bottom is:

$$\mathcal{T}_{own} :\equiv \left(\left\{((), [e_w]_{\mathrm{P}} \cap \mathsf{WExpr}_{\ell^*}) \mid e_w \in \mathsf{WExpr}_{\ell^*}\right\}, \longrightarrow''_{\mathrm{m}}\right)$$

where for any e_1, e_2, the transition relation $\longrightarrow''_{\mathrm{m}}$ relates two states if and only if there exists e'_1, e'_2 such that $e'_1 \in [e_1]_{\mathrm{P}} \cap \mathsf{WExpr}_{\ell^*}$, $e'_2 \in [e_2]_{\mathrm{P}} \cap \mathsf{WExpr}_{\ell^*}$, and $(), e'_1 \longrightarrow_{\mathrm{m}} (), e'_2$. *By definition*, all reduction sequences captured by this transition system preserve the well-formedness judgement since the states are subsets of WExpr_{ℓ^*}.

That the instantiation $\lambda_{\mathrm{m}}[\mathscr{A}ctc;\mathscr{T}c]$ complies with the single-owner policy can be established by proving that for any $e^*_w \in \mathsf{WExpr}_{\ell^*}$, h_{own} defined by $((), [e]_{\mathrm{P}}) \longmapsto ((), [e]_{\mathrm{P}} \cap \mathsf{WExpr}_{\ell^*})$ is a well-defined function from the minimum subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}owner; \mathscr{T}o]$ containing $((), [e^*_w]_{\mathrm{P}})$ to $\mathcal{T}_{own}$, and that h_{own} is a homomorphism. To understand why, recall from Definition 3.3 that a subsystem is closed under transitions. Therefore, the minimum subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}owner; \mathscr{T}o]$ that includes the state $((), [e^*_w]_{\mathrm{P}})$ precisely contains the reduction sequence starting from $((), e^*_w)$. As a result, when h_{own} is a homomorphism from this subsystem to $\mathcal{T}_{own}$, it follows that all subsequent expressions in the reduction sequence are members of WExpr_{ℓ^*}. Hence, the well-formedness judgement is preserved.

3.3 Reusing Metatheories for Composite Languages

Proving properties of contract systems using transition system homomorphisms has the benefit that the proofs can be easily reused as the composition of homomorphisms is again another homomorphism. To demonstrate this concept, let us revisit the annotation language $\mathscr{A}octc$ from Figure 2.5 in page 30, Section 2.3.

Since $\mathscr{A}octc$ is morally a combination of $\mathscr{A}ctc$ and $\mathscr{A}owner$, both the non-masking property and

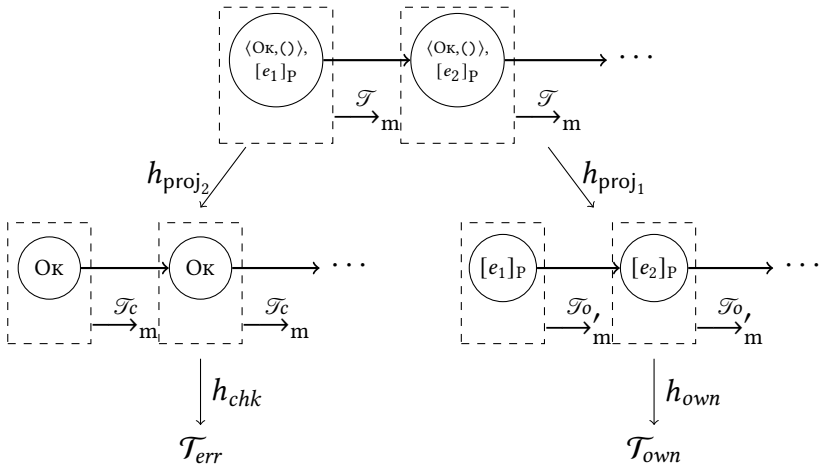

Figure 3.4: The induced transition systems of the composite instantiation.

the compliance of the single-owner policy discussed in Section 3.2 should hold for $\mathscr{A}octc$ as well. In other words, the contract checking status of $\mathscr{A}octc$ would not be reset Ok if it ever becomes Err, and the ownership labels annotated on boundaries always adhere to the single-owner policy.

Nonetheless, repeating the proofs from $\mathscr{A}ctc$ and $\mathscr{A}owner$ again for $\mathscr{A}octc$ is a waste of effort. As I argue in Section 2.3, the transition steps of $\mathscr{A}octc$ can be formed by appropriately setting the π_A, π_S, and *put* functions in Figure 2.6 in page 32. As a result, any reduction step of $\lambda_m[\mathscr{A}octc;\mathscr{T}oc]$ gives rise to a reduction step of $\lambda_m[\mathscr{A}ctc;\mathscr{T}c]$ through the π_A and π_S functions and, intuitively, it should be possible to lift properties about the reduction sequences of $\lambda_m[\mathscr{A}ctc;\mathscr{T}c]$ to the corresponding sequences of $\lambda_m[\mathscr{A}octc;\mathscr{T}oc]$.

Fortunately, this is indeed the case if we take a close look at the two induced transition systems, $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}octc;\mathscr{T}oc]$ and $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc;\mathscr{T}c]$. With functions like (π_A, π_S), the transition system $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}octc;\mathscr{T}oc]$ comes with two weak homomorphisms, h_{proj_2} and h_{proj_1}, to the transition systems $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}owner;\mathscr{T}o]$ and $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc;\mathscr{T}c]$, respectively.

Figure 3.4 depicts the transition systems and homomorphisms. The one on the left projects the contract part of the annotations, and the other one on the right projects the ownership labels part. Recall from Figure 3.2 that the non-masking property is proved using the homomorphism h_{chk} from $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}ctc;\mathscr{T}c]$ to $\mathcal{T}_{err}$. By composing h_{chk} and h_{proj_2}, we obtain a weak homomorphism $h_{chk} \circ h_{\mathsf{proj}_2}$ from $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}octc;\mathscr{T}oc]$ to $\mathcal{T}_{err}$. Thus, the non-masking property holds for the instantiation

$\lambda_{\mathsf{m}}[\mathscr{A}octc;\mathscr{T}oc]$. Similarly, through the composition homomorphism $h_{own} \circ h_{\mathsf{proj}_1}$, the single-owner property holds for $\lambda_{\mathsf{m}}[\mathscr{A}octc;\mathscr{T}oc]$.

In summary, viewing contract systems as transition systems introduces a unified approach to proving the contract systems' properties. Building on this formulation, the transition systems in Figure 3.4 illustrate how properties proven for the individual instantiation can be transferred to the composite one by composing homorphisms. More generally, when working with composite contract systems, the problem of bringing properties from the original contract system to the composite one is reduced to studying the relationship between the respective induced transition systems.

3.4 A Class of Homomorphisms for the Monitor Calculus

As Sections 3.2 and 3.3 show, the transition system representation of contracts helps one build reusable metatheories. Unfortunately, when proving properties in this approach, constructing the appropriate transition systems that capture the desired properties and establishing the respective homomorphisms are two challenging and sometimes tedious steps. To overcome this difficulty, I systematically construct a class of *interpretation-satisfying* transition systems that capture properties one wants to prove by construction together with the corresponding the homomorphisms from the induced transition system. In fact, the interpretation-satisfying transition systems and the homomorphisms to them are generalization of h_{chk} and h_{own} from Figure 3.2. Each transition system and each homomorphism at the bottom of Figure 3.2 represents a useful kind properties in the metatheory of contract systems.

The construction is adapted from a common practice that applies logic to study the behavior of transition systems [Sifakis 1983; van Benthem and Bergstra 1994]. Typically, one designs a logic and takes transition systems as its semantics using a *satisfaction relation* that determines whether a formula is satisfied from a given state. Then, one examines the classes of transition systems that satisfy different formulas. This approach reduces the problem of understanding

$$
\begin{array}{lll}
\mathcal{I}_{\ell_p} \vDash \lambda x.e & \text{iff} & \mathcal{I}_{\ell_p} \vDash e \\
\mathcal{I}_{\ell_p} \vDash e_1\, e_2 & \text{iff} & \mathcal{I}_{\ell_p} \vDash e_1 \text{ and } \mathcal{I}_{\ell_p} \vDash e_2 \\
\mathcal{I}_{\ell_p} \vDash \mathbf{B}\#\langle \ell_q, \ell_r \rangle\, \{e\} & \text{iff} & \ell_p = \ell_q \text{ and } \mathcal{I}_{\ell_r} \vDash e \\
\mathcal{I}_{\ell_p} \vDash \mathsf{proxy}\big(\langle \ell_q, \ell_r \rangle,\ \lambda x.e\big) & \text{iff} & \ell_p = \ell_q \text{ and } \mathcal{I}_{\ell_r} \vDash e \\
\vdots & &
\end{array}
$$

Figure 3.5: A family of interpretations for proving the single-owner policy.

the behavior of transition systems into the design of logic systems and has applications to the theory of concurrent processes [Hennessy and Milner 1985; van Benthem et al. 1994], model checking [Clarke et al. 1986], runtime verification and temporal logic [Pnueli 1977; Lamport 1994].

For the monitor calculus, I adapt the preceding approach and define the satisfaction relation over expressions. It is written as $\mathcal{I} \vDash e$ where $\mathcal{I}$ refers to an *interpretation* of the annotation languages, and e is any expression. Let me assume that there is an ambient logic—the metalanguage—that is referenced when proving properties about the monitor calculus. The relation $\mathcal{I} \vDash e$ asserts that e satisfies the interpretation $\mathcal{I}$. Here, an interpretation $\mathcal{I}$ includes two functions, $\mathbb{B}[\![\, A, e \,]\!]$ and $\mathbb{P}[\![\, A, e^m \,]\!]$, that map any boundary expression $\mathbf{B}\#A\,\{e\}$ and any proxy value $\mathsf{proxy}(A,\ e^m)$ to propositions of the metalanguage. The satisfaction relation further lifts these two functions to all expression forms. Therefore, it helps one systematically select specific subsets of expressions through specially defined interpretations. Generally speaking, an interpretation $\mathcal{I}$ can describe the desired properties with appropriate choices of the $\mathbb{B}$ and $\mathbb{P}$ functions. Then, the set expressions that satisfy the property captured by $\mathcal{I}$ is simply $\{e \mid e \in \mathsf{Expr} \wedge \mathcal{I} \vDash e\}$.

An implication of this observation is that no additional judgements are needed to characterize specific subsets of the expressions. Rather, it is sufficient to supply an appropriate annotation interpretation to the satisfaction relation. For now, I illustrate this idea by reformulating the well-formedness judgment, $\ell_p \Vdash e$ that captures the single-owner policy in Figure 3.3 in terms of the satisfaction relation with the interpretation given in Figure 3.5. Although the presentation does not fully match the technical definition of interpretations, it demonstrates the overall idea.

The family of interpretations $\{\mathcal{I}_{\ell_p}\}_{\ell_p \in \mathsf{Label}}$ given in Figure 3.5 is defined structurally on the expressions e. For example, $\mathcal{I}_{\ell_p} \vDash e_1\, e_2$ holds if and only if both $\mathcal{I}_{\ell_p} \vDash e_1$ and $\mathcal{I}_{\ell_p} \vDash e_2$ hold.

The cases for boundaries ($\mathbf{B}\#\langle \ell_q, \ell_r \rangle\ \{e\}$) and proxies ($\mathsf{proxy}\big(\langle \ell_q, \ell_r \rangle,\ \lambda x.e\big)$) are defined similarly except that $\mathcal{I}_{\ell_p}$ supplies the additional condition $\ell_p = \ell_q$ in the shaded area, thereby enforcing the single-owner policy. With this interpretation, $\mathcal{I}_{\ell_p} \vDash e$ holds if and only if $\ell_p \Vdash e$ holds. Thus, the well-formedness judgement $\ell_p \Vdash e$ is subsumed by instantiating the satisfaction relation with the family of interpretations $\{\mathcal{I}_{\ell_p}\}_{\ell_p \in \mathsf{Label}}$.

[*Note:* Readers familiar with logic may find the satisfaction relation of the monitor calculus different from the common definition in several places. Typically, a transition system, $\mathcal{M}$, instead of $\mathcal{I}$ is placed on the left-hand side of $\vDash$ and $\mathcal{M}$ is considered to be a model of a formula F if it satisfies F at every state. For the monitor calculus, $\mathcal{I}$ is the interpretation of boundary and proxy terms. The transition systems that satisfy an interpretation are implicit in the satisfaction relation. Moreover, the right-hand side of $\vDash$ is usually a logic formula that can include modal operators to express the behavior of state transitions. In the monitor calculus, the right-hand side of $\vDash$ are expressions. As the counterpart of proof systems for logic, the $\longrightarrow_{\mathrm{m}}$ relation and the $\longrightarrow_{\mathrm{p}}$ relation both act as the deduction system of expressions. — *end note*]

Recall that the overall goal of introducing the annotation interpretations and the satisfaction relation is to systematically construct a class of transition systems that have homomorphisms from the induced transition system in order to establish properties of the instantiated monitor calculus. Taking the preservation of the single-owner policy in Figure 3.2 (b) as an example, the family of interpretations $\{\mathcal{I}_{\ell_p}\}_{\ell_p \in \mathsf{Label}}$ has captured the well-formedness judgement $\ell_p \Vdash e$. That is, $\ell_p \Vdash e$ holds if and only if $\mathcal{I}_{\ell_p} \vDash e$ holds. Hence, replacing set WExpr_{ℓ^*} by the definition $\{e \mid e \in \mathsf{Expr} \wedge \mathcal{I}_{\ell^*} \vDash e\}$ does not affect the construction of the transition system $\mathcal{T}_{own}$ and the homomorphism h_{own}. However, since $\{\mathcal{I}_{\ell_p}\}_{\ell_p \in \mathsf{Label}}$ is just one of the many possible interpretations, this immediately generalizes the proof in the second part of the Section 3.2 to all other properties that can be described through the combinations of annotation interpretations and the satisfaction relation!

To see how the generalization works, let me rephrase the proof in Section 3.2. In this section, however, I shall replace the special-purpose set of expressions WExpr_{ℓ^*} with the more general

definition. Specifically, Let $\mathsf{IExpr}(\mathcal{I})$ be the set of expressions that satisfy $\mathcal{I}$, i.e.,

$$\mathsf{IExpr}(\mathcal{I}) :\equiv \{e \mid \mathcal{I} \vDash e\}.$$

If an initial expression e^* belongs to the set $\mathsf{IExpr}(\mathcal{I})$, I shall prove that the monitor calculus preserves this membership relationship when taking reduction steps by constructing a homomorphism h_{sat} from the transition system $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ to one that satisfies $\mathcal{I}$ by construction.

Specifically, the transition system $\mathcal{T}_{sat}$ that satisfies $\mathcal{I}$ **by construction** is defined by intersecting the states of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ with $\mathsf{IExpr}(\mathcal{I})$. Formally, the definition of $\mathcal{T}_{sat}$ is:

$$\mathcal{T}_{sat} :\equiv \left(\{(s, [e]_{\mathrm{P}} \cap \mathsf{IExpr}(\mathcal{I})) \mid s \in \mathsf{State} \wedge e \in \mathsf{IExpr}(\mathcal{I})\}, \longrightarrow''_{\mathrm{m}}\right)$$

where for any e_1, e_2, the transition relation $\longrightarrow''_{\mathrm{m}}$ relates two states if and only if there exists e'_1, e'_2 such that $e'_1 \in [e_1]_{\mathrm{P}} \cap \mathsf{IExpr}(\mathcal{I})$, $e'_2 \in [e_2]_{\mathrm{P}} \cap \mathsf{IExpr}(\mathcal{I})$, and $s, e'_1 \longrightarrow_{\mathrm{m}} s', e'_2$. *By definition*, all reduction sequences captured by this transition system preserve the satisfaction of $\mathcal{I}$ as the states are subsets of $\mathsf{IExpr}(\mathcal{I})$.

To finish the proof and bring the property captured by $\mathcal{I}$ from $\mathcal{T}_{sat}$ back to the instantiation $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$, I only need to prove that for any $e^* \in \mathsf{IExpr}(\mathcal{I})$, the map h_{sat} defined by $(s, [e]_{\mathrm{P}}) \longmapsto (s, [e]_{\mathrm{P}} \cap \mathsf{IExpr}(\mathcal{I}))$ is a well-defined function from the smallest subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathscr{T}]$ containing $(s, [e^*]_{\mathrm{P}})$ to $\mathcal{T}_{sat}$, and that h_{sat} is a homomorphism.

Summarizing the development so far, proving a property in the transition-system-based theory includes only two steps. First, define an interpretation $\mathcal{I}$ to capture the desired property through appropriately chosen $\mathbb{B}$ and $\mathbb{P}$ functions. Second, with the interpretation $\mathcal{I}$, the construction of the interpretation-satisfying transition system $\mathcal{T}_{sat}$ and the candidate homomorphism h_{sat} are defined above; verify that h_{sat} is indeed a homomorphism. Overall, the introduction of interpretations provides a systematic way for constructing transition-system-based proofs.

In Chapter 5, I refine the procedure developed in this section further by introducing two conditions on interpretations that can help verify the well-definedness of h_{sat} and that it is a homomorphism. The framework developed in Chapter 5 also incorporates the preorder structure that h_{chk} utilizes, reducing the boilerplate needed by the proofs.

Part II

A Transition-System View of Contract Systems

Chapter 4

The Monitor Calculus, Formally

In this chapter, I present the monitor calculus, its syntax, operational semantics, and its parameters. Section 4.1 opens the chapter by giving the syntax and the program-related reduction relation of the calculus. Section 4.2 formally defines annotation languages and the operational semantics of the monitor calculus. Last, Section 4.3 briefly discusses how annotation languages can be made extensible through the projections of annotation languages.

Based on the framework in this chapter, I have mechanized the monitor calculus as an Agda library. My Agda library implementation allows one to define annotation languages in direct style and build reusable metatheory for different contract systems. All annotation languages and theorems presented in Chapters 6 to 8 are mechanized as reusable components using this library.

4.1 Syntax and Operational Semantics

The *monitor calculus*, or λ_{m} for short, is a calculus that supports monitoring of higher-order values. It is a parameterized extension of the call-by-value simply typed lambda calculus that includes unit, natural numbers, pairs, disjoint sums, recursive types and immutable reference cells. The calculus is parameterized by annotation languages, which specifies the behavior of the monitoring system. It is covered in the next section.

Figure 4.1 defines the syntax of λ_{m} and Figure 4.2 gives its type system. I shall explain the lan-

$$
\begin{array}{rcl}
\tau & ::= & t \mid \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 \mu t.\tau \\
A & \in & \mathsf{Ann} \\
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 \mathsf{assert}\,e \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 \mid e;\,e_1 \\
e^m & ::= & \mathsf{box}(e) \mid \lambda x.e \\
\\
n, m & ::= & \mathsf{zero} \mid \mathsf{suc}(n) \\
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) \\
E & ::= & \mathsf{suc}(E) \mid \mathsf{fold}_{\mathsf{nat}}(E,\ e_z,\ x\,y.\,e_s) \mid \mathsf{assert}\,E \mid \langle E, e\rangle \mid \langle v, E\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 & E\,e \mid v\,E \mid \mathsf{unroll}(E) \mid \mathsf{roll}_\tau(E) \mid E;\,e_1 \\
 & \mid & \mathbf{B}\#A\,\{E\} \\
r & ::= & \mathbf{B}\#A\,\{()\} \mid \mathbf{B}\#A\,\{n\} \mid \mathbf{B}\#A\,\{\,\langle v_1,\ v_2\rangle\,\} \mid \mathbf{B}\#A\,\{\,\mathsf{inl}(v)\,\} \mid \mathbf{B}\#A\,\{\,\mathsf{inr}(v)\,\} \\
 & \mid & \mathbf{B}\#A\,\{\,\mathsf{roll}_\tau(v)\,\} \mid \mathbf{B}\#A\,\{\,\mathsf{box}(v)\,\} \mid \mathbf{B}\#A\,\{\,\lambda x.e\,\} \\
 & \mid & \mathbf{B}\#A\,\{\,\mathsf{proxy}(A',\ \mathsf{box}(e))\,\} \mid \mathbf{B}\#A\,\{\,\mathsf{proxy}(A',\ \lambda x.e)\,\} \\
 & \mid & \mathsf{unbox}(\,\mathsf{proxy}(A,\ \mathsf{box}(e))\,) \mid \mathsf{proxy}(A,\ \lambda x.e)\ v
\end{array}
$$

Figure 4.1: The syntax of the monitor calculus

guage constructs together with their typing rules. Before that, let me go over what each metavariable denotes in the figure. The metavariable τ ranges over types; A ranges over annotations, Ann, which is part of the annotation language. Then, the metavariable e ranges over expressions. Additionally, e^m syntactically distinguishes terms that can be monitored.

The next four metavariables define the grammar of terms that are relevant to the evaluation of the programs. The metavariables n and m syntactically recognize values that are numbers; v ranges over all values; E denotes evaluation contexts. The metavariable r denotes redexes related to the evaluation of boundaries and proxies. There are two types of evaluation in λ_{m}, one concerns boundaries and proxies and the other concerns non-boundary expressions. The $\longrightarrow_{\mathrm{p}}$ relation defines the evaluation of non-boundary expressions whereas r captures all other

$$\frac{\vdash A : \mathsf{Ann}_\tau \qquad \vdash e : \tau}{\Gamma \vdash \mathbf{B}\#A\,\{e\} : \tau} \qquad \frac{\vdash A : \mathsf{Ann}_\tau \qquad \vdash e^m : \tau}{\Gamma \vdash \mathsf{proxy}(A,\ e^m) : \tau} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{suc}(e) : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{assert}\, e : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma,\, x : \mathsf{nat},\, y : \tau \vdash e_s : \tau}{\Gamma \vdash \mathsf{fold}_{\mathsf{nat}}(e,\ e_z,\ x\,y.\,e_s) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(e) : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathsf{inl}(e_1) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{inr}(e_2) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma,\, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma,\, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\, e\, \mathsf{of}\, \{x.e_1 \mid y.e_2\} : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{box}(e) : \mathsf{Box}\,\tau}$$

$$\frac{\Gamma \vdash e : \mathsf{Box}\,\tau}{\Gamma \vdash \mathsf{unbox}(e) : \tau} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma,\, x : \tau_a \vdash e : \tau_r}{\Gamma \vdash \lambda x.e : \tau_a \to \tau_r} \qquad \frac{\Gamma \vdash e : \tau_a \to \tau_r \qquad \Gamma \vdash e_a : \tau_a}{\Gamma \vdash e\, e_a : \tau_r}$$

$$\frac{\Gamma \vdash e : \tau[\mu t.\tau\,/\,t]}{\Gamma \vdash \mathsf{roll}_\tau(e) : \mu t.\tau} \qquad \frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \mathsf{unroll}(e) : \tau[\mu t.\tau\,/\,t]} \qquad \frac{\Gamma,\, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, x.e : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e;\, e_1 : \tau_1}$$

Figure 4.2: The typing rules of the monitor calculus

reducible expressions. For more on this, see the Progress lemma and Section 4.2.

Let me now explain the grammar of λ_{m} and its typing rules. The two most important constructs of λ_{m} are boundaries and proxies. To begin with, boundary terms are written as $\mathbf{B}\#A\,\{e\}$. Since boundaries separate expressions into different components, the inner expression (e) has to be closed ($\vdash e : \tau$). A boundary term with inner expression of type τ is annotated with an annotation of type Ann_τ. The types of annotations (Ann_τ) are part of the annotation language and are indexed by the types of the expressions they annotate. Proxy terms are written as $\mathsf{proxy}(A,\ e^m)$. Similar to boundaries, the monitored term (e^m) must be closed ($\vdash e^m : \tau$). Moreover, they must be syntactic immutable reference cells, $\mathsf{box}(e)$, or syntactic functions, $\lambda x.e$.

Next, natural numbers are created using either zero or $\mathsf{suc}(e)$. The fold of natural numbers is written as $\mathsf{fold}_{\mathsf{nat}}(e,\ e_z,\ x\,y.\,e_s)$. The expression e_z handles the base case of fold and e_s handles the recursive case. I include the predecessor in the recursive case for convenience, so e_s additionally has $x : \mathsf{nat}$ and $y : \tau$ in the environment which binds the predecessor and the result of recursion, respectively. The expression $\mathsf{assert}\, e$ asserts that e evaluates to a non-zero number.

Recursive types in λ_{m} are created by $\mathsf{roll}_\tau(e)$ and eliminated by $\mathsf{unroll}(e)$. For a type τ that contains one free type variable t, the constructor roll takes an expression of type $\tau[\mu t.\tau \,/\, t]$ and wraps it into an expression of type $\mu t.\tau$. The eliminator unroll does the opposite. Recursive types in the monitor calculus are strict; $\mathsf{roll}_\tau(e)$ fully evaluates its argument.

The monitor calculus also includes immutable reference cells (or references for short) for the purpose of monitoring arbitrary values. That is, while only functions and references can be monitored, any kinds of values can be stored in references. Subsequently, the access of thosed that are stored in references are tracked by the monitor calculus. A reference storing values of type τ has type $\mathsf{Box}\,\tau$. They are created using $\mathsf{box}(e)$ and their contents can be accessed by $\mathsf{unbox}(e)$. In the calculus, immutable references are essentially structs with a single field.

Immutable reference cells can be used to encode lazy contracts [Findler et al. 2008]. For example, when modeling contracts in the monitor calculus, a contracted pair $\langle v_1, v_2\rangle$ will immediately check both of its components (cf. the [R-Cross-Cons] rule in Figure 4.4). On the contrary, for a contracted pair that stores references, i.e. $\langle \mathsf{box}(v_1), \mathsf{box}(v_2)\rangle$, the contract will be checked only when the respective component is accessed using $\mathsf{unbox}(-)$.

Finally, the remaining constructs are standard. The expression $()$ has type unit and is the unit value. Pairs and the projection of pairs are $\langle e_1, e_2\rangle$ and $\pi_i(e)$, respectively. Disjoint sums are created by either $\mathsf{inl}(e)$ or $\mathsf{inr}(e)$ and eliminated through the case expression, $\mathsf{case}\, e\, \mathsf{of}\, \{x.e_1 \mid y.e_2\}$. Variables, λ-functions and function applications have their usual syntax and typing rules. λ_{m} also include the fixed-point combinator, $\mathsf{fix}\, x.e$, to support general recursion. Sequencing, $e;\, e_1$, is included for convenience.

I prove the Substitution lemma following the framework developed by McBride [2005]; Allais et al. [2017, 2018]. A parallel Renaming lemma is presented, then an extension operation on the context (Proposition 4.2), and finally the parallel Substitution lemma of the expressions.

Lemma 4.1 (Renaming)**.** *Let* $\Gamma :\equiv x_1 : \tau_1, \ldots, x_n : \tau_n$ *and* $\Gamma' :\equiv y_1 : \tau'_1, \ldots, y_m : \tau'_m$ *be given. If* $\Gamma \vdash e : \tau$ *and* $\Gamma' \vdash y_{a_i} : \tau_i$ *for some* $1 \le a_i \le m$ *for* $i = 1 \ldots n$ *then* $\Gamma' \vdash e\left[y_{a_1} \ldots y_{a_n} \,/\, x_1 \ldots x_n\right] : \tau$.

Proposition 4.2. *Let* $\Gamma' \vdash e_i : \tau_i$ *for* $i = 1 \ldots n$ *be given. Then, for any* τ_0 *and any fresh variable*

$$
\begin{array}{rcl}
\mathsf{fold}_{\mathsf{nat}}(\mathsf{zero},\ e_z,\ x\,y.\,e_s) & \longrightarrow_{\mathrm{p}} & e_z \\
\mathsf{fold}_{\mathsf{nat}}(\mathsf{suc}(n),\ e_z,\ x\,y.\,e_s) & \longrightarrow_{\mathrm{p}} & e_s[n\,/\,x][\mathsf{fold}_{\mathsf{nat}}(n,\ e_z,\ x\,y.\,e_s)\,/\,y] \\
\mathsf{assert}\ \mathsf{suc}(n) & \longrightarrow_{\mathrm{p}} & () \\
\pi_1(\langle v_1, v_2\rangle) & \longrightarrow_{\mathrm{p}} & v_1 \\
\pi_2(\langle v_1, v_2\rangle) & \longrightarrow_{\mathrm{p}} & v_2 \\
\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \{x.e_1 \mid y.e_2\} & \longrightarrow_{\mathrm{p}} & e_1[v\,/\,x] \\
\mathsf{case}\ \mathsf{inr}(v)\ \mathsf{of}\ \{x.e_1 \mid y.e_2\} & \longrightarrow_{\mathrm{p}} & e_2[v\,/\,y] \\
\mathsf{unbox}(\mathsf{box}(v)) & \longrightarrow_{\mathrm{p}} & v \\
(\lambda x.e)\,v & \longrightarrow_{\mathrm{p}} & e[v\,/\,x] \\
\mathsf{unroll}(\mathsf{roll}_\tau(v)) & \longrightarrow_{\mathrm{p}} & v \\
\mathsf{fix}\,x.e & \longrightarrow_{\mathrm{p}} & e[\mathsf{fix}\,x.e\,/\,x] \\
v;\ e & \longrightarrow_{\mathrm{p}} & e
\end{array}
$$

Figure 4.3: The program-related reduction relation

$x_0 : \tau_0$, there is a sequence $\Gamma'' \vdash e_i : \tau_i$ for $i = 0 \ldots n$ where $\Gamma'' :\equiv \Gamma', x_0 : \tau_0$ and $e_0 :\equiv x_0$.

Proposition 4.2 simply extends the context of the given sequence of typing derivations with a new variable. It is used for weakening the contexts in the proof of the Substitution lemma.

Lemma 4.3 (Substitution)**.** *Let $\Gamma :\equiv x_1 : \tau_1, \ldots, x_n : \tau_n$ and some Γ' be given. If $\Gamma \vdash e : \tau$ and $\Gamma' \vdash e_i : \tau_i$ for $i = 1 \ldots n$ then $\Gamma' \vdash e[e_1 \ldots e_n\,/\,x_1 \ldots x_n] : \tau$.*

Figure 4.3 defines *the $\longrightarrow_p$ relation*, or the *program-related reduction relation* that covers the evaluation of non-boundary expressions. Since they do not involve annotations, the reduction rules are the same across all instantiations of λ_{m}. Note that the $\longrightarrow_{\mathrm{p}}$ relation is slightly different from the introduction in Section 2.2 in that it does not evaluate under evaluation contexts. The reduction relation that accounts for whole expressions for the entire calculus is written as $\mathscr{T} \vdash s, e \longrightarrow s', e'$. I defer their explanation to the next section as they include the global states (s) and the transition steps ($\mathscr{T}$) from the annotation language.

The reduction rules in the $\longrightarrow_{\mathrm{p}}$ relation are standard, but a few of them are worth mentioning. First, to reduce $\mathsf{fold}_{\mathsf{nat}}(\mathsf{suc}(n),\ e_z,\ x\,y.\,e_s)$, the $\longrightarrow_{\mathrm{p}}$ relation evaluates to e_s and replaces y with the result of recursion, or $\mathsf{fold}_{\mathsf{nat}}(n,\ e_z,\ x\,y.\,e_s)$. For conveniences, it also replaces x by the predecessor, n. Next, the assert v expression reduces if and only if v is $\mathsf{suc}(n)$. When v is zero, the

evaluation gets stuck. Last, the expression $\mathsf{unbox}(v')$ simply extracts the content of the reference cell when v' is $\mathsf{box}(v)$. Because references are immutable, I directly treat $\mathsf{box}(v)$ as values and use substitution-based semantics rather than introducing a separate store.

The $\longrightarrow_{\mathrm{p}}$ relation admits the typical properties. It preserves types; it is deterministic; values do not reduce under $\longrightarrow_{\mathrm{p}}$. All of these are to be expected, as boundaries and proxies have yet to come into play. Lemma 4.4 and Proposition 4.5 give their formal statements.

Lemma 4.4 (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow_p e'$ *then* $\vdash e' : \tau$*. Extending to evaluation contexts, if* $\vdash E[e_r] : \tau$*,* $\vdash e_r : \tau_r$ *and* $e_r \longrightarrow_p e'$ *then* $\vdash E[e'] : \tau$*.*

Proposition 4.5. *The* $\longrightarrow_p$ *relation has the following properties.*

1. *For all values* v*,* $v \not\longrightarrow_p$*.*
2. *If* $e \longrightarrow_p e_1$ *and* $e \longrightarrow_p e_2$ *then* $e_1 = e_2$*.*
3. *If* $e = E_1[e_1] = E_2[e_2]$ *with* $e_1 \longrightarrow_p e_1'$ *and* $e_2 \longrightarrow_p e_2'$ *then* $E_1\left[e_1'\right] = E_2\left[e_2'\right]$*.*

In addition to the Preservation lemma, the other type safety property is the Progress lemma. Intuitively, an expression e either is a value, getting stuck due to assert zero, or can make progress. However, only non-boundary expressions can always make progress. When a value needs to flow out of an enclosing boundary or an elimination operation is applied to a proxy, the monitor calculus has to consult its parameter—the annotation language—so the Progress lemma only identifies the redex r in the evaluation context.

Lemma 4.6 (Progress)**.** *If* $\vdash e : \tau$ *then one of the following is true.*

- e *is a value, i.e.* e *adheres to the grammar of the non-terminal* v*.*
- $e = E[e_r]$ *and* $e_r \longrightarrow_p e'$*.*
- $e = E[\mathsf{assert}\ \mathsf{zero}]$*.*
- $e = E[r]$*.*

4.2 The Language of Annotations

Annotation languages, what the monitor calculus is parameterized over, specify the set of annotations on boundaries and proxies, the global states of the monitoring systems, and determines the computation rules of the annotations and the global states. However, I have been inprecise about what the computation rules are in annotation languages until this point. To clear up the matter, I introduce *rule templates* to better characterize computation rules. Then, I specify a set of rule templates for the monitor calculus that each annotation language follows when defining their specific version of the computation rules.

4.2.1 Rule Templates

Informally speaking, a rule template is a schema where the metavariables ranging over the annotations and the states are designated as placeholders for more concrete definitions. Each of the rule template in the monitor calculus describes how a value flows out of a boundary or how an elimination operation on a proxy creates new boundaries. When the annotation languages define computation rules following the rule templates, these placeholder metavariables are replaced by more specific (meta-level) expressions to describe the actual computation.

To give an intuitive idea of what rule templates look like, recall that Section 2.2 presents an annotation languages that describes a contract system and an annotation language that tracks the ownership information. In Figure 2.4, both annotation languages gives their own version of the [R-Proxy-β] rule that defines how to apply proxy functions. The two versions are similar except that the $\mathcal{T}ctc$ version decomposes the contract $\kappa_a \to\!\!/_{\mathrm{c}}\, \kappa_r$ into separate pieces whereas the $\mathcal{T}own$ version propagates and swaps the ownership labels. Here is a copy of the two rules.

$$\mathcal{T}ctc \quad \textsc{Ok}, \mathsf{proxy}(\kappa_a \to\!\!/_{\mathrm{c}}\, \kappa_r,\ \lambda x.e)\ v \quad \longrightarrow_{\mathrm{m}} \quad \textsc{Ok}, \mathbf{B}\#\kappa_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#\kappa_a\ \{\ v\ \})\ \}$$

$$\mathcal{T}own \quad (), \mathsf{proxy}\big(\langle \ell_n, \ell_p\rangle,\ \lambda x.e\big)\ v \quad \longrightarrow_{\mathrm{m}} \quad (), \mathbf{B}\#\langle \ell_n, \ell_p\rangle\ \big\{\ (\lambda x.e)\ \big(\mathbf{B}\#\langle \ell_p, \ell_n\rangle\ \{\ v\ \}\big)\ \big\}$$

Here is the corresponding rule template of [R-Proxy-β].

$$[\text{R-Proxy-}\beta] \quad s, \mathsf{proxy}(A,\ \lambda x.e)\ v \quad \longrightarrow_{\mathrm{m}} \quad s', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#A_a\ \{\ v\ \})\ \}$$

The rule template of [R-Proxy-β] has the metavariables A, A_r, A_a in place of contracts in $\mathscr{T}ctc$ and ownership labels in $\mathscr{T}own$. The template also has the metavariables s, s' in place of the states. For any specific [R-Proxy-β] rule that is defined along the rule template, these five metavariables are the only places that can be changed. For example, in $\mathscr{T}ctc$, the placeholder A is replaced by $\kappa_a \rightarrow\!\!\!/_{\mathrm{c}}\, \kappa_r$ and the placeholders A_a, A_r are replaced by the metavariables κ_a and κ_a, respectively. Similarly, in $\mathscr{T}own$, the placeholders A and A_r are replaced by $\langle \ell_n, \ell_p \rangle$ and A_a is replaced by $\langle \ell_p, \ell_n \rangle$.

The same correspondence between the computation rules and the rule templates applies to other types of boundaries and proxies as well. As another example, consider the [R-Cross-Nat] rule from $\mathscr{T}ctc$ that checks the annotated contract when a number crosses a boundary. The corresponding rule template replaces Ok and Err with the metavariables s and s'. The contract κ is also replaced with the metavariable A. Spelling everything out below, the rule in the first two lines are the specific computation rule from $\mathscr{T}ctc$ and the third line is the corresponding rule template in the monitor calculus.

$$\begin{array}{lll} \text{[R-Cross-Nat]} & \textsc{Ok}, \mathbf{B}\#\kappa\,\{\,n\,\} \longrightarrow_{\mathrm{m}} \textsc{Ok}, n & \text{if } n \text{ satisfies } \kappa \\ & \textsc{Ok}, \mathbf{B}\#\kappa\,\{\,n\,\} \longrightarrow_{\mathrm{m}} \textsc{Err}, n & \text{if } n \text{ does not satisfy } \kappa \\ & \text{where } \kappa \text{ is a predicate} & \end{array}$$

$$\text{[R-Cross-Nat]} \quad s, \mathbf{B}\#A\,\{\,n\,\} \longrightarrow_{\mathrm{m}} s', n$$

Of course, the [R-Cross-Nat] rule in the first two lines seemingly consist of *two* rules. How can it be an instance of a *single* rule template in the third line? The answer is that the [R-Cross-Nat] rule can be framed more compactly in one rule by capturing the contract, the number crossing the boundary and the states in a relation. Let *Checked* be a relation over annotations, numbers and states defined as $\{(n, \kappa, \textsc{Ok}, \textsc{Ok}) \mid n \text{ satisfies } \kappa\} \cup \{(n, \kappa, \textsc{Ok}, \textsc{Err}) \mid n \text{ fails } \kappa\}$, then the [R-Cross-Nat] rule in $\mathscr{T}ctc$ can be rewritten using *Checked* to match the rule template.

$$\text{[R-Cross-Nat]} \quad s, \mathbf{B}\#\kappa\,\{\,n\,\} \longrightarrow_{\mathrm{m}} s', n \quad \text{where} \quad (n, \kappa, s, s') \in \mathit{Checked}$$

Generally speaking, the replacement of placeholder metavariables by other meta-level expressions can be viewed as constraining the placeholder metavariables with an additional relation

and that the relation is also a premise of the computation rule. Both the [R-Cross-Nat] rule and the [R-Proxy-β] rule are exemplars of this pattern. The [R-Cross-Nat] rule in $\mathscr{T}ctc$ is captured by the *Checked* relation, and the [R-Proxy-β] rule can be similarly captured by the relation $\mathit{ArrowCtc} :\equiv \{(\kappa, \kappa_a, \kappa_r, \text{Ok}, \text{Ok}) \mid \kappa = \kappa_a \rightarrow\!\!\!\!/_{\mathsf{c}}\ \kappa_r\}$.

$$[\text{R-Proxy-}\beta] \quad s, \mathsf{proxy}(\kappa,\ \lambda x.e)\ v \quad \longrightarrow_{\mathrm{m}} \quad s', \mathbf{B}\#\kappa_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#\kappa_a\ \{\ v\ \})\ \}$$

$$\text{where } (\kappa, \kappa_a, \kappa_r, s, s') \in \mathit{ArrowCtc}$$

Expressing the above in terms of rule templates, the contracts should be replaced by placeholder metavariables and the *ArrowCtc* relation should be replaced by some general relation R.

$$[\text{R-Proxy-}\beta] \quad s, \mathsf{proxy}(A,\ \lambda x.e)\ v \quad \longrightarrow_{\mathrm{m}} \quad s', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#A_a\ \{\ v\ \})\ \}$$

$$\text{where } (A, A_a, A_r, s, s') \in R_{x,e,v}$$

Note that R must govern all placeholder metavariables, i.e. those that range over the annotations and the states. Optionally, R can refer to any other metavariables in the rule template as explicated in its index.

More importantly, for a fixed set of rule templates, using relations like R gives a uniform representation of specialized computation rules. In the monitor calculus, the annotation languages need only provide one such relation for each of the rule template when defining their own variant of the computation rules. I shall collectively call these relations in the annotation languages as *transition steps* and range over them with $\mathscr{T}$. I will also refer to the set of specialized computation rules as *the* $\longrightarrow_m$ *relation*, or the *monitor-related reduction relation*, and use the notation $\mathscr{T} \vdash s, e \longrightarrow_{\mathrm{m}} s', e'$ to denote reductions captured by the $\longrightarrow_{\mathrm{m}}$ relation.

4.2.2 Languages of Annotations

Definition 4.7 (Language of Annotations)**.** An *annotation language* is a pair $(\mathscr{A}, \mathscr{T})$ where $\mathscr{A}$ is the set of annotations and the set of states, (Ann, State), and $\mathscr{T}$ is the transition steps that specifies the relation of placeholder metavariables in the rule templates.

When instantiating the monitor calculus with an annotation language $(\mathscr{A}, \mathscr{T})$, I shall denote

the *instantiated monitor calculus* by $\lambda_{\text{m}}[\mathscr{A};\mathscr{T}]$.

Definition 4.8. For an annotation language $\mathscr{T}$, the reduction relation of $\lambda_{\text{m}}[\mathscr{A};\mathscr{T}]$ is denoted by $\longrightarrow$. It contains the $\longrightarrow_{\text{p}}$ relation and the $\longrightarrow_{\text{m}}$ relation. In the [R-Bdr] rule, the non-terminal r includes redexes related to boundaries and proxies. See Figure 4.1 for its productions.

$$\frac{e \longrightarrow_{\text{p}} e'}{\mathscr{T} \vdash s, E[e] \longrightarrow s, E[e']}\ [\text{R-Redex}] \qquad \frac{\mathscr{T} \vdash s, r \longrightarrow_{\text{m}} s', e'}{\mathscr{T} \vdash s, E[r] \longrightarrow s', E[e']}\ [\text{R-Bdr}]$$

Proposition 4.9. *If* $\vdash r : \tau$ *and* $\mathscr{T} \vdash s, r \longrightarrow_m s', e'$ *then* $\vdash e' : \tau$.

Lemma 4.10 (Preservation)**.** *If* $\vdash e : \tau$ *and* $\mathscr{T} \vdash s, e \longrightarrow s', e'$ *then* $\vdash e' : \tau$.

Connecting rule templates to computation rules through relations over placeholder metavariables allows one to establish meta-theoretic properties of the monitor calculus itself. For example, these relations provide a concrete formulation of annotation language compositions in Section 5.4. When discussing the applications of the monitor calculus, however, I shall avoid explicitly writing down the relations that are used in specializing the rule templates.

For the purpose of understanding the monitor calculus and its application to reusable metatheories of contracts, presenting the computation rules in the style of Figure 2.4 is sufficient. For the interested readers, the Agda implementation gives the precise definition of placeholder metavariables and rule templates in `Syntax.Template` and `Annotation.Language`.

The complete list of rule templates that the monitor calculus includes is given in Figure 4.4. All of the rule templates have the form $s, r \longrightarrow_{\text{m}} s', e$ where r is a redex related to boundaries or proxies. In the figure, the column on the left shows the name and the column on the right shows the corresponding rule template. Roughly speaking, there is one rule for each kind of values describing how it flows out of a boundary. Similarly, for each kind of monitorable value, there is one rule describing how an elimination operation applies to a proxy.

Let me give a high-level overview of the rule templates. For the [R-Cross-Unit] rule and the [R-Cross-Nat] rule, the value directly flows out of the enclosing boundary. The computation rules correspond to these two rule templates may optionally change the states, but no new boundaries are created. For the next four rule templates, the monitor calculus pushes the bound-

Name	$s, r \longrightarrow_{\mathrm{m}} s', e'$
[R-Cross-Unit]	$s, \mathbf{B}\#A\ \{\ ()\ \} \longrightarrow_{\mathrm{m}} s', ()$
[R-Cross-Nat]	$s, \mathbf{B}\#A\ \{\ n\ \} \longrightarrow_{\mathrm{m}} s', n$
[R-Cross-Cons]	$s, \mathbf{B}\#A\ \{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} s', \langle\ \mathbf{B}\#A_1\ \{v_1\}\ ,\ \mathbf{B}\#A_2\ \{v_2\}\ \rangle$
[R-Cross-Inl]	$s, \mathbf{B}\#A\ \{\ \mathsf{inl}(v)\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{inl}(\mathbf{B}\#A'\ \{v\})$
[R-Cross-Inr]	$s, \mathbf{B}\#A\ \{\ \mathsf{inr}(v)\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{inr}(\mathbf{B}\#A'\ \{v\})$
[R-Cross-Roll]	$s, \mathbf{B}\#A\ \{\ \mathsf{roll}_\tau(v)\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{roll}_\tau(\mathbf{B}\#A'\ \{v\})$
[R-Cross-Box]	$s, \mathbf{B}\#A\ \{\ \mathsf{box}(v)\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{proxy}(A',\ \mathsf{box}(v))$
[R-Cross-Lam]	$s, \mathbf{B}\#A\ \{\ \lambda x.e\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{proxy}(A',\ \lambda x.e)$
[R-Proxy-Unbox]	$s, \mathsf{unbox}(\mathsf{proxy}(A,\ \mathsf{box}(e))) \longrightarrow_{\mathrm{m}} s', \mathbf{B}\#A'\ \{\ \mathsf{unbox}(\mathsf{box}(e))\ \}$
[R-Proxy-β]	$s, \mathsf{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} s', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#A_a\ \{v\})\ \}$
[R-Merge-Box]	$s, \mathbf{B}\#A\ \{\ \mathsf{proxy}(A',\ \mathsf{box}(e))\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{proxy}(A'',\ \mathsf{box}(e))$
[R-Merge-Lam]	$s, \mathbf{B}\#A\ \{\ \mathsf{proxy}(A',\ \lambda x.e)\ \} \longrightarrow_{\mathrm{m}} s', \mathsf{proxy}(A'',\ \lambda x.e)$

In the rule templates, the annotations A, A', A_a, A_r, and the global states s, s' are collectively called the *placeholder metavariables*.

Figure 4.4: The rule templates of the monitor-related reduction relation

ary into the constructor. The values in the constructor may undergo further inspection while the constructor flows out as-is. Next, the [R-Cross-Box] rule handles immutable reference cells and the [R-Cross-Lam] rule handles higher-order functions. These two types are monitorable, and the monitor calculus wraps the value in the boundary with a new proxy to intercept any follow-up elimination operations on them.

At the bottom, the [R-Proxy-Unbox] rule describes how an unbox operation eliminates a proxied reference and the [R-Proxy-β] rule describes how to apply a proxy function. When accessing the content of a proxied reference, the [R-Proxy-Unbox] rule applies the unbox operation to the reference in the proxy and creates a new boundary around the whole term to guard the result of unbox(−). Similarly, when applying a proxy function, the [R-Proxy-β] rule sends the argument v to the function in the proxy. Additionally, the [R-Proxy-β] rule creates two new boundaries, one guarding the argument and one guarding the result of application.

Finally, the [R-Merge-Box] rule and the [R-Merge-Lam] rule merge boundaries into proxies.

When a proxy flows out of a boundary, the monitor calculus merges the two annotations into a new one on the proxy instead of creating nested proxies. It is not just the two R-Merge rules avoid creating nested proxies, but the grammar of $\mathsf{proxy}(A,\ e^m)$ in the monitor calculus only allow immutable references and functions to appear inside proxies in the first place.

Readers familiar with contracts may notice that in the monitor calculus, the design of proxies is different from that of other contract systems. As Chapter 6 shows, this design decision does not lose any expressiveness because the annotation language can record a list of annotations on boundaries and proxies. Additionally, merging annotations on boundaries and proxies allows for expressing different merging strategies using the annotation languages. Chapter 8 utilizes this design to model space-efficient contracts.

4.2.3 Examples

As an demonstration of the definitions, I present two example annotation languages, one is trivial and the other simply counts the number of monitor-related reduction steps.

The Trivial Annotation Language. The trivial language $(\mathscr{A}unit, \mathscr{T}unit)$ where $\mathscr{A}unit :\equiv (A, s)$ is defined by setting all annotations and the global state to just unit, `()`. There is no information being attached in the annotations, and nothing is tracked in the global states. It serves as a kind

$$A ::= \text{(unspecified; a parameter)} \qquad \mathbb{N} \ni s ::= k$$

$$\begin{aligned}
&\text{[R-Cross-Unit]} && k, \mathbf{B}\#A\,\{\ ()\ \} \longrightarrow_{\mathrm{m}} k+1, ()\\
&\text{[R-Cross-Nat]} && k, \mathbf{B}\#A\,\{\ n\ \} \longrightarrow_{\mathrm{m}} k+1, n\\
&\text{[R-Cross-Cons]} && k, \mathbf{B}\#A\,\{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} k+1, \langle\, \mathbf{B}\#A_1\,\{v_1\}\,,\, \mathbf{B}\#A_2\,\{v_2\}\rangle\\
&\text{[R-Cross-Inl]} && k, \mathbf{B}\#A\,\{\ \mathsf{inl}(v)\ \} \longrightarrow_{\mathrm{m}} k+1, \mathsf{inl}(\mathbf{B}\#A'\,\{v\})\\
&\text{[R-Cross-Inr]} && k, \mathbf{B}\#A\,\{\ \mathsf{inr}(v)\ \} \longrightarrow_{\mathrm{m}} k+1, \mathsf{inl}(\mathbf{B}\#A'\,\{v\})\\
&\text{[R-Cross-Roll]} && k, \mathbf{B}\#A\,\{\,\mathsf{roll}_\tau(v)\,\} \longrightarrow_{\mathrm{m}} k+1, \mathsf{roll}_\tau(\mathbf{B}\#A'\,\{v\})\\
&\text{[R-Cross-Box]} && k, \mathbf{B}\#A\,\{\,\mathsf{box}(v)\,\} \longrightarrow_{\mathrm{m}} k+1, \mathsf{proxy}(A',\ \mathsf{box}(v))\\
&\text{[R-Cross-Lam]} && k, \mathbf{B}\#A\,\{\,\lambda x.e\,\} \longrightarrow_{\mathrm{m}} k+1, \mathsf{proxy}(A',\ \lambda x.e)\\
&\text{[R-Proxy-Unbox]} && k, \mathsf{unbox}(\mathsf{proxy}(A,\ \mathsf{box}(e))) \longrightarrow_{\mathrm{m}} k+1, \mathbf{B}\#A'\,\{\,\mathsf{unbox}(\mathsf{box}(e))\,\}\\
&\text{[R-Proxy-}\beta\text{]} && k, \mathsf{proxy}(A,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} k+1, \mathbf{B}\#A_r\,\{\ (\lambda x.e)\ (\mathbf{B}\#A_a\,\{v\})\ \}\\
&\text{[R-Merge-Box]} && k, \mathbf{B}\#A\,\{\,\mathsf{proxy}(A',\ \mathsf{box}(e))\,\} \longrightarrow_{\mathrm{m}} k+1, \mathsf{proxy}(A'',\ \mathsf{box}(e))\\
&\text{[R-Merge-Lam]} && k, \mathbf{B}\#A\,\{\,\mathsf{proxy}(A',\ \lambda x.e)\,\} \longrightarrow_{\mathrm{m}} k+1, \mathsf{proxy}(A'',\ \lambda x.e)
\end{aligned}$$

The definition of the annotation language $(\mathscr{A}cnt, \mathscr{T}cnt)$ where $\mathscr{A}cnt :\equiv (A, s)$.

Figure 4.5: The annotation language that counts monitor-related reduction steps.

of "unit" of the annotation languages.

$$A ::= () \qquad s ::= ()$$

$$\begin{aligned}
&\text{[R-Cross-Unit]} && (), \mathbf{B}\#()\,\{\ ()\ \} \longrightarrow_{\mathrm{m}} (), ()\\
&\text{[R-Cross-Nat]} && (), \mathbf{B}\#()\,\{\ n\ \} \longrightarrow_{\mathrm{m}} (), n\\
&\text{[R-Cross-Cons]} && (), \mathbf{B}\#()\,\{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} (), \langle\, \mathbf{B}\#()\,\{v_1\}\,,\, \mathbf{B}\#()\,\{v_2\}\rangle\\
&\text{[R-Cross-Lam]} && (), \mathbf{B}\#()\,\{\,\lambda x.e\,\} \longrightarrow_{\mathrm{m}} (), \mathsf{proxy}((),\ \lambda x.e)\\
&\text{[R-Proxy-}\beta\text{]} && (), \mathsf{proxy}((),\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} (), \mathbf{B}\#()\,\{\ (\lambda x.e)\ (\mathbf{B}\#()\,\{v\})\ \}\\
&\text{[R-Merge-Lam]} && (), \mathbf{B}\#()\,\{\,\mathsf{proxy}((),\ \lambda x.e)\,\} \longrightarrow_{\mathrm{m}} (), \mathsf{proxy}((),\ \lambda x.e)
\end{aligned}$$

Counting Monitor-Related Reduction Steps. A slightly more interesting presented in Fig-

ure 4.5 is the counting language, an annotation language that counts the number of monitor-related reduction steps. It leaves the definition of the annotations unspecified—as a parameter that can be further instantiated by other annotation languages—and the global state tracks how many $\longrightarrow_{\mathrm{m}}$ steps there have been by incrementing the global state for each $\longrightarrow_{\mathrm{m}}$ reduction. In Chapter 8, I use the counting language to measure the cost of space-efficient contracts since the monitor calculus encapsulates all computation needed by the annotations in the transition steps.

Figure 4.5 has presented the transition steps of the counting language, $\mathcal{T}cnt$, in a more conventional style. The formal definition of $\mathcal{T}cnt$ is actually a set of twelve relations as follows, one for each rule template:

name	**relation**
[R-Cross-Unit]	$R \coloneqq \{ (A, k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Nat]	$R_n \coloneqq \{ (A, k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Cons]	$R_{v_1,v_2} \coloneqq \{ (A, A_1, A_2, k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Inl]	$R_v \coloneqq \{ (A, A', k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Inr]	$R_v \coloneqq \{ (A, A', k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Roll]	$R_v \coloneqq \{ (A, A', k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Box]	$R_v \coloneqq \{ (A, A', k, k+1) \mid k \in \mathbb{N} \}$
[R-Cross-Lam]	$R_{x,e} \coloneqq \{ (A, A', k, k+1) \mid k \in \mathbb{N} \}$
[R-Proxy-Unbox]	$R_e \coloneqq \{ (A, A', k, k+1) \mid k \in \mathbb{N} \}$
[R-Proxy-β]	$R_{x,e,v} \coloneqq \{ (A, A_r, A_a, k, k+1) \mid k \in \mathbb{N} \}$
[R-Merge-Box]	$R_e \coloneqq \{ (A, A', A'', k, k+1) \mid k \in \mathbb{N} \}$
[R-Merge-Lam]	$R_{x,e} \coloneqq \{ (A, A', A'', k, k+1) \mid k \in \mathbb{N} \}$

4.3 Projections of Annotation Languages

Having formally defined annotation languages, I turn to the characterization of their composition. Section 2.3 already explains how composition can be done by writing the annotation languages in an extensible style in the first place. However, how can one be certain that an individual language such as $\mathscr{T}c'$ from Figure 2.6 in page 32 is correctly defined? How can one express the intuition that $\mathscr{A}octc$ from Section 2.3 is the composition of $\mathscr{A}ctc$ and $\mathscr{A}owner$? To answer these questions, I introduce the projection of annotation languages in Definition 4.11, a concept derived from the usage of the π_A, π_S, and *put* functions in Section 2.3 that mathematically formulate the relationship between a composite annotation language and its constituents.

Definition 4.11 utilizes the relations in the transition steps of an annotation language. When an annotation language is a combination of multiple basic annotation languages, there should be a projection from the composite language to the basic languages such that precomposing the projection with the relations of the composite languages gives the relations of the basic languages.

Definition 4.11 (Projection of Annotation Language)**.** Let $(\mathscr{A}, \mathscr{T})$ and $(\mathscr{A}', \mathscr{T}')$ be two annotation languages where $\mathscr{A} :\equiv (\mathsf{Ann}, \mathsf{State})$ and $\mathscr{A}' :\equiv (\mathsf{Ann}', \mathsf{State}')$. An annotation-language *projection* from $(\mathscr{A}, \mathscr{T})$to $(\mathscr{A}', \mathscr{T}')$ is a pair of functions (π_A, π_S) such that

1. $\pi_A : \mathsf{Ann} \to \mathsf{Ann}'$ maps the annotations of $\mathscr{A}$ to the annotations of $\mathscr{A}'$.
2. $\pi_S : \mathsf{State} \to \mathsf{State}'$ maps the states of $\mathscr{A}$ to the states of $\mathscr{A}'$.
3. For any relation R in $\mathscr{T}$, if some annotations and states are related by R, their images under $\pi_A(-)$ and $\pi_S(-)$ are also related by the corresponding relation R' in $\mathscr{T}'$.

Since the monitor-related redution relation encapsulates all transition steps under a single name, condition (3) means that if $\mathscr{T} \vdash s_1, e_1 \longrightarrow_m s_2, e_2$ then $\mathscr{T}' \vdash \pi_S(s_1), \pi_{expr}(e_1) \longrightarrow_m \pi_S(s_2), \pi_{expr}(e_2)$ where the $\pi_{expr}(-)$ function recursively applies $\pi_A(-)$ to the given expression.

Example 4.12. For any annotation language $(\mathscr{A}, \mathscr{T})$, there is a trivial projection (π_A, π_S) from $(\mathscr{A}, \mathscr{T})$ to $(\mathscr{A}unit, \mathscr{T}unit)$, the trivial annotation language introduced in Section 4.2.3.

Example 4.13. Let Label be the set of labels, $\mathsf{Ctc}\,\tau$ be the set of contracts for values of type τ, and Status be the set of contract-checking status. The pair of functions, $(\pi_{\mathsf{A}}, \pi_{\mathsf{S}})$, taken from Section 2.3, is a projection from $(\mathscr{A}octc, \mathscr{T}oc)$ to $(\mathscr{A}ctc, \mathscr{T}c)$.

$$\begin{aligned}
&\pi_{\mathsf{A}} \;:\; (\mathsf{Label} \times \mathsf{Label}) \times \mathsf{Ctc}\,\tau \longrightarrow \mathsf{Ctc}\,\tau \\
&\pi_{\mathsf{A}}(\langle\langle \ell_n, \ell_p \rangle, \kappa \rangle) \;=\; \kappa
\end{aligned}$$

$$\begin{aligned}
&\pi_{\mathsf{S}} \;:\; \mathsf{unit} \times \mathsf{Status} \longrightarrow \mathsf{Status} \\
&\pi_{\mathsf{S}}(\langle (), st \rangle) \;=\; st
\end{aligned}$$

It should be noted that projections of annotation languages only determine the relationship between the composite language and its parts. The projections themselves do not indicate how the constituent languages are put together and, as Chapter 8 shows, there are multiple possibilities when composing complex annotation languages. Nonetheless, the concept of projection is sufficient for developing reusable metatheories of annotation languages.

Chapter 5

The Transition-System Representation of Contract Systems

In this chapter, I develop a transition-system-based metatheory of the monitor calculus. Section 5.1 explains how instantiations of the monitor calculus induce transition systems that can be used to prove properties about the instantiated calculus. Section 5.2 introduces annotation interpretations and the satisfaction relation. Together, they interpret annotation languages as property-carrying propositions in the language. Subsequently, Section 5.3 relates annotation interpretations to transition systems and develops a systematic method for proving properties of the instantiated calculus. Finally, Section 5.4 explains how metatheories of annotation languages can be reused by lifting projections of annotation languages to a relation between the induced transition systems.

5.1 Relating Calculus Instantiations to Transition Systems

For any annotation language $(\mathscr{A}, \mathscr{T})$ with $\mathscr{A} :\equiv (\mathsf{Ann}, \mathsf{State})$, the instantiated monitor calculus $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ induces a transition system, denoted by $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$, whose states are formed by $s \in \mathsf{State}$ together with the equivalence classes of expressions generated by the $\longrightarrow_{\mathsf{p}}$ relation. Section 3.2 has examined the observation that specific properties of the instantiated monitor calculus,

$\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$, can be reflected as specific behaviors of the induced transition system, $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$. Due to this observation, proving properties of $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ amounts to proving that $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ possesses certain behavior.

In this section, I formally define the induced transition system of an instantiated monitor calculus. To ease the exposition, I also give the formal definition of the equivalence classes of expressions generated by the $\longrightarrow_{\mathrm{p}}$ relation. Furthermore, throughout this section, all definitions are stated with respect to a particular, but unspecified, annotation language $(\mathscr{A},\mathscr{T})$ where $\mathscr{A} :\equiv (\mathsf{Ann},\mathsf{State})$.

Definition 5.1. In order to formulate the equivalence classes of expressions generated by the $\longrightarrow_{\mathrm{p}}$ relation, I define two accompanying relations based on the $\longrightarrow_{\mathrm{p}}$ relation.

1. The $\longrightarrow_{\mathrm{P}}$ relation is the closure of $\longrightarrow_{\mathrm{p}}$ over evaluation contexts, i.e. $e_1 \longrightarrow_{\mathrm{P}} e_2$ iff $e_1 = E\left[e_1'\right]$, $e_2 = E\left[e_2'\right]$ and $e_1' \longrightarrow_{\mathrm{p}} e_2'$ for some E, e_1' and e_2'.
2. The $\sim_{\mathrm{P}}$ relation is the reflexive, symmetric, and transitive closure of $\longrightarrow_{\mathrm{P}}$.

Definition 5.2. For any closed expression e, $[e]_{\mathrm{P}}$ denote the equivalence class of e with respect to $\sim_{\mathrm{P}}$, i.e. i.e. $e' \in [e]_{\mathrm{P}} \iff e' \sim_{\mathrm{P}} e$.

With the definition of the equivalence classes, the states of the induced transition system can be expressed as the set of the pairs $(s,[e]_{\mathrm{P}})$ for $s \in \mathsf{State}$ and $e \in \mathsf{Expr}$. The transition is roughly the same as the $\longrightarrow_{\mathrm{m}}$ relation, except that it has to be appropriately lifted to the equivalence classes.

Definition 5.3 (Induced Transition System)**.** For any annotation language $(\mathscr{A},\mathscr{T})$, the *induced transition system* of $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$, which I shall denote as $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$, is the transition system

$$(\{\,(s,[e]_{\mathrm{P}}) \mid s \in \mathsf{State}, e \in \mathsf{Expr}\,\},\ \longrightarrow_{\mathrm{M}})$$

where $\longrightarrow_{\mathrm{M}}$ is a binary relation over the states of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ such that $s_1,[e_1]_{\mathrm{P}} \longrightarrow_{\mathrm{M}} s_2,[e_2]_{\mathrm{P}}$ if and only if there exists E, e_1' and e_2' such that $E\left[e_1'\right] \in [e_1]_{\mathrm{P}}$, $E\left[e_2'\right] \in [e_2]_{\mathrm{P}}$, and $\mathscr{T} \vdash s_1, e_1' \longrightarrow_{\mathrm{m}} s_2, e_2'$. That is, the $\longrightarrow_{\mathrm{M}}$ relation is constructed by first closing the $\longrightarrow_{\mathrm{m}}$ relation over evaluation

contexts and then lifting the result to work with $[e]_{\mathrm{P}}$ for $e \in \mathsf{Expr}$.

5.2 Interpretation of Annotation Languages

Having introduced induced transition systems in Section 5.1, now I develop a systematic method for proving that the induced transition system of an instantiated monitor calculus has certain behavior using annotation interpretations. Recall from Section 3.2 that for an annotation language $(\mathscr{A}, \mathscr{T})$, properties of the instantiated calculus ($\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$) are manifested as specific behaviors of the induced transition system ($\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$) and these behaviors are established using homomorphisms that map $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ to some better understood transition systems. In this section, I formally define annotation interpretations and the satisfaction relation introduced in Section 3.4, both of which are the central tools for systematically building a new class of well-understood transition systems.

[*Note:* I will formulate the definitions using type-theoretic concepts and notations [Univalent Foundations Program 2013, Chapter 1 and Appendix A] where $\mathcal{U}$ denotes the universe(s) of types, $\bot$ is the empty type, $\top$ is the unit type, $A \times B$ is the product type, and $A + B$ is the sum type. Furthermore, $\sum_{a:A} B$ is the Σ-type and $\prod_{a:A} B$ is the Π-type (or, the dependent function type). Under the propositions-as-types reading, inhabitants of $\mathcal{U}$ can be thought of as propositions; $\bot : \mathcal{U}$ represents falsity; $\top : \mathcal{U}$ represents the true proposition; for $A : \mathcal{U}$ and $B : \mathcal{U}$, the type $A \times B : \mathcal{U}$ represents the conjunction of A and B; similarly, $A + B : \mathcal{U}$ represents disjunction of A and B; $\sum_{a:A} B : \mathcal{U}$ represents existential quantification and $\prod_{a:A} B : \mathcal{U}$ represents universal quantification. — *end note.*]

I initiate the discussion with the formal definition of annotation interpretations. As in the previous section, I will assume that an arbitrary but fixed annotation language $(\mathscr{A}, \mathscr{T})$ is given throughout this section where $\mathscr{A} :\equiv (\mathsf{Ann}, \mathsf{State})$.

An annotation interpretations of $(\mathscr{A}, \mathscr{T})$ includes a predicate over of $s \in \mathsf{State}$ for identifying a subset of State of interest, a preorder over the subset of State, two functions that separately

interpret boundary terms and proxy terms as propositions in the metalanguage, and a law that regulate the interaction of thetwo functions and the reduction relation. The preorder and the two functions can be chosen to formulate the properties of $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ that one wants to prove or, in terms of the induced transition system, some specific behaviors of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ that one wants to establish.

I divide the formal definition of annotation interpretations into a raw definition and the law it has to satisfy. Annotation Pre-interpretation covers the raw definition part that gives the signatures of the functions in annotation interpretations and explains their roles. Interpretation Law state the law that an annotation interpretation must adhere to.

Definition 5.4 (Annotation Interpretation)**.** An *annotation interpretation* is an annotation pre-interpretation $\mathcal{I}$ that satisfies the interpretation law.

Definition 5.5 (Annotation Pre-interpretation)**.** An *annotation pre-interpretation*, $\mathcal{I}$, is a four-tuple $(\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ where their types are:

$$\begin{array}{lcl}
\mathbb{S} & : & \mathsf{State} \to \mathcal{U} \\
\preccurlyeq & : & \sum_{s:\mathsf{State}} \mathbb{S}(s) \to \sum_{s:\mathsf{State}} \mathbb{S}(s) \to \mathcal{U} \\
\mathbb{B}[\![-,-]\!] & : & \mathsf{Ann}_\tau \to \mathsf{Expr} \to \mathcal{U} \\
\mathbb{P}[\![-,-]\!] & : & \mathsf{Ann}_\tau \to \mathsf{Expr} \to \mathcal{U}
\end{array}$$

In Definition 5.5, the $\mathbb{S}$ function is a predicate over State. Per the convention from type theory, predicates over State are expressed as functions of type $\mathsf{State} \to \mathcal{U}$. Note that functions of type $\mathsf{State} \to \mathcal{U}$ can produce the types $\top : \mathcal{U}$ and $\bot : \mathcal{U}$ which are the "true" and "false" propositions in type theory, so this representation of predicates is the type-theoretic analogue of functions that assign truth values.

On top of the $\mathbb{S}$ function, the $\preccurlyeq$ relation is a preorder of $\sum_{s:\mathsf{State}} \mathbb{S}(s)$. The dependent sum type $\sum_{s:\mathsf{State}} \mathbb{S}(s)$ represents the subset of State whose elements all satisfy $\mathbb{S}$. In Section 3.2, as motivated by the non-masking property of $\mathscr{A}ctc$, the preorder $\preccurlyeq$ can express specific behavior of the global states. For example, Figure 3.2 (a) gives a preorder asserting that the contract annotation language never changes $s \in \mathsf{State}$ from Err to Ok.

Since the preorder $\preccurlyeq$ in Definition 5.5 is defined on the subset of State whose elements all satisfy $\mathbb{S}$, I will also use s and s' to denote elements of $\sum_{s:\mathsf{State}} \mathbb{S}(s)$ when discussing states that adhere to the preorder $\preccurlyeq$. For example, I shall write $s \preccurlyeq s'$ when it is clear from the context that both $\mathbb{S}(s)$ and $\mathbb{S}(s')$ hold without formally supplying the inhabitants of $\sum_{s:\mathsf{State}} \mathbb{S}(s)$.

Next, the $\mathbb{B}[\![-, -]\!]$ function interprets a boundary term as a proposition of the metalanguage which can describe a property about the annotations and the nested expressions of the given boundary term. Specifically, for any boundary term, $\mathbf{B}\#A\,\{e\}$, $\mathbb{B}[\![A, e]\!]$ returns some type $C : \mathcal{U}$. Under the propositions-as-types reading, the type C is just a proposition involving A and e, hence $\mathbb{B}[\![A, e]\!]$ essentially expresses certain invariant of A and e. The $\mathbb{P}[\![-, -]\!]$ function is defined similarly but for proxy terms rather than boundary terms; for any term $\mathsf{proxy}(A,\ e^m)$, $\mathbb{P}[\![A, e^m]\!]$ is a proposition of the metalanguage that can carry additional invariant about A and e^m.

Example 5.6 concretely illustrates how $\mathbb{B}[\![-, -]\!]$ and $\mathbb{P}[\![-, -]\!]$ can help capture relationships between annotations and expressions using the interpretation, $\mathcal{I}val$, that captures proxies containing only *syntactic* values. Note that the grammar of the monitor calculus only ensures that proxies contain terms of form $\mathsf{box}(e)$ or $\lambda x.e$, but it is not immediate that a $\mathsf{box}(e)$ in a proxy will always be a value during evaluation. Going forward, the $\mathcal{I}val$ interpretation helps me prove that the monitor calculus does ensure the strictness of proxies in Proposition 5.20.

Example 5.6. Let the annotation interpretation $\mathcal{I}val :\equiv (\mathbb{S}_{val}, \preccurlyeq_{val}, \mathbb{B}_{val}, \mathbb{P}_{val})$ be defined as:

$$\begin{array}{lcl} \mathbb{S}_{val}(s) & :\equiv & \top \\ s \preccurlyeq_{val} s' & :\equiv & \top \\ \mathbb{B}_{val}[\![A, e]\!] & :\equiv & \top \\ \mathbb{P}_{val}[\![A, e]\!] & :\equiv & \mathsf{is\text{-}value}(e) \end{array}$$

The $\mathbb{P}_{val}$ function captures proxies that carry syntactic values. Here, the $\mathsf{is\text{-}value}(-)$ predicate holds iff a given term is a syntactic value; I shall omit its definition.

Recall that the goal of introducing annotation interpretations is to systematically construct property-satisfying transition systems such that the transition system induced by the instantiated monitor calculus has homomorphisms into them. Because the properties that the annotation interpretations can deal with also includes the monotonicity of state transitions as promised at

$$
\begin{array}{lll}
\mathcal{I} \vDash \mathbf{B}\#A\,\{e\} & \text{iff} & \mathbb{B}[\![\, A, e \,]\!] \text{ and } \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{proxy}(A,\ e^m) & \text{iff} & \mathbb{P}[\![\, A, e^m \,]\!] \text{ and } \mathcal{I} \vDash e^m \\
\mathcal{I} \vDash () & & \\
\mathcal{I} \vDash \mathsf{zero} & & \\
\mathcal{I} \vDash \mathsf{suc}(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{fold}_{\mathsf{nat}}(e,\ e_z,\ x_n\, y.\, e_s) & \text{iff} & \mathcal{I} \vDash e,\ \mathcal{I} \vDash e_z \text{ and } \mathcal{I} \vDash e_s, \\
\mathcal{I} \vDash \mathsf{assert}\, e & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \langle e_1, e_2 \rangle & \text{iff} & \mathcal{I} \vDash e_1 \text{ and } \mathcal{I} \vDash e_2 \\
\mathcal{I} \vDash \pi_i(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{inl}(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{inr}(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{case}\, e\, \mathsf{of}\, \{x.e_1 \mid y.e_2\} & \text{iff} & \mathcal{I} \vDash e,\ \mathcal{I} \vDash e_1 \text{ and } \mathcal{I} \vDash e_2, \\
\mathcal{I} \vDash \mathsf{box}(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{unbox}(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash x & & \text{(where } x \text{ is bound)} \\
\mathcal{I} \vDash \lambda x.e & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash e\, e_a & \text{iff} & \mathcal{I} \vDash e \text{ and } \mathcal{I} \vDash e_a \\
\mathcal{I} \vDash \mathsf{unroll}(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{roll}_\tau(e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash \mathsf{fix}\, x.e & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash e;\ e_1 & \text{iff} & \mathcal{I} \vDash e \text{ and } \mathcal{I} \vDash e_1
\end{array}
$$

Figure 5.1: The satisfaction relation

the end of Section 3.4, the functions in an annotation pre-interpretation must satisfy the Interpretation Law.

Definition 5.7 (Interpretation Law)**.** Let $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ be any pre-interpretation. The *interpretation law* regulating the $\preccurlyeq$ and the $\mathbb{B}$ and $\mathbb{P}$ functions are formally stated as:

- $\preccurlyeq$ is a preorder.
- $\mathbb{B}$ **is sound with respect to the transition steps** $\mathcal{T}$.

  If $\mathbb{S}(s)$, $\mathbb{S}(s')$, $s \preccurlyeq s'$ and $\mathcal{T} \vdash s, e \longrightarrow s', e'$, then $\mathbb{B}[\![\, A, e \,]\!]$ implies $\mathbb{B}[\![\, A, e' \,]\!]$.

As discussed in Section 3.4, the *satisfaction relation* lifts an interpretation of annotations, $\mathcal{I}$,

to all expressions in the instantiation $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$. Figure 5.1 shows its definition. The satisfaction relation is a binary relation $\mathcal{I} \vDash e$ that determines whether or not an expression e satisfies an interpretation $\mathcal{I}$ where the satisfaction of an expression means that the $\mathbb{B}$ and the $\mathbb{P}$ functions hold for all annotations appearing in e. Taking $\mathcal{I}val$ from Example 5.6 as an example, $\mathcal{I}val \vDash e$ would hold if an only if all proxies in e only stores syntactic values.

The satisfaction relation is defined structurally over expressions. Most cases work in a straightforward manner, but there are two noteworthy exceptions. To say that a boundary expression is satisfied by $\mathcal{I}$, that is, $\mathcal{I} \vDash \mathbf{B}\#A\,\{e\}$ holds, the satisfaction relation extracts the $\mathbb{B}$ function from the annotation interpretation $\mathcal{I}$ and requires that $\mathbb{B}[\![\, A, e \,]\!]$ holds, in addition to the requirement that $\mathcal{I} \vDash e$ recursively holds. Similarly, for $\mathcal{I} \vDash \mathrm{proxy}(A,\ e^m)$ to hold, the satisfaction relation requires that both the interpreted result $\mathbb{P}[\![\, A, e^m \,]\!]$ and the recursive component $\mathcal{I} \vDash e^m$ holds.

As another example of annotation interpretations, I define the $\mathcal{I}empty$ interpretation that interprets all boundary and proxy terms as falsity in Example 5.8. If a term e contains any boundary expressino or any proxy value, $\mathcal{I}empty \vDash e$ is unsatisfiable. Therefore, $\mathcal{I}empty$ helps the satisfaction relation precisely capture terms that do not contain boundaries and proxies.

Example 5.8. Let the annotation interpretation $\mathcal{I}empty :\equiv (\mathbb{S}_{empty}, \preccurlyeq_{empty}, \mathbb{B}_{empty}, \mathbb{P}_{empty})$ be:

$$\begin{array}{lcl}
\mathbb{S}_{empty}(s) & :\equiv & \top \\
s \preccurlyeq_{empty} s' & :\equiv & \top \\
\mathbb{B}_{empty}[\![\, A, e \,]\!] & :\equiv & \bot \\
\mathbb{P}_{empty}[\![\, A, e^m \,]\!] & :\equiv & \bot
\end{array}$$

As in Section 4.1, I follow the framework developed by McBride [2005]; Allais et al. [2017, 2018] to prove the Substitution lemma and the accompanying lemmas such as renaming and extension. These lemmas are similar to Lemmas 4.1 and 4.3 and Proposition 4.2 except that they are substituting the satisfaction relation.

Lemma 5.9 (Renaming)**.** *Let* $\Gamma :\equiv (x_1 : \tau_1), \ldots, (x_n : \tau_n)$ *and* $\Gamma' :\equiv (y_1 : \tau'_1), \ldots, (y_m : \tau'_m)$ *be given. Assume that* $\Gamma \vdash e : \tau$ *and* $\Gamma' \vdash y_{a_i} : \tau_i$ *for some* $1 \le a_i \le m$ *for* $i = 1 \ldots n$*. If* $\mathcal{I} \vDash e$ *then* $\mathcal{I} \vDash e\big[y_{a_1} \ldots y_{a_n} \,/\, x_1 \ldots x_n\big]$.

Proposition 5.10. *Assume that* $\Gamma' \vdash e_i : \tau_i$ *for* $i = 1, \ldots, n$ *and let any* $x_0 : \tau_0$ *be given.*

Let $\Gamma'' \vdash e'_0 : \tau_0, \ldots, \Gamma'' \vdash e'_n : \tau_n$ *be the sequence given by Proposition 4.2 where* $\Gamma'' :\equiv \Gamma', x_0 : \tau_0$, $e'_0 :\equiv x_0$ *and* $e'_i :\equiv e_i$ *for* $i = 1, \ldots, n$. *Then, if* $\mathcal{I} \vDash e_i$ *for* $i = 1, \ldots, n$, *there is a sequence* $\mathcal{I} \vDash e'_i$ *for* $i = 0, \ldots, n$ *as well.*

Lemma 5.11 (Substitution)**.** *Let* $\Gamma :\equiv x_1 : \tau_1, \ldots, x_n : \tau_n$ *and some* Γ' *be given. Assume that* $\Gamma \vdash e : \tau$ *and* $\Gamma' \vdash e_i : \tau_i$ *for* $i = 1 \ldots n$. *If* $\mathcal{I} \vDash e$ *and* $\mathcal{I} \vDash e_i$ *for* $i = 1 \ldots n$ *then* $\mathcal{I} \vDash e[e_1 \ldots e_n \,/\, x_1 \ldots x_n]$.

Proposition 5.12 (Decomposition)**.** *If* $\mathcal{I} \vDash e$ *and* $e = E[e_r]$ *then* $\mathcal{I} \vDash e_r$.

5.3 Soundness of the Interpretations

For an annotation language $(\mathscr{A}, \mathscr{T})$, an annotation interpretation of it describes some specific invariant about the terms of the instantiated calculus $(\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}])$. This description usually reflects a property about $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ that one wants to prove. Section 3.4 foreshadows an approach for constructing transition systems using annotation interpretations and the satisfaction relation such that there exists homomorphisms from a subsystem of the induced transition system $(\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}])$ of $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ to the just constructed transition systems, thereby establishing the desired property about $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$.

Both the satisfaction relation and annotation interpretations have been discussed in Section 5.2. In this section, I explain how to utilize the satisfaction relation to construct a transition system that carries the properties characterized by a given annotation interpretation **by construction**. I shall refer to this transition system as the interpretation-satisfying transition system and denote it using the notation $\mathcal{T}_{\mathsf{sat}}[\mathscr{A};\mathscr{T};\mathcal{I}]$. Finally, Theorem 5.19 exhibits a homomorphism from a subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ to $\mathcal{T}_{\mathsf{sat}}[\mathscr{A};\mathscr{T};\mathcal{I}]$.

To start with, I shall explain how to construct the interpretation-satisfying transition system from the satisfaction relation. Concretely, its definition follows that of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ except that it only uses global states that satisfy $\mathbb{S}(-)$ from $\mathcal{I}$, and refers to alternative equivalence classes of expressions that satisfy $\mathcal{I}$.

Definition 5.13. For any closed expression e such that $\mathcal{I} \vDash e$, $[e]_{\mathcal{I}}$ is the equivalence class of e containing only expressions that satisfy $\mathcal{I}$, i.e. $e' \in [e]_{\mathcal{I}} \iff e' \sim_{\mathrm{P}} e$ and $\mathcal{I} \vDash e'$.

Definition 5.14. For any interpretation $\mathcal{I}$, the *interpretation-satisfying transition system* which I denote as $\mathcal{T}_{\mathrm{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}]$ is

$$(\{\, (s, [e]_{\mathcal{I}}) \mid s \in \mathsf{State} \wedge e \in \mathsf{Expr} \wedge \mathbb{S}(s) \wedge \mathcal{I} \vDash e \,\}, \longrightarrow_{\mathrm{M}'})$$

where $\longrightarrow_{\mathrm{M}'}$ is a binary relation over the set of states such that $s_1, [e_1]_{\mathcal{I}} \longrightarrow_{\mathrm{M}'} s_2, [e_1]_{\mathcal{I}}$ if there exists E, e_1' and e_2' such that $E\left[e_1'\right] \in [e_1]_{\mathcal{I}}$, $E\left[e_2'\right] \in [e_1]_{\mathcal{I}}$ and $\mathscr{T} \vdash s_1, e_1' \longrightarrow_{\mathrm{m}} s_2, e_2'$, i.e. e_1' and e_2' can take a monitor-related reduction step.

Given ($\mathscr{A}$,$\mathscr{T}$) and an interpretation $\mathcal{I}$, the states of $\mathcal{T}_{\mathrm{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}]$ only contain expressions that satisfy $\mathcal{I}$. Any state that fails $\mathcal{I}$ is removed, and transitions to these states are pruned. Thus, reduction sequences captured by $\mathcal{T}_{\mathrm{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}]$ preserve $\mathcal{I}$ by construction, and establishing a homomorphism from a subsystem of $\mathcal{T}_{\mathrm{ind}}[\mathscr{A}; \mathscr{T}]$ to $\mathcal{T}_{\mathrm{sat}}[\mathscr{A}; \mathscr{T}; \mathcal{I}]$ transfers $\mathcal{I}$ back to $\mathcal{T}_{\mathrm{ind}}[\mathscr{A}; \mathscr{T}]$.

To facilitate the construction of homomorphisms from subsystems of the induced transition systems to the interpretation-satisfying transition systems, I characterize well-behaved interpretations by the monotonicity and the soundness properties. Informally speaking, an interpretation $\mathcal{I}$ is monotonic if the monitor-related reductions of $\mathscr{T}$ respect the preorder of $\mathcal{I}$. In a similar manner, $\mathcal{I}$ is sound if the monitor-related reductions of $\mathscr{T}$ preserve the satisfaction relation. Theorems 5.17 and 5.18 subsequently lift the monotonicity and the soundness of an interpretation to the full reduction relation, and Theorem 5.19 exhibits the desired homomorphisms for all monotonic and sound interpretations.

Definition 5.15 (Monotonic Interpretation)**.** An annotation interpretation $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is *monotonic* if for any s_1, s_2, e_1, e_2, if $\mathbb{S}(s_1)$ holds, $\mathscr{T} \vdash s_1, e_1 \longrightarrow_{\mathrm{m}} s_2, e_2$ and $\mathcal{I} \vDash e_1$ then $\mathbb{S}(s_2)$ holds and $s_1 \preccurlyeq s_2$.

Definition 5.16 (Sound Interpretation)**.** An annotation interpretation $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is *sound* if for any s_1, s_2, e_1, e_2 such that $\mathbb{S}(s_1)$ and $\mathbb{S}(s_2)$ hold, if $s_1 \preccurlyeq s_2$, $\mathscr{T} \vdash s_1, e_1 \longrightarrow_{\mathrm{m}} s_2, e_2$

and $\mathcal{I} \vDash e_1$ then $\mathcal{I} \vDash e_2$.

Next, I prove the Monotonicity and the Soundness theorems that lift the monotonicity and the soundness of annotation interpretations to sequences of $\longrightarrow$ reduction steps, i.e., the reduction relation of $\lambda_m[\mathscr{A};\mathscr{T}]$ from Definition 4.8. Using these two results, Theorem 5.19 establishes the existence of a homomorphism from a subsystem of the induced transition system to the interpretation-satisfying transition system.

Theorem 5.17 (Monotonicity). *Let $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ be a monotonic and sound interpretation. For any sequence of reduction steps $\mathscr{T} \vdash s, e \longrightarrow^* s', e'$, if $\mathbb{S}(s)$ and $\mathcal{I} \vDash e$ then $\mathbb{S}(s')$ and $s \preccurlyeq s'$.*

Theorem 5.18 (Soundness). *Let $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ be a monotonic and sound interpretation. For any sequence of reduction steps $\mathscr{T} \vdash s, e \longrightarrow^* s', e'$, if $\mathbb{S}(s)$ and $\mathcal{I} \vDash e$ then $\mathcal{I} \vDash e'$.*

Theorem 5.19. *Let $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ be a monotonic and sound interpretation. Assume that for some s_0 and e_0, both $\mathbb{S}(s_0)$ and $\mathcal{I} \vDash e_0$ hold. Let $\mathcal{T}$ be the minimum subsystem of $\mathcal{T}_{\text{ind}}[\mathscr{A};\mathscr{T}]$ that contains $(s_0, [e_0]_P)$, the map $h : (s, [e]_P) \longmapsto (s, [e]_{\mathcal{I}})$ for any e such that $\mathcal{I} \vDash e$ holds is a well-defined function from $\mathcal{T}$ to $\mathcal{T}_{\text{sat}}[\mathscr{A};\mathscr{T};\mathcal{I}]$. Moreover, h is a homomorphism.*

Proof. I begin the proof with the verification of the well-definedness of h. This includes two steps: (i) h is defined for all states of $\mathcal{T}$, and (ii) for any state $(s, [e]_{\text{P}})$ of $\mathcal{T}$ and any e' such that $e' \sim_{\text{P}} e$ and $\mathcal{I} \vDash e'$, $h(s, [e]_{\text{P}}) = h(s, [e']_{\text{P}})$.

Since both $\mathbb{S}(s_0)$ and $\mathcal{I} \vDash e_0$ hold, the domain of h is non-empty. To see (i), let $(s_1, [e_1]_{\text{P}})$ be any state of $\mathcal{T}$ such that $\mathcal{I} \vDash e_1$ holds and $(s_1, [e_1]_{\text{P}}) \in \text{dom}(h)$. Because $h(s_1, [e_1]_{\text{P}})$ is a state of $\mathcal{T}_{\text{sat}}[\mathscr{A};\mathscr{T};\mathcal{I}]$, $\mathbb{S}(s_1)$ also holds. If $s_1, [e_1]_{\text{P}} \longrightarrow_{\text{M}} s_2, [e_2]_{\text{P}}$, there exists E, e_1', e_2' such that $e_1 \sim_{\text{P}} E\left[e_1'\right]$, $e_2 \sim_{\text{P}} E\left[e_2'\right]$, and $\mathscr{T} \vdash s_1, e_1' \longrightarrow_{\text{m}} s_2, e_2'$. Since $\longrightarrow_{\text{p}}$ and $\longrightarrow_{\text{m}}$ are disjoint, it must be the case that $e_1 \longrightarrow_{\text{P}}^* E\left[e_1'\right]$. Hence, $\mathscr{T} \vdash e_1 \longrightarrow^* E\left[e_2'\right]$. By the Monotonicity theorem, $\mathbb{S}(s_2)$ holds. By the Soundness theorem, $\mathcal{I} \vDash E\left[e_2'\right]$ holds. To see (ii), it suffices to note that by definition, $h(s, [e]_{\text{P}}) = h(s, [e']_{\text{P}})$ if and only if $[e]_{\mathcal{I}} = [e']_{\mathcal{I}}$. Therefore, (ii) trivially holds because it is equivalent to $e' \in [e]_{\mathcal{I}}$.

Next, I verify that h is a homomorphism from $\mathcal{T}$ to $\mathcal{T}_{\text{sat}}[\mathscr{A};\mathscr{T};\mathcal{I}]$. That h preserves the tran-

sitions of $\mathcal{T}$ has been proven as part of (i) of its well-definedness. Conversely, suppose that $s_1, [e_1]_{\mathcal{I}} \longrightarrow'_{\mathrm{M}} s_2, [e_2]_{\mathcal{I}}$. By the definition of $\longrightarrow'_{\mathrm{M}}$, there exists E, e'_1, e'_2 such that $E\left[e'_1\right] \in [e_1]_{\mathcal{I}}$, $E\left[e'_2\right] \in [e_2]_{\mathcal{I}}$, and $\mathcal{T} \vdash s_1, e'_1 \longrightarrow_{\mathrm{m}} s_2, e'_2$. Consequently, $s_1, [e_1]_{\mathrm{P}} \longrightarrow_{\mathrm{M}} s_2, [e_2]_{\mathrm{P}}$. □

Proposition 5.20. *In Example 5.6, the annotation interpretation $\mathcal{I}val$ is monotonic and sound.*

Let me illustrate the theorems by showing that for any annotation language $(\mathscr{A}, \mathscr{T})$, the monitor calculus will only store values in proxies. Note that a separate proof is required to show that proxies only carry values because the grammar does not restrict what proxies can contain. By using the $\mathcal{I}val$ interpretation from Example 5.6, expressions whose proxies only contain values are those characterized by $\mathcal{I}val \vDash e$. Consequently, applying Theorem 5.19 to Proposition 5.20 guarantees that as along as one starts with an expression that has the desired property (e.g. one where there are no proxies), the monitor calculus will never wrap proxies around an unevaluated reference cell.

A Note on the Agda Implementation. In Definitions 5.15 and 5.16, I directly define the monotonicity and soundness of an interpretation in terms of the satisfaction of expressions with respect to the monitor-related reduction steps. However, the satisfaction relation is defined recursively over expressions, so only the boundary and proxy forms in a given expression need to be addressed. The Agda mechanization actually takes advantage of this and eliminates boilerplate in the proof obligations it presents to the user. For example, the proof goals that the Agda implementation generated for the soundness with respect to the [R-Cross-Cons] rule from Figure 4.4 only includes the interpretation of boundaries:

Reduction Rule	$\mathcal{T} \vdash s, \mathbf{B}\#A\,\{\,\langle v_1, v_2\rangle\,\} \longrightarrow_{\mathrm{m}} s',\ \langle\ \mathbf{B}\#A_1\,\{\,v_1\,\},\ \mathbf{B}\#A_2\,\{\,v_2\,\}\ \rangle$
Assumption	$\mathbb{S}(s),\ \ \mathbb{S}(s'),\ \ s \preccurlyeq s',\ \ \text{and}\ \mathbb{B}[\![\, A, \langle v_1, v_2\rangle\,]\!]$
Obligation	$\mathbb{B}[\![\, A_1, v_1\,]\!]$ and $\mathbb{B}[\![\, A_2, v_2\,]\!]$

To understand the derivation of the simplified proof goals, recall that an interpretation $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is sound with respect to $\mathcal{T}$ means that if $s \preccurlyeq s'$ *and* $\mathcal{I} \vDash \mathbf{B}\#A\,\{\langle v_1, v_2\rangle\}$ both hold then $\mathcal{I} \vDash \langle \mathbf{B}\#A_1\,\{v_1\}, \mathbf{B}\#A_2\,\{v_2\}\rangle$ holds as well. By definition, the latter boils down to checking both

$\mathcal{I} \vDash \mathbf{B}\#A_1\,\{v_1\}$ and $\mathcal{I} \vDash \mathbf{B}\#A_2\,\{v_2\}$. In other words, it is sufficient to look at the boundary and proxy forms for monitor-related reduction steps.

This is not the whole story yet. In the boundary case, for $\mathcal{I} \vDash \mathbf{B}\#A\,\{e\}$ to hold it means that both $\mathbb{B}[\![\, A, e \,]\!]$ and $\mathcal{I} \vDash e$ have to hold. When it comes to the [R-Cross-Cons] rule, the soundness property expands to the statement that if both $\mathbb{B}[\![\, A, \langle v_1, v_2 \rangle \,]\!]$ and $\mathcal{I} \vDash \langle v_1, v_2 \rangle$ hold, then all of $\mathbb{B}[\![\, A_1, v_1 \,]\!]$, $\mathbb{B}[\![\, A_2, v_2 \,]\!]$, $\mathcal{I} \vDash v_1$ and $\mathcal{I} \vDash v_2$ have to hold. But the assumption $\mathcal{I} \vDash \langle v_1, v_2 \rangle$ actually cancels out the obligations $\mathcal{I} \vDash v_1$ and $\mathcal{I} \vDash v_2$, hence only the predicates resulting from $\mathbb{B}[\![\, -, - \,]\!]$ are needed in the soundness statement.

In the end, for the interpretation $\mathcal{I}$ to be sound with respect to the [R-Cross-Cons] rule in Figure 4.4, one needs to prove that if $\mathbb{B}[\![\, A, \langle v_1, v_2 \rangle \,]\!]$ then $\mathbb{B}[\![\, A_1, v_1 \,]\!]$ and $\mathbb{B}[\![\, A_2, v_2 \,]\!]$. Put differently, one only needs to look at boundary (and proxy) terms and interpret them using the annotation interpretation. The same process generalizes not only to other the rule templates in Figure 4.4, but also to the monotonicity property.

5.4 Reusing Metatheories by Composing Homomorphisms

One last missing piece in my discussion about the monitor calculus is the method for building reusable metatheories of annotation languages. In this section, I show that when a language of annotations is built from other basic languages of annotations, the corresponding annotation-language projections give rise to weak homomorphisms from the induced transition system of the composed language to the induced transition systems of the basic languages.

Let me briefly review the framework for proving properties of annotation languages. Previously, Sections 5.1 to 5.3 demonstrate that for an annotation language $(\mathscr{A}, \mathscr{T})$, a desired property of the instantiation calculus ($\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$) is first reframed as some specific behavior of the induced transition system ($\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$). Then, for an appropriately defined annotation interpretation $\mathcal{I}$, the interpretation-satisfying transition system ($\mathcal{T}_{\mathsf{sat}}[\mathscr{A};\mathscr{T};\mathcal{I}]$) will exhibit the particular transition behavior specified by $\mathcal{I}$ *by construction*. Thus, establishing a homomorphism from

$\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ to $\mathcal{T}_{\mathsf{sat}}[\mathscr{A};\mathscr{T};\mathscr{I}]$ in turn proves the desired property about $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$.

Since it is sufficient to study the behavior of the induced transition system, the problem of building modular and reusable metatheories is reduced to understanding the behavior of the induced transition system in a reusable manner. To solve this problem, Section 2.3 observes that the composition of annotation languages gives rise to transition system homomorphisms between the transition systems induced by the respective annotation languages.

As it turns out, for a composite annotation language, when properties about its constituent annotation languages are proven through homomorphisms from the induced transition systems of the constituent languages to other transition systems (e.g. the interpretation-satisfying system), composing these homomorphisms with the one between the induced transition system of the composite annotation language and that of the constituent languages transitively establishes properties about the composite annotation language.

Sections 2.3, 3.2 and 3.3 altogether is a concrete instantiation of this idea. Because Section 3.2 proves properties about $\mathscr{A}ctc$ and $\mathscr{A}owner$, Section 3.3 reuses these homomorphisms to bring the properties about the individual language to the language that combines contracts and blames, demonstrating the reusability of their metatheories.

To close the loop, I prove that there are weak homomorphisms from the induced transition system of a composite annotation language to the induced transition system of the constituent annotation languages in Theorem 5.21, where the composite language and its constituents are formally related using the concept of projections of annotation languages introduced in Section 4.3.

Theorem 5.21. *Let* $(\mathscr{A},\mathscr{T})$ *and* $(\mathscr{A}',\mathscr{T}')$ *where* $\mathscr{A} :\equiv (\mathsf{Ann},\mathsf{State})$ *and* $\mathscr{A}' :\equiv (\mathsf{Ann}',\mathsf{State}')$ *be two annotation languages such that there is a projection,* $(\pi_{\mathsf{A}},\pi_{\mathsf{S}})$*, from* $(\mathscr{A},\mathscr{T})$ *to* $(\mathscr{A},\mathscr{T}')$*, the function*

$$h_{\mathsf{proj}} : (s,[e]_P) \longmapsto (\pi_{\mathsf{S}}(s),[\pi_{\mathsf{expr}}(e)]_P)$$

from $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ *to* $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}';\mathscr{T}']$ *is a weak homomorphism. Here, the function* $\pi_{\mathsf{expr}}(-)$ *recursively applies* $\pi_{\mathsf{A}}(-)$ *to all annotations in a given expression.*

Proof. It is easy to verify that h_{proj} is well-defined by induction over the $\longrightarrow_{\mathsf{p}}$ relation.

Let $\longrightarrow_{\mathrm{M}}$ be the transition relation of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A};\mathscr{T}]$ and $\longrightarrow_{\mathrm{M}'}$ be the transition relation of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}';\mathscr{T}']$. For any two states of $\lambda_{\mathsf{m}}[\mathscr{A};\mathscr{T}]$ such that $(s_1, [e_1]_{\mathrm{P}}) \longrightarrow_{\mathrm{M}} (s_2, [e_2]_{\mathrm{P}})$, I shall prove that $h_{\mathsf{proj}}(s_1, [e_1]_{\mathrm{P}}) \longrightarrow_{\mathrm{M}'} h(s_2, [e_2]_{\mathrm{P}})$ as follows.

When $(s_1, [e_1]_{\mathrm{P}}) \longrightarrow_{\mathrm{M}} (s_2, [e_2]_{\mathrm{P}})$, there exists E, e_1', e_2' such that $E\left[e_1'\right] \in [e_1]_{\mathrm{P}}$, $E\left[e_2'\right] \in [e_2]_{\mathrm{P}}$, and $\mathscr{T} \vdash s_1, e_1' \longrightarrow_{\mathrm{m}} s_2, e_2'$. By Definition 4.11 (3), $\mathscr{T}' \vdash \pi_{\mathrm{S}}(s_1), \pi_{\mathsf{expr}}(e_1') \longrightarrow_{\mathrm{m}} \pi_{\mathrm{S}}(s_2), \pi_{\mathsf{expr}}(e_2')$. Moreover, it is easy to prove that $\pi_{\mathsf{expr}}(E)\left[\pi_{\mathsf{expr}}(e_1')\right] = \pi_{\mathsf{expr}}(E\left[e_1'\right]) \in [\pi_{\mathsf{expr}}(e_1)]_{\mathrm{P}}$ by induction. Similarly, $\pi_{\mathsf{expr}}(E)\left[\pi_{\mathsf{expr}}(e_2')\right] = \pi_{\mathsf{expr}}(E\left[e_2'\right]) \in [\pi_{\mathsf{expr}}(e_2)]_{\mathrm{P}}$. Therefore, by the definition of $\longrightarrow_{\mathrm{M}'}$, $h_{\mathsf{proj}}(s_1, [e_1]_{\mathrm{P}}) \longrightarrow_{\mathrm{M}'} h_{\mathsf{proj}}(s_2, [e_2]_{\mathrm{P}})$. □

In Section 2.3, Figure 2.5 demonstrates how to form composite annotation languages with the example $(\mathscr{A}\mathit{octc}, \mathscr{T}\mathit{oc})$ that combines both blame tracking from $\mathscr{A}\mathit{owner}$ and contract checking from $\mathscr{A}\mathit{ctc}$. Section 3.3 further discusses how to reuse the metatheories developed in Section 3.2 for the composite language, $\mathscr{A}\mathit{octc}$.

In Figure 3.4, because $\mathscr{A}\mathit{octc}, \mathscr{T}\mathit{oc}$ is formed by composing $(\mathscr{A}\mathit{owner}, \mathscr{T}\mathit{o})$ and $(\mathscr{A}\mathit{ctc}, \mathscr{T}\mathit{c})$, the map h_{proj_1} that erases contracts and their checking status is a weak homomorphism from $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{octc}; \mathscr{T}\mathit{oc}]$ to $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{owner}; \mathscr{T}\mathit{o}]$. Similarly, the map h_{proj_2} that erases owenrship labels is a weak homomorphism mapping $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{octc}; \mathscr{T}\mathit{oc}]$ to $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}\mathit{ctc}; \mathscr{T}\mathit{c}]$.

Theorem 5.21 justifies the existence and the correctness of h_{proj_2} and h_{proj_1}. As of the h_{chk} and h_{own} homomorphisms from Figure 3.2 that separately proves non-masking property of $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{ctc};\mathscr{T}\mathit{c}]$ and the single-owner policy of $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{owner};\mathscr{T}\mathit{o}]$, both homomorphisms are instances of Theorem 5.19.

Part III

Applications to Contract Metatheories

Chapter 6

Findler-Felleisen Contract System and the Non-masking Property

In this chapter, I illustrate the development of transition-system-based reusable metatheory by modeling Findler and Felleisen [2002] higher-order contracts in the monitor calculus and proving its non-masking property. The model is an annotation language $(\mathscr{A}ctc, \mathscr{T}c)$ that characterizes the contract checking and the monitoring capabilities where $\mathscr{T}c$ is the transition steps of $\mathscr{A}ctc$. Contracts in $\mathscr{A}ctc$ are represented as annotations on boundaries and proxies. The contract checking results are tracked by the state of the instantiated calculus, which can be either Ok or Err(ℓ) for some label ℓ.

$\mathscr{A}ctc$ partly handles a distinctive feature of Findler and Felleisen [2002] contract system: any custom predicate can serve as contracts. In other words, their contract system is *open* and the contracts can even be dynamically computed in host languages. This capability greatly improved the practicality of contracts since library developers can leverage host language features to organize and abstract contracts.

To allow custom predicates, contracts in $\mathscr{A}ctc$ can refer to arbitrary terms in the instantiated calculus $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$. While this arrangement does not capture the full first-class contracts like what Findler and Felleisen [2002] do, allowing custom predicates showcases the expressiveness of

annotation languages: the transition steps of an annotation language can integrate other calculi and even the monitor calculus itself.

Technically, the variant of contract systems presented in this chapter is closer to Dimoulas et al. [2011, 2012]'s design in that contracts are both simply typed and separated from the evaluation of ordinary expressions. Although adding a type system to higher-order contracts in the formalism deviates from the implementation of the Findler and Felleisen [2002] contract system in Racket, the typed variant greatly simplifies the analysis the contract system while still shedding light on its design.

I prove the non-masking property for the full $\mathscr{A}ctc$ language in this chapter. Specifically, I show that $\mathscr{A}ctc$ may change the state from Ok to Err(ℓ) for some ℓ. Once the state is set to Err(ℓ), $\mathscr{A}ctc$ never reverts it back to Ok or changes it to Err(ℓ') for a different ℓ'. This is the monotonicity property introduced in Section 3.2 which matches the state changes of $\lambda_{\mathsf{m}}[\mathscr{A}ctc;\mathscr{T}c]$ with the blame-raising behavior of typical contract systems.

$\mathscr{A}ctc$ is the also basis for the study of blames and the space-efficient contracts, and I will built on it in Chapters 7 and 8. Chapter 7 constructs an annotation language for blame and composes it with $\mathscr{A}ctc$ to analyze the interaction between blame and contract violations. Chapter 8 defines an annotation language for Greenberg [2016]'s variant of space-efficient contracts and combines it with $\mathscr{A}ctc$ to show that the behaviors of the two contract systems are equivalent. In these two applications, $\mathscr{A}ctc$'s monotonicity property transfers to the composite annotation languages, demonstrating how my framework supports proof reuse across different contract systems.

6.1 The Syntax of Contracts

Figure 6.1 presents the definition of $\mathscr{A}ctc$ and the syntax of contracts. In this chapter, I shall let the metavariables A_{c} and s_{c} separately range over the annotations and the states of $\mathscr{A}ctc$, respectively. The annotations A_{c} are lists of contracts which I will explain shortly. The states s_{c} can be either Ok or Err(ℓ) for some label ℓ. Last, the metavariable κ ranges over contracts.

$$\mathscr{A}ctc \;::\equiv\; (A_c, s_c) \qquad \begin{array}{rcl} A_c & ::= & [\kappa_1, \ldots, \kappa_m] \\ s_c & ::= & \text{Status} \end{array} \qquad \frac{\vdash^c \kappa_1 : \text{Ctc}\,\tau \qquad \cdots \qquad \vdash^c \kappa_m : \text{Ctc}\,\tau}{\vdash [\kappa_1, \ldots, \kappa_m] : \text{Ann}_\tau}$$

$$\begin{array}{rcl} \text{Status} \ni st & ::= & \textsc{Ok} \mid \textsc{Err}(\ell) \\ \kappa & ::= & \text{unit/c} \mid \text{flat}^\ell(x.\, e) \mid \kappa_1 \times\!\!/c\, \kappa_2 \mid \kappa_1 +\!\!/c\, \kappa_2 \mid \text{box/c}\,\kappa \mid \kappa_a \to\!\!/c\, \kappa_r \mid t \mid \mu/c\, t.\kappa \end{array}$$

Figure 6.1: The syntax contract annotation language, $\mathscr{A}ctc$

$$\boxed{\Delta :\equiv \{t_1, \ldots, t_n\}} \qquad \boxed{\Delta \vdash^c \kappa : \text{Ctc}\,\tau}$$

$$\frac{}{\Delta \vdash^c \text{unit/c} : \text{Ctc unit}} \qquad \frac{x : \text{nat} \vdash e : \text{nat}}{\Delta \vdash^c \text{flat}^\ell(x.\, e) : \text{Ctc nat}} \qquad \frac{\Delta \vdash^c \kappa_1 : \text{Ctc}\,\tau_1 \qquad \Delta \vdash^c \kappa_2 : \text{Ctc}\,\tau_2}{\Delta \vdash^c \kappa_1 \times\!\!/c\, \kappa_2 : \text{Ctc}\,(\tau_1 \times \tau_2)}$$

$$\frac{\Delta \vdash^c \kappa_1 : \text{Ctc}\,\tau_1 \qquad \Delta \vdash^c \kappa_2 : \text{Ctc}\,\tau_2}{\Delta \vdash^c \kappa_1 +\!\!/c\, \kappa_2 : \text{Ctc}\,(\tau_1 + \tau_2)} \qquad \frac{\Delta \vdash^c \kappa : \text{Ctc}\,\tau}{\Delta \vdash^c \text{box/c}\,\kappa : \text{Ctc}\,(\text{Box}\,\tau)}$$

$$\frac{\Delta \vdash^c \kappa_a : \text{Ctc}\,\tau_a \qquad \Delta \vdash^c \kappa_r : \text{Ctc}\,\tau_r}{\Delta \vdash^c \kappa_a \to\!\!/c\, \kappa_r : \text{Ctc}\,(\tau_a \to \tau_r)} \qquad \frac{t \in \Delta}{\Delta \vdash^c t : \text{Ctc}\,t} \qquad \frac{\Delta, t \vdash^c \kappa : \text{Ctc}\,\tau}{\Delta \vdash^c \mu/c\, t.\kappa : \text{Ctc}\,(\mu t.\tau)}$$

Figure 6.2: The typing rules of contracts

To understand why the annotations are lists of contracts, let us work through a few examples of the instantiated calculus $\lambda_m[\mathscr{A}ctc;\mathscr{T}c]$. The first one is rewritten from the contracted function in Figure 2.1 in page 26. This function expects an odd number and promises to produce an even number, so a corresponding contract is attached to it via a proxy. Here, I assume that isOdd and isEven denote the appropriate expressions of the monitor calculus that check whether the given number is odd (or even), and + denotes the expression that sums two numbers.

$$\text{proxy}\big([\text{flat}^{\ell_1}(x.\,\text{isOdd}\,x) \to\!\!/c\, \text{flat}^{\ell_2}(x.\,\text{isEven}\,x)],\ \lambda x.x + 2\big)$$

$\mathscr{A}ctc$ has two differences comparing to the informal presentation in Figure 2.1 in page 26. Since the monitor calculus disallows nested proxies by design, one difference is that annotations in $\mathscr{A}ctc$ are lists of contracts, so the contract attached to $\lambda x.x + 2$ from Figure 2.1 appears in a singleton list in $\mathscr{A}ctc$. The other difference is that the predicates isOdd and isEven are wrapped inside the constructor flat where the additional labels ℓ_1 and ℓ_2 identify the source locations of the predicates and the variable x binds the number to be checked.

Boundaries and proxies can be attached with more than one contract, as the syntax of *Actc* suggests. As a second example, consider the following function, which is enclosed in two boundaries, and hence guarded by two different contracts.

$$\mathbf{B}\#[\kappa_2 \to\!\!\!/_{\mathsf{c}}\, \kappa_2']\ \{\quad \mathbf{B}\#[\kappa_1 \to\!\!\!/_{\mathsf{c}}\, \kappa_1']\ \{\ \lambda x.e\ \}\quad \}$$

In this program, when $\lambda x.e$ flows out of the boundaries, the monitor calculus creates a proxy. As a proxy collects all attached contracts in its annotation as a list, the above expression reduces to a proxy whose annotation is the two-element list containing $\kappa_1 \to\!\!\!/_{\mathsf{c}}\, \kappa_1'$ and $\kappa_2 \to\!\!\!/_{\mathsf{c}}\, \kappa_2'$.

$$\mathsf{proxy}\big([\kappa_1 \to\!\!\!/_{\mathsf{c}}\, \kappa_1', \kappa_2 \to\!\!\!/_{\mathsf{c}}\, \kappa_2'],\ \ \lambda x.e\,\big)$$

In the annotation, the contract $\kappa_1 \to\!\!\!/_{\mathsf{c}}\, \kappa_1'$ comes first because $\lambda x.e$ reaches it prior to meeting the other contract. In this sense, when a proxy is annotated with a list of contracts, the ones closer to the front of the list can be thought of as the inner contracts that resides closer to the proxied value whereas the ones closer to the rear of the list are the outer contracts.

This order of contracts in the annotations is more apparent when it comes to the order in which *Actc* checks flat contracts. When there are multiple predicates, *Actc* checks them from the left to the right. For example, the following boundary guards the number n with the predicates $\mathsf{flat}^{\ell_1}(x.\, e_1)$ and $\mathsf{flat}^{\ell_2}(x.\, e_2)$.

$$\mathbf{B}\#[\mathsf{flat}^{\ell_1}(x.\, e_1), \mathsf{flat}^{\ell_2}(x.\, e_2)]\ \{\ n\ \}$$

When n passes the boundary, *Actc* will first check e_1 against n and then check e_2 against n. If $e_1[n\,/\,x]$ reduces to 0, the contract checking process is aborted without running $e_2[n\,/\,x]$.

Having seen examples of *Actc*, we turn to the details of *Actc*. Figure 6.2 shows the typing rules of contracts. The evaluation of contracts are covered by the transition steps of *Actc* in Section 6.2. In *Actc*, contracts are typed using the judgment $\Delta \vdash^{\mathsf{c}} \kappa : \mathsf{Ctc}\,\tau$ where $\mathsf{Ctc}\,\tau$ types contracts that guard values of type τ. The set Δ in the typing judgement contains the type variables that the contract κ and the type τ can refer to.

The contracts annotated on boundaries and proxies are all typed using the empty context. To see this, recall that the typing rule of boundaries from Figure 4.2 in page 53 is

$$\frac{\vdash A : \mathsf{Ann}_\tau \qquad \vdash e : \tau}{\Gamma \vdash \mathbf{B}\#A\,\{e\} : \tau}\,.$$

The premise $\vdash A : \mathsf{Ann}_\tau$ types the annotation of boundaries, or list of contracts in $\mathscr{A}ctc$. Its typing rule is shown at the top of Figure 6.1. For A a list of contracts $[\kappa_1, \ldots, \kappa_m]$, the rule types each contract using only the empty environment, ignoring Γ. Similarly, the typing rule of proxies types the annotations on proxies as Ann_τ. Therefore, all annotated contracts must be closed.

Let us go over the syntax of contracts together with their typing rules. Since contracts in $\mathscr{A}ctc$ are typed, there is one kind of contract for each type in the monitor calculus. The contract for the unit type is $\mathsf{unit/c} : \mathsf{Ctc}\,\mathsf{unit}$. The contracts for natural numbers are called *flat contracts* and are represented as $\mathsf{flat}^\ell(x.\,e) : \mathsf{Ctc}\,\mathsf{nat}$. For a flat contract $\mathsf{flat}^\ell(x.\,e)$, the label ℓ identifies the origin of the contract when reporting contract violations. The expression e is a predicate that takes a natural number input $x : \mathsf{nat}$ and produces a natural number. The predicate e is not allowed to use other identifiers in the context, thus it is typed using the judgment $x : \mathsf{nat} \vdash e : \mathsf{nat}$. For a number n, the contract $\mathsf{flat}^\ell(x.\,e)$ fails if $e[n\,/\,x]$ evaluates to zero and passes otherwise.

Next, the contract for pairs and disjoint sums are written as $\kappa_1 \times\!\!/\mathsf{c}\ \kappa_2 : \mathsf{Ctc}\,(\tau_1 \times \tau_2)$ and $\kappa_1 +\!\!/\mathsf{c}\ \kappa_2 : \mathsf{Ctc}\,(\tau_1 + \tau_2)$ and have their expected meaning. The contract for immutable reference cells is $\mathsf{box/c}\,\kappa : \mathsf{Ctc}\,(\mathsf{Box}\,\tau)$ where $\kappa : \mathsf{Ctc}\,\tau$ is the contract of the value stored in the reference cell. The arrow contract for functions is $\kappa_a \to\!\!/\mathsf{c}\ \kappa_r : \mathsf{Ctc}\,(\tau_a \to \tau_a)$ where $\kappa_a : \mathsf{Ctc}\,\tau_a$ is the *domain contract* and $\kappa_r : \mathsf{Ctc}\,\tau_r$ is the *range contract*. Finally, the contract for recursive types is $\mu/\mathsf{c}\ t.\kappa : \mathsf{Ctc}\,(\mu t.\tau)$. When typing the body κ of the recursive contract, the typing rule extends the context with the additional type variable t just like how the recursive type $\mu t.\tau$ binds the type variable t in τ.

Similar to the Substitution lemma of expressions in page 55 and the Substitution lemma of the satisfaction relation in page 74, I prove the substitution lemma of contracts in three steps via the Renaming lemma, the extension operation (Proposition 6.2) and finally the Substitution lemma.

Lemma 6.1 (Renaming). *Let* $\Delta :\equiv \{t_1, \ldots, t_n\}$ *and* $\Delta' :\equiv \{t'_1, \ldots, t'_m\}$ *be given. Assume that there is a sequence* $1 \le a_i \le m$ *for* $i = 1 \ldots n$. *If* $\Delta \vdash^{\mathsf{c}} \kappa : \mathsf{Ctc}\,\tau$ *then* $\Delta' \vdash^{\mathsf{c}} \kappa\left[t'_{a_1}, \ldots, t'_{a_n}\,/\,t_1, \ldots, t_n\right] : \mathsf{Ctc}\left(\tau\left[t'_{a_1}, \ldots, t'_{a_n}\,/\,t_1, \ldots, t_n\right]\right)$.

Proposition 6.2. *Assume that* $\Delta' \vdash^{c} \kappa_i : \mathrm{Ctc}\ \tau_i$ *for* $i = 1 \ldots n$ *and let any* $t_0 \notin \Delta'$ *be given. There is a sequence* $\Delta'' \vdash^{c} \kappa_i : \mathrm{Ctc}\ \tau_i$ *for* $i = 0 \ldots n$ *where* $\Delta'' :\equiv \Delta', t_0$ *and* $\kappa_0 :\equiv t_0$.

Lemma 6.3 (Substitution)**.** *Let* $\Delta :\equiv \{t_1, \ldots, t_n\}$ *and some* Δ' *be given. Assume that* $\Delta' \vdash^{c} \kappa_i : \mathrm{Ctc}\ \tau_i$ *for* $i = 1 \ldots n$. *If* $\Delta \vdash^{c} \kappa : \mathrm{Ctc}\ \tau$ *then* $\Delta' \vdash^{c} \kappa[\kappa_1, \ldots, \kappa_n \,/\, t_1, \ldots, t_n] : \mathrm{Ctc}\,(\tau[\tau_1, \ldots, \tau_n \,/\, t_1, \ldots, t_n])$.

6.2 The Contract Checking Transition Steps

In this section, I turn to the dynamics of $\mathscr{A}ctc$, or its transition steps, $\mathscr{T}c$. Figure 6.4 first defines the dynamics of contracts in the form of computation rules, then Figure 6.5 defines the metafunctions that implement flat contract checks.

Since contracts in $\mathscr{A}ctc$ are arbitrary predicates from $\lambda_{\mathrm{m}}[\mathscr{A}ctc;\mathscr{T}c]$, $\mathscr{T}c$ incorporates the reduction relation of the monitor calculus, i.e. the $\longrightarrow$ relation, to evaluate the predicates. This makes $\longrightarrow$ a recursive definition because the $\longrightarrow_{\mathrm{m}}$ relation in it refers to $\mathscr{T}c$, which in turn is defined using $\longrightarrow$. Since the theory developed in Chapter 5 does not account for this, I stratify $\mathscr{T}c$ into layers to avoid the cyclic dependency.

The transition steps of $\mathscr{A}ctc$ are indexed by the maximum depth of nested contract occurrences. The base case $\mathscr{T}c_0$ is an empty relation that disallows any $\longrightarrow_{\mathrm{m}}$ step from taking place. The inductive case, $\mathscr{T}c_{i+1}$ (which is coming shortly in Figure 6.4), uses the instantiation $\lambda_{\mathrm{m}}[\mathscr{A}ctc; \mathscr{T}c_i]$ to evaluate the predicates in flat contracts. Thus, contracts are technically modeled by a family of annotation languages $\{(\mathscr{A}ctc, \mathscr{T}c_i)\}_{i \geq 0}$ although all of them are morally identical.

$$
\begin{aligned}
f :\equiv\ & \mathsf{proxy}\big([\mathsf{flat}^{\ell_1}(z.\ div360'\ z)\ \rightarrow\!\!/\mathrm{c}\ \mathsf{any/c}^{\ell_2}],\ \lambda x.e\big) \\
\text{where}\quad div360' :\equiv\ & \mathsf{proxy}\big([\mathsf{flat}^{\ell_3}(z.\ isPos\ z)\ \rightarrow\!\!/\mathrm{c}\ \mathsf{any/c}^{\ell_4}],\ \lambda x.div360\ x\big) \\
isPos :\equiv\ & \lambda w.\ \mathsf{fold}_{\mathsf{nat}}(w,\ 0,\ x\ y.\ 1) \\
\mathsf{any/c}^{\ell} :\equiv\ & \mathsf{flat}^{\ell}(z.\ 1)
\end{aligned}
$$

Figure 6.3: An example of nested contracts

Figure 6.3 better explains the role of stratification and what nested contracts look like. Assuming that the function f in the example only accepts factors of 360 as inputs, the contract guarding f checks its argument with $\mathsf{flat}^{\ell_1}(z.\ div360'\ z)$. This flat contract tests the argument of f with the

predicate *div360′* to ensure that the input is a factor of 360. In the predicate *div360′*, the function *div360* implements the actual divisibility test which only works for positive integers. Hence, *div360′* guards *div360* with the contract $\mathsf{flat}^{\ell_3}(z.\ \mathit{isPos}\, z) \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_4}$ where *isPos* ensures that the inputs are positive.

The contract of f—one that uses *div360′*—is an instance of nested contracts in $\mathscr{A}ctc$. The function f is protected by a contract, $\mathsf{flat}^{\ell_1}(z.\ \mathit{div360}'\ z) \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_2}$, and this contract in turn contains yet another contract, $\mathsf{flat}^{\ell_3}(z.\ \mathit{isPos}\, z) \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_4}$. Therefore, the contract nesting depth of f is 2, and similarly the contract nesting depth of *div360′* is 1.

The contract nesting depth of an expression affects the minimum index of the subscript in $\mathscr{T}c_i$ that is needed to evaluate the expression. For example, when checking the contract $\mathsf{flat}^{\ell_1}(z.\ \mathit{div360}'\ z)$, $\mathscr{A}ctc$ needs to evaluate *div360′* and, since is it contracted, another contract checking process is initiated to evaluate $\mathsf{flat}^{\ell_3}(z.\ \mathit{isPos}\, z)$ which finally contains only base expressions.

Going back from *isPos*, the evaluation of $\mathsf{flat}^{\ell_3}(z.\ \mathit{isPos}\, z)$ requires only the base reduction steps (i.e. the $\longrightarrow_{\mathrm{p}}$ relation), so $\mathscr{T}c_0$ is sufficient. Calls to *div360′*, however, need to be evaluated using the transition steps $\mathscr{T}c_1$ at minimum since checking it requires the evaluation of $\mathsf{flat}^{\ell_3}(z.\ \mathit{isPos}\, z)$. Transitively, calls to f need at least the transition steps $\mathscr{T}c_2$ as evaluating f requires two nested contract checking process.

The Transition Steps $\mathscr{T}c_i$. I shall now walk through the definition of $\mathscr{T}c$. The base case of the transition steps $\mathscr{T}c_0$ is the empty relation, so the reduction relation $\mathscr{T}c_0 \vdash \_, \_ \longrightarrow \_, \_$ includes all of the $\longrightarrow_{\mathrm{p}}$ relation, but it cannot take any $\longrightarrow_{\mathrm{m}}$ step. For the inductive case, i.e. $\mathscr{T}c_{i+1}$, the details are presented in Figure 6.4. Amongst the rules, the [R-Cross-Nat] rule is the core of $\mathscr{A}ctc$ that formalizes the enforcement of flat contract. When a number n passes a boundary annotated with the contracts $[\kappa_1, \ldots, \kappa_m]$, the [R-Cross-Nat] rule invokes the $\mathit{checkCtcs}_{\mathscr{T}c_i}$ function to update the state from Ok to the contract checking result, s'. The $\mathit{checkCtcs}_{\mathscr{T}c_i}$ function takes a list of flat contracts, a number, and runs the flat contracts from left to right using the transition steps $\mathscr{T}c_i$. The $\mathit{checkCtcs}_{\mathscr{T}c_i}$ returns the relation between the original state and the updated state, so [R-Cross-Nat] pulls out the updated state with $(\mathrm{Ok}, s') \in \mathit{checkCtcs}_{\mathscr{T}c_i,\, get,\, put}([\kappa_1, \ldots, \kappa_m],\ n)$. I

$$
\begin{array}{rl}
\text{[R-Cross-Unit]} & \textsc{Ok}, \mathbf{B}\#[\mathsf{unit}/\mathsf{c}, \ldots, \mathsf{unit}/\mathsf{c}]\ \{\ ()\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, ()
\\[2ex]
\text{[R-Cross-Nat]} & \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{\ n\ \} \longrightarrow_{\mathrm{m}} s', n
\\
\text{where} & (\textsc{Ok}, s') \in \mathit{checkCtcs}_{\mathscr{TC}_i,\, \mathit{get},\, \mathit{put}}([\kappa_1, \ldots, \kappa_m],\ n)
\\[2ex]
\text{[R-Cross-Cons]} & \textsc{Ok}, \mathbf{B}\#[\left(\kappa_1 \times\!/\mathsf{c}\ \kappa_1'\right), \ldots, \left(\kappa_m \times\!/\mathsf{c}\ \kappa_m'\right)]\ \{\ \langle v_1, v_2 \rangle\ \} \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \langle\ \mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{v_1\},\ \mathbf{B}\#[\kappa_1', \ldots, \kappa_m']\ \{v_2\}\ \rangle
\\[2ex]
\text{[R-Cross-Inl]} & \textsc{Ok}, \mathbf{B}\#[\left(\kappa_1 +\!/\mathsf{c}\ \kappa_1'\right), \ldots, \left(\kappa_m +\!/\mathsf{c}\ \kappa_m'\right)]\ \{\ \mathsf{inl}(v)\ \} \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathsf{inl}(\mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{v\})
\\[2ex]
\text{[R-Cross-Inr]} & \textsc{Ok}, \mathbf{B}\#[\left(\kappa_1 +\!/\mathsf{c}\ \kappa_1'\right), \ldots, \left(\kappa_m +\!/\mathsf{c}\ \kappa_m'\right)]\ \{\ \mathsf{inr}(v)\ \} \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathsf{inr}(\mathbf{B}\#[\kappa_1', \ldots, \kappa_m']\ \{v\})
\\[2ex]
\text{[R-Cross-Roll]} & \textsc{Ok}, \mathbf{B}\#[(\mu/\mathsf{c}\ t.\kappa_1), \ldots, (\mu/\mathsf{c}\ t.\kappa_m)]\ \{\ \mathsf{roll}_\tau(v)\ \} \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathsf{roll}_\tau(\mathbf{B}\#[\kappa_1[(\mu/\mathsf{c}\ t.\kappa_1)\ /\ t], \ldots, \kappa_m[(\mu/\mathsf{c}\ t.\kappa_m)\ /\ t]]\ \{\ v\ \})
\\[2ex]
\text{[R-Cross-Box]} & \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{\ \mathsf{box}(v)\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{proxy}([\kappa_1, \ldots, \kappa_m],\ \mathsf{box}(v))
\\[2ex]
\text{[R-Cross-Lam]} & \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{\ \lambda x.e\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{proxy}([\kappa_1, \ldots, \kappa_m],\ \lambda x.e)
\\[2ex]
\text{[R-Proxy-Unbox]} & \textsc{Ok}, \mathsf{unbox}(\mathsf{proxy}([\mathsf{box}/\mathsf{c}\ \kappa_1, \ldots, \mathsf{box}/\mathsf{c}\ \kappa_m],\ \mathsf{box}(e))) \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{\ \mathsf{unbox}(\mathsf{box}(e))\ \}
\\[2ex]
\text{[R-Proxy-}\beta\text{]} & \textsc{Ok}, \mathsf{proxy}\big([\left(\kappa_1 \to\!/\mathsf{c}\ \kappa_1'\right), \ldots, \left(\kappa_m \to\!/\mathsf{c}\ \kappa_m'\right)],\ \lambda x.e\big)\ v \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathbf{B}\#[\kappa_1', \ldots, \kappa_m']\ \{\ (\lambda x.e)\ (\mathbf{B}\#[\kappa_m, \ldots, \kappa_1]\ \{v\})\ \}
\\[2ex]
\text{[R-Merge-Box]} & \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_l]\ \big\{\ \mathsf{proxy}\big([\kappa_1', \ldots, \kappa_m'],\ \mathsf{box}(e)\big)\ \big\} \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathsf{proxy}\big([\kappa_1', \ldots, \kappa_m', \kappa_1, \ldots, \kappa_l],\ \mathsf{box}(e)\big)
\\[2ex]
\text{[R-Merge-Lam]} & \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_l]\ \big\{\ \mathsf{proxy}\big([\kappa_1', \ldots, \kappa_m'],\ \lambda x.e\big)\ \big\} \longrightarrow_{\mathrm{m}}
\\
& \textsc{Ok}, \mathsf{proxy}\big([\kappa_1', \ldots, \kappa_m', \kappa_1, \ldots, \kappa_l],\ \lambda x.e\big)
\end{array}
$$

This figure defines the inductive case of the transition steps $\mathscr{TC}_{i+1}$ of $\mathscr{A}ctc$ for $i \geq 0$. The base case $\mathscr{TC}_0$ is the empty relation.

Figure 6.4: The transition steps, $\mathscr{TC}_{i+1}$, of the contract annotation language

will return the formal definition of $\mathit{checkCtcs}_{\mathscr{TC}_i}$ in Figure 6.5 later.

There are two more rules that are worth paying attention to. The first one is the [R-Proxy-β] rule. When applying a proxy function annotated with the contracts $[\left(\kappa_1 \to\!/\mathsf{c}\ \kappa_1'\right), \ldots, \left(\kappa_m \to\!/\mathsf{c}\ \kappa_m'\right)]$ to an argument v, the [R-Proxy-β] rule creates two new boundaries, one around the function

application and the other around the argument v. The [R-Proxy-β] rule further annotates the boundary around the function application with the range contracts, $[\kappa'_1, \ldots, \kappa'_m]$. The other boundary around v is annotated with the domain contracts, $[\kappa_m, \ldots, \kappa_1]$.

It should be noted that the order of the domain contracts are *reversed*: since the contracts closer to the end of the list of contracts on a proxy are the *outer* ones (cf. the discussion of the examples in page 86), they are closer to the argument in a function application. Therefore, the respective domain contracts should be checked first.

Next, the [R-Merge-Lam] rule merges the annotations when a proxy crosses a boundary. In $\mathscr{A}ctc$, the transition steps $\mathscr{T}c_i$ simply concatenates the two list of contracts. Again, it should be noted that the order of concatenation is *reversed*. The contracts on the boundary, $[\kappa_1, \ldots, \kappa_l]$, are the *outer* ones with respect to the contracts on the proxy, $[\kappa'_1, \ldots, \kappa'_m]$. Therefore, the new list of contracts on the proxy should be $[\kappa'_1, \ldots, \kappa'_m, \kappa_1, \ldots, \kappa_l]$.

The rest of the rules decompose and propagate contracts on boundaries and proxies in accordance with the types. For example, the [R-Cross-Cons] rule takes as inputs the annotations on the boundary, $[(\kappa_1 \mathbin{\not\times_{\mathrm{c}}} \kappa'_1), \ldots, (\kappa_m \mathbin{\not\times_{\mathrm{c}}} \kappa'_m)]$, and the pair that crosses the boundary, $\langle v_1, v_2 \rangle$, and then creates two new boundaries around v_1, v_2 and distributes the corresponding sub-contracts $[\kappa_1, \ldots, \kappa_m]$ and $[\kappa'_1, \ldots, \kappa'_m]$ on the new boundaries. The other R-Cross rules are defined similarly, so I shall omit their explanation.

A Note on Extensibility. Figure 6.4 can be easily turned into an extensible form in the spirit of Section 2.3 by using π_{A} and π_{S} to extract annotations and global states from a larger annotation language akin to how Figure 2.6 achieves extensibility for $\mathscr{T}c'$. In particular, in rule [R-Cross-Nat], the *checkCtcs* metafunction is already defined in terms of π_{S} (written as *get* in the figure) and *put* as *checkCtcs* needs to update the global states. Apart from this rule, all other rules only inspects the contents of annotations and global states without updating them.

The Metafunction for Checking Predicates. Now, let me explain the $checkCtcs_{\mathscr{T}}$ function that captures the contract checking process. Figure 6.5 formalizes $checkCtcs_{\mathscr{T}}$ and its auxiliary definitions as functions that produce binary relations over Status, essentially describing how the state

$$
\begin{aligned}
&\mathit{checkPred}_{\mathscr{T},\,\mathit{get},\,\mathit{put}}(\mathsf{flat}^{\ell}(x.\,e),\ n) :\equiv \\
&\qquad \{(s, s') \mid (\mathscr{T} \vdash s, e[n\,/\,x] \longrightarrow^{*} s', \mathsf{suc}(n')) \wedge \mathit{get}(s') = \textsc{Ok}\ \} \\
&\quad \cup\ \{(s, s'') \mid (\mathscr{T} \vdash s, e[n\,/\,x] \longrightarrow^{*} s', \mathsf{zero}) \quad \wedge \mathit{get}(s') = \textsc{Ok} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge s'' = \mathit{put}(s', \textsc{Err}(\ell))\ \} \\
&\quad \cup\ \{(s, s') \mid (\mathscr{T} \vdash s, e[n\,/\,x] \longrightarrow^{*} s', e') \quad \wedge \mathit{get}(s') = \textsc{Err}(\ell')\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{checkCtcs}_{\mathscr{T},\,\mathit{get},\,\mathit{put}}([],\ n) :\equiv \mathit{id} \\
&\mathit{checkCtcs}_{\mathscr{T},\,\mathit{get},\,\mathit{put}}([\mathsf{flat}^{\ell}(x.\,e), \kappa_2, \ldots, \kappa_m],\ n) :\equiv \\
&\quad \left(\left(\mathit{guard}(\textsc{Ok}) \circ \mathit{checkPred}_{\mathscr{T},\,\mathit{get},\,\mathit{put}}(\mathsf{flat}^{\ell}(x.\,e),\ n)\right) \cup \bigcup_{\ell' \in \mathsf{Label}} \mathit{guard}(\textsc{Err}(\ell')) \right) \circ \\
&\qquad \mathit{checkCtcs}_{\mathscr{T},\,\mathit{get},\,\mathit{put}}([\kappa_2, \ldots, \kappa_m],\ n) \\
&\text{where } \mathit{id} :\equiv \{(s, s) \mid s \in \mathsf{State}\} \\
&\qquad \mathit{guard}(\mathit{st}) :\equiv \{(s, s) \mid s \in \mathsf{State} \wedge \mathit{get}(s) = \mathit{st}\}
\end{aligned}
$$

Figure 6.5: The contract checking relation

evolves during the contract checking process. When the [R-Cross-Nat] rule uses $\mathit{checkCtcs}_{\mathscr{T}}$ in the transition steps $\mathscr{T}_{c_i}$, for example, wires up the transition of states by pulling out the pair of Status from the result of $\mathit{checkCtcs}_{\mathscr{T}}$:

$$
\begin{aligned}
&[\text{R-Cross-Nat}] \quad \textsc{Ok}, \mathbf{B}\#[\kappa_1, \ldots, \kappa_m]\ \{\ n\ \} \longrightarrow_{\mathrm{m}} s', n \\
&\qquad\qquad \text{where} \quad (\textsc{Ok}, s') \in \mathit{checkCtcs}_{\mathscr{T}_{c_i},\,\mathit{get},\,\mathit{put}}([\kappa_1, \ldots, \kappa_m],\ n).
\end{aligned}
$$

The two auxiliary definitions and the composition of relations forms a small imperative-flavored language for transition relations. To be concrete, the *id* relation can be thought of as the do-nothing command that leaves the state unchanged. The other auxiliary relation, $\mathit{guard}(s)$, imperatively restricts the state to be s. Finally, the *composition* of relations[1] captures the *sequencing* of transitions: for binary relations R_1 and R_2, their composition $R_1 \circ R_2$ is the relation for which the state changes first in accordance with R_1 and then according to R_2.

Figure 6.5 gives the definition of the $\mathit{checkCtcs}_{\mathscr{T}}$ metafunction using the notation discussed in the previous paragraph. It takes a transition steps $\mathscr{T}$ (in the subscript), a list of flat contracts, a natural number n, and sequentially enforces the contracts from left to right against the given num-

[1] Formally, $R_1 \circ R_2$ is the relation $\{\ (s, s') \mid \exists s''.\ sR_1s'' \wedge s''R_2s'\ \}$.

ber. When the list of contracts is empty, $\mathit{checkCtcs}_{\mathscr{T},\mathit{get},\mathit{put}}([],\ n)$ returns the identity relation, i.e. leaves the state unchanged. Otherwise, $\mathit{checkCtcs}_{\mathscr{T},\mathit{get},\mathit{put}}([\mathsf{flat}^{\ell}(x.\,e), \kappa_2, \ldots, \kappa_m],\ n)$ is the relation that checks $\mathsf{flat}^{\ell}(x.\,e)$ against n followed by the recursive call $\mathit{checkCtcs}_{\mathscr{T},\mathit{get},\mathit{put}}([\kappa_2, \ldots, \kappa_m],\ n)$.

In this chapter, for the instance $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{ctc}; \mathscr{T}c_{i+1}]$, by setting $\mathit{get} = \mathit{id}$ and $\mathit{put} = \mathsf{const}\ \mathit{id}$, $\mathit{checkCtcs}_{\mathscr{T}c_{i+1},\ \mathit{id},\ \mathsf{const}\ \mathit{id}}([\kappa_1, \ldots, \kappa_m],\ n)$ is the binary relation over Status such that

1. $(\textsc{Ok}, \textsc{Ok}) \in \mathit{checkCtcs}_{\mathscr{T}c_{i+1},\ \mathit{id},\ \mathsf{const}\ \mathit{id}}([\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)],\ n)$ if for all $1 \leq j \leq m$, there exists n'_j such that $\mathscr{T} \vdash \textsc{Ok}, e_j[n\,/\,x] \longrightarrow^* \textsc{Ok}, \mathsf{suc}(n'_j)$.
2. $(\textsc{Ok}, \textsc{Err}(\ell_k)) \in \mathit{checkCtcs}_{\mathscr{T}c_{i+1},\ \mathit{id},\ \mathsf{const}\ \mathit{id}}([\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)],\ n)$ if $\mathscr{T} \vdash \textsc{Ok}, e_k[n\,/\,x] \longrightarrow^* \textsc{Ok}, \mathsf{zero}$ and for all $1 \leq j \leq k-1$, there exists n'_j such that $\mathscr{T} \vdash \textsc{Ok}, e_j[n\,/\,x] \longrightarrow^* \textsc{Ok}, \mathsf{suc}(n'_j)$.
3. $(\textsc{Ok}, \textsc{Err}(\ell')) \in \mathit{checkCtcs}_{\mathscr{T}c_{i+1},\ \mathit{id},\ \mathsf{const}\ \mathit{id}}([\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)],\ n)$ if $\mathscr{T} \vdash \textsc{Ok}, e_k[n\,/\,x] \longrightarrow^* \textsc{Err}(\ell'), e'_k$ and for all $1 \leq j \leq k-1$, there exists n'_j such that $\mathscr{T} \vdash \textsc{Ok}, e_j[n\,/\,x] \longrightarrow^* \textsc{Ok}, \mathsf{suc}(n'_j)$.

and *checkPred* simplifies to the following:

$$
\begin{array}{rcll}
\mathit{checkPred}_{\mathscr{T}}(\mathsf{flat}^{\ell}(x.\,e),\ n) & :\equiv & \{(s, \textsc{Ok}) & \mid\ \mathscr{T} \vdash s, e[n\,/\,x] \longrightarrow^* \textsc{Ok}, \mathsf{suc}(n')\} \\
& \cup & \{(s, \textsc{Err}(\ell)) & \mid\ \mathscr{T} \vdash s, e[n\,/\,x] \longrightarrow^* \textsc{Ok}, \mathsf{zero}\} \\
& \cup & \{(s, \textsc{Err}(\ell')) & \mid\ \mathscr{T} \vdash s, e[n\,/\,x] \longrightarrow^* \textsc{Err}(\ell'), e'\}
\end{array}
$$

There are two cases when $\mathit{checkCtcs}_{\mathscr{T},\mathit{get},\mathit{put}}([\mathsf{flat}^{\ell}(x.\,e), \kappa_2, \ldots, \kappa_m],\ n)$ handles $\mathsf{flat}^{\ell}(x.\,e)$. If the current state is $\textsc{Ok}$ as identified by $\mathit{guard}(\textsc{Ok})$, the $\mathit{checkCtcs}_{\mathscr{T}}$ metafunction invokes the $\mathit{checkPred}_{\mathscr{T}}$ metafunction to enforce $\mathsf{flat}^{\ell}(x.\,e)$ on n. Otherwise, if the state is $\textsc{Err}(\ell')$ for some ℓ' as identified by $\bigcup_{\ell' \in \mathsf{Label}} \mathit{guard}(\textsc{Err}(\ell'))$, the $\mathit{checkCtcs}_{\mathscr{T}}$ function does nothing. Note that the $\mathit{checkCtcs}_{\mathscr{T}}$ function always traverses over the entire list of contracts. Whenever a contract fails while $\mathit{checkPred}_{\mathscr{T}}$ enforces it, the remaining contracts are skipped since they are the $\mathit{checkPred}_{\mathscr{T}}$ check is guarded by $\mathit{guard}(\textsc{Ok})$. The $\mathit{checkCtcs}_{\mathscr{T}}$ function simply passes the failure state ($\textsc{Err}(\ell')$) over through $\bigcup_{\ell' \in \mathsf{Label}} \mathit{guard}(\textsc{Err}(\ell'))$.

$$
\begin{array}{lll}
\mathcal{I} \vDash \mathsf{flat}^{\ell}(x.\,e) & \text{iff} & \mathcal{I} \vDash e \\
\mathcal{I} \vDash t & & \text{(where } t \text{ is bound)} \\
\mathcal{I} \vDash \mathsf{unit/c} & & \\
\mathcal{I} \vDash \kappa_1 \times\!/\mathsf{c}\ \kappa_2 & \text{iff} & \mathcal{I} \vDash \kappa_1 \text{ and } \mathcal{I} \vDash \kappa_2 \\
\mathcal{I} \vDash \kappa_1 +\!/\mathsf{c}\ \kappa_2 & \text{iff} & \mathcal{I} \vDash \kappa_1 \text{ and } \mathcal{I} \vDash \kappa_2 \\
\mathcal{I} \vDash \mathsf{box/c}\ \kappa & \text{iff} & \mathcal{I} \vDash \kappa \\
\mathcal{I} \vDash \kappa_a \to\!/\mathsf{c}\ \kappa_r & \text{iff} & \mathcal{I} \vDash \kappa_a \text{ and } \mathcal{I} \vDash \kappa_r \\
\mathcal{I} \vDash \mu/\mathsf{c}\ t.\kappa & \text{iff} & \mathcal{I} \vDash \kappa
\end{array}
$$

Figure 6.6: The satisfaction relation of contracts

The $\mathit{checkPred}_{\mathcal{T},\,\mathit{get},\,\mathit{put}}(\mathsf{flat}^{\ell}(x.\,e),\,n)$ function does the actual work of enforcing $\mathsf{flat}^{\ell}(x.\,e)$ against n. In its definition, $\mathit{checkPred}_{\mathcal{T},\,\mathit{get},\,\mathit{put}}(\mathsf{flat}^{\ell}(x.\,e),\,n)$ evaluates $e[n\,/\,x]$ using the $\longrightarrow$ relation under the transition steps $\mathcal{T}$ and there are three potential outcomes: if, starting with s, the evaluation terminates with the state Ok and a non-zero number $\mathsf{suc}(n')$ for some n', the overall result of $\mathit{checkPred}_{\mathcal{T},\,\mathit{get},\,\mathit{put}}(\mathsf{flat}^{\ell}(x.\,e),\,n)$ is $\{(s, \textsc{Ok})\}$. Otherwise, if the evaluation terminates with the state Ok and the number zero, the result of $\mathit{checkPred}_{\mathcal{T},\,\mathit{get},\,\mathit{put}}(\mathsf{flat}^{\ell}(x.\,e),\,n)$ is $\{(s, \textsc{Err}(\ell))\}$. Last, if the evaluation terminates in the state $\textsc{Err}(\ell')$ for some ℓ', the overall result is $\{(s, \textsc{Err}(\ell'))\}$.

6.3 The Satisfaction Relation of Contracts

Since the annotations in $\mathscr{A}\mathit{ctc}$—or list of contracts—can include arbitrary terms, I define the satisfaction relation of contracts analogous to that of the expressions from Figure 5.1 in page 72. The satisfaction relation over contracts is an essential tool for modularly establishing the metatheories of blames and space-efficient contracts in Chapters 7 and 8.

Figure 6.6 presents the definition of the satisfaction relation of contracts. For an annotation interpretation $\mathcal{I}$ and a contract κ, the relation $\mathcal{I} \vDash \kappa$ asserts that the contract κ satisfies the interpretation $\mathcal{I}$. The satisfaction of a contract κ is defined compositionally over κ. For example, a function contract $\kappa_a \to\!/\mathsf{c}\ \kappa_r$ is satisfied if and only if the subcontracts κ_a and κ_r are both satisfied.

The base case for flat contracts, $\mathsf{flat}^{\ell}(x.\, e)$, is where an arbitrary term e may appear. In this case, the flat contract satisfies the interpretation $\mathcal{I}$—or $\mathcal{I} \vDash \mathsf{flat}^{\ell}(x.\, e)$—if and only if the term e itself satisfies $\mathcal{I}$, i.e. $\mathcal{I} \vDash e$ holds.

Now, I shall prove the substitution lemmas for the satisfaction relation of contracts.

Lemma 6.4 (Renaming)**.** *Let $\Delta :\equiv \{t_1, \ldots, t_n\}$ and $\Delta' :\equiv \{t'_1, \ldots, t'_m\}$ be given. Assume that there is a sequence $1 \leq a_i \leq m$ for $i = 1 \ldots n$. If $\Delta \vdash^{\mathsf{c}} \kappa : \mathsf{Ctc}\,\tau$ and $\mathcal{I} \vDash \kappa$ then $\mathcal{I} \vDash \kappa\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right]$.*

Proposition 6.5. *Assume that $\Delta' \vdash^{\mathsf{c}} \kappa_i : \mathsf{Ctc}\,\tau_i$ for $i = 1, \ldots, n$ and let any $t_0 \notin \Delta'$ be given. Let $\Delta'' \vdash^{\mathsf{c}} \kappa'_i : \mathsf{Ctc}\,\tau_i$ for $i = 0, \ldots, n$ be the sequence given by Proposition 6.2 where $\Delta'' :\equiv \Delta', t_0$, and $\kappa'_i :\equiv \kappa_i$ for $i = 1, \ldots, n$.*

Then, if $\mathcal{I} \vDash \kappa_i$ for $i = 1, \ldots, n$, there is a sequence $\mathcal{I} \vDash \kappa'_i$ for $i = 0, \ldots, n$ as well.

Lemma 6.6 (Substitution)**.** *Let $\Delta :\equiv \{t_1, \ldots, t_n\}$ and some Δ' be given. Assume that $\Delta \vdash^{\mathsf{c}} \kappa : \mathsf{Ctc}\,\tau$ and that $\Delta' \vdash^{\mathsf{c}} \kappa_i : \mathsf{Ctc}\,\tau_i$ for $i = 1, \ldots, n$.*

Then, if $\mathcal{I} \vDash \kappa$ and $\mathcal{I} \vDash \kappa_i$ for $i = 1, \ldots, n$ then $\mathcal{I} \vDash \kappa[\kappa_1, \ldots, \kappa_n \,/\, t_1, \ldots, t_n]$.

6.4 Monotonicity

In this section, I prove that the transition steps $\{\mathscr{T}c_i\}_{i \geq 0}$ changes the global state of $\mathscr{A}ctc$ monotonically using the framework developed in Sections 5.2 and 5.3. In particular, the state of $\mathscr{A}ctc$ can only change from Ok to $\textsc{Err}(\ell)$ for some ℓ and, once the global state is set to $\textsc{Err}(\ell)$ by the transition steps $\{\mathscr{T}c_i\}_{i \geq 0}$, it is never changed back to Ok or $\textsc{Err}(\ell')$ for a different ℓ'.

The monotonicity of the annotation language $\{(\mathscr{A}ctc, \mathscr{T}c_i)\}_{i \geq 0}$ is the full-fledged version of the monotonic property discussed in Figure 3.2 (a) of Section 3.2 in page 39. Earlier, the enforcement of contracts in Section 3.2 leaves out potential nested checks that could be triggered due to the evaluation of predicates from flat contracts. Therefore, the monotonicity of $\mathscr{A}ctc$ in Figure 3.2 (a) involves only top-level reductions. In this chapter, contracts in $\{(\mathscr{A}ctc, \mathscr{T}c_i)\}_{i \geq 0}$ can contain arbitrary expressions. Consequently, additional treatment is needed during the evaluation of predicates in flat contracts to guarantee that the global state is monotonic at any point.

To prove the monotonicity theorem in my framework, I first define an annotation interpretation $\mathcal{I}mono_i$ that captures how the global state evolves using a partial order following the steps in Section 5.2. Formally, Definition 6.7 presents the definition of $\mathcal{I}mono_i$. The preorder $s \preccurlyeq_{mono} s'$ specifies that the global state can remain unchanged, or it can (only) change from Ok to Err(ℓ). The interpretation of boundary terms, $\mathbb{B}_{i+1}$, recursively applies $\mathcal{I}mono_i$ to each contract in the annotation using the satisfaction relation on contracts from Figure 6.6 in page 94 in Section 6.3.

Definition 6.7. The annotation interpretation $\mathcal{I}mono_i :\equiv (\mathbb{S}_{mono}, \preccurlyeq_{mono}, \mathbb{B}_i, \mathbb{P}_i)$ is defined as

$$
\begin{array}{lcl}
\mathbb{S}_{mono}(s) & :\equiv & \top \\
s \preccurlyeq_{mono} s' & :\equiv & (s = s') \vee (\exists \ell.\ s = \textsc{Ok} \wedge s' = \textsc{Err}(\ell)) \\
\mathbb{B}_0[\![\ A, e\ ]\!] & :\equiv & \bot \\
\mathbb{B}_{i+1}[\![\ [\kappa_1, \ldots, \kappa_m], e\ ]\!] & :\equiv & (\mathcal{I}mono_i \vDash \kappa_1) \wedge \ldots \wedge (\mathcal{I}mono_i \vDash \kappa_m) \\
\mathbb{P}_i[\![\ A, e^m\ ]\!] & :\equiv & \mathbb{B}_i[\![\ A, e^m\ ]\!]
\end{array}
$$

Next, to complete the proof using my framework, I prove that the annotation interpretation $\mathcal{I}mono_i$ is both monotonic and sound in Theorem 6.8 following the discussion in Section 5.3.

Theorem 6.8. *The annotation interpretation $\mathcal{I}mono_i$ is monotonic and sound for all $i \geq 0$.*

Finally, by Theorem 6.8 above and Theorem 5.19 in page 76, there is a homomorphism from $\mathcal{T}_{\text{ind}}[\mathcal{A}ctc; \mathcal{T}c_i]$, the induced transition system of the annotation language $\mathcal{T}c_i$, to the transition system $\mathcal{T}_{\text{sat}}[\mathcal{A}ctc; \mathcal{T}c_i; \mathcal{I}mono_i]$, the one which is characterized by the annotation interpretation $\mathcal{I}mono_i$. Since $\mathcal{I}mono_i$ ensures that the global state is monotonic, it follows that the annotation language $\mathcal{T}c_i$ changes the global state only monotonically as well.

Chapter 7

The Correct Blame of Contracts

In Findler and Felleisen [2002]'s calculus, contract violations come with *blame* to help programmers locate the source code containing the bug.[1] In their work, each contract is assigned two labels that separately identify the two parties of the contract. The intention is that each party corresponds to a specific source file and, when a particular party is blamed, the bug to be fixed is in the corresponding source file.

$$
\begin{array}{l}
\mathsf{proxy}([\kappa],\ \lambda x.x)\ (\lambda y.y)\ 5 \\
\quad \text{where}\ \ \kappa :\equiv \left(\mathsf{isEven}^{\ell_1} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_2}\right) \rightarrow\!\!/\mathsf{c}\ \left(\mathsf{isEven}^{\ell_3} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_4}\right) \\
\qquad\quad \mathsf{isEven}^{\ell} :\equiv \mathsf{flat}^{\ell}(z.\ \text{——elided——}) \\
\qquad\quad \mathsf{isOdd}^{\ell} :\equiv \mathsf{flat}^{\ell}(z.\ \text{——elided——}) \\
\qquad\quad \mathsf{any}/\mathsf{c}^{\ell} :\equiv \mathsf{flat}^{\ell}(z.\ 1)
\end{array}
$$

Figure 7.1: Blame of higher-order functions

For example, consider the program in Figure 7.1. Because there are no source files in the monitor calculus, I use proxies to syntactically delimit the program into multiple regions where each region contains the code that originates from a distinct source file. When there is a contract violation, the error will contain one of the four labels. Findler and Felleisen [2002] assert that the labels ℓ_1 and ℓ_4 indicate that the bug comes from the region inside the proxy. In a similar spirit, contract violations with the labels ℓ_2 or ℓ_3 indicate that the bug is in the region outside the proxy.

From their assertion, Findler and Felleisen [2002] design an algorithm that assigns appropriate

[1] Lazarek et al. [2019] empirically evaluate the effectiveness of blame for debugging programs using the Rational Programmer methodology. Unfortunately, they discover that blame does not significantly outperform stack traces.

$$
\begin{aligned}
&\textsc{Ok},\ \mathsf{proxy}([\kappa'],\ \lambda x.x)\ (\lambda y.y)\ 5 \longrightarrow \\
&\textsc{Ok},\ \left(\mathbf{B}\#[\mathsf{isEven}^{\ell_C} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_S}]\ \{\ (\lambda x.x)\ \left(\mathbf{B}\#[\mathsf{isEven}^{\ell_S} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_C}]\ \{\lambda y.y\}\right)\ \}\right)\ 5 \longrightarrow^* \\
&\textsc{Ok},\ \left(\mathbf{B}\#[\mathsf{isEven}^{\ell_C} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_S}]\ \{\ \mathsf{proxy}\left([\mathsf{isEven}^{\ell_S} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_C}],\ \lambda y.y\right)\ \}\right)\ 5 \longrightarrow \\
&\textsc{Ok},\ \mathsf{proxy}\left([\mathsf{isEven}^{\ell_S} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_C}, \mathsf{isEven}^{\ell_C} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_S}],\ \lambda y.y\right)\ 5 \longrightarrow \\
&\textsc{Ok},\ \mathbf{B}\#[\mathsf{isOdd}^{\ell_C}, \mathsf{any}/\mathsf{c}^{\ell_S}]\ \left\{(\lambda y.y)\ \left(\mathbf{B}\#[\mathsf{isEven}^{\ell_C}, \mathsf{isEven}^{\ell_S}]\ \{\,5\,\}\right)\right\} \longrightarrow \\
&\textsc{Err}(\ell_C),\ \mathbf{B}\#[\mathsf{isOdd}^{\ell_C}, \mathsf{any}/\mathsf{c}^{\ell_S}]\ \{(\lambda y.y)\ 5\}
\end{aligned}
$$

Figure 7.3: The reduction sequence of the program in Figure 7.2.

blame labels on contracts to point out the erroneous source file. More specifically, their algorithm would assign the same label to the predicates that are currently labeled ℓ_1 and ℓ_4 in Figure 7.1. Similarly, their algorithm also gives the same label to the predicates currently labeled ℓ_2 and ℓ_3, ensuring that each region delimited by proxies or boundaries matches a unique label. Then, these labels matching the regions can be thought as the names of the source files.

$$
\begin{aligned}
&\mathsf{proxy}([\kappa'],\ \lambda x.x)\ (\lambda y.y)\ 5 \\
&\quad \text{where } \kappa' :\equiv \left(\mathsf{isEven}^{\ell_S} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_C}\right) \rightarrow\!\!/\mathsf{c}\ \left(\mathsf{isEven}^{\ell_C} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_S}\right)
\end{aligned}
$$

Figure 7.2: Refined labels on contracts

Figure 7.2 shows the same program that contains the contract with the refined assignment of the labels. In the figure, the contract κ' is the same as the contract κ from Figure 7.1 except that the labels $\ell_1, \ldots, \ell_4$ are replaced by ℓ_S and ℓ_C. The label ℓ_S refers to the region inside the proxy whereas the label ℓ_C refers to the other region in the figure. When there is a contract violation, the error would be either $\textsc{Err}(\ell_S)$ or $\textsc{Err}(\ell_C)$, telling the user which part of the code needs to be fixed.

Let me take a close look at the program from Figure 7.2 to see why Findler and Felleisen [2002] call the labels "blame". When reducing the program in Figure 7.2, the number 5 fails the contract isEven^{ℓ_C}. Therefore, the monitor calculus reports the error $\textsc{Err}(\ell_C)$. To see why this is the case, see Figure 7.3 that expands more on the details of the reduction. The program effectively checks the number 5 against the two isEven^{ℓ_S} predicates and then applies $\lambda y.y$ to 5. Afterwards, the function application result is checked by isOdd^{ℓ_C} and $\mathsf{any}/\mathsf{c}^{\ell_S}$, respectively.

From the reduction sequence in Figure 7.3, changing the code in the region inside the proxy cannot truly fix the bug since ℓ_C refers to the region outside the proxy. Instead, changing the code

in the proxy can only introduce another error or non-termination. Therefore, the labels assigned by Findler and Felleisen [2002]'s algorithm correctly identify the problematic part of the code.

$$\mathsf{proxy}([\kappa'],\ \lambda x.x)\ (\lambda y.\mathsf{suc}(y))\ 4$$
$$\text{where } \kappa' :\equiv \left(\mathsf{isEven}^{\ell_S} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_C}\right) \rightarrow\!\!/\mathsf{c}\ \left(\mathsf{isEven}^{\ell_C} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_S}\right)$$

Figure 7.4: The fixed program

In Figure 7.1, changing 5 to an even number like 4 makes it passes the two contracts, isEven^{ℓ_C} and isEven^{ℓ_S}. However, the entire program still fails the contract isOdd^{ℓ_C}, indicating that there is another bug in the region identified by ℓ_C. Now, consider the program in Figure 7.4 where, in addition to changing the number to 4, the function $\lambda y.y$ is also replaced by $\lambda y.\mathsf{suc}(y)$. This time, the error is finally fixed and the region represented by ℓ_C is no longer blamed.

To justify the intuition in Findler and Felleisen [2002]'s blame assignment algorithm, Dimoulas et al. [2011, 2012] introduce the idea of *ownership* of run-time values. In their work, each subexpression is assigned an owner that represents which region has "control" over how the value can be used. When a region owns a run-time value, it can affect how the value is produced through the code in the corresponding region much like how changes of the program in Figure 7.4 fixes the bug in Figure 7.2. Then, Dimoulas et al. [2011, 2012] show that when a region is blamed in the contract violation, that region must be the owner of the value which fails the contract. Therefore, since the owner of a value determines how the value is produced, contract blame effectively points to the region that contains the buggy code.

In this chapter, I illustrate how to instantiate my framework to capture blame and ownership, and how to develop a modular proof of their correctness. In my approach, blame and ownership are formalized using two separate annotation languages: one is the language $(\mathscr{A}bctc, \mathscr{T}b)$ that characterizes the results of Findler and Felleisen [2002]'s blame assignment algorithm; the other is the language $(\mathscr{A}owner, \mathscr{T}owner)$ that formulates a notion of ownership inspired by Dimoulas et al. [2011, 2012]'s theory. Then, I combine these two annotation languages to prove a version of the blame correctness theorem [Dimoulas et al. 2011, 2012].

7.1 The Blame Annotation Language

In this section, I introduce the blame annotation language, $(\mathscr{A}bctc, \mathscr{T}b)$, to encode Findler and Felleisen [2002]'s blame assignment algorithm in my framework. When a contract is attached to a value in Findler and Felleisen [2002]'s contract calculus, their algorithm couples the contract with two labels that individually identify the party that produces the value and the party that consumes it. The program from Figure 7.4 in page 99 is a concrete example of this assignment: the label ℓ_S referring to the region inside the proxy is the provider party of the contracted function $(\lambda x.x)$ whereas the label ℓ_C referring to the region outside the proxy is the consumer (i.e. the region that contains the code $\cdots\ (\lambda y.\mathsf{suc}(y))\ 4$).

To encode blame assignment, I extend the annotation language $\{(\mathscr{A}ctc, \mathscr{T}c_i)\}_{i\geq 0}$ for contracts from Chapter 6 with *blame objects*. Each object is a record consisting of two labels that name a *positive party* and a *negative party*. When a blame object is paired with a contract, these two parties correspond to the producer and the consumer, respectively. As a natural constraint, the labels of the blame objects shall match the output of Findler and Felleisen [2002]'s blame tracking algorithm, assuming that the predicates in the contracts have a consistent assignment of labels.

$$
\begin{aligned}
&\mathsf{proxy}([\langle b, \kappa'\rangle],\ \lambda x.x)\ \ (\lambda y.y)\ 5\\
&\quad \text{where}\ \ b :\equiv \{\mathsf{pos} = \ell_S;\ \mathsf{neg} = \ell_C\}\\
&\qquad\quad \kappa' :\equiv \left(\mathsf{isEven}^{\ell_S} \rightarrow\!\!/\mathsf{c}\ \mathsf{isOdd}^{\ell_C}\right) \rightarrow\!\!/\mathsf{c}\ \left(\mathsf{isEven}^{\ell_C} \rightarrow\!\!/\mathsf{c}\ \mathsf{any}/\mathsf{c}^{\ell_S}\right)
\end{aligned}
$$

Figure 7.5: Contracts with blame objects

Figure 7.5 illustrates the blame object extension using the program from Figure 7.2 where the contract on the proxy κ' is paired with the blame object b. In this example, the pos field of b contains ℓ_S, the label of the positive party. Similarly, the neg field of b contains ℓ_C, the label of the negative party. From Figure 7.2, the contracts isEven^{ℓ_S} and $\mathsf{any}/\mathsf{c}^{\ell_S}$ are labeled ℓ_S since failing these contracts indicates that the bug is the positive party—the region named ℓ_S—and *mutatis mutandis* for the contracts isOdd^{ℓ_C} and isEven^{ℓ_C}.

For the blame annotation language, when the labels on the predicates in a contract like κ' are assigned by Findler and Felleisen [2002]'s algorithm, the corresponding blame object b can be

$$
\begin{array}{rcl}
\mathscr{A}bctc & :\equiv & (A_{bctc}, s_{bctc}) \\
A_{bctc} & ::= & [\langle b_1, \kappa_1 \rangle, \ldots, \langle b_m, \kappa_m \rangle] \\
s_{bctc} \in \text{Status} & ::= & \textsc{Ok} \mid \textsc{Err}(\ell) \\
b & ::= & \{\text{pos} = \ell_p;\ \text{neg} = \ell_n\} \\
\mathit{blameSwap}(b) & :\equiv & \{\text{pos} = b.\text{neg};\ \text{neg} = b.\text{pos}\}
\end{array}
$$

Figure 7.6: The syntax of the blame annotation language, $\mathscr{A}bctc$.

seen as the witness of this fact. Later, I introduce the consistency judgement, $b \vdash \kappa'$ consistent, to formalize this idea in Figure 7.7. This intuition is further utilized to guide the design of the transition steps of the blame annotation language, ensuring that the consistency is maintained throughout the reduction.

Alternatively, blame objects can be used as the primary device for tracking the source of errors when reporting contract violations, unlike the reliance on labeled predicates in Chapter 6. Then, the transition steps of the blame annotation language would be defined in a way that propagates the blame objects in accordance with how run-time values flow across each region delimited by proxies and boundaries. This approach is conceptually closer to what Findler and Felleisen [2002] and Dimoulas et al. [2011, 2012] have done in their work.

Now, let me present the formal definition of the blame annotation language, $\mathscr{A}bctc$. The syntax of $\mathscr{A}bctc$ is given in Figure 7.6. An annotation of $\mathscr{A}bctc$ is a list of blame-contract pairs where a blame is a record with a pos field and a neg field storing labels. The records are created using the syntax, $\{\text{pos} = \ell_p;\ \text{neg} = \ell_n\}$, for any ℓ_p, ℓ_n and accessed using the usual dot syntax. Both the syntaxes of contracts and global states are the same as the $\mathscr{A}ctc$ annotation language; see Figure 6.1 in page 85 for their full grammar.

To ensure that a blame object matches a contract and that a contract has a consistent assignment of labels from Findler and Felleisen [2002]'s algorithm, I introduce the *consistency judgement*, $b \vdash \kappa$ consistent, in Figure 7.7. Most inference rules of this judgement directly follow the syntax of contracts, but there are two important ones. For one, when proving that a blame object b matches the label of a flat contract or, more formally, $b \vdash \text{flat}^{\ell}(x.\, e)$ consistent, the consistency

$$\boxed{b \vdash \kappa \text{ consistent}}$$

$$\frac{b.\text{pos} = \ell}{b \vdash \text{flat}^{\ell}(x.\, e) \text{ consistent}} \qquad \frac{\mathit{blameSwap}(b) \vdash \kappa_a \text{ consistent} \qquad b \vdash \kappa_r \text{ consistent}}{b \vdash (\kappa_a \to\!\!/\mathsf{c}\ \kappa_r) \text{ consistent}}$$

$$\frac{}{b \vdash t \text{ consistent}} \qquad \frac{}{b \vdash \text{unit}/\mathsf{c} \text{ consistent}} \qquad \frac{b \vdash \kappa_1 \text{ consistent} \qquad b \vdash \kappa_2 \text{ consistent}}{b \vdash (\kappa_1 \times\!\!/\mathsf{c}\ \kappa_2) \text{ consistent}}$$

$$\frac{b \vdash \kappa_1 \text{ consistent} \qquad b \vdash \kappa_2 \text{ consistent}}{b \vdash (\kappa_1 +\!\!/\mathsf{c}\ \kappa_2) \text{ consistent}} \qquad \frac{b \vdash \kappa \text{ consistent}}{b \vdash (\text{box}/\mathsf{c}\ \kappa) \text{ consistent}} \qquad \frac{b \vdash \kappa \text{ consistent}}{b \vdash (\mu/\mathsf{c}\ t.\kappa) \text{ consistent}}$$

Figure 7.7: Consistency of blame objects and the labels on contracts

judgement requires that the positive party of b matches the label of the contract, i.e. $b.\text{pos} = \ell$.

The reader may question why there are no rules that inspect the neg field. The answer is in the other important rule, the function contract rule. To prove that a blame object b matches a function contract, i.e. $b \vdash (\kappa_a \to\!\!/\mathsf{c}\ \kappa_r)$ consistent, the judgement checks the consistency of the two sub-contracts, κ_a and κ_r. But, when checking the consistency of κ_a, the judgement *swaps* the positive party and the negative party of b through $\mathit{blameSwap}(b)$ from Figure 7.6. Put differently, the role of provider and the role of consumer are reversed for function arguments.

Incidentally, the formulation of blame objects and the consistency judgement in this section is closer to the implementation of higher-order contracts in the Racket programming language [Felleisen et al. 2015, 2018] when compared to Findler and Felleisen [2002]'s contract calculus or Dimoulas et al. [2011, 2012]'s CPCF. Of course, Racket's blame object implementation is much more sophisticated and includes other details such as the context information in blame error messages for indicating the violated part of the failed contract. Nevertheless, the party-swapping operation is identical to Racket's implementation.

$$\frac{\dfrac{\dfrac{\overline{b.\text{pos} = \ell_S}}{b \vdash \text{isEven}^{\ell_S} \text{ consistent}} \qquad \dfrac{\overline{b_1.\text{pos} = \ell_C}}{b_1 \vdash \text{isOdd}^{\ell_C} \text{ consistent}}}{b_1 \vdash \left(\text{isEven}^{\ell_S} \to\!\!/\mathsf{c}\ \text{isOdd}^{\ell_C}\right) \text{ consistent}} \qquad \dfrac{\dfrac{\overline{b_1.\text{pos} = \ell_C}}{b_1 \vdash \text{isEven}^{\ell_C} \text{ consistent}} \qquad \dfrac{\overline{b.\text{pos} = \ell_S}}{b \vdash \text{any}/\mathsf{c}^{\ell_S} \text{ consistent}}}{b \vdash \left(\text{isEven}^{\ell_C} \to\!\!/\mathsf{c}\ \text{any}/\mathsf{c}^{\ell_S}\right) \text{ consistent}}}{b \vdash \left(\left(\text{isEven}^{\ell_S} \to\!\!/\mathsf{c}\ \text{isOdd}^{\ell_C}\right) \to\!\!/\mathsf{c}\ \left(\text{isEven}^{\ell_C} \to\!\!/\mathsf{c}\ \text{any}/\mathsf{c}^{\ell_S}\right)\right) \text{ consistent}}$$

where $b :\equiv \{\text{pos} = \ell_S;\ \text{neg} = \ell_C\}$
$b_1 :\equiv \mathit{blameSwap}(b) \equiv \{\text{pos} = \ell_C;\ \text{neg} = \ell_S\}$

Figure 7.8: The consistency of κ' from Figure 7.2.

As a running example of the consistency judgement, Figure 7.8 displays a proof of the consistency of the contract κ' from Figures 7.2 and 7.5. The blame object b that proves the consistency of κ' is the same one from Figure 7.5. In the proof, I let b' be the blame object obtained by swapping the two parties of b. As one would expect, the consistency of isOdd^{ℓ_C} and isEven^{ℓ_C} are proved by b'.

Having explained blame objects and the consistency of contracts, I shall return to the dynamics of $\mathscr{A}bctc$. The transition steps of $\mathscr{A}bctc$, or $\mathscr{T}b_i$ for $i \geq 0$, are augmented from $\mathscr{T}c_i$ in Figure 6.4, page 90 to account for the addition of blame objects to the annotations. Figure 7.9 lists the rules of $\mathscr{T}b_{i+1}$ for $i \geq 0$ and $\mathscr{T}b_0$ is the empty relation. The transition steps $\mathscr{T}b_i$ are stratified in the same manner as how $\mathscr{T}c_i$ does to allow arbitrary predicates to be used as contracts.

The blame objects do not take part in the enforcement of the contracts. In the [R-Cross-Nat] rule, the same $checkCtcs_{\mathscr{T}b_i}$ function from Figure 6.5 in page 92 is used to check the contracts $\kappa_1, \ldots, \kappa_m$. This time, the transition steps $\mathscr{T}b_i$ instead of $\mathscr{T}c_i$ are used to evaluate possibly nested contracts as highlighted in the subscript of *checkCtcs*. It is necessary to evaluate nested contracts with $\mathscr{T}b_i$ since their annotations also would contain blame objects. Otherwise, the [R-Cross-Nat] rule works in the same way as $\mathscr{T}c_i$ from Figure 6.4 in page 90.

As shown in Figure 7.9, the rules of $\mathscr{T}b_{i+1}$ for $i \geq 0$ propagate the blame objects together with the corresponding contracts. For example, in the [R-Cross-Cons] rule, the blame object b_i is paired with the contract $\kappa_i \mathbin{\not\!\!\times_{\mathsf{c}}} \kappa'_i$ for $1 \leq i \leq m$ on the left-hand side. Thus, b_i is paired with κ_i and κ'_i, respectively, on the right-hand side. Similarly, other rules except the [R-Proxy-β] rule propagate the blame objects following the same pattern.

In the [R-Proxy-β] rule, additional care is needed when propagating the blame objects. Specifically, the role of provider and the role of consumer are reversed for function arguments. Hence, when propagating the contracts and the blame objects to the boundary around the argument (i.e. $\mathbf{B}\#A_{bctc2}\,\{v\}$), not only the order of the contracts has to be reversed, but the blame objects also have to be flipped using $blameSwap(b_i)$ to match the reversal of the roles.

[R-Cross-Unit] $\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{\ ()\ \} \longrightarrow_{\text{m}} \text{Ok}, ()$

[R-Cross-Nat] $\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{\ n\ \} \longrightarrow_{\text{m}} s', n$
where $(\text{Ok}, s') \in \mathit{checkCtcs}_{\mathscr{T}b_i,\, get,\, put}([\kappa_1, \ldots, \kappa_m],\ n)$

[R-Cross-Cons] $\text{Ok}, \mathbf{B}\#[\langle b_1, (\kappa_1 \times\!/\text{c}\ \kappa_1')\rangle, \ldots, \langle b_m, (\kappa_m \times\!/\text{c}\ \kappa_m')\rangle]\ \{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\text{m}}$
$\text{Ok}, \langle\ \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{v_1\},\ \mathbf{B}\#[\langle b_1, \kappa_1'\rangle, \ldots, \langle b_m, \kappa_m'\rangle]\ \{v_2\}\ \rangle$

[R-Cross-Inl] $\text{Ok}, \mathbf{B}\#[\langle b_1, (\kappa_1 +\!/\text{c}\ \kappa_1')\rangle, \ldots, \langle b_m, (\kappa_m +\!/\text{c}\ \kappa_m')\rangle]\ \{\ \text{inl}(v)\ \} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{inl}(\mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{v\})$

[R-Cross-Inr] $\text{Ok}, \mathbf{B}\#[\langle b_1, (\kappa_1 +\!/\text{c}\ \kappa_1')\rangle, \ldots, \langle b_m, (\kappa_m +\!/\text{c}\ \kappa_m')\rangle]\ \{\ \text{inr}(v)\ \} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{inr}(\mathbf{B}\#[\langle b_1, \kappa_1'\rangle, \ldots, \langle b_m, \kappa_m'\rangle]\ \{v\})$

[R-Cross-Roll] $\text{Ok}, \mathbf{B}\#[\langle b_1, (\mu/\text{c}\ t_1.\kappa_1)\rangle, \ldots, \langle b_m, (\mu/\text{c}\ t_m.\kappa_m)\rangle]\ \{\ \text{roll}_\tau(v)\ \} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{roll}_\tau(\mathbf{B}\#A_{bctc}\ \{v\})$
where $A_{bctc} :\equiv [\langle b_1, \kappa_1[(\mu/\text{c}\ t_1.\kappa_1)\ /\ t_1]\rangle, \ldots, \langle b_m, \kappa_m[(\mu/\text{c}\ t_m.\kappa_m)\ /\ t_m]\rangle]$

[R-Cross-Box] $\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{\ \text{box}(v)\ \} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{proxy}([\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle],\ \text{box}(v))$

[R-Cross-Lam] $\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{\ \lambda x.e\ \} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{proxy}([\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle],\ \lambda x.e)$

[R-Proxy-Unbox] $\text{Ok}, \text{unbox}(\text{proxy}([\langle b_1, \text{box}/\text{c}\ \kappa_1\rangle, \ldots, \langle b_m, \text{box}/\text{c}\ \kappa_m\rangle],\ \text{box}(e))) \longrightarrow_{\text{m}}$
$\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\ \{\ \text{unbox}(\text{box}(e))\ \}$

[R-Proxy-β] $\text{Ok}, \text{proxy}\big([\langle b_1, (\kappa_1 \to\!/\text{c}\ \kappa_1')\rangle, \ldots, \langle b_m, (\kappa_m \to\!/\text{c}\ \kappa_m')\rangle],\ \lambda x.e\big)\ v \longrightarrow_{\text{m}}$
$\text{Ok}, \mathbf{B}\#A_{bctc1}\ \{\ (\lambda x.e)\ \ (\mathbf{B}\#A_{bctc2}\ \{v\})\ \}$
where $A_{bctc1} :\equiv [\langle b_1, \kappa_1'\rangle, \ldots, \langle b_m, \kappa_m'\rangle]$
$A_{bctc2} :\equiv [\langle \mathit{blameSwap}(b_m), \kappa_m\rangle, \ldots, \langle \mathit{blameSwap}(b_1), \kappa_1\rangle]$

[R-Merge-Box] $\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_l, \kappa_l\rangle]\ \big\{\ \text{proxy}\big([\langle b_1', \kappa_1'\rangle, \ldots, \langle b_m', \kappa_m'\rangle],\ \text{box}(e)\big)\ \big\} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{proxy}\big([\langle b_1', \kappa_1'\rangle, \ldots, \langle b_m', \kappa_m'\rangle, \langle b_1, \kappa_1\rangle, \ldots, \langle b_l, \kappa_l\rangle],\ \text{box}(e)\big)$

[R-Merge-Lam] $\text{Ok}, \mathbf{B}\#[\langle b_1, \kappa_1\rangle, \ldots, \langle b_l, \kappa_l\rangle]\ \big\{\ \text{proxy}\big([\langle b_1', \kappa_1'\rangle, \ldots, \langle b_m', \kappa_m'\rangle],\ \lambda x.e\big)\ \big\} \longrightarrow_{\text{m}}$
$\text{Ok}, \text{proxy}\big([\langle b_1', \kappa_1'\rangle, \ldots, \langle b_m', \kappa_m'\rangle, \langle b_1, \kappa_1\rangle, \ldots, \langle b_l, \kappa_l\rangle],\ \lambda x.e\big)$

This figure defines the inductive case of the transition steps $\mathscr{T}b_{i+1}$ of $\mathscr{A}bctc$ for $i \geq 0$. The base case $\mathscr{T}b_0$ is the empty relation.

Figure 7.9: The transition relation of $\mathscr{A}bctc$.

7.2 Blame Consistency

In this section, I prove that the transition steps $\mathcal{T}b_{i+1}$ from Figure 7.9, page 104 preserve the consistency judgement for all $i \geq 0$. The implications of this property are two folds. First, it justifies how $\mathcal{T}b_{i+1}$ distributes the blame objects on top of the original, unchanged rules for propagating contracts. This includes the exchange of the positive party and the negative party in the [R-Proxy-β] rule. Second, the preservation property conversely justify the definition of the consistency judgement itself. That is, contract violations reported using the static assignment of labels via the consistency judgement coincide with the outputs of Findler and Felleisen [2002]'s blame tracking algorithm. This is especially important since the judgement is inspired by, but not identical to, Findler and Felleisen [2002]'s algorithm.

Both implications are reenforced when I prove the single-owner property in Section 7.4 and connect it to blame objects in Section 7.6. The separate justification follows from the fact that the ownership annotation language is designed to directly track the name of each region delimited by proxies and boundaries, independent of blame and contracts. Hence, that blame and contracts are compatible with ownership better certifies their correctness.

The preservation of the consistency of recursive contracts, however, is rather subtle. Recursive contracts that describe first-order types indeed preserve the consistency judgement, but this is not always the case when higher-order functions are involved. This subtlety can be more easily recognized by considering how contracted values flow between boundaries.

To be more concrete, when a value v of a first-order recursive type crosses a boundary, all values that v contains also cross the boundary along the same direction. To the contrary, when a higher-order function f crosses a boundary, values (e.g. arguments of function applications) can flow backwards into the boundary. As a consequence, recursive contracts that involve higher-order functions do not necessarily blame a fixed owner.

To warm up the discussion, consider the contract *EvenList* that describes even-number lists.

$$\mathit{EvenList} :\equiv \mu/\mathsf{c}\ t.\mathsf{unit}/\mathsf{c} +/\mathsf{c} \left(\mathsf{isEven}^{\ell} \times/\mathsf{c}\ t\right)$$

When a list protected by *EvenList* crosses a boundary, all numbers in it also cross the same boundary in the same direction. Consequently, a single label ℓ on the predicate isEven can correctly designate the owner of the numbers to be blamed. This observation can be made precise by checking that unrolling *EvenList* gives $\mathsf{unit}/\!\!\mathsf{c} +\!\!/\mathsf{c} \left(\mathsf{isEven}^{\ell} \times\!\!/\mathsf{c}\ \mathit{EvenList}\right)$ in which the occurrence of *EvenList* has the same owner as isEven^{ℓ}.

In this example, if isEven^{ℓ} and t in *EvenList* were not owned by the same region, consistency would break after unrolling *EvenList*. Unfortunately, t could actually occur in a contravariant position in the presence of function contracts. This suggests a direction for constructing contracts and programs that fail to preserve the consistency judgement. To see a concrete witness of the contravariance issue, consider the program in Figure 7.10.

$$
\begin{aligned}
&\mathsf{unroll}(\mathbf{B}\#[\langle b, \mathit{RecFn}\rangle]\ \{\ \mathsf{roll}_{\tau_t}(f)\ \})\ \mathit{one} \\
&\quad \text{where}\quad \mathit{RecFn} &&:\equiv\ \mu/\!\!\mathsf{c}\, t.(t \to\!\!/\mathsf{c}\ \mathsf{isEven}^{\ell_S}) \\
&\qquad\qquad b &&:\equiv\ \{\mathsf{pos} = \ell_S;\ \mathsf{neg} = \ell_C\} \\
&\qquad\qquad \tau_t &&:\equiv\ t \to \mathsf{nat} \\
&\qquad\qquad f &&:\equiv\ \lambda h.\ \mathsf{unroll}(h)\ (\mathsf{roll}_{\tau_t}(\lambda\_.0)) \\
&\qquad\qquad \mathit{one} &&:\equiv\ \mathsf{roll}_{\tau_t}(\lambda g.1)
\end{aligned}
$$

Figure 7.10: An example recursive contract that blames the wrong party.

In Figure 7.10, the program is morally the same as $(\mathbf{B}\#[\langle b, \mathit{RecFn}\rangle]\ \{\ \lambda h.\ h\ (\lambda\_.0)\ \})\ (\lambda g.1)$ except that the contracted function $\lambda h.\ h\ (\lambda\_.0)$ is encapsulated in the recursive type $\mu t.(t \to \mathsf{nat})$ and hence there are extra roll/unroll wrappers. The recursive contract *RecFn* on $\lambda h.\ h\ (\lambda\_.0)$ unrolls to $\mathit{RecFn} \to\!\!/\mathsf{c}\ \mathsf{isEven}^{\ell_S}$, thus it asserts that both $\lambda h.\ h\ (\lambda\_.0)$ and its argument $\lambda g.1$ should produce even numbers. However, the actual argument $\lambda g.1$ violates this contract.

Intuitively, when $\lambda g.1$ breaks the contract, the blame should point to the region containing it to highlight the problematic code. The blame object b suggests that contract violations from $\lambda g.1$ should blame ℓ_C. However, the contract on $\lambda g.1$ is obtained by unrolling *RecFn* and taking the domain contract which is again *RecFn* itself. The blame is thus wrong and, as a corollary, we have found an example where the supposedly consistent contract *RecFn* unrolls to an inconsistent contract $\mathit{RecFn} \to\!\!/\mathsf{c}\ \mathsf{isEven}^{\ell_S}$.

The actual problem of *RecFn* is that all of its unfolding use the label ℓ_S. However, when un-

folding an odd number of times, the label ℓ_C should have been used instead. This is inevitable, nonetheless, since there is only one available label on the predicate isEven^{ℓ_S} in *RecFn*. It is therefore not possible to statically assign labels for *RecFn* while respecting blame correctness.

$$
\begin{array}{ll}
\textsc{Ok}, & \mathsf{unroll}(\mathbf{B}\#[\langle b, \mathit{RecFn}\rangle]\ \{\ \mathsf{roll}_{\tau_t}(f)\ \})\ \mathit{one} \longrightarrow^* \\
\textsc{Ok}, & \mathsf{proxy}\big([\langle b, \mathit{RecFn} \rightarrow\!\!\!/c\ \mathsf{isEven}^{\ell_S}\rangle],\ f\big)\ \mathit{one} \longrightarrow \\
\textsc{Ok}, & \mathbf{B}\#[\langle b, \mathsf{isEven}^{\ell_S}\rangle]\ \{\ f\ (\mathbf{B}\#[\langle b_1, \mathit{RecFn}\rangle]\ \{\ \mathit{one}\ \})\ \} \longrightarrow^* \\
\textsc{Ok}, & \mathbf{B}\#[\langle b, \mathsf{isEven}^{\ell_S}\rangle]\ \{\ f\ (\mathsf{proxy}([\langle b_1, \mathit{RecFn} \rightarrow\!\!\!/c\ \mathsf{isEven}^{\ell_S}\rangle],\ \lambda g.1))\ \} \longrightarrow \\
\textsc{Ok}, & \mathbf{B}\#[\langle b, \mathsf{isEven}^{\ell_S}\rangle]\ \{\ \mathsf{proxy}([\langle b_1, \mathit{RecFn} \rightarrow\!\!\!/c\ \mathsf{isEven}^{\ell_S}\rangle],\ \lambda g.1)\ (\mathsf{roll}_{\tau_t}(\lambda\_.0))\ \} \longrightarrow \\
\textsc{Ok}, & \mathbf{B}\#[\langle b, \mathsf{isEven}^{\ell_S}\rangle]\ \{\ \mathbf{B}\#[\langle b_1, \mathsf{isEven}^{\ell_S}\rangle]\ \{\ (\lambda g.1)\ (\mathbf{B}\#[\langle b, \mathit{RecFn}\rangle]\ \{\ \mathsf{roll}_{\tau_t}(\lambda\_.0)\ \})\ \}\ \} \longrightarrow^* \\
\textsc{Err}(\ell_S), & \mathbf{B}\#[\langle b, \mathsf{isEven}^{\ell_S}\rangle]\ \{\ 1\ \}
\end{array}
$$

where $b :\equiv \{\mathsf{pos} = \ell_S;\ \mathsf{neg} = \ell_C\}$ and $b_1 :\equiv \mathit{blameSwap}(b)$.

Figure 7.11: The reduction sequence of the program from Figure 7.10.

Figure 7.11 displays the detailed reduction sequence. In the figure, the reduction steps unroll *RecFn* on f, giving the contract $\mathit{RecFn} \rightarrow\!\!\!/c\ \mathsf{isEven}^{\ell_S}$. Unfortunately, invoking this function requires the argument, *one*, to satisfy the contract *RecFn*. This means that when *one* violates the contract as the subsequent reduction steps show, the party ℓ_S be blamed. However, ℓ_S is the *callee* of this function application and yet *one* comes from the *caller*, ℓ_C. In other words, the reduction sequence in Figure 7.11 blames the wrong party.

To restore blame correctness, I impose an additional restriction on recursive contracts to avoid this situation. Figure 7.12 defines the *covariant judgement*, $\delta \vdash^p \kappa$ signed, that formally restricts the shape of recursive contracts. When $\delta \vdash^p \kappa$ signed holds, all variables bound by recursive contracts only occur in covariant position with respect to their binders. Contracts like *RecFn* are thus excluded from my formalization of contracts.

Contracts that satisfy the covariance judgement preserve the consistency judgement during reductions. Therefore, in the proof of the preservation, the covariance judgement is added as another invariant on top of the consistency judgement. I shall formally state and prove this property in Proposition 7.8.

The metavariable p, or the *sign* (of a contract), can be either + or −. If the sign of a contract is positive (+), it means that κ resides in a covariant position. Otherwise, it means that κ

$$p \quad ::= \quad + \mid -$$

$$(-p) :\equiv \begin{cases} -, & \text{if } p = + \\ +, & \text{if } p = - \end{cases}$$

$$\mathit{signedBlame}(+,\ b) :\equiv b$$
$$\mathit{signedBlame}(-,\ b) :\equiv \mathit{blameSwap}(b)$$

$$\boxed{\delta :\equiv \{t_1 : p_1, \ldots, t_m : p_m\}} \quad \boxed{\delta \vdash^p \kappa \text{ signed}}$$

$$\frac{\delta(t) = p}{\delta \vdash^p t \text{ signed}} \qquad \frac{\delta, t : p \vdash^p \kappa \text{ signed}}{\delta \vdash^p (\mu/\mathsf{c}\, t.\kappa) \text{ signed}} \qquad \frac{\delta \vdash^{-p} \kappa_a \text{ signed} \qquad \delta \vdash^p \kappa_r \text{ signed}}{\delta \vdash^p (\kappa_a \rightarrow\!/\mathsf{c}\, \kappa_r) \text{ signed}}$$

$$\frac{}{\delta \vdash^p \mathsf{unit}/\mathsf{c} \text{ signed}} \qquad \frac{}{\delta \vdash^p \mathsf{flat}^\ell(x.\, e) \text{ signed}} \qquad \frac{\delta \vdash^p \kappa_1 \text{ signed} \qquad \delta \vdash^p \kappa_2 \text{ signed}}{\delta \vdash^p (\kappa_1 \times\!/\mathsf{c}\, \kappa_2) \text{ signed}}$$

$$\frac{\delta \vdash^p \kappa_1 \text{ signed} \qquad \delta \vdash^p \kappa_2 \text{ signed}}{\delta \vdash^p (\kappa_1 +\!/\mathsf{c}\, \kappa_2) \text{ signed}} \qquad \frac{\delta \vdash^p \kappa \text{ signed}}{\delta \vdash^p (\mathsf{box}/\mathsf{c}\, \kappa) \text{ signed}}$$

Figure 7.12: Covariance of recursive contracts

resides in a contravariant position. The context δ tracks the sign of each bound variable. The $\mathit{signedBlame}(p,\ b)$ metafunction flips the given blame object b iff p is negative, just like how a blame object should be flipped when it is used in the domain of a function contract.

In the covariance judgement, $\delta \vdash^p (\mu/\mathsf{c}\, t.\kappa)$ signed holds when the judgement holds for the contract κ under the extended context $\delta, t : p$. This context records that t is introduced under the sign p. To prove $\delta \vdash^p t$ signed, the covariance judgement looks up the context δ to make sure that t is used in a position that has the same sign as its binder. Finally, to prove $\delta \vdash^p (\kappa_a \rightarrow\!/\mathsf{c}\, \kappa_r)$ signed, the covariant judgement checks the two sub-contracts in the premises. Most importantly, the first premise, $\delta \vdash^{-p} \kappa_a$ signed, negates p to indicate that κ_a is covariant with respect to the contract $\kappa_a \rightarrow\!/\mathsf{c}\, \kappa_r$.

The covariance judgement possesses the conventional Substitution property as proved in Lemmas 7.2 and 7.4 and Proposition 7.3. Most theorem statements are standard except that the renaming of the variables in Lemma 7.2 must preserve the signs as required by the condition $p_i = p'_{a_i}$. Similarly, in Lemma 7.4 the substituted contracts κ_i must satisfy the covariant judgement under the respective sign $(t_i : p_i) \in \delta$.

Proposition 7.1. *If* $\delta \vdash^{p} \kappa$ signed *then* $-\delta \vdash^{-p} \kappa$ signed *where* $(-\delta)(x) :\equiv -\delta(x)$.

Lemma 7.2 (Renaming)**.** *Let* $\delta :\equiv \{t_1 : p_1, \ldots, t_n : p_n\}$ *and* $\delta' :\equiv \{t'_1 : p'_1, \ldots, t'_m : p'_m\}$ *be given. Assume that there is a sequence* $1 \leq a_i \leq m$ *such that* $p_i = p'_{a_i}$ *for* $i = 1 \ldots n$*. If* $\delta \vdash^{p} \kappa$ signed *then* $\delta' \vdash^{p} \kappa\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right]$ signed.

Proposition 7.3. *Let* $\delta' \vdash^{p_i} \kappa_i$ signed *for* $i = 1 \ldots n$ *be given. Then, for any fresh variable* $t_0 : p_0$*, there is a sequence* $\delta'' \vdash^{p_i} \kappa_i$ signed *for* $i = 0 \ldots n$ *where* $\delta'' :\equiv \delta, t_0 : p_0$ *and* $\kappa_0 :\equiv t_0$.

Lemma 7.4 (Substitution)**.** *Let* $\delta :\equiv \{t_1 : p_1, \ldots, t_n : p_n\}$ *be given. If* $\delta' \vdash^{p_i} \kappa_i$ signed *for* $1 \leq i \leq n$ *and* $\delta \vdash^{p} \kappa$ signed *then* $\delta' \vdash^{p} \kappa[\kappa_1, \ldots, \kappa_n \,/\, t_1, \ldots, t_n]$ signed.

The covariance judgement $\delta \vdash^{p} \kappa$ signed enables me to prove the Substitution lemma through a series of auxiliary results, Lemmas 7.5 and 7.7 and Proposition 7.6. In Lemma 7.7, all contracts κ and $\kappa_1, \ldots, \kappa_n$ shall be covariant. Moreover, the consistency of the contracts $\kappa_1, \ldots, \kappa_n$ shall be proven using the appropriately swapped blame, $\mathit{signedBlame}(p_i,\, b)$ for $i = 1 \ldots n$, in accordance with the sign of the substituted variables. With these additional conditions, one can prove that the result of the substitution stays consistent.

Lemma 7.5 (Renaming)**.** *Let* $\Delta :\equiv \{t_1, \ldots, t_n\}$ *and* $\Delta' :\equiv \{t'_1, \ldots, t'_m\}$ *be given. Assume that there is a sequence* $1 \leq a_i \leq m$ *for* $i = 1 \ldots n$*. If* $b \vdash \kappa$ consistent *where* $\Delta \vdash^{c} \kappa : \mathrm{Ctc}\,\tau$ *then* $b \vdash \kappa\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right]$ consistent.

Proposition 7.6. *Let* $\mathit{signedBlame}(p_i,\, b) \vdash \kappa_i$ consistent *for* $i = 1 \ldots n$ *be given. For any fresh variable* $t_0 : p_0$*, there is a sequence* $\mathit{signedBlame}(p_i,\, b) \vdash \kappa_i$ consistent *for* $i = 0 \ldots n$ *where* $\kappa_0 :\equiv t_0$.

Lemma 7.7 (Substitution)**.** *Let* $\delta :\equiv \{t_1 : p_1, \ldots, t_n : p_n\}$ *be given. Assume that* $\delta \vdash^{p} \kappa$ signed *and* $\mathit{signedBlame}(p,\, b) \vdash \kappa$ consistent*. If* $\delta' \vdash^{p_i} \kappa_i$ signed *and* $\mathit{signedBlame}(p_i,\, b) \vdash \kappa_i$ consistent *for* $i = 1 \ldots n$ *then* $\mathit{signedBlame}(p,\, b) \vdash \kappa[\kappa_1, \ldots, \kappa_n \,/\, t_1, \ldots, t_n]$ consistent.

Having proved the Substitution lemma, I can now show that consistent recursive contracts stay consistent during evaluation. Since the core is the substitution of recursive contracts as the [R-Cross-Roll] rule from Figure 7.9 in page 104 defines, Proposition 7.8 states that when

unrolling a consistent recursive contract $\mu/\mathrm{c}\, t.\kappa$, the result $\kappa[(\mu/\mathrm{c}\, t.\kappa)\ /\ t]$ is still consistent as long as the variable t occurs only in covariant positions in κ, i.e. $\delta \vdash^{p} \kappa$ signed holds.

Proposition 7.8. *If* $\delta \vdash^{p} \kappa$ signed *and* $\mathit{signedBlame}(p,\ b) \vdash (\mu/\mathrm{c}\, t.\kappa)$ consistent *both hold then we have* $\mathit{signedBlame}(p,\ b) \vdash (\kappa[(\mu/\mathrm{c}\, t.\kappa)\ /\ t])$ consistent.

The proof of the preservation starts with Definition 7.9 which defines an annotation interpretation mapping boundary terms and proxy terms to the covariant judgements and the consistency judgements. Theorem 7.10 further shows that the annotation interpretation $\mathcal{I}\mathit{con}_i$ is sound. In the framework of the monitor calculus, it then follows from Theorem 5.19 that the annotation language $\mathscr{A}\mathit{bctc}$ preserves consistency.

Definition 7.9. The annotation interpretation $\mathcal{I}\mathit{con}_i :\equiv (\mathbb{S}_{con},\ \preccurlyeq_{con},\ \mathbb{B}_i,\ \mathbb{P}_i)$ is defined as

$$
\begin{aligned}
&\mathbb{S}_{con}(s) && :\equiv \quad \top \\
&s \preccurlyeq_{con} s' && :\equiv \quad \top \\
&\mathbb{B}_0[\![\ A_{bctc}, e\ ]\!] && :\equiv \quad \bot \\
&\mathbb{B}_{i+1}[\![\ [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle], e\ ]\!] && :\equiv \\
&\qquad (\vdash^{+} \kappa_1 \text{ signed}) \ \wedge\ \ldots\ \wedge\ (\vdash^{+} \kappa_m \text{ signed}) \wedge \\
&\qquad (b_1 \vdash \kappa_1 \text{ consistent}) \wedge \cdots \wedge (b_m \vdash \kappa_m \text{ consistent}) \wedge \\
&\qquad (\mathcal{I}\mathit{con}_i \vDash \kappa_1)\ \wedge\ \ldots\ \wedge\ (\mathcal{I}\mathit{con}_i \vDash \kappa_m) \\
&\mathbb{P}_i[\![\ A_{bctc}, e^m\ ]\!] && :\equiv \quad \mathbb{B}_i[\![\ A_{bctc}, e^m\ ]\!]
\end{aligned}
$$

For $i \geq 0$, the $\mathbb{B}_{i+1}[\![\ A_{bctc}, e\ ]\!]$ function maps boundaries to the assertion that the annotated contracts all satisfy the covariant judgement and that their labels are all consistent with respect to the blame objects, giving the preservation of the consistency judgement and its prerequisites a precise formulation. Formally, for any annotation $A_{bctc} :\equiv [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]$, the $\mathbb{B}_{i+1}[\![\ A_{bctc}, e\ ]\!]$ function asserts that $\vdash^{+} \kappa_j$ signed holds for all $1 \leq j \leq n$, i.e. all κ_j only uses recursive contracts (if any) covariantly, and that each blame object b_i proves $b_i \vdash \kappa_j$ consistent for each $1 \leq j \leq n$. Here, the + sign in $\vdash^{+} \kappa_j$ signed does not matter since Proposition 7.1 guarantees that $\vdash^{+} \kappa_j$ signed holds if and only if $\vdash^{-} \kappa_j$ signed holds.

In addition to the covariant judgement and the consistency judgement, the $\mathbb{B}_{i+1}[\![\ A_{bctc}, e\ ]\!]$ function requires that for all $1 \leq j \leq n$, the contract κ_j recursively satisfies the interpretation

$\mathcal{I}con_i :\equiv (\mathbb{S}_{con}, \preccurlyeq_{con}, \mathbb{B}_i, \mathbb{P}_i)$ through the satisfaction relation for contracts, $\mathcal{I}con_i \vDash \kappa_j$, from Section 6.3. This ensures that other contracts nested in the flat predicates inside κ_j also satisfy the covariant and the consistency judgements.

Theorem 7.10. *The annotation interpretation $\mathcal{I}con_i$ is monotonic and sound for all $i \geq 0$.*

Proof Idea. Recall from the definition of Sound Interpretation in page 75 that the soundness proof of $\mathcal{I}con_{i+1}$ essentially means proving that $\mathcal{I}con_{i+1} \vDash e_1$ implies $\mathcal{I}con_{i+1} \vDash e_2$ whenever $\mathcal{T}bctc_{i+1} \vdash s_1, e_1 \longrightarrow_{\mathrm{m}} s_2, e_2$. In the case of the [R-Cross-Roll] rule, one needs to show that

$$\mathbb{B}_{i+1}[\![\ [\langle b_1, (\mu/\mathrm{c}\, t_1.\kappa_1)\rangle, \ldots, \langle b_m, (\mu/\mathrm{c}\, t_m.\kappa_m)\rangle], \mathsf{roll}_\tau(v)\ ]\!]$$

implies

$$\mathbb{B}_{i+1}[\![\ [\langle b_1, \kappa_1[(\mu/\mathrm{c}\, t_1.\kappa_1)\ /\ t_1]\rangle, \ldots, \langle b_m, \kappa_m[(\mu/\mathrm{c}\, t_m.\kappa_m)\ /\ t_m]\rangle], v\ ]\!],$$

but this can be easily established using Proposition 7.8. □

7.3 The Ownership Annotation Language

The correctness criteria of blame can be further refined through Dimoulas et al. [2011, 2012]'s theory of ownership. Their theory annotates expressions with owners to help track the parties that actually have control over the run-time value. Then, Dimoulas et al. [2011, 2012] study whether a contract calculus always blame a party that owns the value failing the contract.

In this section, I take inspiration from their theory and design the ownership annotation language, ($\mathscr{A}owner$, $\mathscr{T}owner$) The $\mathscr{A}owner$ language stands by itself and is orthogonal to contracts and blame from Section 7.1; it formalizes a notion of ownership in the monitor calculus using annotations. Specifically, $\mathscr{A}owner$ assigns labels to regions delimited by proxies and boundaries where the term in each distinct region serves as the source code that originates from a separate file. In effect, $\mathscr{A}owner$ utilizes the design of proxies and boundaries to encode a representation of "source files" in the syntax.

In Section 7.4, I investigate the correctness of the $\mathscr{A}owner$ language by showing that correct

$$
\begin{array}{rcl}
\mathscr{A}\mathit{owner} & :\equiv & (A_{own}, s_{own}) \\
A_{own} & ::= & \langle \ell_n, \ell_p \rangle \\
s_{own} & ::= & () \\
\ell_n, \ell_p & \in & \mathsf{Label}
\end{array}
$$

Figure 7.13: The syntax of the ownership annotation language.

annotations can be interpreted as owners of delimited regions and that the transition steps of $\mathscr{A}\mathit{owner}$ preserve correct annotations. Later, in Section 7.6, I combine the $\mathscr{A}\mathit{owner}$ language in this section with blame objects from the $\mathscr{A}\mathit{bctc}$ language in Section 7.1 to prove a version of the Blame Correctness theorem [Dimoulas et al. 2011, 2012] using my framework.

Figure 7.13 shows the syntax of the ownership annotation language. In this language, the global state is just unit ($()$) and the annotations are pairs of labels. When a pair $\langle \ell_n, \ell_p \rangle$ is annotated on a proxy, it means that the *owner* of the region outside the proxy is ℓ_n and the *owner* of the region inside the proxy is ℓ_p. In other words, the labels in the pair designate the owners of the two regions delimited by the proxy. Similarly, when a pair of labels is annotated on a boundary, it names the two regions delimited by the boundary. Taking the program from Figure 7.2 in page 98 as an example, its ownership annotation is $\langle \ell_C, \ell_S \rangle$ since the region outside the proxy is named ℓ_C whereas the region inside the proxy is named ℓ_S:

$$\mathsf{proxy}(\langle \ell_C, \ell_S \rangle,\ \lambda x.x)\ (\lambda y.y)\ 5$$

From this example, it can be seen that the formulation of ownership in $\mathscr{A}\mathit{owner}$ is a little different from Dimoulas et al. [2011, 2012]'s presentation. In $\mathscr{A}\mathit{owner}$, the labels of the owners are represented using annotations, and an owner must include an entire piece of regions delimited by proxies and boundaries. To the contrary, Dimoulas et al. [2011, 2012]'s CPCF has a separate construct, $|e|^{\ell}$, to specify the owner of an arbitrary expression. Their construct is more flexible and can give any sub-expression an owner whereas $\mathscr{A}\mathit{owner}$ only tracks the owner of each region.

$$
\begin{array}{l}
(), \mathsf{proxy}(\langle \ell_C, \ell_S \rangle,\ \lambda x.x)\ (\lambda y.y)\ 5 \longrightarrow \\
(), (\mathbf{B}\#\langle \ell_C, \ell_S \rangle\ \{\ (\lambda x.x)\ (\mathbf{B}\#\langle \ell_S, \ell_C \rangle\ \{\ \lambda y.y\ \})\ \})\ 5 \longrightarrow^* \\
(), (\mathbf{B}\#\langle \ell_C, \ell_S \rangle\ \{\ \mathsf{proxy}(\langle \ell_S, \ell_C \rangle,\ \lambda y.y)\ \})\ 5 \longrightarrow \\
(), \mathsf{proxy}(\langle \ell_C, \ell_C \rangle,\ \lambda y.y)\ 5 \longrightarrow^* (), 5
\end{array}
$$

Figure 7.14: The reduction sequence of the program adapted from Figure 7.2.

To give a taste of how ownership annotations work in *Λowner*, Figure 7.14 depicts the reduction sequence of the program from Figure 7.2 where its annotation is appropriately adapted to ownership labels. After one step, $\lambda y.y$ flows through the $\langle \ell_C, \ell_S \rangle$ boundary and gets protected by another boundary to separate it from the enclosing function application. Thus, the new ownership labels on the inner boundary is $\langle \ell_S, \ell_C \rangle$ since $\lambda y.y$ inside the inner boundary originates from the region owned by ℓ_C and $\lambda x.x$ next to the inner boundary belongs to the region owned by ℓ_S.

The second line in Figure 7.14 suggests how to read off the annotations in *Λowner* to recognize the owner of each region. Specifically, the labels $\langle \ell_C, \ell_S \rangle$ on the outer boundary indicates that the outermost region is named ℓ_C and the region between the two boundaries is named ℓ_S. Similarly, the labels $\langle \ell_S, \ell_C \rangle$ on the inner boundary indicates that the innermost region, one which contains $\lambda y.y$, is owned by ℓ_C. After coloring the region named ℓ_C in brown and the region named ℓ_S in blue, the program looks like:

$$(\mathbf{B}\#\langle \ell_C, \ell_S \rangle \{ \ (\lambda x.x) \ (\mathbf{B}\#\langle \ell_S, \ell_C \rangle \{ \lambda y.y \}) \ \}) \ 5$$

For a program to have a sensible designation of owners, the labels on nested proxies and nested boundaries must match the labels annotated on their enclosing context. In the preceding example, the outer boundary has the annotation $\langle \ell_C, \ell_S \rangle$ which designates ℓ_S as the owner of the middle region that contains $\lambda x.x$. The inner boundary also designates ℓ_S as the owner of the same region with the annotation $\langle \ell_S, \ell_C \rangle$. Thus, these annotations give a consistent designation of owners.

The colorings only exist when nested proxies and boundaries have matching labels. If, to the contrary, the labels on the enclosing context of $\lambda x.x$ are not the same as the labels on the nested proxies or boundaries, there is no way to read off the owner that controls $\lambda x.x$ from the labels.

For example, if the labels on the innermost boundary in the preceding program is $\langle \ell_P, \ell_Q \rangle$ instead, both ℓ_S from the enclosing context and ℓ_P from the inner boundary would be the owner of $\lambda x.x$, conflicting with the requirement that one region has exactly one owner. The following program more concretely colors the problematic labels and the region they identified in red:

$$\big(\mathbf{B}\#\langle \ell_C, \ell_S \rangle \{ \ (\lambda x.x) \ \big(\mathbf{B}\#\langle \ell_P, \ell_Q \rangle \{ \lambda y.y \}\big) \ \}\big) \ 5$$

The consistent designation of owners is the manifestation of Dimoulas et al. [2011, 2012]'s *single-owner property* in my framework. The reduction rules of $\mathscr{A}owner$ are designed using this connection between the ownership labels and the owners of the delimited regions to ensure that sub-expressions owned by a region continue to be owned by the same region during reductions.

To complete the introduction to the ownership annotation language, I shall go over $\mathscr{T}owner$ from Figure 7.15, the transition steps of $\mathscr{A}owner$, to explain how ownership labels propagate in the example from Figure 7.14. The formal rules of $\mathscr{T}owner$ govern how the ownership labels are maintained when naming the new regions that might appear when the reduction step creates new boundaries or proxies.

$$
\begin{array}{rl}
\text{[R-Cross-Unit]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ () \ \} \longrightarrow_{\mathrm{m}} (), () \\
\text{[R-Cross-Nat]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ n \ \} \longrightarrow_{\mathrm{m}} (), n \\
\text{[R-Cross-Cons]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \langle v_1, v_2 \rangle \ \} \longrightarrow_{\mathrm{m}} (), \langle\ \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{v_1\}, \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{v_2\}\ \rangle \\
\text{[R-Cross-Inl]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \mathsf{inl}(v) \ \} \longrightarrow_{\mathrm{m}} (), \mathsf{inl}(\mathbf{B}\#\langle \ell_n, \ell_p \rangle \{v\}) \\
\text{[R-Cross-Inr]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \mathsf{inr}(v) \ \} \longrightarrow_{\mathrm{m}} (), \mathsf{inr}(\mathbf{B}\#\langle \ell_n, \ell_p \rangle \{v\}) \\
\text{[R-Cross-Roll]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \mathsf{roll}_\tau(v) \ \} \longrightarrow_{\mathrm{m}} (), \mathsf{roll}_\tau(\mathbf{B}\#\langle \ell_n, \ell_p \rangle \{v\}) \\
\text{[R-Cross-Box]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \mathsf{box}(v) \ \} \longrightarrow_{\mathrm{m}} (), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \mathsf{box}(v)\big) \\
\text{[R-Cross-Lam]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \lambda x.e \ \} \longrightarrow_{\mathrm{m}} (), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.e\big) \\
\text{[R-Proxy-Unbox]} & (), \mathsf{unbox}(\mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \mathsf{box}(e)\big)) \longrightarrow_{\mathrm{m}} \\
& (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \{\ \mathsf{unbox}(\mathsf{box}(e)) \ \} \\
\text{[R-Proxy-}\beta\text{]} & (), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.e\big)\ v \longrightarrow_{\mathrm{m}} \\
& (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{\ (\lambda x.e)\ \left(\mathbf{B}\#\langle \ell_p, \ell_n \rangle \{v\}\right)\ \right\} \\
\text{[R-Merge-Box]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{ \mathsf{proxy}\Big(\langle \ell'_n, \ell'_p \rangle,\ \mathsf{box}(e)\Big) \right\} \longrightarrow_{\mathrm{m}} \\
& (), \mathsf{proxy}\Big(\langle \ell_n, \ell'_p \rangle,\ \mathsf{box}(e)\Big) \\
& \text{if} \quad \ell_p = \ell'_n \\
\text{[R-Merge-Lam]} & (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{ \mathsf{proxy}\Big(\langle \ell'_n, \ell'_p \rangle,\ \lambda x.e\Big) \right\} \longrightarrow_{\mathrm{m}} (), \mathsf{proxy}\Big(\langle \ell_n, \ell'_p \rangle,\ \lambda x.e\Big) \\
& \text{if} \quad \ell_p = \ell'_n
\end{array}
$$

Figure 7.15: The transition relation of $\mathscr{A}owner$, $\mathscr{T}owner$.

Taking the [R-Cross-Cons] rule as an example, it pushes the boundary around $\langle v_1, v_2 \rangle$ inside

and creates two separate boundaries around v_1 and v_2. Note that $\langle v_1, v_2 \rangle$ and its two components are all owned by ℓ_p since the annotation on the boundary is $\langle \ell_n, \ell_p \rangle$. Consequently, the labels on the new boundaries around v_1 and v_2 should also be $\langle \ell_n, \ell_p \rangle$ to designate ℓ_p as their owner.

The same logic applies to all other rules in Figure 7.15. The [R-Proxy-β] rule is another notable example where there are three regions on the right-hand side of the term. In this rule, the proxy annotated with $\langle \ell_n, \ell_p \rangle$ is applied to v. After one step, the argument v flows from the region ℓ_n into the region ℓ_p. The two new boundaries divide the entire term in three regions: the region outside the whole term, the region between the two boundaries, and the region inside the innermost boundary. Using Felleisen and Friedman [1987]'s notation of program contexts, the middle region refers to the part that contains the code $(\lambda x.e)$ []. Since v shall be owned by ℓ_n and $(\lambda x.e)$ [] shall be owned by ℓ_p, the rule swaps ℓ_n and ℓ_p on the inner boundary.

The [R-Merge-Box] and [R-Merge-Lam] rules are a little subtle since they merge a boundary onto a proxy. The two rules reduce *only* when there is a sensible designation of owner for the region between the boundary and the proxy. Thus, an additional (pre-)condition $\ell_p = \ell'_n$ is included in the rules. With this condition, the left-hand side of the [R-Merge-Box] and [R-Merge-Lam] rules has three owners: the entire expression is owned by ℓ_n; the region in-between is owned by ℓ_p, which equals ℓ'_n, and the region inside the proxy is owned by ℓ'_p.

The R-Merge rules are where the single-owner property intervenes in the reductions of the *𝒜owner* annotation language, much like how the property identifies what reductions are legitimate in Dimoulas et al. [2012]'s CPCF. For example, the earlier incorrectly-annotated program reduces to a proxy whose ownership labels do not match that of the enclosing boundary:

$$(), \left(\mathbf{B}\#\langle \ell_\mathsf{C}, \ell_\mathsf{S} \rangle \left\{ \; (\lambda x.x) \; \left(\mathbf{B}\#\langle \ell_\mathsf{P}, \ell_\mathsf{Q} \rangle \left\{ \, \lambda y.y \, \right\}\right) \; \right\}\right) \; 5 \longrightarrow^*$$

$$(), \left(\mathbf{B}\#\langle \ell_\mathsf{C}, \ell_\mathsf{S} \rangle \left\{ \; \mathsf{proxy}\left(\langle \ell_\mathsf{P}, \ell_\mathsf{Q} \rangle, \;\; \lambda y.y \right) \; \right\}\right) \; 5 \not\longrightarrow$$

When the proxy reaches the boundary, since the label ℓ_S does not match ℓ_P, the program can no longer reduce through the [R-Merge-Lam] rule as the side condition of the rule is not satisfied. Therefore, only programs with correct labels are guaranteed to make progress. As a remark,

the Agda mechanization includes a proof of the progress theorem in `Blame.Progress`, showing that programs adhering to the single-owner policy indeed can make progress. Unfortunately, the progress theorem cannot be modeled using an annotation interpretation.

Unfortunately, proving the preservation of the single-owner property is beyond the capability of the framework that I introduced in Chapter 4. In the [R-Proxy-β] rule, the argument v should belong to the owner ℓ_n, but the framework does not allow for the expression of this:

Reduction Rule $\quad (), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.e\big)\ v \longrightarrow_{\mathrm{m}} (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{ (\lambda x.e)\ \left(\mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{v\}\right) \right\}$

Assumption $\quad \big(\mathcal{I} \vDash \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.e\big)\big)$ and $(\mathcal{I} \vDash v)$

Obligation (part) $\quad \mathcal{I} \vDash \mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{v\}$

When attempting to construct an annotation interpretation $\mathcal{I}$ to prove that the [R-Proxy-β] rule preserves the single-owner property, the boundary around v on the right-hand side requires v be owned by ℓ_n as the proof obligation states. However, the highlighted premise $\mathcal{I} \vDash v$ by no means relates v to the owner ℓ_n and fails the attempt.

The problem is inherent in the design of the satisfaction relation ($\mathcal{I} \vDash e$) given in Section 5.2. In its presented form, the satisfaction relation only allows one to express that the function $\lambda x.e$ *inside the proxy* belongs to the owner ℓ_p. Since v is outside the proxy, nothing is told by the relation and the annotation interpretation.

An immediate solution would be equipping the satisfaction relation with indices. In the next section, I extend my framework with *indexed annotation interpretations* to prove that $\mathscr{T}owner$ preserves the single-owner property. In other words, as long as one starts with a program whose annotations consistently assign a single owner to each region, $\mathscr{T}owner$ is designed to guarantee that each region continues to have a single owner after reductions.

7.4 Indexed Interpretations and Single-Owner Policy

In this section, I state and prove that the ownership annotation language ($\mathscr{A}owner, \mathscr{T}owner$) preserves the single-owner property by augmenting the proof framework from Sections 5.2 and 5.3

with *indices*. This extension adds indices to annotations in annotation interpretations and to the satisfaction relation itself. The corresponding Monotonicity and Soundness theorems are also amended to work with indexed annotation interpretations.

Definition 7.11 defines indexed annotation interpretations, the generalized version of annotation interpretation from page 70. It equips each interpretation with a set of indices $\mathcal{J}$ and a family of binary relations over the indices, $\{\lhd_A\}_{A:\mathsf{Ann}_\tau}$. In indexed annotation interpretations, an annotation A no longer appears by itself but always shows up with an additional pair of indices (j, j') such that $j \lhd_A j'$. I shall write $A^{j,j'}$ to denote the triple (A, j, j') where $j \lhd_A j'$.

Definition 7.11 (Indexed Interpretation)**.** Given an annotation language $(\mathscr{A}, \mathscr{T})$, an *indexed annotation interpretation* $\mathcal{I}$ is a six-tuple $(\mathcal{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ satisfying the interpretation law in Definition 7.12. Below are the types of each component:

$$
\begin{array}{lcl}
\mathcal{J} & : & \mathcal{U} \\
\lhd_A & : & \mathcal{J} \to \mathcal{J} \to \mathcal{U} \\
\mathbb{S} & : & \mathsf{State} \to \mathcal{U} \\
\preccurlyeq & : & \sum_{s:\mathsf{State}} \mathbb{S}(s) \to \sum_{s:\mathsf{State}} \mathbb{S}(s) \to \mathcal{U} \\
\mathbb{B}[\![\, A^{j,j'}, e \,]\!] & : & \mathcal{U} \\
\mathbb{P}[\![\, A^{j,j'}, e^m \,]\!] & : & \mathcal{U}
\end{array}
$$

Since annotations are augmented with indices in annotation interpretations, the relevant interpretation law need adjusting to account for this change as well. Specifically, references to annotations in the $\mathbb{B}$ and $\mathbb{P}$ functions now take indexed annotations ($A^{j,j'}$) in the laws.

Definition 7.12 (Interpretation Law)**.** Let $\mathcal{I} :\equiv (\mathcal{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ be any indexed annotation interpretation. The interpretation law, suitably extended from Definition 5.7 in page 72, are:

- $\preccurlyeq$ is a preorder.
- $\mathbb{B}$ **is sound with respect to the transition steps** $\mathscr{T}$.

  If $\mathbb{S}(s)$, $\mathbb{S}(s')$, $s \preccurlyeq s'$ and $\mathscr{T} \vdash s, e \longrightarrow s', e'$, then $\mathbb{B}[\![\, A^{j,j'}, e \,]\!]$ implies $\mathbb{B}[\![\, A^{j,j'}, e' \,]\!]$.

Next, Figure 7.16 displays the *indexed satisfaction relation*, the generalized version of the sat-

$$
\begin{array}{lll}
\mathcal{I} \vDash^{j} \mathbf{B}\#A\,\{e\} & \text{iff} & j \lhd_A j',\ \mathbb{B}[\![\, A^{j,j'}, e \,]\!] \text{ and } \mathcal{I} \vDash^{j'} e \\
\mathcal{I} \vDash^{j} \mathsf{proxy}(A,\ e^m) & \text{iff} & j \lhd_A j',\ \mathbb{P}[\![\, A^{j,j'}, e^m \,]\!] \text{ and } \mathcal{I} \vDash^{j'} e^m \\
\mathcal{I} \vDash^{j} () & & \\
\mathcal{I} \vDash^{j} \mathsf{zero} & & \\
\mathcal{I} \vDash^{j} \mathsf{suc}(e) & \text{iff} & \mathcal{I} \vDash^{j} e \\
 & \vdots & \\
\mathcal{I} \vDash^{j} \lambda x.e & \text{iff} & \mathcal{I} \vDash^{j} e \\
\mathcal{I} \vDash^{j} e\ e_a & \text{iff} & \mathcal{I} \vDash^{j} e \text{ and } \mathcal{I} \vDash^{j} e_a \\
 & \vdots & \\
\mathcal{I} \vDash^{j} e;\ e_1 & \text{iff} & \mathcal{I} \vDash^{j} e \text{ and } \mathcal{I} \vDash^{j} e_1
\end{array}
\quad \Big\} \text{ modified } \textit{mutatis mutandis}
$$

The indexed satisfaction relation is generalized from Figure 5.1 in page 72 by equipping the relation itself with an index. All except the case for $\mathbf{B}\#A\,\{e\}$ and $\mathsf{proxy}(A,\ e^m)$ are modified by adding the same index j to both sides.

Figure 7.16: The indexed satisfaction relation.

isfaction relation from page 72. Other than taking indexed annotation interpretations as inputs, the generalization equips the relation itself with an index j. For the indexed satisfaction relation, a term e satisfies an interpretation $\mathcal{I}$ *at the index* j. I shall formally denote this as $\mathcal{I} \vDash^{j} e$.

Figure 7.16 highlights the amended pieces with shading to signify the differences with the original relation. In non-boundary cases, the same index is added both sides of the definition. These cases are all as modified in the same manner as 7.16 shows. For example, $\mathcal{I} \vDash^{j} e\ e_a$ holds if and only if both $\mathcal{I} \vDash^{j} e$ and $\mathcal{I} \vDash^{j} e_a$ hold. In case of boundaries and proxies, an additional condition $j \lhd_A j'$ is included to express that the index of the context, j, is related to the index of the sub-expression, j', via the relation $\{\lhd_A\}_{A:\mathsf{Ann}_\tau}$ from $\mathcal{I}$.

Lemma 7.13 (Renaming). *Let* $\Gamma :\equiv (x_1 : \tau_1), \ldots, (x_n : \tau_n)$ *and* $\Gamma' :\equiv (y_1 : \tau'_1), \ldots, (y_m : \tau'_m)$ *be given. Assume that* $\Gamma \vdash e : \tau$ *and* $\Gamma' \vdash y_{a_i} : \tau_i$ *for some* $1 \leq a_i \leq m$ *for* $i = 1 \ldots n$. *If* $\mathcal{I} \vDash^{j} e$ *then* $\mathcal{I} \vDash^{j} e\big[y_{a_1} \ldots y_{a_n} \,/\, x_1 \ldots x_n\big]$.

Proposition 7.14. *Assume that* $\Gamma' \vdash e_i : \tau_i$ *for* $i = 1, \ldots, n$ *and let any* $x_0 : \tau_0$ *be given.*

Let $\Gamma'' \vdash e'_0 : \tau_0, \ldots, \Gamma'' \vdash e'_n : \tau_n$ *be the sequence given by Proposition 4.2 where* $\Gamma'' :\equiv \Gamma', x_0 : \tau_0$,

$e'_0 :\equiv x_0$ and $e'_i :\equiv e_i$ for $i = 1, \ldots, n$. Then, if $\mathcal{I} \vDash^j e_i$ for $i = 1, \ldots, n$, there is a sequence $\mathcal{I} \vDash^j e'_i$ for $i = 0, \ldots, n$ as well.

Lemma 7.15 (Substitution). *Let $\Gamma :\equiv x_1 : \tau_1, \ldots, x_n : \tau_n$ and some Γ' be given. Assume that $\Gamma \vdash e : \tau$ and $\Gamma' \vdash e_i : \tau_i$ for $i = 1 \ldots n$. If $\mathcal{I} \vDash^j e$ and $\mathcal{I} \vDash^j e_i$ for $i = 1 \ldots n$ then $\mathcal{I} \vDash^j e[e_1 \ldots e_n \,/\, x_1 \ldots x_n]$.*

Proposition 7.16. *For any $j, j' \in \mathcal{J}$ and any value $n : \mathsf{nat}$, if $\mathcal{I} \vDash^j n$ then $\mathcal{I} \vDash^{j'} n$.*

Proposition 7.17 (Decomposition). *If $\mathcal{I} \vDash^j e$ and $e = E[e_r]$, there exists j_r such that $\mathcal{I} \vDash^{j_r} e_r$.*

With the generalization of the satisfaction relation, the indices can be instantiated with labels to identify the owner of an expression. The soundness proof of an annotation interpretation for the [R-Proxy-β] rule now has an additional index ℓ_n on the relation itself to connect the proxy and the argument of the function application to their enclosing context. More concretely, let the indices be the ownership labels and let $\{\lhd_{A_{own}}\}_{A_{own}}$ relate the labels in ownership annotations, i.e. $\ell_n \lhd_{\langle \ell_n, \ell_p \rangle} \ell_p$ for arbitrary $\langle \ell_n, \ell_p \rangle$, then the proof obligation of the soundness property is:

Reduction Rule $\quad (), \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.e\big)\ v \longrightarrow_{\mathrm{m}} (), \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{ (\lambda x.e)\ \big(\mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{v\}\big) \right\}$

Assumption $\quad \big(\mathcal{I} \vDash^{\ell_n} \mathsf{proxy}\big(\langle \ell_n, \ell_p \rangle,\ \lambda x.e\big)\big)$ and $\big(\mathcal{I} \vDash^{\ell_n} v\big)$

Obligation $\quad \mathcal{I} \vDash^{\ell_n} \mathbf{B}\#\langle \ell_n, \ell_p \rangle \left\{ (\lambda x.e)\ \big(\mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{v\}\big) \right\}$

The proof obligation is a little lengthy, but a more familiar goal shows up after expanding the satisfaction relation and plugging in the definition of $\{\lhd_{A_{own}}\}_{A_{own}}$:

$$\mathbb{B}[\![\ \langle \ell_n, \ell_p \rangle^{\ell_n, \ell_p},\ (\lambda x.e)\ \big(\mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{v\}\big)\ ]\!] \wedge \big(\mathcal{I} \vDash^{\ell_p} \lambda x.e\big) \wedge \big(\mathcal{I} \vDash^{\ell_p} \mathbf{B}\#\langle \ell_p, \ell_n \rangle\ \{v\}\big)$$

Compared to the situation of the indexless satisfaction relation at the end of the previous section (Section 7.3), the highlighted premise $(\mathcal{I} \vDash^{\ell_n} v)$ explicitly states that the argument (v) is satisfied at index ℓ_n. Thus, the highlighted part of the expanded proof obligation above becomes provable.

Before instantiating the augmented framework to prove the preservation of the single-owner property, I shall document the reformulated monotonicity and soundness properties, and also rephrase the corresponding theorems (Theorems 7.20 and 7.21). These properties and theorems

are mostly identical to their counterpart in Section 5.3 except that the ones presented here have incorporated the indexed satisfaction relation into their statements.

Definition 7.18 (Monotonic Interpretation)**.** An indexed interpretation $\mathcal{I} :\equiv (\mathcal{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is *monotonic* if for any s_1, s_2, e_1, e_2, if $\mathbb{S}(s_1)$ holds, $\mathcal{T} \vdash s_1, e_1 \longrightarrow_{\mathrm{m}} s_2, e_2$ and $\mathcal{I} \vDash^j e_1$ then $\mathbb{S}(s_2)$ holds and $s_1 \preccurlyeq s_2$.

Definition 7.19 (Sound Interpretation)**.** An indexed interpretation $\mathcal{I} :\equiv (\mathcal{J}, \{\lhd_A\}_{A:\mathsf{Ann}_\tau}, \mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is *sound* if for any s_1, s_2, e_1, e_2 such that $\mathbb{S}(s_1)$ and $\mathbb{S}(s_2)$ hold, if $s_1 \preccurlyeq s_2$, $\mathcal{T} \vdash s_1, e_1 \longrightarrow_{\mathrm{m}} s_2, e_2$ and $\mathcal{I} \vDash^j e_1$ then $\mathcal{I} \vDash^j e_2$.

Theorem 7.20 (Monotonicity)**.** *Let $\mathcal{I}$ be a monotonic and sound indexed interpretation. For any reduction sequence $\mathcal{T} \vdash s, e \longrightarrow^* s', e'$, if $\mathbb{S}(s)$ and $\mathcal{I} \vDash^j e$ then $\mathbb{S}(s')$ and $s \preccurlyeq s'$.*

Theorem 7.21 (Soundness)**.** *Let $\mathcal{I}$ be a monotonic and sound indexed interpretation. For any reduction sequence $\mathcal{T} \vdash s, e \longrightarrow^* s', e'$, if $\mathbb{S}(s)$ and $\mathcal{I} \vDash^j e$ then $\mathcal{I} \vDash^j e'$.*

Definition 7.22. For any closed expression e such that $\mathcal{I} \vDash^j e$, $[e]_{\mathcal{I},j}$ is the equivalence class of e containing only expressions that satisfy $\mathcal{I}$ at j, i.e. $e' \in [e]_{\mathcal{I},j} \iff e' \sim_{\mathrm{P}} e$ and $\mathcal{I} \vDash^j e'$.

Theorem 7.23. *Let $\mathcal{I} :\equiv (\mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ be a monotonic and sound interpretation. Assume that for some s_0, j_0 and e_0, both $\mathbb{S}(s_0)$ and $\mathcal{I} \vDash^{j_0} e_0$ hold. Let $\mathcal{T}$ be the minimum subsystem of $\mathcal{T}_{\mathsf{ind}}[\mathscr{A}; \mathcal{T}]$ that contains $(s_0, [e_0]_P)$, the map $h : (s, [e]_P) \longmapsto (s, [e]_{\mathcal{I},j})$ for any e such that $\mathcal{I} \vDash^j e$ holds is a well-defined function from $\mathcal{T}$ to $\mathcal{T}_{\mathsf{sat}}[\mathscr{A}; \mathcal{T}; \mathcal{I}, j]$. Moreover, h is a homomorphism.*

With all the machinery, we are finally in a position to state and prove the preservation of the single-owner property of the $\mathscr{A}owner$ language. To be precise, Definition 7.24 gives the indexed annotation interpretation, $\mathcal{I}sngl$, that instantiates the augmented framework to capture single ownership. In $\mathcal{I}sngl$, the indices are just labels and $j \lhd_{A_{own}} j'$ precisely when (j, j') are the labels in A_{own}. The $\mathbb{B}$ function of $\mathcal{I}sngl$ similarly states that the indices at boundaries are the annotated labels. Then, Theorem 7.25 shows that $\mathcal{I}sngl$ is monotonic and sound to conclude the preservation proof.

Definition 7.24. The annotation interpretation $\mathcal{I}sngl :\equiv (\mathcal{J}_{sngl},\ \{\lhd_{A_{own}}\}_{A_{own}},\ \mathbb{S}_{sngl},\ \preccurlyeq_{sngl},\ \mathbb{B},\ \mathbb{P})$ is defined as

$$
\begin{array}{lcl}
\mathcal{J}_{sngl} & :\equiv & \mathsf{Label} \\
j \lhd_{A_{own}} j' & :\equiv & j = \ell_n \wedge j' = \ell_p \\
\quad \text{where } A_{own} = \langle \ell_n, \ell_p \rangle & & \\
\mathbb{S}_{sngl}(s) & :\equiv & \top \\
s \preccurlyeq_{sngl} s' & :\equiv & \top \\
\mathbb{B}[\![\ \langle \ell_n, \ell_p \rangle^{j,j'}, e\ ]\!] & :\equiv & j = \ell_n \wedge j' = \ell_p \\
\mathbb{P}[\![\ A^{j,j'}_{own}, e\ ]\!] & :\equiv & \mathbb{B}[\![\ A^{j,j'}_{own}, e^m\ ]\!]
\end{array}
$$

Theorem 7.25. *The annotation interpretation $\mathcal{I}sngl$ is monotonic and sound.*

The definition $\mathcal{I}sngl$ may seem trivial as it straightforwardly identifies the indices with the ownership labels. In Section 7.6, I shall present a more interesting example that instantiates the augmented framework to relate labels in blame objects with the ownership labels on top of the single-owner property and prove the Blame Correctness theorem.

7.5 Capturing Monitoring Strategies in the Framework

Equipping the satisfaction relation with indices yields an interesting connection between the monitoring strategies of higher-order values and the proof framework that is organized around the indexed satisfaction relation. Recall that in the monitor calculus proof framework, proving a specific property about an instantiation of the calculus amounts to constructing an annotation interpretation and proving that it is both monotonic and sound. In the generalized framework discussed in Section 7.4, the introduction of indices adds new proof obligations to the soundness proof of indexed annotation interpretations, reflecting the monitoring strategies of higher-order values in the monitor calculus. In this section, I shall illustrate this connection by working with an alternative [R-Cross-Cons] rule and examining how it affects the proof obligations.

To begin with, suppose that the monitor calculus employs the following alternative [R-Cross-Cons′] rule that directly discards boundaries around the pairs instead of creating two new bound-

aries around the sub-components,[2]

$$\mathcal{T} \vdash s, \mathbf{B}\#A\,\{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} s', \langle v_1, v_2\rangle$$

To show that an indexed interpretation $\mathcal{I}$ is sound for this [R-Cross-Cons′] rule, one needs to prove that $\mathcal{I} \vDash^j \mathbf{B}\#A\,\{\ \langle v_1, v_2\rangle\ \}$ implies $\mathcal{I} \vDash^j \langle v_1, v_2\rangle$. After expanding the definition of the satisfaction relation, this becomes

$$\mathbb{B}[\![\ A^{j,j'}, \langle v_1, v_2\rangle\ ]\!] \wedge (j \lhd_A j') \wedge \left(\mathcal{I} \vDash^{j'} v_1\right) \wedge \left(\mathcal{I} \vDash^{j'} v_2\right) \text{ implies } \left(\mathcal{I} \vDash^{j} v_1\right) \wedge \left(\mathcal{I} \vDash^{j} v_2\right)$$

Here, note that v_1 and v_2 satisfy $\mathcal{I}$ at the index j' in the assumptions on the left-hand side and yet they are required to satisfy $\mathcal{I}$ at the index j by the proof obligation on the right-hand side. In other words, when the boundary annotated with A around v_1 and v_2 is removed by the [R-Cross-Cons′] rule, the corresponding satisfaction results, i.e. $\mathcal{I} \vDash^{j'} v_k$ and $\mathcal{I} \vDash^{j} v_k$, change their indices from j' to j where $j \lhd_A j'$.

The relationship between the indices in the proof obligation of the [R-Cross-Cons] rule is in sharp contrast to the one for the (alternative) [R-Cross-Cons′] rule. Specifically, the [R-Cross-Cons] rule creates two new boundaries with the annotations A_1 and A_2 around v_1 and v_2:

$$\mathcal{T} \vdash s, \mathbf{B}\#A\,\{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} s', \langle\ \mathbf{B}\#A_1\,\{v_1\}, \mathbf{B}\#A_2\,\{v_2\}\ \rangle$$

The soundness for the [R-Cross-Cons] rule asks for the proof of $\mathcal{I} \vDash^j \langle \mathbf{B}\#A_1\,\{v_1\}, \mathbf{B}\#A_2\,\{v_2\}\rangle$ assuming $\mathcal{I} \vDash^j \mathbf{B}\#A\,\{\langle v_1, v_2\rangle\}$. By inverting the assumption, there exists j' such that $j \lhd_A j'$. Then, by a similar argument, the proof obligation simplifies to $\exists j_k.\ (j \lhd_{A_k} j_k) \wedge (\mathcal{I} \vDash^{j_k} v_k)$ assuming $\mathcal{I} \vDash^{j'} v_k$ for $1 \le k \le 2$. Hence, that the indices change from j' to j_k where $j \lhd_A j'$ and $j \lhd_{A_k} j_k$ indicates that v_k stays inside a boundary both before and after the reduction step. Furthermore, the annotation on the boundary changes from A to A_k.

The same index-changing phenomenon happens for the [R-Cross-Nat] rule, but it does not affect the soundness proof. To prove that an interpretation $\mathcal{I}$ is sound for the [R-Cross-Nat]

[2]This rule is an approximation of an untyped–typed–untyped interaction scenario that Greenman et al. [2019] discovered for Vitousek et al. [2017]'s Transient checking strategy.

rule, the same argument yields the goal

$$\mathbb{B}[\![\, A^{j,j'}, n \,]\!] \wedge (j \lhd_A j') \wedge \left(\mathcal{I} \vDash^{j'} n\right) \text{ implies } \left(\mathcal{I} \vDash^{j} n\right)$$

provided $\mathcal{T} \vdash s, \mathbf{B}\#A\{n\} \longrightarrow_{\mathrm{m}} s', n$. Nonetheless, for base values like natural numbers, one can always prove $\mathcal{I} \vDash^{j} n$ regardless of the index. Comparing to other cases such as the [R-Cross-Cons′] rule, it is not possible to generally prove the satisfaction of higher-order values at a different index. For example, the obligations $\mathcal{I} \vDash^{j} v_k$ from the [R-Cross-Cons′] rule could contain nested boundaries in v_k and hence they have to be dealt with on a case-by-case basis.

In all but the [R-Cross-Nat] rule, the indices on the satisfaction relation encode the information of the boundaries around the sub-expressions and how they evolve during reduction steps. In summary, the indexed-interpretation extension reifies the monitoring strategy of higher-order values—how boundaries evolve—concretely as the relatioship between the indices on the corresponding satisfaction relation proof obligations.

7.6 Correct Blame and Single-Owner Policy

In this section, I put the blame annotation language $(\mathscr{A}bctc, \mathscr{T}b_i)$ from Section 7.1 and the ownership annotation language $(\mathscr{A}owner, \mathscr{T}owner)$ from Section 7.3 together to prove a result that is similar to Dimoulas et al. [2011, 2012]'s Blame Correctness theorem. Specifically, I shall show that the transition steps $\mathscr{T}b_i$ in page 104 distribute blame objects in a way that is consistent with how the transition steps $\mathscr{T}owner$ in page 114 propagates the ownership labels.

$$\begin{aligned}&\mathsf{proxy}(\langle\langle \ell_{\mathsf{C}}, \ell_{\mathsf{S}}\rangle,\ [\langle b, \kappa'\rangle]\rangle,\ \lambda x.x)\ (\lambda z.\mathsf{suc}(z))\ 4\\ &\quad \text{where}\ \ b :\equiv \{\mathsf{pos} = \ell_{\mathsf{S}};\ \mathsf{neg} = \ell_{\mathsf{C}}\}\\ &\qquad\quad \kappa' :\equiv \left(\mathsf{isEven}^{\ell_{\mathsf{S}}} \rightarrow\!\!/_{\mathsf{c}}\ \mathsf{isOdd}^{\ell_{\mathsf{C}}}\right) \rightarrow\!\!/_{\mathsf{c}}\ \left(\mathsf{isEven}^{\ell_{\mathsf{C}}} \rightarrow\!\!/_{\mathsf{c}}\ \mathsf{any}/\mathsf{c}^{\ell_{\mathsf{S}}}\right)\end{aligned}$$

Figure 7.17: An example program of the combined annotation language

Figure 7.17 illustrates what it means for blame objects and ownership labels to be consistent. The example program combines $(\mathscr{A}bctc, \mathscr{T}b_i)$and $(\mathscr{A}owner, \mathscr{T}owner)$, hence its annotation is a pair comprising the ownership labels $\langle \ell_{\mathsf{C}}, \ell_{\mathsf{S}}\rangle$ and a list of blame-contract pairs, $[\langle b, \kappa'\rangle]$. In this program, the pos field of b points to the region inside the proxy, i.e. the part that contains $\lambda x.x$. The neg

$$
\begin{array}{rcll}
\mathscr{A} & :\equiv & (A, s) & \\
A & ::= & \langle A_{own}, A_{bctc} \rangle & A_{bctc} ::= [\langle b_1, \kappa_1 \rangle, \ldots, \langle b_m, \kappa_m \rangle] \\
s \in \text{Status} & ::= & \text{Ok} \mid \text{Err}(\ell) & A_{own} ::= \langle \ell_n, \ell_p \rangle
\end{array}
$$

Figure 7.18: The syntax of the combined annotation language, $(\mathscr{A}, \mathscr{T}_i)$.

field similarly points to the enclosing context of the proxy, or the function application that calls the proxy with $\lambda z.\text{suc}(z)$ and 4. Most importantly, these two labels in b match the ownership labels $\langle \ell_C, \ell_S \rangle$ which indicate that the owner of the context is ℓ_C and that the owner of $\lambda x.x$ is ℓ_S.

The theorem that I am proving in this section states that this correspondence between the ownership annotation $\langle \ell_C, \ell_S \rangle$ and the labels in the blame object b will be preserved during evaluation. Additionally, when there is a sequence of the blame objects in the annotations, the neg label of any blame object in the sequence always matches the pos label of the object that follows it. This preservation property together with the preservation of the consistency judgement from Section 7.2 and the preservation of the single-owner property from Section 7.4 capture Dimoulas et al. [2011, 2012]'s Blame Correctness theorem.

It is worth mentioning that my alternative formulation decomposes Dimoulas et al. [2011, 2012]'s Blame Correctness theorem into three reusable parts. On one hand, the correct tracking of blame labels on the monitors of CPCF in their Blame Correctness theorem is captured by the preservation of the consistency judgement for the blame annotation language $(\mathscr{A}bctc, \mathscr{T}b_i)$from Sections 7.1 and 7.2. On the other hand, the single-owner property in their work is modeled and proved using the ownership annotation language $(\mathscr{A}owner, \mathscr{T}owner)$from Sections 7.3 and 7.4. These two annotation languages and their properties are developed independently, and they are reusable through the framework developed in Section 5.4 when combined with other annotation languages. In particular, the third part of my formulation of the Blame Correctness theorem in this section is proved for the annotation language that pairs up $(\mathscr{A}bctc, \mathscr{T}b_i)$and $(\mathscr{A}owner, \mathscr{T}owner)$, and only when joining all three properties we regain the full Blame Correctness theorem.

Figure 7.18 briefly lists the syntax of the combined annotation language. The syntax of annotations, A, simply pairs up the ownership annotation A_{own} from $\mathscr{A}owner$ and the blame annotation A_{bctc} from $\mathscr{A}bctc$. The global state of the combined language is just Status, the same one from

$\mathscr{A}bctc$. Similar to the annotations, the transition steps $\mathscr{T}_i$ of $\mathscr{A}$ is the combination of $\mathscr{T}owner$ in page 114 and $\mathscr{T}b_i$ in page 104. The two transition steps are simply put together to simultaneously propagate the A_{own} part and the A_{bctc} part of the annotation. For convenience, Figure 7.18 recaps the definition of A_{own} and A_{bctc}.

For the combined annotation language in Figure 7.18, the list of blame-contract pairs in the A_{bctc} part of the annotations accumulates when a proxy crosses a boundary. In this situation, when the blame labels match the ownership labels in the A_{own} part of the annotations, the labels of the blame objects on the proxy will align with the labels on the boundary. Thus, that the blame objects in the annotations have aligned labels is an essential part of my formulation of the Blame Correctness theorem. To be more concretely, consider the following reduction sequence:

$$
\begin{array}{l}
\textsc{Ok},\ \mathbf{B}\#\langle\langle \ell_C, \ell_S\rangle, [\langle b, \mathsf{isEven}^{\ell_C} \rightarrow\!\!\!/c\ \mathsf{any}/c^{\ell_S}\rangle]\rangle\ \{ \\
\qquad \mathsf{proxy}\big(\langle\langle \ell_S, \ell_C\rangle, [\langle b_1, \mathsf{isEven}^{\ell_S} \rightarrow\!\!\!/c\ \mathsf{isOdd}^{\ell_C}\rangle]\rangle,\ \lambda y.y\big) \\
\quad \} \longrightarrow \\
\textsc{Ok},\ \mathsf{proxy}\big(\langle \ell_C, \ell_C, [\langle b_1, \mathsf{isEven}^{\ell_S} \rightarrow\!\!\!/c\ \mathsf{isOdd}^{\ell_C}\rangle, \langle b, \mathsf{isEven}^{\ell_C} \rightarrow\!\!\!/c\ \mathsf{any}/c^{\ell_S}\rangle]\rangle,\ \lambda y.y\big) \\
\qquad\qquad\qquad\qquad \text{where } b :\equiv \{\mathsf{pos} = \ell_S;\ \mathsf{neg} = \ell_C\} \text{ and } b_1 :\equiv \mathit{blameSwap}(b).
\end{array}
$$

Figure 7.19: Aligned blame object sequences

In the program in Figure 7.19, the labels of the blame objects b and b_1 match the ownership labels (colored brown). The ownership labels on the boundary and the proxy, in turn, uniquely identify the owner of each region delimited by the proxy and the boundary. After one reduction step, the sequence of blame objects annotated on the proxy is $[b_1, b]$. In this sequence, the neg field of b_1 equals the pos field of b. In other words, this sequence of blame objects is aligned.

To capture aligned sequences of blame objects, I introduce the judgement BlameSeq in Figure 7.20. For blame objects $b_1, \ldots, b_n$ to be aligned, the neg field of any object in the sequence must match the pos field of the object that follows it. That is, $b_1, \ldots, b_n$ are aligned if $b_1.\mathsf{neg} = b_2.\mathsf{pos}, \ldots,$ and $b_{n-1}.\mathsf{neg} = b_n.\mathsf{pos}$. To define this inductively, the BlameSeq judgement takes two additional indices, ℓ_p and ℓ_q, to track the pos field of the *first* object in the sequence and the neg

$$
\frac{}{\mathsf{BlameSeq}(\ell,\ \ell,\ [])}
\qquad
\frac{b_1 = \{\mathsf{pos} = \ell_p;\ \mathsf{neg} = \ell_q\} \qquad \mathsf{BlameSeq}(\ell_q,\ \ell_r,\ [b_2, \ldots, b_m])}{\mathsf{BlameSeq}(\ell_p,\ \ell_r,\ [b_1, b_2, \ldots, b_m])}
$$

Figure 7.20: Aligned blame object sequences.

field of the *last* object in the sequence. Specifically, $\mathsf{BlameSeq}(\ell_p,\ \ell_q,\ [b_1, \ldots, b_n])$ holds if and only if $\ell_p = b_1.\mathsf{pos}$, $\ell_q = b_n.\mathsf{neg}$ and $b_i.\mathsf{neg} = b_{i+1}.\mathsf{pos}$ for all $1 \leq i < n$.

Proposition 7.26. *If* $\mathsf{BlameSeq}(\ell_p,\ \ell_q,\ [b_1, \ldots, b_m])$ *and* $\mathsf{BlameSeq}(\ell_q,\ \ell_r,\ [b'_1, \ldots, b'_n])$ *then* $\mathsf{BlameSeq}(\ell_p,\ \ell_r,\ [b_1, \ldots, b_m, b'_1, \ldots, b'_n])$.

Proposition 7.27. *If* $\mathsf{BlameSeq}(\ell_p,\ \ell_q,\ [b_1, \ldots, b_m])$ *then* $\mathsf{BlameSeq}(\ell_q,\ \ell_p,\ [\mathit{blameSwap}(b_m), \ldots, \mathit{blameSwap}(b_1)])$.

Definition 7.28 defines the indexed annotation interpretation $\mathcal{I}\mathit{own}_i$ that captures aligned sequences of blame objects. Specifically, the relation $\lhd_A$ in $\mathcal{I}\mathit{own}_i$ relates the indices $j, j' \in \mathsf{Label}$ if the sequence of blame objects in A are aligned with indices j and j', i.e. if $\mathsf{BlameSeq}(j',\ j,\ [b_1, \ldots, b_m])$. Other than the indices of the annotations, the $\mathbb{B}_{i+1}[\![\ A^{j,j'}, e\ ]\!]$ function asserts that the indices j, j' on the annotation A matches the ownership part of A, i.e. $j = \ell_n$ and $j' = \ell_p$ for $A = \langle A_{\mathit{own}}, A_{\mathit{bctc}}\rangle = \langle\langle \ell_n, \ell_p\rangle, A_{\mathit{bctc}}\rangle$. This ensures that when the $\lhd_A$ relation aligns the blame objects in the A_{bctc} part of A, the ownership labels ℓ_n, ℓ_p match the labels of the first and the last blame objects.

Definition 7.28. The annotation interpretation $\mathcal{I}\mathit{own}_i :\equiv (\mathcal{J}_{\mathit{own}},\ \{\lhd_A\}_{A:\mathsf{Ann}_\tau},\ \mathbb{S}_{\mathit{own}},\ \preccurlyeq_{\mathit{own}},\ \mathbb{B}_i, \mathbb{P}_i)$ is defined as

$$
\begin{array}{lcl}
\mathcal{J}_{\mathit{own}} & :\equiv & \mathsf{Label} \\
j \lhd_A j' & :\equiv & \mathsf{BlameSeq}(j',\ j,\ [b_1, \ldots, b_m]) \\
\quad \text{where } A = \langle A_{\mathit{own}}, A_{\mathit{bctc}}\rangle = \langle A_{\mathit{own}}, [\langle b_1, \kappa_1\rangle, \ldots, \langle b_m, \kappa_m\rangle]\rangle & & \\
\mathbb{S}_{\mathit{own}}(s) & :\equiv & \top \\
s \preccurlyeq_{\mathit{own}} s' & :\equiv & \top \\
\mathbb{B}_0[\![\ A, e\ ]\!] & :\equiv & \bot \\
\mathbb{B}_{i+1}[\![\ A^{j,j'}, e\ ]\!] & :\equiv & (\exists j_1.\ \mathcal{I}\mathit{own}_i \vDash^{j_1} \kappa_1) \ \wedge\ \ldots\ \wedge\ (\exists j_m.\ \mathcal{I}\mathit{own}_i \vDash^{j_m} \kappa_m) \wedge \\
 & & \qquad (j = \ell_n \wedge j' = \ell_p) \\
\quad \text{where } A = \langle A_{\mathit{own}}, A_{\mathit{bctc}}\rangle = \langle\langle \ell_n, \ell_p\rangle, A_{\mathit{bctc}}\rangle & & \\
\mathbb{P}_i[\![\ A^{j,j'}, e^m\ ]\!] & :\equiv & \mathbb{B}_i[\![\ A^{j,j'}, e^m\ ]\!]
\end{array}
$$

Finally, Theorem 7.29 proves that the interpretation $\mathcal{I}\mathit{own}_i$ is monotonic and sound for all $i \geq 0$, thus completing the proof of the Blame Correctness theorem.

Theorem 7.29. *The annotation interpretation* $\mathcal{I}\mathit{own}_i$ *is monotonic and sound for all* $i \geq 0$.

Chapter 8

Space-Efficient Contracts

Over the past two decades, researches have identified the accumulation of *redundant proxies* around values as a significant source of overhead of contracts and gradual types, incurring unnecessary space and time cost [Herman et al. 2010; Siek and Wadler 2010; Siek et al. 2015a; Greenberg 2015, 2016; Feltey et al. 2018]. Indeed, the issue is bad both in theory [Siek et al. 2009; Siek and Wadler 2010; Siek et al. 2015a, 2021] and in practice [Findler et al. 2008; Takikawa et al. 2016, 2015, Figure 10].

Since contracts allow arbitrary user-defined predicates, the correct determination of redundancy is more complex for contracts than gradual typing in order to preserve the correct blame behavior. Luckily, Greenberg [2015, 2016]'s space-efficient latent contracts hold the essential ideas for avoiding redundancy by merging adjacent proxies and removing duplicate checks in the contracts on the proxies. Feltey et al. [2018] take Greenberg [2016]'s idea further and develop collapsible contracts in Racket [Felleisen et al. 2015, 2018]'s contract system, resolving certain cases that suffer exponential slowdown.

Space-efficient contracts occupy a rather unique position in the world of contracts. They represent a class of contract systems that employ non-trivial operations on contracts themselves at run time; their metatheory concerns non-functional properties of the contract systems; they are indispensable for practical programs that use contracts. Accordingly, they make an excellent ex-

ercise for the monitor calculus. In this chapter, I instantiate the monitor calculus to capture space-efficient contracts and apply the transition-system framework to build its metatheory based on Greenberg [2016]'s work. Similar to the proofs in Chapter 7, the metatheory in this chapter is built primarily on various instantiations of the monitor calculus that extends a common annotation language without resorting to external definitions.

Section 8.1 starts with discussion with a gentle introduction to space-efficient contracts. Section 8.2 formally introduces $(\mathscr{A}se, \mathscr{T}s)$, an annotation language that captures space-efficient contracts. Section 8.3 further discusses several auxiliary definitions about the contracts together with their properties in preparation for the proofs of space efficiency and correct time complexity.

Section 8.4 proves the space efficiency of the instantiation $\lambda_m[\mathscr{A}se; \mathscr{T}s]$. The proof is dependent on two parameters for bounding the contracts: $\mathcal{K}$, the set of all distinct predicates in all contracts, and H, the height of the tallest contract in the initial program. As long as there are no recursive contracts, the height of the contracts are non-increasing, and the proof of space efficiency proceeds by constructing an interpretation that bounds the size of each contract within $O(|\mathcal{K}| \cdot 2^H)$.

Section 8.5 tackles time complexity. It introduces $(\mathscr{A}ccs, \mathscr{T}ccs)$, an extension of $(\mathscr{A}se, \mathscr{T}s)$ that counts in its global states the (flat) contract checks and the number of primitive operations used by $\lambda_m[\mathscr{A}ccs;\mathscr{T}ccs]$ for merging space-efficient contracts. Let k denote the number of monitor-related steps in any reduction sequence. Section 8.5 proves, again via interpretations of annotations, that $\lambda_m[\mathscr{A}ccs;\mathscr{T}ccs]$ checks at most $O(k \cdot |\mathcal{K}|)$ predicates and uses no more than $O(k \cdot |\mathcal{K}|^2 \cdot 2^H)$ operations for maintaining space efficiency. In other words, adding contracts to a program will not introduce exponential slowdown.

Finally, Section 8.6 shows that space-efficient contracts signal the same errors as extended CPCF through the instantiation $\lambda_m[\mathscr{A}scctc;\mathscr{T}sc]$ that combines both $\lambda_m[\mathscr{A}se;\mathscr{T}s]$ and $\lambda_m[\mathscr{A}ctc; \mathscr{T}c_1]$ and compares their contract checking status.

8.1 Space-Efficient Contracts in Action

To understand why contracts can accumulate on proxies, consider the following example written in the syntax of $\lambda_{\mathsf{m}}[\mathscr{A}ctc; \mathscr{T}c_1]$, the instantiation for ordinary contracts from Chapter 6:

$$\mathsf{proxy}([\kappa_{oe} \rightarrow\!\!\!/_{\mathsf{c}}\, \kappa_{oe}],\ \lambda h.h)$$

In this function, κ_{oe} is a shorthand for the contract *isOdd* $\rightarrow\!\!\!/_{\mathsf{c}}$ *isEven*. The flat contracts *isOdd* and *isEven* blame ℓ_{O} or ℓ_{E} unless their inputs are odd numbers or even numbers, respectively. The contracted function $\lambda h.h$ takes another function and immediately returns it. Whenever $\lambda h.h$ is called, its argument is attached with the contract κ_{oe}, and the same contract is again attached to the result of the application. Thus, the output of $\lambda h.h$ will end up receiving *two* identical copies of κ_{oe}. The following reduction sequence shows this situation in more detail:

$$\begin{aligned}
&\textsc{Ok}, \mathsf{proxy}([\kappa_{oe} \rightarrow\!\!\!/_{\mathsf{c}}\, \kappa_{oe}],\ \lambda h.h)\ (\lambda y.\mathsf{suc}(\mathsf{suc}(y)))\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#[\kappa_{oe}]\ \{\ (\lambda h.h)\ (\mathbf{B}\#[\kappa_{oe}]\ \{\ \lambda y.\mathsf{suc}(\mathsf{suc}(y))\ \})\ \}\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#[\kappa_{oe}]\ \{\ (\lambda h.h)\ \mathsf{proxy}([\kappa_{oe}],\ \lambda y.\mathsf{suc}(\mathsf{suc}(y)))\ \}\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#[\kappa_{oe}]\ \{\ \mathsf{proxy}([\kappa_{oe}],\ \lambda y.\mathsf{suc}(\mathsf{suc}(y)))\ \}\\
\longrightarrow\ &\textsc{Ok}, \mathsf{proxy}([\kappa_{oe}, \kappa_{oe}],\ \lambda y.\mathsf{suc}(\mathsf{suc}(y)))
\end{aligned}$$

During reduction, the domain of $\kappa_{oe} \rightarrow\!\!\!/_{\mathsf{c}}\, \kappa_{oe}$ is attached to the argument, resulting in a new proxy that is wrapped around $\lambda y.\mathsf{suc}(\mathsf{suc}(y))$. The function $\lambda h.h$ immediately returns the proxy in the next step, and the outer boundary stacks another copy of κ on the proxy. Intuitively, only one copy of κ_{oe} is needed around $\lambda y.\mathsf{suc}(\mathsf{suc}(y))$, and keeping the duplication may result in useless memory growth in a program that adds such a contract in a loop.

Note that redundant contracts not only consume more memory but also degrade the performance. For example, whenever the contracted $\lambda y.\mathsf{suc}(\mathsf{suc}(y))$—the output of $\lambda h.h$—is applied, the predicates *isOdd* and *isEven* will be checked twice, and the second check always produces the

$$
\begin{aligned}
&\textsc{Ok}, \mathrm{proxy}\big((\kappa'_{oe} \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ \kappa'_{oe}),\ \lambda h.h\big)\ (\lambda y.\mathrm{suc}(\mathrm{suc}(y)))\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#\kappa'_{oe}\ \{\ (\lambda h.h)\ \big(\mathbf{B}\#\kappa'_{oe}\ \{\ \lambda y.\mathrm{suc}(\mathrm{suc}(y))\ \}\big)\ \}\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#\kappa'_{oe}\ \{\ (\lambda h.h)\ \mathrm{proxy}\big(\kappa'_{oe},\ \lambda y.\mathrm{suc}(\mathrm{suc}(y))\big)\ \}\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#\kappa'_{oe}\ \{\ \mathrm{proxy}\big(\kappa'_{oe},\ \lambda y.\mathrm{suc}(\mathrm{suc}(y))\big)\ \}\\
(\star)\quad &\text{-- } \mathit{join}(\kappa'_{oe}, \kappa'_{oe}) = \mathit{join}([\mathit{isOdd}] \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ [\mathit{isEven}], [\mathit{isOdd}] \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ [\mathit{isEven}])\\
&\qquad\qquad = \mathit{join}([\mathit{isOdd}], [\mathit{isOdd}]) \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ \mathit{join}([\mathit{isEven}], [\mathit{isEven}])\\
&\qquad\qquad = \mathit{joinFlats}([\mathit{isOdd}], [\mathit{isOdd}]) \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ \mathit{joinFlats}([\mathit{isEven}], [\mathit{isEven}])\\
&\qquad\qquad = [\mathit{isOdd}] \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ [\mathit{isEven}]\\
\longrightarrow\ &\textsc{Ok}, \mathrm{proxy}([\mathit{isOdd}] \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ [\mathit{isEven}],\ \lambda y.\mathrm{suc}(\mathrm{suc}(y)))
\end{aligned}
$$

Figure 8.1: An example reduction sequence of space-efficient contracts.

same result as the first:

$$
\begin{aligned}
&\textsc{Ok}, \mathrm{proxy}([\kappa_{oe}, \kappa_{oe}],\ \lambda y.\mathrm{suc}(\mathrm{suc}(y)))\ 5\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#[\mathit{isEven}, \mathit{isEven}]\ \{\ (\lambda y.\mathrm{suc}(\mathrm{suc}(y)))\ (\mathbf{B}\#[\mathit{isOdd}, \mathit{isOdd}]\ \{\ 5\ \})\ \}\\
\longrightarrow\ &\textsc{Ok}, \mathbf{B}\#[\mathit{isEven}, \mathit{isEven}]\ \{\ (\lambda y.\mathrm{suc}(\mathrm{suc}(y)))\ 5\ \}\\
\longrightarrow^*\ &\textsc{Ok}, \mathbf{B}\#[\mathit{isEven}, \mathit{isEven}]\ \{\ 7\ \}\\
\longrightarrow\ &\textsc{Err}(\ell_{\mathrm{E}}), 7
\end{aligned}
$$

The space-efficient contracts in this chapter aim at removing redundant checks. To discover redundancy, the space-efficient contracts change their syntax. For example, let κ'_{oe} be the shorthand of $[\mathit{isOdd}] \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ [\mathit{isEven}]$, the above function would have been written as

$$\mathrm{proxy}\big(\kappa'_{oe} \rightarrow\!\!\!/\!\!\!\text{s}\!\!\!\text{e}\ \kappa'_{oe},\ \lambda h.h\big)\ .$$

Instead of annotating a list of contracts, any boundary (and similarly any proxy) is always annotated with one space-efficient contract. Following the syntax changes, the leaves of contracts now contain lists of flat predicates to track all needed checks. The reduction sequence in Figure 8.1 shows how space-efficient contracts are enforced. In step ($\star$) in the figure, the contract

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

Figure 8.2: Syntax of the space-efficient annotation language, $\mathscr{A}se$

$$\boxed{\Delta :\equiv \{t_1, \ldots, t_n\}} \qquad \boxed{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa : \mathsf{SECtc}\ \tau}$$

$$\frac{}{\Delta \vdash^{\mathsf{se}} \mathsf{unit}/\mathsf{se} : \mathsf{SECtc}\ \mathsf{unit}} \qquad \frac{x_1 : \mathsf{nat} \vdash e_1 : \mathsf{nat} \qquad \ldots \qquad x_m : \mathsf{nat} \vdash e_m : \mathsf{nat}}{\Delta \vdash^{\mathsf{se}} [\mathsf{flat}^{\ell}(x_1.\, e_1), \ldots, \mathsf{flat}^{\ell}(x_m.\, e_m)] : \mathsf{SECtc}\ \mathsf{nat}}$$

$$\frac{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_1 : \mathsf{SECtc}\ \tau_1 \qquad \Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_2 : \mathsf{SECtc}\ \tau_2}{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_1 \times/\mathsf{se}\ {}^{\mathsf{se}}\kappa_2 : \mathsf{SECtc}\ (\tau_1 \times \tau_2)} \qquad \frac{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_1 : \mathsf{SECtc}\ \tau_1 \qquad \Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_2 : \mathsf{SECtc}\ \tau_2}{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_1 +/\mathsf{se}\ {}^{\mathsf{se}}\kappa_2 : \mathsf{SECtc}\ (\tau_1 + \tau_2)}$$

$$\frac{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa : \mathsf{SECtc}\ \tau}{\Delta \vdash^{\mathsf{se}} \mathsf{box}/\mathsf{se}\ {}^{\mathsf{se}}\kappa : \mathsf{SECtc}\ (\mathsf{Box}\ \tau)} \qquad \frac{t \in \Delta}{\Delta \vdash^{\mathsf{se}} t : \mathsf{SECtc}\ t}$$

$$\frac{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_a : \mathsf{SECtc}\ \tau_a \qquad \Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_r : \mathsf{SECtc}\ \tau_r}{\Delta \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa_a \to/\mathsf{se}\ {}^{\mathsf{se}}\kappa_r : \mathsf{SECtc}\ (\tau_a \to \tau_r)} \qquad \frac{\Delta, t \vdash^{\mathsf{se}} {}^{\mathsf{se}}\kappa : \mathsf{SECtc}\ \tau}{\Delta \vdash^{\mathsf{se}} \mu/\mathsf{se}\, t.{}^{\mathsf{se}}\kappa : \mathsf{SECtc}\ (\mu t.\tau)}$$

Figure 8.3: The typing rules of contracts

on the boundary (colored blue) is *merged into* the contract on the proxy (colored brown) using a metafunction *join*. The operation $\mathit{join}(\kappa'_{oe}, \kappa'_{oe})$ traverses the given contracts, appends the lists of predicates in the leaves, and removes any duplication from the appended predicate lists. As a result, the contract on the proxy around $\lambda y.\mathsf{suc}(\mathsf{suc}(y))$ in the final step only contains one copy of the predicates in its domain and range.

8.2 Syntax and Transition Steps of Space-Efficient Contracts

Figure 8.2 presents the syntax of the space-efficient annotation language, $\mathscr{A}se :\equiv (A, s)$. In its syntax, an annotation A is just a contract ${}^{\mathsf{se}}\kappa$ where the metavariable ${}^{\mathsf{se}}\kappa$ ranges over space-efficient contracts. The global state s is either $\textsc{Ok}$ or $\textsc{Err}(\ell)$, which is the same as the contract annotation language $\mathscr{A}ctc$ from Chapter 6.

The syntax of space-efficient contract ${}^{\text{se}}\kappa$ has one constructor for each type. For the nat type, the space-efficient contract ${}^{\text{se}}\kappa$ is a list of predicates over the input. For other types, the constructors of ${}^{\text{se}}\kappa$ follow the recursive definition of the corresponding types. For example, a space-efficient contract for pairs of type $\tau_1 \times \tau_2$ is ${}^{\text{se}}\kappa_1 \times\!\!/\text{se}\ {}^{\text{se}}\kappa_2$ for some ${}^{\text{se}}\kappa_1 : \text{SECtc}\,\tau_1$ and ${}^{\text{se}}\kappa_2 : \text{SECtc}\,\tau_2$ where $\text{SECtc}\,\tau$ denotes the type of space-efficient contracts for values of type τ. For consistency, the names of the constructors of the space-efficient contracts have the suffix "/se".

Next, Lemmas 8.1 and 8.3 and Proposition 8.2 present the Substitution lemmas for space-efficient contracts

Lemma 8.1 (Renaming)**.** *Let* $\Delta :\equiv \{t_1, \ldots, t_n\}$ *and* $\Delta' :\equiv \{t'_1, \ldots, t'_m\}$ *be given. Assume that there is a sequence* $1 \le a_i \le m$ *for* $i = 1 \ldots n$*. If* $\Delta \vdash^{\text{se}} {}^{\text{se}}\kappa : \text{SECtc}\,\tau$ *then* $\Delta' \vdash^{\text{se}} {}^{\text{se}}\kappa\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right] : \text{SECtc}\left(\tau\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right]\right)$.

Proposition 8.2. *Assume that* $\Delta' \vdash^{\text{se}} \kappa_i : \text{SECtc}\,\tau_i$ *for* $i = 1 \ldots n$ *and let any* $t_0 \notin \Delta'$ *be given. There is a sequence* $\Delta'' \vdash^{\text{se}} {}^{\text{se}}\kappa_i : \text{SECtc}\,\tau_i$ *for* $i = 0 \ldots n$ *where* $\Delta'' :\equiv \Delta', t_0$ *and* ${}^{\text{se}}\kappa_0 :\equiv t_0$.

Lemma 8.3 (Substitution)**.** *Let* $\Delta :\equiv \{t_1, \ldots, t_n\}$ *and some* Δ' *be given. Assume that* $\Delta' \vdash^{\text{se}} {}^{\text{se}}\kappa_i : \text{SECtc}\,\tau_i$ *for* $i = 1 \ldots n$*. If* $\Delta \vdash^{\text{se}} {}^{\text{se}}\kappa : \text{SECtc}\,\tau$ *then*

$$\Delta' \vdash^{\text{se}} {}^{\text{se}}\kappa[\kappa_1, \ldots, \kappa_n \,/\, t_1, \ldots, t_n] : \text{SECtc}\,(\tau[\tau_1, \ldots, \tau_n \,/\, t_1, \ldots, t_n])$$

Figure 8.4 displays the transition steps, $\mathcal{T}s$, of the space-efficient contracts. In its definition, $\lambda_{\mathsf{m}}[\mathcal{A}se;\mathcal{T}s]$ relies on a decidable predicate *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, except that the Racket implementation always returns true when the predicates have closures at the same location in

$$
\begin{array}{rl}
\text{[R-Cross-Unit]} & \textsc{Ok}, \mathbf{B}\#\mathsf{unit}/\!\!\text{\textae}\ \{\ ()\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, ()\\
\text{[R-Cross-Nat]} & \textsc{Ok}, \mathbf{B}\#[\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)]\ \{\ n\ \} \longrightarrow_{\mathrm{m}} s', n\\
\text{where} & (\textsc{Ok}, s') \in \mathit{checkCtcs}_{\emptyset,\,\mathit{get},\,\mathit{put}}([\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)],\ n)\\
\text{[R-Cross-Cons]} & \textsc{Ok}, \mathbf{B}\#(\text{\textae}\kappa_1 \times/\!\!\text{\textae}\ \text{\textae}\kappa_2)\ \{\ \langle v_1, v_2\rangle\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \langle\ \mathbf{B}\#\text{\textae}\kappa_1\ \{v_1\}, \mathbf{B}\#\text{\textae}\kappa_2\ \{v_2\}\ \rangle\\
\text{[R-Cross-Inl]} & \textsc{Ok}, \mathbf{B}\#(\text{\textae}\kappa_1 +/\!\!\text{\textae}\ \text{\textae}\kappa_2)\ \{\ \mathsf{inl}(v)\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{inl}(\mathbf{B}\#\text{\textae}\kappa_1\ \{v\})\\
\text{[R-Cross-Inr]} & \textsc{Ok}, \mathbf{B}\#(\text{\textae}\kappa_1 +/\!\!\text{\textae}\ \text{\textae}\kappa_2)\ \{\ \mathsf{inr}(v)\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{inr}(\mathbf{B}\#\text{\textae}\kappa_2\ \{v\})\\
\text{[R-Cross-Roll]} & \textsc{Ok}, \mathbf{B}\#(\mu/\!\!\text{\textae}\ t.\text{\textae}\kappa)\ \{\ \mathsf{roll}_\tau(v)\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{roll}_\tau(\mathbf{B}\#\ (\text{\textae}\kappa[(\mu/\!\!\text{\textae}\ t.\text{\textae}\kappa)\ /\ t])\ \{v\})\\
\text{[R-Cross-Box]} & \textsc{Ok}, \mathbf{B}\#\text{\textae}\kappa\ \{\ \mathsf{box}(v)\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{proxy}(\text{\textae}\kappa,\ \mathsf{box}(v))\\
\text{[R-Cross-Lam]} & \textsc{Ok}, \mathbf{B}\#\text{\textae}\kappa\ \{\ \lambda x.e\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{proxy}(\text{\textae}\kappa,\ \lambda x.e)\\
\text{[R-Proxy-Unbox]} & \textsc{Ok}, \mathsf{unbox}(\mathsf{proxy}(\mathsf{box}/\!\!\text{\textae}\ \text{\textae}\kappa,\ \mathsf{box}(e))) \longrightarrow_{\mathrm{m}}\\
& \textsc{Ok}, \mathbf{B}\#\text{\textae}\kappa\ \{\ \mathsf{unbox}(\mathsf{box}(e))\ \}\\
\text{[R-Proxy-}\beta\text{]} & \textsc{Ok}, \mathsf{proxy}(\text{\textae}\kappa_a \to/\!\!\text{\textae}\ \text{\textae}\kappa_r,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathbf{B}\#\text{\textae}\kappa_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#\text{\textae}\kappa_a\ \{v\})\ \}\\
\text{[R-Merge-Box]} & \textsc{Ok}, \mathbf{B}\#\text{\textae}\kappa_1\ \{\ \mathsf{proxy}(\text{\textae}\kappa_2,\ \mathsf{box}(e))\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{proxy}(\text{\textae}\kappa,\ \mathsf{box}(e))\\
\text{where} & \text{\textae}\kappa :\equiv \mathsf{evalTick}[\![\ \checkmark\ \mathit{join}(\text{\textae}\kappa_2, \text{\textae}\kappa_1)\ ]\!]\\
\text{[R-Merge-Lam]} & \textsc{Ok}, \mathbf{B}\#\text{\textae}\kappa_1\ \{\ \mathsf{proxy}(\text{\textae}\kappa_2,\ \lambda x.e)\ \} \longrightarrow_{\mathrm{m}} \textsc{Ok}, \mathsf{proxy}(\text{\textae}\kappa,\ \lambda x.e)\\
\text{where} & \text{\textae}\kappa :\equiv \mathsf{evalTick}[\![\ \checkmark\ \mathit{join}(\text{\textae}\kappa_2, \text{\textae}\kappa_1)\ ]\!]
\end{array}
$$

Figure 8.4: The transition steps, $\mathscr{T}s$, of space-efficient contracts

memory, rather than comparing their bodies.

The reduction rules $\mathscr{T}s$ are similar to $\mathscr{T}c_i$. For example, the [R-Cross-Nat] invokes the *checkCtcs* metafunction from Figure 6.5 on page 92 to check the list of predicates annotated on the boundary. The *checkCtcs* metafunction computes the the transition of the global states, $(\textsc{Ok}, s')$, and the [R-Cross-Nat] updates the configuration accordingly. The subscript that $\mathscr{T}s$ supplies to *checkCtcs* is an empty relation, $\emptyset$, which effectively disallows *any* nested contract checks during the evaluation of the predicates. This simplifies the proofs but still allows custom predicates to be used, except that the custom predicates cannot contain other contracts.

The other [R-Cross] rules decompose and distribute pieces of the space-efficient contracts in the conventional manner. In the [R-Cross-Cons] rule, the contract $\text{\textae}\kappa_1 \times/\!\!\text{\textae}\ \text{\textae}\kappa_2$ on $\langle v_1, v_2\rangle$ is broken down into $\text{\textae}\kappa_1$ and $\text{\textae}\kappa_2$ on v_1 and v_2. Similarly, the [R-Cross-Inl] and [R-Cross-Inr] rules break down $\text{\textae}\kappa_1 +/\!\!\text{\textae}\ \text{\textae}\kappa_2$ and annotate either $\text{\textae}\kappa_1$ or $\text{\textae}\kappa_2$ on the new boundary. The [R-Cross-Roll]

Tick	:	$\mathcal{U} \to \mathcal{U}$	A definition of (the meta-level) type Tick A produces a value of type A while tracking the total number of ticks.
✓	:	Tick $A \to$ Tick A	The ✓ operation increments the tick by one.
evalTick	:	Tick $A \to A$	The evalTick function executes a Tick computation and extracts the final result.
execTick	:	Tick $A \to \mathbb{N}$	The execTick function executes a Tick computation and returns the total tick count.

Figure 8.5: The operation of the Tick monad in the meta-language

rule expands the recursive contract. Finally, the [R-Cross-Box] and [R-Cross-Lam] rules attach the given contract to the new proxy.

Next, the [R-Proxy-Unbox] rule extracts the contract ${}^{\text{se}}\kappa$ of the stored value from the contract box$/_{\text{se}}\,{}^{\text{se}}\kappa$ and pushes the unbox operation into the boundary. Following the same pattern, the [R-Proxy-β] rule extracts the contracts ${}^{\text{se}}\kappa_a$ and ${}^{\text{se}}\kappa_r$ from the arrow contract ${}^{\text{se}}\kappa_a \to /_{\text{se}}\,{}^{\text{se}}\kappa_r$, and puts them separately on the boundaries around the argument and the result of the function application.

Last, the [R-Merge] rules are what make space-efficient contracts different from the ordinary contracts in Chapter 6. When a proxy reaches a boundary, these rules merge the space-efficient contract annotated on the boundary with the contract annotated on the proxy using the *join* metafunction. Figures 8.6 and 8.7 display the definitions of *join*, *joinFlats*, and *drop* which work with each other to remove redundant predicates from the list of predicates in the leaves.

Figure 8.6 shows the signature of *join* and the two auxiliary functions that handle the actual removal of the duplication. All three of *join*, *joinFlats*, and *drop* are written in monadic style using the Tick monad displayed from Figure 8.5. The Tick monad is adapted from Danielsson [2008]'s *Thunk* monad and it comes an internal *tick counter* together with three operations: the ✓ operation increments the internal counter by 1; evalTick executes the computation and returns the final result; execTick returns the count of the ticks (✓) that a computation uses. The functions generally work like their purely functional variants except that the definitions automatically track the total number of ✓ operations, which represents the count of primitive operations that the functions need. For example, evalTick$[\![$ ✓ $\mathit{join}({}^{\text{se}}\kappa_2, {}^{\text{se}}\kappa_1)\,]\!]$ computes the merging result of ${}^{\text{se}}\kappa_2$ and

$$
\begin{array}{l}
\mathit{join} \;:\; \mathsf{SECtc}\,\tau \to \mathsf{SECtc}\,\tau \to \mathsf{Tick}\,(\mathsf{SECtc}\,\tau) \\
\mathit{join}(^{\mathsf{se}}\kappa, {}^{\mathsf{se}}\kappa') \;\equiv\; \ldots \qquad \text{(in Figure 8.7)}
\end{array}
$$

$$
\begin{array}{l}
\mathit{drop} \;:\; \mathsf{List}\,(\mathsf{SECtc}\,\mathsf{nat}) \to \mathsf{Expr}\,\mathsf{nat} \to \mathsf{Tick}\,(\mathsf{List}\,(\mathsf{SECtc}\,\mathsf{nat})) \\
\mathit{drop}([\,], \qquad\qquad\qquad e) \;\equiv\; \checkmark\,\mathbf{return}\;[\,] \\
\mathit{drop}([\mathsf{flat}^{\ell}(x.\,e'), \kappa_2, \ldots, \kappa_m], e) \;\equiv\; \mathbf{do} \\
\quad b \leftarrow \checkmark\,\mathit{isStronger}(e,\, e') \\
\quad \mathbf{if}\; b \\
\quad\quad \mathbf{then}\; \checkmark\,\mathit{drop}([\kappa_2, \ldots, \kappa_m], e) \\
\quad\quad \mathbf{else\ do}\; \mathit{collapsedPreds} \leftarrow \checkmark\,\mathit{drop}([\kappa_2, \ldots, \kappa_m], e) \\
\quad\quad\quad\quad \checkmark\,\mathbf{return}\;\; \mathsf{flat}^{\ell}(x.\,e') :: \mathit{collapsedPreds}
\end{array}
$$

$$
\begin{array}{l}
\mathit{joinFlats} \;:\; \mathsf{List}\,(\mathsf{SECtc}\,\mathsf{nat}) \to \mathsf{List}\,(\mathsf{SECtc}\,\mathsf{nat}) \to \mathsf{Tick}\,(\mathsf{List}\,(\mathsf{SECtc}\,\mathsf{nat})) \\
\mathit{joinFlats}([\,], \qquad\qquad\qquad [\kappa_1, \ldots, \kappa_m]) \;\equiv\; \checkmark\,\mathbf{return}\;[\kappa_1, \ldots, \kappa_m] \\
\mathit{joinFlats}([\mathsf{flat}^{\ell'}(x.\,e), \kappa'_2, \ldots, \kappa'_k], [\kappa_1, \ldots, \kappa_m]) \;\equiv\; \mathbf{do} \\
\quad \mathit{mergedPreds} \leftarrow \checkmark\,\mathit{joinFlats}([\kappa'_2, \ldots, \kappa'_k], [\kappa_1, \ldots, \kappa_m]) \\
\quad \mathit{collapsedPreds} \leftarrow \checkmark\,\mathit{drop}(\mathit{mergedPreds}, e) \\
\quad \checkmark\,\mathbf{return}\;\; \mathsf{flat}^{\ell'}(x.\,e) :: \mathit{collapsedPreds}
\end{array}
$$

Figure 8.6: The *joinFlats* function that removes redundant predicates.

$^{\mathsf{se}}\kappa_1$ in rule [R-Merge-Box] and rule [R-Merge-Lam], and $\mathsf{execTick}[\![\, \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_2, {}^{\mathsf{se}}\kappa_1) \,]\!]$ returns the number of primitive operations needed by *join*.

The *join* metafunction is adapted from Greenberg [2016]'s work. Since it simply traverses the given inputs and calls *joinFlats* when reaching the leaves, I shall focus on the definitions of *joinFlats* first. Specifically, *joinFlats* takes two lists of flat contracts and, for each in the first list, calls *drop* on the result of recursion to filter out the predicates that *isStronger* reports as redundant.

Finally, Figure 8.7 gives the definition of the *join* metafunction. It takes two space-efficient contracts, and returns a space-efficient contract that exhibits the same contract-checking behavior as checking the two input contracts in order. Since space-efficient contracts are typed in my framework, *join* only needs to consider merging contracts that have the same constructor. Specifically, the first clause $\mathit{join}(t, t)$ in Figure 8.7 handles the *variable case* of space-efficient contracts, not matching arbitrary equal inputs. As one would expect, *join* simply traverses its inputs and calls *joinFlats* upon reaching the nat type case.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{join} \;:\; \mathsf{SECtc}\,\tau \to \mathsf{SECtc}\,\tau \to \mathsf{Tick}\,(\mathsf{SECtc}\,\tau)} \\
\mathit{join}(t, & t) & \equiv \;\checkmark\,\mathbf{return}\; t \\
\mathit{join}(\mathsf{unit}/\!\!\mathsf{se}, & \mathsf{unit}/\!\!\mathsf{se}) & \equiv \;\checkmark\,\mathbf{return}\; \mathsf{unit}/\!\!\mathsf{se} \\
\mathit{join}([\kappa'_1, \ldots, \kappa'_k], & [\kappa_1, \ldots, \kappa_m]) & \equiv \;\checkmark\,\mathit{joinFlats}([\kappa'_1, \ldots, \kappa'_k], [\kappa_1, \ldots, \kappa_m]) \\
\mathit{join}((^{\mathsf{se}}\kappa_1 \times\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_3), & (^{\mathsf{se}}\kappa_2 \times\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_4)) & \equiv \;\mathbf{do}\; ^{\mathsf{se}}\kappa_l \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_1, ^{\mathsf{se}}\kappa_2) \\
 & & \quad\; ^{\mathsf{se}}\kappa_r \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_3, ^{\mathsf{se}}\kappa_4) \\
 & & \quad\; \checkmark\,\mathbf{return}\;\; ^{\mathsf{se}}\kappa_l \times\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_r \\
\mathit{join}((^{\mathsf{se}}\kappa_1 +\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_3), & (^{\mathsf{se}}\kappa_2 +\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_4)) & \equiv \;\mathbf{do}\; ^{\mathsf{se}}\kappa_l \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_1, ^{\mathsf{se}}\kappa_2) \\
 & & \quad\; ^{\mathsf{se}}\kappa_r \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_3, ^{\mathsf{se}}\kappa_4) \\
 & & \quad\; \checkmark\,\mathbf{return}\;\; ^{\mathsf{se}}\kappa_l +\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_r \\
\mathit{join}(\mathsf{box}/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_1, & \mathsf{box}/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_2) & \equiv \;\mathbf{do}\; ^{\mathsf{se}}\kappa \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_1, ^{\mathsf{se}}\kappa_2) \\
 & & \quad\; \checkmark\,\mathbf{return}\;\; \mathsf{box}/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa \\
\mathit{join}((^{\mathsf{se}}\kappa_1 \to\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_3), & (^{\mathsf{se}}\kappa_2 \to\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_4)) & \equiv \;\mathbf{do}\; ^{\mathsf{se}}\kappa_a \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_2, ^{\mathsf{se}}\kappa_1) \\
 & & \quad\; ^{\mathsf{se}}\kappa_r \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_3, ^{\mathsf{se}}\kappa_4) \\
 & & \quad\; \mathbf{return}\;\; ^{\mathsf{se}}\kappa_a \to\!/\!\!\mathsf{se}\; ^{\mathsf{se}}\kappa_r \\
\mathit{join}((\mu/\!\!\mathsf{se}\; t.^{\mathsf{se}}\kappa_1), & (\mu/\!\!\mathsf{se}\; t.^{\mathsf{se}}\kappa_2)) & \equiv \;\mathbf{do}\; ^{\mathsf{se}}\kappa \leftarrow \checkmark\,\mathit{join}(^{\mathsf{se}}\kappa_1, ^{\mathsf{se}}\kappa_2) \\
 & & \quad\; \checkmark\,\mathbf{return}\;\; \mu/\!\!\mathsf{se}\; t.^{\mathsf{se}}\kappa
\end{array}
$$

Figure 8.7: The *join* function that merges space-efficient contracts.

8.3 Interlude: Size Parameters of Space-Efficient Contracts

Since the space and time complexity of $\lambda_{\mathsf{m}}[\mathscr{A}\mathit{se};\mathscr{T}\mathit{s}]$ depends on the height of contracts and the length of predicates in the leaves of contracts, I need to introduce several auxiliary definitions for bounding the height and the length of predicates of a contract. Moreover, the complexity results only hold in the *absence* of recursive contracts, so another judgement is needed. In this section, I give the definition of the helper judgements.

Capturing Non-recursive Contracts. Figure 8.8 defines the judgement NonRec that captures non-recursive contracts. Concretely, $\mathsf{NonRec}(^{\mathsf{se}}\kappa)$ holds if and only if there are no recursive contracts (and thus no variables) anywhere in $^{\mathsf{se}}\kappa$.

The Height of Space-Efficient Contracts. Figure 8.9 defines the MaxHt judgement that captures all contracts whose height is smaller than or equal to the given bound. That is, the judgement $\mathsf{MaxHt}(^{\mathsf{se}}\kappa, h)$ holds if and only if the height of $^{\mathsf{se}}\kappa$ is at most h. It can be related to the

$$\boxed{\mathsf{NonRec}(\text{ᵉ}\kappa)}$$

$$\frac{}{\mathsf{NonRec}(\mathsf{unit}/\text{ᵉ})} \qquad \frac{}{\mathsf{NonRec}([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)])} \qquad \frac{\mathsf{NonRec}(\text{ᵉ}\kappa_1) \qquad \mathsf{NonRec}(\text{ᵉ}\kappa_2)}{\mathsf{NonRec}(\text{ᵉ}\kappa_1 \times/\text{ᵉ}\ \text{ᵉ}\kappa_2)}$$

$$\frac{\mathsf{NonRec}(\text{ᵉ}\kappa_1) \qquad \mathsf{NonRec}(\text{ᵉ}\kappa_2)}{\mathsf{NonRec}(\text{ᵉ}\kappa_1 +/\text{ᵉ}\ \text{ᵉ}\kappa_2)} \qquad \frac{\mathsf{NonRec}(\text{ᵉ}\kappa)}{\mathsf{NonRec}(\mathsf{box}/\text{ᵉ}\ \text{ᵉ}\kappa)} \qquad \frac{\mathsf{NonRec}(\text{ᵉ}\kappa_a) \qquad \mathsf{NonRec}(\text{ᵉ}\kappa_r)}{\mathsf{NonRec}(\text{ᵉ}\kappa_a \rightarrow/\text{ᵉ}\ \text{ᵉ}\kappa_r)}$$

Figure 8.8: Identifying non-recursive contracts.

$$\boxed{\mathsf{MaxHt}(\text{ᵉ}\kappa,\ h) \text{ where } \text{ᵉ}\kappa : \mathsf{SECtc}\ \tau \text{ and } h : \mathbb{N}}$$

$$\frac{}{\mathsf{MaxHt}(t,\ h)} \qquad \frac{}{\mathsf{MaxHt}(\mathsf{unit}/\text{ᵉ},\ h)} \qquad \frac{}{\mathsf{MaxHt}([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)],\ h)}$$

$$\frac{\mathsf{MaxHt}(\text{ᵉ}\kappa_1,\ h) \qquad \mathsf{MaxHt}(\text{ᵉ}\kappa_2,\ h)}{\mathsf{MaxHt}(\text{ᵉ}\kappa_1 \times/\text{ᵉ}\ \text{ᵉ}\kappa_2,\ 1+h)} \qquad \frac{\mathsf{MaxHt}(\text{ᵉ}\kappa_1,\ h) \qquad \mathsf{MaxHt}(\text{ᵉ}\kappa_2,\ h)}{\mathsf{MaxHt}(\text{ᵉ}\kappa_1 +/\text{ᵉ}\ \text{ᵉ}\kappa_2,\ 1+h)}$$

$$\frac{\mathsf{MaxHt}(\text{ᵉ}\kappa,\ h)}{\mathsf{MaxHt}(\mathsf{box}/\text{ᵉ}\ \text{ᵉ}\kappa,\ 1+h)} \qquad \frac{\mathsf{MaxHt}(\text{ᵉ}\kappa_a,\ h) \qquad \mathsf{MaxHt}(\text{ᵉ}\kappa_r,\ h)}{\mathsf{MaxHt}(\text{ᵉ}\kappa_a \rightarrow/\text{ᵉ}\ \text{ᵉ}\kappa_r,\ 1+h)} \qquad \frac{\mathsf{MaxHt}(\text{ᵉ}\kappa,\ h)}{\mathsf{MaxHt}((\mu/\text{ᵉ}\ t.\ \text{ᵉ}\kappa),\ 1+h)}$$

Figure 8.9: The height of space-efficient contracts.

conventional definition of the height$(\cdot)$ function over contracts. Specifically, given the definition

$$\begin{array}{lcl}
\mathsf{height}\ :\ \mathsf{SECtc}\ \tau \to \mathbb{N} & & \\
\mathsf{height}(t) & \equiv & 0 \\
\mathsf{height}(\mathsf{unit}/\text{ᵉ}) & \equiv & 0 \\
\mathsf{height}\big([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)]\big) & \equiv & 0 \\
\mathsf{height}(\text{ᵉ}\kappa_1 \times/\text{ᵉ}\ \text{ᵉ}\kappa_2) & \equiv & 1 + \max\left(\mathsf{height}(\text{ᵉ}\kappa_1), \mathsf{height}(\text{ᵉ}\kappa_2)\right) \\
\mathsf{height}(\text{ᵉ}\kappa_1 +/\text{ᵉ}\ \text{ᵉ}\kappa_2) & \equiv & 1 + \max\left(\mathsf{height}(\text{ᵉ}\kappa_1), \mathsf{height}(\text{ᵉ}\kappa_2)\right) \\
\mathsf{height}(\mathsf{box}/\text{ᵉ}\ \text{ᵉ}\kappa) & \equiv & 1 + \mathsf{height}(\text{ᵉ}\kappa) \\
\mathsf{height}(\text{ᵉ}\kappa_a \rightarrow/\text{ᵉ}\ \text{ᵉ}\kappa_r) & \equiv & 1 + \max\left(\mathsf{height}(\text{ᵉ}\kappa_a), \mathsf{height}(\text{ᵉ}\kappa_r)\right) \\
\mathsf{height}(\mu/\text{ᵉ}\ t.\text{ᵉ}\kappa) & \equiv & 1 + \mathsf{height}(\text{ᵉ}\kappa)
\end{array}$$

it is the case that $\mathsf{MaxHt}(\text{ᵉ}\kappa,\ \mathsf{height}(\text{ᵉ}\kappa))$ holds for any $\text{ᵉ}\kappa$ and, conversely, if $\mathsf{MaxHt}(\text{ᵉ}\kappa,\ h)$ for some h then $\mathsf{height}(\text{ᵉ}\kappa) \leq h$.

When there are no recursive contracts, I can prove that the height of contracts in a program is non-increasing. Formally, if H is the height of the tallest contract in an initial program, I can define an interpretation $\mathcal{I}ht$ that bounds the height of all contracts by H.

$$\boxed{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa) \text{ where } J : \mathsf{List}\,(\mathsf{Expr\,nat}) \to \mathcal{U} \text{ and } {}^{\mathfrak{se}}\kappa : \mathsf{SECtc}\,\tau}$$

$$\frac{J([e_1, \ldots, e_m])}{\mathsf{AllFlats}(J, [\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)])} \qquad \frac{}{\mathsf{AllFlats}(J, t)} \qquad \frac{}{\mathsf{AllFlats}(J, \mathsf{unit}/\mathfrak{se})}$$

$$\frac{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_1) \qquad \mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_2)}{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_1 \times/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_2)} \qquad \frac{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_1) \qquad \mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_2)}{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_1 +/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_2)}$$

$$\frac{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa)}{\mathsf{AllFlats}(J, \mathsf{box}/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa)} \qquad \frac{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_a) \qquad \mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_r)}{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa_a \to/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_r)} \qquad \frac{\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa)}{\mathsf{AllFlats}(J, (\mu/\mathfrak{se}\, t.\, {}^{\mathfrak{se}}\kappa))}$$

$$\boxed{\mathsf{NonEmpty}(xs) \text{ where } xs : \mathsf{List}\, C} \qquad \boxed{\mathsf{UniqSub}(xs, ys) \text{ where } xs\, ys : \mathsf{List}\, C}$$

$$\frac{}{\mathsf{NonEmpty}(x :: xs)} \qquad \frac{xs +\!\!+ xs^c = zs \qquad zs \leftrightsquigarrow ys}{\mathsf{UniqSub}(xs, ys)}$$

Figure 8.10: Constraining the list of predicates in space-efficient contracts.

Definition 8.4. The annotation interpretation $\mathcal{I}ht :\equiv (\mathbb{S}_{ht},\ \preccurlyeq_{ht},\ \mathbb{B},\ \mathbb{P})$ is defined as

$$\begin{array}{lcl}
\mathbb{S}_{ht}(s) & :\equiv & \top \\
s \preccurlyeq_{ht} s' & :\equiv & \top \\
\mathbb{B}[\![\ {}^{\mathfrak{se}}\kappa, e\ ]\!] & :\equiv & \mathsf{NonRec}({}^{\mathfrak{se}}\kappa) \wedge \mathsf{MaxHt}({}^{\mathfrak{se}}\kappa,\ H) \\
\mathbb{P}[\![\ {}^{\mathfrak{se}}\kappa, e^m\ ]\!] & :\equiv & \mathbb{B}[\![\ {}^{\mathfrak{se}}\kappa, e^m\ ]\!]
\end{array}$$

Theorem 8.5. *The interpretation $\mathcal{I}ht$ is monotonic and sound.*

Bounding Lengths of Predicates in Space-Efficient Contracts. To bound the lengths of the list of flat predicates in a contract, I introduce $\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa)$ in Figure 8.10, a judgement which takes another judgement J and applies it to all leaves of a contract. That is, $\mathsf{AllFlats}(J, {}^{\mathfrak{se}}\kappa)$ holds if and only if $J([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)])$ holds for all list of predicates in ${}^{\mathfrak{se}}\kappa$. For example, $\mathsf{NonEmpty}(xs)$ asserts that the given list is non-empty, hence $\mathsf{AllFlats}(\mathsf{NonEmpty}, {}^{\mathfrak{se}}\kappa)$ means that all leaves of ${}^{\mathfrak{se}}\kappa$ contain at least one predicate. As another example, $\mathsf{UniqSub}(xs, ys)$ takes two lists and asserts that xs contains only the distinct elements of ys. In its definition, xs^c and ys are existentially quantified, and $zs \leftrightsquigarrow ys$ means that zs is a permutation of ys. Essentially, UniqSub is formalized by making sure that xs is a prefix of a permutation of ys. The UniqSub judgement is particularly useful for proving space efficiency.

8.4 Space Efficiency

With the help of the judgements for bounding the height and the length of predicates introduced in Section 8.3, I can present the proof of space efficiency for the instantiation $\lambda_{\mathsf{m}}[\mathscr{A}se;\mathscr{T}s]$. Let $\mathsf{size}(\cdot)$ be the function that computes the size of a contract defined as follows:

$$
\begin{array}{lcl}
\mathsf{size} \;:\; \mathsf{SECtc}\,\tau \to \mathbb{N} & & \\
\mathsf{size}(t) & \equiv & 1 \\
\mathsf{size}(\mathsf{unit}/\mathfrak{se}) & \equiv & 1 \\
\mathsf{size}\big([\mathsf{flat}^{\ell_1}(x.\,e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\,e_m)]\big) & \equiv & m \\
\mathsf{size}({}^{\mathfrak{se}}\kappa_1 \times/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_2) & \equiv & 1 + \mathsf{size}({}^{\mathfrak{se}}\kappa_1) + \mathsf{size}({}^{\mathfrak{se}}\kappa_2) \\
\mathsf{size}({}^{\mathfrak{se}}\kappa_1 +/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_2) & \equiv & 1 + \mathsf{size}({}^{\mathfrak{se}}\kappa_1) + \mathsf{size}({}^{\mathfrak{se}}\kappa_2) \\
\mathsf{size}(\mathsf{box}/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa) & \equiv & 1 + \mathsf{size}({}^{\mathfrak{se}}\kappa) \\
\mathsf{size}({}^{\mathfrak{se}}\kappa_a \to/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_r) & \equiv & 1 + \mathsf{size}({}^{\mathfrak{se}}\kappa_a) + \mathsf{size}({}^{\mathfrak{se}}\kappa_r) \\
\mathsf{size}(\mu/\mathfrak{se}\ t.{}^{\mathfrak{se}}\kappa) & \equiv & 1 + \mathsf{size}({}^{\mathfrak{se}}\kappa)
\end{array}
$$

Then, with the indexed interpretation $\mathcal{I}size_C$ I shall prove that, if H is the height of the tallest contract in the initial program, and that $\mathcal{K}$ is similarly the set of all distinct predicates, the size of all contracts are always bounded by $O(|\mathcal{K}| \cdot 2^H)$.

Definition 8.6. The indexed interpretation $\mathcal{I}size_C :\equiv (\mathbb{N}, \{\lhd_{C,A}\}_A, \mathbb{S}, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is defined as

$$
\begin{array}{lcl}
m \lhd_{C,A} m' & :\equiv & C = m \wedge C = m' \\
\mathbb{S}(s) & :\equiv & \top \\
s \preccurlyeq s' & :\equiv & \top \\
\mathbb{B}[\![\, {}^{\mathfrak{se}}\kappa^{C,C}, e \,]\!] & :\equiv & \mathsf{NonRec}({}^{\mathfrak{se}}\kappa) \wedge \\
 & & \mathsf{AllFlats}(\mathsf{UniqSub}(-, \mathcal{K}), {}^{\mathfrak{se}}\kappa) \wedge \mathsf{MaxHt}({}^{\mathfrak{se}}\kappa,\, H) \wedge \\
 & & \mathsf{size}({}^{\mathfrak{se}}\kappa) \leq C \cdot 2^H \cdot |\mathcal{K}| \\
\mathbb{P}[\![\, {}^{\mathfrak{se}}\kappa^{C,C}, e^m \,]\!] & :\equiv & \mathbb{B}[\![\, {}^{\mathfrak{se}}\kappa^{C,C}, e^m \,]\!]
\end{array}
$$

Theorem 8.7. *There exists $c_0 > 0$ such that the indexed interpretation $\mathcal{I}size_{c_0}$ is monotonic and sound.*

8.5 The Time Complexity of Space-Efficient Contracts

Tracking Run-time Cost of Contracts. Figure 8.11 displays the instantiation $\lambda_{\mathsf{m}}[\mathscr{A}ccs;\mathscr{T}ccs]$ that I use for proving time complexity bounds for space-efficient contracts. To reason about time

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

$$
\begin{array}{rl}
\text{[R-Cross-Inl]} & \langle \textsc{Ok}, c, w, k\rangle, \mathbf{B}\#\,({}^{\mathfrak{se}}\kappa_1 \not\rightarrow\!\!\!\!{}^{\mathfrak{se}}\ {}^{\mathfrak{se}}\kappa_2)\ \{\ \mathsf{inl}(v)\ \} \longrightarrow_{\mathrm{m}} \\
& \langle \textsc{Ok}, c, w, k+1\rangle, \mathsf{inl}(\mathbf{B}\#{}^{\mathfrak{se}}\kappa_1\ \{v\}) \\
\text{[R-Proxy-}\beta\text{]} & \langle \textsc{Ok}, c, w, k\rangle, \mathsf{proxy}({}^{\mathfrak{se}}\kappa_a \not\rightarrow\!\!\!\!{}^{\mathfrak{se}}\ {}^{\mathfrak{se}}\kappa_r,\ \lambda x.e)\ v \longrightarrow_{\mathrm{m}} \\
& \langle \textsc{Ok}, c, w, k+1\rangle, \mathbf{B}\#{}^{\mathfrak{se}}\kappa_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#{}^{\mathfrak{se}}\kappa_a\ \{v\})\ \} \\
\text{[R-Merge-Lam]} & \langle \textsc{Ok}, c, w, k\rangle, \mathbf{B}\#{}^{\mathfrak{se}}\kappa_1\ \{\ \mathsf{proxy}({}^{\mathfrak{se}}\kappa_2,\ \lambda x.e)\ \} \longrightarrow_{\mathrm{m}} \\
& \langle \textsc{Ok}, c, w + w_{join}, k+1\rangle, \mathsf{proxy}({}^{\mathfrak{se}}\kappa,\ \lambda x.e) \\
\text{where} & {}^{\mathfrak{se}}\kappa :\equiv \mathsf{evalTick}[\![\ \checkmark\, join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)\ ]\!] \\
& w_{join} :\equiv \mathsf{execTick}[\![\ \checkmark\, join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)\ ]\!] \\
\text{[R-Cross-Nat]} & \langle \textsc{Ok}, c, w, k\rangle, \mathbf{B}\#[\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)]\ \{\ n\ \} \longrightarrow_{\mathrm{m}} \\
& \langle st', c' + m, w', k' + 1\rangle, n \\
\text{where} & (\langle \textsc{Ok}, c, w, k\rangle, \langle st', c', w', k'\rangle) \in \\
& \qquad \mathit{checkCtcs}_{\emptyset,\, get,\, put}([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)],\ n) \\
& \mathit{get}(\langle st, c, w, k\rangle) :\equiv s \\
& \mathit{put}(\langle st, c, w, k\rangle, st') :\equiv \langle st', c, w, k\rangle
\end{array}
$$

The definition of the annotation language $(\mathscr{A}ccs, \mathscr{T}ccs)$ that counts contract checks and the primitive operations *join* uses where $\mathscr{A}ccs :\equiv (A, s)$.

Figure 8.11: The annotation language $(\mathscr{A}ccs, \mathscr{T}ccs)$ that tracks various run-time costs.

complexity, $\lambda_{\mathrm{m}}[\mathscr{A}ccs;\mathscr{T}ccs]$ extends $\lambda_{\mathrm{m}}[\mathscr{A}se;\mathscr{T}s]$ with three counters in the global states for tracking run time cost: 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.

For each rule in the transition steps, $\mathscr{T}ccs$ increments k by 1. In rule [R-Cross-Nat], $\mathscr{T}ccs$ adds m to the counter c since the rule checks m more flat contracts. In rule [R-Merge-Box] and [R-Merge-Lam], in addition to merging the two contracts with $\mathsf{evalTick}[\![\ \checkmark\, join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)\ ]\!]$, the two rules call the metafunction $\mathsf{execTick}[\![\ \checkmark\, join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)\ ]\!]$ to calculate the number of primitive operations needed for computing $join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)$ and keep the sum in the counter w. Here, I place the tick operation (✓) following Danielsson [2008]'s convention (or, as Charguéraud and Pottier [2019] argue, placing a tick operation at the entry of every function is another reasonable option).

Bounding the Number of (Flat) Contract Checks. To show that adding contracts does not incur exponential slowdown in theory, it is necessary to prove that the contract system does not introduce an enormous amount of contract checks. Fortunately, this is actually a consequence of space efficiency: since $\lambda_{\mathsf{m}}[\mathscr{A}se;\mathscr{T}s]$ removes duplicate predicates from all contracts appearing in the program, every leaf of any contract can store at most $|\mathcal{K}|$ distinct predicates in the list. Therefore, each boundary of type nat needs to check no more than $|\mathcal{K}|$ predicates, which is merely a constant.

Proving this fact in the framework through the instantiation $\lambda_{\mathsf{m}}[\mathscr{A}ccs;\mathscr{T}ccs]$ is straightforward. Since the monitor calculus checks the whole list of predicates in one monitor-related reduction step, I only need to define the interpretation $\mathscr{I}chkbnd$ that holds the invariant $c \leq k \cdot |\mathcal{K}|$:

Definition 8.8. The annotation interpretation $\mathscr{I}chkbnd :\equiv (\mathbb{S}_{chk},\ \preccurlyeq_{chk},\ \mathbb{B},\ \mathbb{P})$ is defined as

$$
\begin{array}{lcl}
\mathbb{S}_{chk}(\langle st, c, w, k\rangle) & :\equiv & c \leq k \cdot |\mathcal{K}| \\
\langle st, c, w, k\rangle \preccurlyeq_{chk} \langle st', c', w', k'\rangle & :\equiv & \top \\
\mathbb{B}[\![\ {}^{\mathfrak{se}}\kappa, e\ ]\!] & :\equiv & \mathsf{AllFlats}(\mathsf{UniqSub}(-, \mathcal{K}), {}^{\mathfrak{se}}\kappa) \\
\mathbb{P}[\![\ {}^{\mathfrak{se}}\kappa, e^m\ ]\!] & :\equiv & \mathbb{B}[\![\ {}^{\mathfrak{se}}\kappa, e^m\ ]\!]
\end{array}
$$

Theorem 8.9. *$\mathscr{I}chkbnd$ is monotonic and sound.*

Bounding the Cost of *join*. Another non-trivial piece of the cost of space-efficient contracts is the time needed for merging two contracts—the essential part for maintaining space efficiency. To make the calculation of the cost of *join* simpler, I follow Guéneau et al. [2018] and Guéneau [2019] to work with multivariate asymptotic complexity.

Theorem 8.10. *The time complexity of $\mathit{drop}(xs, e)$ is $O(\mathsf{length}(xs))$.*

Theorem 8.11. *The time complexity of $\mathit{joinFlats}(xs,\ ys)$ is $O\big((\mathsf{length}(xs) + \mathsf{length}(ys))^2\big)$.*

Theorem 8.12. *Assume that $\mathsf{MaxHt}({}^{\mathfrak{se}}\kappa_2,\ H)$, $\mathsf{AllFlats}(\mathsf{lengthIn}(1, U, -), {}^{\mathfrak{se}}\kappa_2)$, $\mathsf{MaxHt}({}^{\mathfrak{se}}\kappa_1,\ H)$, $\mathsf{AllFlats}(\mathsf{lengthIn}(1, U, -), {}^{\mathfrak{se}}\kappa_1)$, and $1 \leq U$ where $\mathsf{lengthIn}(L, U, xs)$ holds if and only if $L \leq \mathsf{length}(xs) \leq U$. Then, the time complexity of $\mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)$ is*

$$O\Big(U^2 \cdot 2^H\Big).$$

Theorem 8.12 bounds the time complexity of $join(^{ss}\kappa_2, ^{ss}\kappa_1)$ in $O(U^2 \cdot 2^h)$ where U represents the upper bound of the length of the lists of predicates in $^{ss}\kappa_2, ^{ss}\kappa_1$, and H is the upper bound of the height of $^{ss}\kappa_2, ^{ss}\kappa_1$. This suggests the following interpretation for the aggregate cost for maintaining space efficiency:

Definition 8.13. The annotation interpretation $\mathcal{I}sebnd_C :\equiv (\mathbb{S}_C, \preccurlyeq, \mathbb{B}, \mathbb{P})$ is defined as

$$
\begin{array}{lcl}
\mathbb{S}_C(\langle st, c, w, k\rangle) & :\equiv & w \le C \cdot k \cdot |\mathcal{K}|^2 \cdot 2^H \\
\langle st, c, w, k\rangle \preccurlyeq \langle st', c', w', k'\rangle & :\equiv & \top \\
\mathbb{B}[\![\ ^{ss}\kappa, e\ ]\!] & :\equiv & \mathsf{NonRec}(^{ss}\kappa) \wedge \\
 & & \mathsf{AllFlats}(\mathsf{NonEmpty}, ^{ss}\kappa) \wedge \\
 & & \mathsf{AllFlats}(\mathsf{UniqSub}(-, \mathcal{K}), ^{ss}\kappa) \wedge \\
 & & \mathsf{MaxHt}(^{ss}\kappa,\ H) \\
\mathbb{P}[\![\ ^{ss}\kappa, e^m\ ]\!] & :\equiv & \mathbb{B}[\![\ ^{ss}\kappa, e^m\ ]\!]
\end{array}
$$

In $\mathcal{I}sebnd_C$, H is again the height of the tallest contract in the initial program, and $\mathcal{K}$ denotes the set of all distinct predicates. The interpretation $\mathcal{I}sebnd_C$ morally asserts the cost for merging contracts, w, is bounded by $O(k \cdot |\mathcal{K}|^2 \cdot 2^H)$, which is again linear in k. In summary, space-efficient contracts only slow down programs by a constant factor in theory.

Theorem 8.14. *There exists $c_0 > 0$ such that the interpretation $\mathcal{I}sebnd_{c_0}$ is monotonic and sound.*

Note. As Howell [2008] noted, the multivariate big O notation does not immediately satisfy common properties needed for the compositional analysis of algorithm complexity. Thus, I follow Guéneau [2019]'s approach in the actual proof. Using the big O notation in Theorem 8.14, however, introduces additional constraints in the interpretation $\mathcal{I}sebnd$ since the time complexity results hold only for sufficiently large inputs. Fortunately, the conditions $\mathsf{AllFlats}(\mathsf{NonEmpty}, ^{ss}\kappa)$ for all $^{ss}\kappa$ and that $|\mathcal{K}| \ge 1$ are sufficient for applying Theorem 8.12 in the proof of Theorem 8.14. In general, however, the size lower bounds may need to be generalized for an alternative proof.

8.6 Equivalence to Findler and Felleisen [2002]'s 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. In this section, I prove that space-efficient contracts behave the same as Findler and Felleisen [2002]'s contract system using the monitor calculus and the proof framework developed in Chapter 5.

To prove the equivalence, I follow the ideas from Pottier and Simonet [2002, 2003]'s non-interference proof and create ($\mathscr{A}sctc$,$\mathscr{T}sc$), an annotation language that runs two contract systems from $\lambda_m[\mathscr{A}ctc; \mathscr{T}c_1]$ and $\lambda_m[\mathscr{A}se;\mathscr{T}s]$ simultaneously. As the monitor calculus separates monitor-related rules from program-related rules, the two contract systems, $\lambda_m[\mathscr{A}ctc; \mathscr{T}c_1]$ and $\lambda_m[\mathscr{A}se;\mathscr{T}s]$, are completely encapsulated in the annotations and the global states of $\lambda_m[\mathscr{A}sctc; \mathscr{T}sc]$. Consequently, their equivalence boils down to proving that the global state of $\lambda_m[\mathscr{A}sctc;\mathscr{T}sc]$ always has two equal components.

The equivalence of the two contract systems is established by instantiating the proof framework from Sections 5.2 and 5.3 with an annotation interpretation named $\mathscr{I}sim$ that characterizes global states with two equal Status components. Then, after proving that $\mathscr{I}sim$ is both monotonic and sound, the proof framework leads to the conclusion that the Status components of the global state are always equal.

The remainder of this section details the proof outlined above. In Section 8.6.1, I present the instantiation $\lambda_m[\mathscr{A}sctc;\mathscr{T}sc]$. Then, Section 8.6.2 briefly reviews the proof framework of the monitor calculus. Sections 8.6.3 and 8.6.4 introduce the simulation relation, $\sim$, and prove its properties in preparation for establishing the monotonicity and soundness of $\mathscr{I}sim$. Finally, Section 8.6.5 assembles all the pieces of the proof.

8.6.1 Syntax of the Combined Annotation Language

The top of Figure 8.12 displays the formal syntax of $\mathscr{A}sctc$. Its annotations pair a space-efficient contract with a list of ordinary contracts, and its global state keeps two distinct copies of contract-

$$
\begin{array}{rcll}
\mathscr{A}scctc & :\equiv & (A, s) & A_{se} ::= {}^{\mathfrak{se}}\kappa \\
A & ::= & \langle A_{se}, A_{sctc}\rangle & A_{sctc} ::= [\kappa_1, \ldots, \kappa_m] \\
s & ::= & \langle st, st'\rangle & \text{Status} \ni st ::= \textsc{Ok} \mid \textsc{Err}(\ell)
\end{array}
$$

$$
\begin{array}{rl}
\text{[R-Cross-Inl]} & \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathbf{B}\#\langle ({}^{\mathfrak{se}}\kappa_1 \mathbin{+\!\!/\mathfrak{se}} {}^{\mathfrak{se}}\kappa_2), [(\kappa_1 \mathbin{+\!\!/\mathfrak{c}} \kappa_1'), \ldots, (\kappa_m \mathbin{+\!\!/\mathfrak{c}} \kappa_m')]\rangle \{\ \mathsf{inl}(v)\ \} \\
& \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathsf{inl}(\mathbf{B}\#\langle {}^{\mathfrak{se}}\kappa_1, [\kappa_1, \ldots, \kappa_m]\rangle \{v\}) \\
\text{[R-Cross-Roll]} & \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathbf{B}\#\langle (\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa), [(\mu/\mathfrak{c}\, t.\kappa_1), \ldots, (\mu/\mathfrak{c}\, t.\kappa_m)]\rangle \{\ \mathsf{roll}_\tau(v)\ \} \\
& \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathsf{roll}_\tau(\mathbf{B}\#\langle ({}^{\mathfrak{se}}\kappa[(\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa)\ /\ t]), [\kappa_1', \ldots, \kappa_m']\rangle \{v\}) \\
\text{where} & \kappa_1' :\equiv \kappa_1[(\mu/\mathfrak{c}\, t.\kappa_1)\ /\ t] \\
& \quad\vdots \\
& \kappa_m' :\equiv \kappa_m[(\mu/\mathfrak{c}\, t.\kappa_m)\ /\ t] \\
\text{[R-Proxy-}\beta\text{]} & \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathsf{proxy}\big(\langle {}^{\mathfrak{se}}\kappa_a \mathbin{\rightarrow\!\!/\mathfrak{se}} {}^{\mathfrak{se}}\kappa_r, [(\kappa_1 \mathbin{\rightarrow\!\!/\mathfrak{c}} \kappa_1'), \ldots, (\kappa_m \mathbin{\rightarrow\!\!/\mathfrak{c}} \kappa_m')]\rangle,\ \lambda x.e\big)\ v \\
& \longrightarrow_{\mathrm{m}}\ \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathbf{B}\#\langle {}^{\mathfrak{se}}\kappa_r, [\kappa_1', \ldots, \kappa_m']\rangle \{\ (\lambda x.e)\ (\mathbf{B}\#\langle {}^{\mathfrak{se}}\kappa_a, [\kappa_m, \ldots, \kappa_1]\rangle \{v\})\ \} \\
\text{[R-Merge-Box]} & \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathbf{B}\#\langle {}^{\mathfrak{se}}\kappa_1, [\kappa_1, \ldots, \kappa_l]\rangle \big\{\ \mathsf{proxy}\big(\langle {}^{\mathfrak{se}}\kappa_2, [\kappa_1', \ldots, \kappa_m']\rangle,\ \mathsf{box}(e)\big)\ \big\} \\
& \longrightarrow_{\mathrm{m}}\ \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathsf{proxy}\big(\langle {}^{\mathfrak{se}}\kappa, [\kappa_1', \ldots, \kappa_m', \kappa_1, \ldots, \kappa_l]\rangle,\ \mathsf{box}(e)\big) \\
\text{where} & {}^{\mathfrak{se}}\kappa :\equiv \mathsf{evalTick}[\![\ \checkmark\ \mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)\ ]\!] \\
\text{[R-Merge-Lam]} & \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathbf{B}\#\langle {}^{\mathfrak{se}}\kappa_1, [\kappa_1, \ldots, \kappa_l]\rangle \big\{\ \mathsf{proxy}\big(\langle {}^{\mathfrak{se}}\kappa_2, [\kappa_1', \ldots, \kappa_m']\rangle,\ \lambda x.e\big)\ \big\} \\
& \longrightarrow_{\mathrm{m}}\ \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathsf{proxy}\big(\langle {}^{\mathfrak{se}}\kappa, [\kappa_1', \ldots, \kappa_m', \kappa_1, \ldots, \kappa_l]\rangle,\ \lambda x.e\big) \\
\text{where} & {}^{\mathfrak{se}}\kappa :\equiv \mathsf{evalTick}[\![\ \checkmark\ \mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1)\ ]\!] \\
\text{[R-Cross-Nat]} & \langle \textsc{Ok}, \textsc{Ok}\rangle, \mathbf{B}\#\langle [\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)], [\kappa_1, \ldots, \kappa_m]\rangle \{\ n\ \} \longrightarrow_{\mathrm{m}} s'', n \\
\text{where} & (\langle \textsc{Ok}, \textsc{Ok}\rangle, s') \in \mathit{checkCtcs}_{\emptyset,\, \mathit{get}_1,\, \mathit{put}_1}([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)],\ n) \\
& (s', s'') \in \mathit{checkCtcs}_{\emptyset,\, \mathit{get}_2,\, \mathit{put}_2}([\kappa_1, \ldots, \kappa_m],\ n) \\
& \mathit{get}_1(\langle st_1, st_2\rangle) :\equiv st_1 \quad \mathit{put}_1(\langle st_1, st_2\rangle, st_1') :\equiv \langle st_1', st_2\rangle \\
& \mathit{get}_2(\langle st_1, st_2\rangle) :\equiv st_2 \quad \mathit{put}_2(\langle st_1, st_2\rangle, st_2') :\equiv \langle st_1, st_2'\rangle
\end{array}
$$

The annotation language ($\mathscr{A}scctc, \mathscr{T}sc$) pairs space-efficient contracts, ($\mathscr{A}se, \mathscr{T}s$), with Findler and Felleisen [2002]'s contracts, ($\mathscr{A}ctc, \mathscr{T}c_1$).

Figure 8.12: The annotation language for proving the equivalence.

checking status. The bottom of Figure 8.12 displays the selected rules of the transition steps of $\mathscr{T}sc$. As one would expect, the transition steps defined in the figure apply the rules from both $\mathscr{T}s$ from Figure 8.4 in page 133 and $\mathscr{T}c_1$ from Figure 6.4 in page 90 to propagate the contracts and separately manage the global states.

The monitor-related reduction relation $\mathscr{T}sc$ conceptually runs space-efficient contracts and ordinary contracts in parallel. For example, the [R-Cross-Inl] rule illustrates how the space-efficient contract ${}^{se}\kappa_1 \mathbin{+\!\!/_{se}} {}^{se}\kappa_2$ and the ordinary contracts $[(\kappa_1 \mathbin{+\!\!/_{c}} \kappa_1'), \ldots, (\kappa_m \mathbin{+\!\!/_{c}} \kappa_m')]$ are separately propagated from the old boundary around $\mathsf{inl}(v)$ to the new boundary around v.

The [R-Proxy-β] rule is similar in spirit. In the rule, the space-efficient contract ${}^{se}\kappa_a \mathbin{\rightarrow\!\!/_{se}} {}^{se}\kappa_r$ is decomposed and distributed to the two new boundaries in the second line. The contracts $[(\kappa_1 \mathbin{\rightarrow\!\!/_{c}} \kappa_1'), \ldots, (\kappa_m \mathbin{\rightarrow\!\!/_{c}} \kappa_m')]$ are similarly decomposed and distributed to the new boundaries. For ordinary contracts, the domain sub-contracts $\kappa_1, \ldots, \kappa_m$ are reversed before being attached to the argument (v) to maintain the correct contract-checking order (cf. Section 6.2).

Next, the [R-Merge-Box] and [R-Merge-Lam] rules joins the space-efficient contracts ${}^{se}\kappa_1$ and ${}^{se}\kappa_2$ using the *join* metafunction just like what $\mathscr{T}se$ does in page 133. At the same time, the [R-Merge-Lam] rule concatenating ordinary contracts $[\kappa_1, \ldots, \kappa_l]$ and $[\kappa_1', \ldots, \kappa_m']$ just like what $\mathscr{T}c_1$ does in page 90.

Finally, the [R-Cross-Nat] rule enforces the (flat) space-efficient contract and ordinary contracts. The rule makes two sequential calls to the *checkCtcs* metafunction from Figure 6.5 in page 92 to check the two lists of predicates. For the first call, rule [R-Cross-Nat] uses get_1 and put_1 to access the space-efficient component in the global status. The second call is similar except that rule [R-Cross-Nat] uses get_2 and put_2.

8.6.2 Overview of the Equivalence Proof

To prove that $\lambda_m[\mathscr{A}se;\mathscr{T}s]$ and $\lambda_m[\mathscr{A}ctc;\mathscr{T}c_1]$ always produce the same contract-checking results, I define the interpretation $\mathscr{I}sim$, which asks that the global state of $\lambda_m[\mathscr{A}scctc;\mathscr{T}sc]$ always contains equal components. As $\mathscr{T}sc$ updates the global states in rule [R-Cross-Nat] with two calls to

$$p \;::=\; + \mid -$$

$$(-p) :\equiv \begin{cases} -, & \text{if } p = + \\ +, & \text{if } p = - \end{cases}$$

$$\mathit{signedReverse}(+,\ [\kappa_1, \ldots, \kappa_m]) :\equiv [\kappa_1, \ldots, \kappa_m]$$
$$\mathit{signedReverse}(-,\ [\kappa_1, \ldots, \kappa_m]) :\equiv [\kappa_m, \ldots, \kappa_1]$$

$$\frac{\delta(t) = p}{\delta \vdash^p t\ \mathsf{signed}} \qquad \frac{\delta, t : p \vdash^p {}^{\mathfrak{se}}\kappa\ \mathsf{signed}}{\delta \vdash^p (\mu/\mathfrak{se}\ t.{}^{\mathfrak{se}}\kappa)\ \mathsf{signed}} \qquad \frac{\delta \vdash^{-p} {}^{\mathfrak{se}}\kappa_a\ \mathsf{signed} \qquad \delta \vdash^p {}^{\mathfrak{se}}\kappa_r\ \mathsf{signed}}{\delta \vdash^p ({}^{\mathfrak{se}}\kappa_a \to/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_r)\ \mathsf{signed}}$$

$$\frac{}{\delta \vdash^p \mathsf{unit}/\mathfrak{se}\ \mathsf{signed}} \qquad \frac{}{\delta \vdash^p [\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_m}(x.\, e_m)]\ \mathsf{signed}} \qquad \frac{\delta \vdash^p {}^{\mathfrak{se}}\kappa\ \mathsf{signed}}{\delta \vdash^p (\mathsf{box}/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa)\ \mathsf{signed}}$$

$$\frac{\delta \vdash^p {}^{\mathfrak{se}}\kappa_1\ \mathsf{signed} \qquad \delta \vdash^p {}^{\mathfrak{se}}\kappa_2\ \mathsf{signed}}{\delta \vdash^p ({}^{\mathfrak{se}}\kappa_1 \times/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_2)\ \mathsf{signed}} \qquad \frac{\delta \vdash^p {}^{\mathfrak{se}}\kappa_1\ \mathsf{signed} \qquad \delta \vdash^p {}^{\mathfrak{se}}\kappa_2\ \mathsf{signed}}{\delta \vdash^p ({}^{\mathfrak{se}}\kappa_1 +/\mathfrak{se}\ {}^{\mathfrak{se}}\kappa_2)\ \mathsf{signed}}$$

Figure 8.13: The covariance judgement for space-efficient contracts.

checkCtcs, I introduce a simulation $\sim$ over the annotations to make sure that the two lists of predicates that $\mathscr{T}sc$ passes to *checkCtcs* produce identical results.

I shall give the definition of $\sim$ later. Instead, let me explain how $\sim$ is involved in the equivalence proof. First, for any related contracts, i.e. ${}^{\mathfrak{se}}\kappa \sim [\kappa_1, \ldots, \kappa_m]$, *checkCtcs* produces identical contract checking results when ${}^{\mathfrak{se}}\kappa$ and $[\kappa_1, \ldots, \kappa_m]$ are contracts over natural numbers (where, in this case, ${}^{\mathfrak{se}}\kappa$ is also a list of predicates by the typing rules). Second, ${}^{\mathfrak{se}}\kappa \sim [\kappa_1, \ldots, \kappa_m]$ is preserved by monitor-related reductions, and thus leading to the conclusion that $\lambda_{\mathrm{m}}[\mathscr{A}scctc;\mathscr{T}sc]$ always contain global states of equal components, i.e. $\lambda_{\mathrm{m}}[\mathscr{A}se;\mathscr{T}s]$ and $\lambda_{\mathrm{m}}[\mathscr{A}ctc;\mathscr{T}c_1]$ are equivalent.

The interpretation $\mathscr{I}sim$ integrates $\sim$ in two places. First, it defines the partial order on the global states generated by $\langle \textsc{Ok}, \textsc{Ok} \rangle \preccurlyeq \langle \textsc{Err}(\ell), \textsc{Err}(\ell) \rangle$. Thus, when $\mathscr{I}sim$ is monotonic, the framework guarantees that the global state of $\lambda_{\mathrm{m}}[\mathscr{A}scctc;\mathscr{T}sc]$ has equal components. Second, $\mathscr{I}sim$ uses $\sim$ as the invariant attached by the $\mathbb{B}$ and $\mathbb{P}$ functions. Thus, $\mathscr{I}sim \vDash e$ holds if and only if all pairs of contracts in the annotations in e are related by $\sim$. As a result, the preservation of $\sim$ by $\longrightarrow_{\mathrm{m}}$ is, by definition, the soundness of $\mathscr{I}sim$. In other words, the equivalence proof still lies within the framework developed in Chapter 5.

A Note on Recursive Contracts. There is one caveat, however: $\sim$ is preserved by $\longrightarrow_{\mathrm{m}}$ only

$$\frac{}{\Gamma^{\varkappa} \vdash [] \sim^{\mathsf{cl}} []} \qquad \frac{\Gamma^{\varkappa}, e \vdash [\kappa_2, \ldots, \kappa_k] \sim^{\mathsf{cl}} [\kappa_2', \ldots, \kappa_m']}{\Gamma^{\varkappa} \vdash [\mathsf{flat}^{\ell}(x.\, e), \kappa_2, \ldots, \kappa_k] \sim^{\mathsf{cl}} [\mathsf{flat}^{\ell}(x.\, e), \kappa_2', \ldots, \kappa_m']}\ [\textsc{Clp-Keep}]$$

$$\frac{e^* \in \Gamma^{\varkappa} \qquad e^* \preccurlyeq e \qquad \Gamma^{\varkappa} \vdash [\kappa_1, \ldots, \kappa_k] \sim^{\mathsf{cl}} [\kappa_2', \ldots, \kappa_m']}{\Gamma^{\varkappa} \vdash [\kappa_1, \ldots, \kappa_k] \sim^{\mathsf{cl}} [\mathsf{flat}^{\ell}(x.\, e), \kappa_2', \ldots, \kappa_m']}\ [\textsc{Clp-Drop}]$$

Figure 8.14: The simulation relation of flat predicates.

when all recursive contracts appearing in the program are covariant. This is the same phenomenon as blame consistency from Section 7.2 where a contravariant recursive contract can produce an incorrect blame. Thus, I add the constraint $\vdash^{+} {}^{\mathfrak{se}}\kappa$ signed to the interpretation $\mathcal{I}sim$ just like to how the blame consistency theorem restricts recursive contracts.

For completeness, I present the definition of $\delta \vdash^{p} {}^{\mathfrak{se}}\kappa$ signed for recursive contracts and an auxiliary function $\mathit{signedReverse}(p, [\kappa_1, \ldots, \kappa_m])$ in Figure 8.13. The definitions are similar to their counterparts in Section 7.2 and are used only for the proof.

Proposition 8.15. *If* $\delta \vdash^{p} {}^{\mathfrak{se}}\kappa$ signed *then* $-\delta \vdash^{-p} {}^{\mathfrak{se}}\kappa$ signed *where* $(-\delta)(x) :\equiv -\delta(x)$.

Lemma 8.16 (Renaming)**.** *Let* $\delta :\equiv \{t_1 : p_1, \ldots, t_n : p_n\}$ *and* $\delta' :\equiv \{t_1' : p_1', \ldots, t_m' : p_m'\}$ *be given. Assume that there is a sequence* $1 \le a_i \le m$ *such that* $p_i = p_{a_i}'$ *for* $i = 1 \ldots n$*. If* $\delta \vdash^{p} {}^{\mathfrak{se}}\kappa$ signed *then* $\delta' \vdash^{p} {}^{\mathfrak{se}}\kappa\left[t_{a_1}', \ldots, t_{a_n}' \,/\, t_1, \ldots, t_n\right]$ signed.

Proposition 8.17. *Let* $\delta' \vdash^{p_i} {}^{\mathfrak{se}}\kappa_i$ signed *for* $i = 1 \ldots n$ *be given. Then, for any fresh variable* $t_0 : p_0$*, there is a sequence* $\delta'' \vdash^{p_i} {}^{\mathfrak{se}}\kappa_i$ signed *for* $i = 0 \ldots n$ *where* $\delta'' :\equiv \delta, t_0 : p_0$ *and* $\kappa_0 :\equiv t_0$.

Lemma 8.18 (Substitution)**.** *Let* $\delta :\equiv \{t_1 : p_1, \ldots, t_n : p_n\}$ *be given. If* $\delta' \vdash^{p_i} {}^{\mathfrak{se}}\kappa_i$ signed *for* $1 \le i \le n$ *and* $\delta \vdash^{p} {}^{\mathfrak{se}}\kappa$ signed *then* $\delta' \vdash^{p} {}^{\mathfrak{se}}\kappa[{}^{\mathfrak{se}}\kappa_1, \ldots, {}^{\mathfrak{se}}\kappa_n \,/\, t_1, \ldots, t_n]$ signed.

8.6.3 Maintaining Equal Contract Checking Status

The base case of the $\sim$ relation delegates the work to the $\sim^{\mathsf{cl}}$ relation. That is, $\sim^{\mathsf{cl}}$ captures equivalent space-efficient contracts and ordinary contracts at the nat type, and the next section uses $\sim^{\mathsf{cl}}$ to define the full relation over contracts of all types, $\sim$.

Figure 8.14 displays the definition of $\sim^{\mathsf{cl}}$. To help distinguish the contracts, I color space-

$$\frac{}{t \sim [t, \ldots, t]} \qquad \frac{\vdash [\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)] \sim^{\mathsf{cl}} [\kappa_1, \ldots, \kappa_m]}{[\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)] \sim [\kappa_1, \ldots, \kappa_m]}$$

$$\frac{}{\mathsf{unit}/\mathfrak{se} \sim [\mathsf{unit}/\mathfrak{c}, \ldots, \mathsf{unit}/\mathfrak{c}]} \qquad \frac{^{\mathfrak{se}}\kappa_a \sim [\kappa_m, \ldots, \kappa_1] \qquad ^{\mathfrak{se}}\kappa_r \sim [\kappa'_1, \ldots, \kappa'_m]}{(^{\mathfrak{se}}\kappa_a \to/\mathfrak{se}\ ^{\mathfrak{se}}\kappa_r) \sim [(\kappa_1 \to/\mathfrak{c}\ \kappa'_1), \ldots, (\kappa_m \to/\mathfrak{c}\ \kappa'_m)]}$$

$$\frac{^{\mathfrak{se}}\kappa_1 \sim [\kappa_1, \ldots, \kappa_m] \qquad ^{\mathfrak{se}}\kappa_2 \sim [\kappa'_1, \ldots, \kappa'_m]}{(^{\mathfrak{se}}\kappa_1 \times/\mathfrak{se}\ ^{\mathfrak{se}}\kappa_2) \sim [(\kappa_1 \times/\mathfrak{c}\ \kappa'_1), \ldots, (\kappa_m \times/\mathfrak{c}\ \kappa'_m)]} \qquad \frac{^{\mathfrak{se}}\kappa_1 \sim [\kappa_1, \ldots, \kappa_m] \qquad ^{\mathfrak{se}}\kappa_2 \sim [\kappa'_1, \ldots, \kappa'_m]}{(^{\mathfrak{se}}\kappa_1 +/\mathfrak{se}\ ^{\mathfrak{se}}\kappa_2) \sim [(\kappa_1 +/\mathfrak{c}\ \kappa'_1), \ldots, (\kappa_m +/\mathfrak{c}\ \kappa'_m)]}$$

$$\frac{^{\mathfrak{se}}\kappa \sim [\kappa_1, \ldots, \kappa_m]}{\mathsf{box}/\mathfrak{se}\ ^{\mathfrak{se}}\kappa \sim [\mathsf{box}/\mathfrak{c}\ \kappa_1, \ldots, \mathsf{box}/\mathfrak{c}\ \kappa_m]} \qquad \frac{^{\mathfrak{se}}\kappa \sim [\kappa_1, \ldots, \kappa_m]}{(\mu/\mathfrak{se}\ t.\, ^{\mathfrak{se}}\kappa) \sim [(\mu/\mathfrak{c}\ t.\kappa_1), \ldots, (\mu/\mathfrak{c}\ t.\kappa_m)]}$$

Figure 8.15: The simulation relation of all contracts.

efficient contracts blue and ordinary contracts brown. As Figure 8.14, shows $\sim^{\mathsf{cl}}$ is a ternary relation. The first part is a context, $\Gamma^{\varkappa}$, which is responsible for tracking the set of predicates that $\sim^{\mathsf{cl}}$ has walked through when recurring into the tail of the list. The second part is a redundancy-free list of predicates. The last part of $\sim^{\mathsf{cl}}$ is the list of all predicates.

During recursion, rule [Clp-Drop] can drop a predicate e from the second part if the context contains a stronger predicate e^*. In this case, removing e has no effect as e^* subsumes the contract checking result of e. Otherwise, rule [Clp-Keep] keeps e in both the second and the third parts. When scanning the tail of the list, rule [Clp-Keep] also adds e to the context.

Lemma 8.19 is the key lemma for proving the equivalence of two contract systems. Specifically, it states that when a space-efficient contract and an ordinary contract are related at base types (i.e. related lists of flat predicates), the checking result of will be equivalent:

Lemma 8.19. *Assume* $(\langle \textsc{Ok}, \textsc{Ok}\rangle, \langle st', \textsc{Ok}\rangle) \in \mathit{checkCtcs}_{\emptyset,\, get_1,\, put_1}([\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)],\ n)$ *and that* $(\langle st', \textsc{Ok}\rangle, s'') \in \mathit{checkCtcs}_{\emptyset,\, get_2,\, put_2}([\kappa_1, \ldots, \kappa_m],\ n)$. *If, it is the case that*

$$\Gamma^{\varkappa} \vdash [\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)] \sim^{\mathsf{cl}} [\kappa_1, \ldots, \kappa_m]$$

and $(\langle st', \textsc{Ok}\rangle, \langle st', \textsc{Ok}\rangle) \in \mathit{checkPred}_{\emptyset\ ,\, get_1,\, put_1}(\mathsf{flat}^{\ell^*}(x.\, e^*),\ n)$ *for all* $e^* \in \Gamma^{\varkappa}$, *there exists* $st'' \in$ Status *such that* $s'' = \langle st'', st''\rangle$.

8.6.4 Preservation of the Simulation Relation

Figure 8.15 shows the definition of $\sim$. In most cases, $\sim$ is defined by replicating the actions of the transition steps, $\mathscr{T}sc$, except that for the nat type $\sim$ uses $\sim^{\mathsf{cl}}$ from the previous section to ensure that space-efficient contracts only removes redundant predicates.

To understand how $\sim$ captures the transition steps $\mathscr{T}sc$ in its definition, consider the reduction

$$\mathscr{A}scctc \vdash s, \mathsf{proxy}(A,\ \lambda x.e)\ v \longrightarrow s', \mathbf{B}\#A_r\ \{\ (\lambda x.e)\ (\mathbf{B}\#A_a\ \{v\})\ \}$$

where A is the annotation $\langle {}^{\mathfrak{se}}\kappa_a \rightarrow\!\!\!/_{\mathfrak{se}}\ {}^{\mathfrak{se}}\kappa_r, [\kappa_1 \rightarrow\!\!\!/_{\mathfrak{c}}\ \kappa'_1, \ldots, \kappa_m \rightarrow\!\!\!/_{\mathfrak{c}}\ \kappa'_m]\rangle$. By the definition of $\mathscr{T}sc$ from Figure 8.12 in page 144, A_r and A_a equal:

$$\begin{array}{rcl} A_r & :\equiv & \langle {}^{\mathfrak{se}}\kappa_r, [\kappa'_1, \ldots, \kappa'_m]\rangle \\ A_a & :\equiv & \langle {}^{\mathfrak{se}}\kappa_a, [\kappa_m, \kappa_{m-1} \ldots, \kappa_1]\rangle \end{array}$$

Comparing the annotations A, A_r, and A_a to the definition of $\sim$, it can be easily seen that when the contracts in A are related by $\sim$, so are the contracts in A_a and A_r. Thus, most rules in the definition of $\sim$ are straightforward, as is the preservation proof.

The [R-Merge-Box] and [R-Merge-Lam] rules are where real work kicks in because $\mathscr{T}sc$ calls $join(\cdot,\cdot)$ to merge two space-efficient contracts. In Lemmas 8.20 to 8.22, I prove that the [R-Merge] rules preserve the simulation relation. Concretely, in Figure 8.12, the term on the left-hand side of the [R-Merge] rules is

$$\mathbf{B}\#\langle {}^{\mathfrak{se}}\kappa_1, [\kappa_1, \ldots, \kappa_l]\rangle\ \left\{\ \mathsf{proxy}\left(\langle {}^{\mathfrak{se}}\kappa_2, [\kappa'_1, \ldots, \kappa'_m]\rangle,\ e^m\right)\ \right\},$$

which reduces to the term

$$\mathsf{proxy}\left(\langle join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1), [\kappa'_1, \ldots, \kappa'_m, \kappa_1, \ldots, \kappa_l]\rangle,\ e^m\right)$$

on the right-hand side. To prove that the reduction step preserves $\sim$, I need to demonstrate that ${}^{\mathfrak{se}}\kappa_1 \sim [\kappa_1, \ldots, \kappa_l]$ and ${}^{\mathfrak{se}}\kappa_2 \sim [\kappa'_1, \ldots, \kappa'_m]$ implies $join({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1) \sim [\kappa'_1, \ldots, \kappa'_m, \kappa_1, \ldots, \kappa_l]$. This breaks down into the following lemmas about *drop*, *joinFlats*, and *join*. To ease the presentation, I shall omit the references to evalTick and ✓ in the statements of the lemmas.

Lemma 8.20. *If* $e \in \Gamma^{\varkappa}$ *and* $\Gamma^{\varkappa} \vdash [\mathsf{flat}^{\ell_1}(x.\, e_1), \ldots, \mathsf{flat}^{\ell_k}(x.\, e_k)] \sim^{\mathsf{cl}} [\kappa_1, \ldots, \kappa_m]$ *then*

$\Gamma^{\varkappa} \vdash \mathit{drop}([\mathrm{flat}^{\ell_1}(x.\, e_1), \ldots, \mathrm{flat}^{\ell_k}(x.\, e_k)], e) \sim^{\mathrm{cl}} [\kappa_1, \ldots, \kappa_m]$

Lemma 8.21. *If* $\Gamma^{\varkappa} \vdash [\mathrm{flat}^{\ell_1}(x.\, e_1), \ldots, \mathrm{flat}^{\ell_k}(x.\, e_k)] \sim^{\mathrm{cl}} [\kappa_1, \ldots, \kappa_m]$ *and* $\vdash [\mathrm{flat}^{\ell'_1}(x.\, e'_1), \ldots, \mathrm{flat}^{\ell'_h}(x.\, e'_h)] \sim^{\mathrm{cl}} [\kappa'_1, \ldots, \kappa'_l]$ *then*

$$\Gamma^{\varkappa} \vdash \mathit{joinFlats}([\mathrm{flat}^{\ell_1}(x.\, e_1), \ldots, \mathrm{flat}^{\ell_k}(x.\, e_k)], [\mathrm{flat}^{\ell'_1}(x.\, e'_1), \ldots, \mathrm{flat}^{\ell'_h}(x.\, e'_h)]) \sim^{\mathrm{cl}} [\kappa_1, \ldots, \kappa_m, \kappa'_1, \ldots, \kappa'_l]$$

Lemma 8.22. *If* ${}^{\mathfrak{se}}\kappa_1 \sim [\kappa_1, \ldots, \kappa_l]$ *and* ${}^{\mathfrak{se}}\kappa_2 \sim [\kappa'_1, \ldots, \kappa'_m]$ *then* $\mathit{join}({}^{\mathfrak{se}}\kappa_2, {}^{\mathfrak{se}}\kappa_1) \sim [\kappa'_1, \ldots, \kappa'_m, \kappa_1, \ldots, \kappa_l]$.

The preservation of the simulation relation for the [R-Cross-Roll] rule is more intricate. Recall that in the [R-Cross-Roll] rule from Figure 8.12, the term on the left-hand side is

$$\mathbf{B}\#\langle(\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa)\,, [(\mu/\mathfrak{c}\, t.\kappa_1), \ldots, (\mu/\mathfrak{c}\, t.\kappa_m)]\rangle\ \{\ \mathrm{roll}_\tau(v)\ \}\,,$$

and the term on the right-hand side is

$$\mathrm{roll}_\tau(\mathbf{B}\#\langle({}^{\mathfrak{se}}\kappa[(\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa)\ /\ t])\,, [(\kappa_1[(\mu/\mathfrak{c}\, t.\kappa_1)\ /\ t])\,, \ldots, (\kappa_m[(\mu/\mathfrak{c}\, t.\kappa_m)\ /\ t])]\rangle\ \{v\})$$

Therefore, for the simulation relation to be preserved by the reduction, it has to be shown that

$$\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa \sim [(\mu/\mathfrak{c}\, t.\kappa_1)\,, \ldots, (\mu/\mathfrak{c}\, t.\kappa_m)]$$

would imply

$${}^{\mathfrak{se}}\kappa[(\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa)\ /\ t] \sim [(\kappa_1[(\mu/\mathfrak{c}\, t.\kappa_1)\ /\ t])\,, \ldots, (\kappa_m[(\mu/\mathfrak{c}\, t.\kappa_m)\ /\ t])].$$

In other words, the preservation with respect to the [R-Cross-Roll] rule requires the proof that the substitution respects the simulation relation.

As previously noted, only recursive contracts that are covariant preserves the simulation relation. Proposition 8.23 formally states the preservation lemma for the [R-Cross-Roll] rule, with the necessary condition that the given space-efficient contract is covariant.

Proposition 8.23. *If* $\delta \vdash^{+} {}^{\mathfrak{se}}\kappa$ signed *and* $\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa \sim [(\mu/\mathfrak{c}\, t.\kappa_1)\,, \ldots, (\mu/\mathfrak{c}\, t.\kappa_m)]$ *then* ${}^{\mathfrak{se}}\kappa[(\mu/\mathfrak{se}\, t.{}^{\mathfrak{se}}\kappa)\ /\ t] \sim [(\kappa_1[(\mu/\mathfrak{c}\, t.\kappa_1)\ /\ t])\,, \ldots, (\kappa_m[(\mu/\mathfrak{c}\, t.\kappa_m)\ /\ t])]$.

Proposition 8.23 is a direct corollary of the Substitution lemma for the simulation relation. Lemmas 8.24 and 8.26 and Proposition 8.25 together prove the Substitution lemma for the simu-

lation relation.

Lemma 8.24 (Renaming). *Let* $\Delta :\equiv \{t_1, \ldots, t_n\}$ *and* $\Delta' :\equiv \{t'_1, \ldots, t'_m\}$ *be given. Assume that there is a sequence* $1 \le a_i \le m$ *for* $i = 1 \ldots n$. *If* ${}^{\text{se}}\kappa \sim [\kappa_1, \ldots, \kappa_m]$ *where* $\Delta \vdash^{\text{se}} {}^{\text{se}}\kappa : \mathsf{SECtc}\,\tau$ *and* $\Delta \vdash^{\text{se}} \kappa_i : \mathsf{SECtc}\,\tau$ *for* $i = 1 \ldots n$ *then*

$$ {}^{\text{se}}\kappa\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right] \sim \left[\left(\kappa_1\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right]\right), \ldots, \left(\kappa_m\left[t'_{a_1}, \ldots, t'_{a_n} \,/\, t_1, \ldots, t_n\right]\right)\right]. $$

Proposition 8.25. *Let* ${}^{\text{se}}\kappa_i \sim \mathit{signedReverse}(p_i, [\kappa_{i,1}, \ldots, \kappa_{i,m}])$ *for* $i = 1 \ldots n$ *be given. For any fresh variable* $t_0 : p_0$, *there is a sequence* ${}^{\text{se}}\kappa_i \sim \mathit{signedReverse}(p_i, [\kappa_{i,1}, \ldots, \kappa_{i,m}])$ *for* $i = 0 \ldots n$ *where* ${}^{\text{se}}\kappa_0 :\equiv t_0$ *and* $\kappa_{0,j} :\equiv t_0$ *for* $j = 0 \ldots m$.

Lemma 8.26 (Substitution). *Let* $\delta :\equiv \{t_1 : p_1, \ldots, t_n : p_n\}$ *and* δ' *be given. Assume that* $\delta \vdash^{p} {}^{\text{se}}\kappa$ signed *and* ${}^{\text{se}}\kappa \sim \mathit{signedReverse}(p, [\kappa_1, \ldots, \kappa_n])$. *If for all* $1 \le i \le n$, $\delta' \vdash^{p_i} {}^{\text{se}}\kappa_i$ signed *and* ${}^{\text{se}}\kappa_i \sim \mathit{signedReverse}(p_i, [\kappa'_{i,1}, \ldots, \kappa'_{i,m}])$ *then*

$$ {}^{\text{se}}\kappa[{}^{\text{se}}\kappa_1, \ldots, {}^{\text{se}}\kappa_n \,/\, t_1, \ldots, t_n] \sim \mathit{signedReverse}(p, [\left(\kappa_1\left[\kappa'_{1,1}, \ldots, \kappa'_{1,m} \,/\, t_1, \ldots, t_n\right]\right), \ldots, \left(\kappa_n\left[\kappa'_{n,1}, \ldots, \kappa'_{n,m} \,/\, t_1, \ldots, t_n\right]\right)]). $$

8.6.5 Completing the Equivalence Proof

To finish the equivalence proof, I present the interpretation $\mathcal{I}\mathit{sim}$. The properties needed for proving its monotonicity and soundness are covered earlier, so the rest is easy. Of course, similar to blame consistency, $\mathcal{I}\mathit{sim}$ includes the judgement $\vdash^{+} {}^{\text{se}}\kappa$ signed in the $\mathbb{B}$ function to make sure all recursive contracts appearing in the program are covariant.

Definition 8.27. The annotation interpretation $\mathcal{I}\mathit{sim} :\equiv (\mathbb{S}_{sim}, \preccurlyeq_{sim}, \mathbb{B}, \mathbb{P})$ is defined as

$$
\begin{array}{lcl}
\mathbb{S}_{sim}(\langle st, st' \rangle) & \Longleftrightarrow & (st = st') \\
s \preccurlyeq_{sim} s' & :\equiv & (s = s') \vee (\exists \ell.\ (s = \langle \textsc{Ok}, \textsc{Ok} \rangle) \wedge (s' = \langle \textsc{Err}(\ell), \textsc{Err}(\ell) \rangle)) \\
\mathbb{B}[\![\, \langle {}^{\text{se}}\kappa, [\kappa_1, \ldots, \kappa_m] \rangle, e \,]\!] & :\equiv & (\mathcal{I}_{\perp} \vDash \kappa_1) \wedge \cdots \wedge (\mathcal{I}_{\perp} \vDash \kappa_m) \wedge \\
 & & (\vdash^{+} {}^{\text{se}}\kappa \text{ signed}) \wedge ({}^{\text{se}}\kappa \sim [\kappa_1, \ldots, \kappa_m]) \\
\mathbb{P}[\![\, A, e^m \,]\!] & :\equiv & \mathbb{B}[\![\, A, e^m \,]\!]
\end{array}
$$

Theorem 8.28. $\mathcal{I}\mathit{sim}$ *is monotonic.*

Proof. For the [R-Cross-Nat] rule, by Lemma 8.19. For other rules, the global states remain

identical. □

Theorem 8.29. *$\mathcal{I}$sim is sound.*

Proof. For the [R-Merge-Box] and [R-Merge-Lam] rules, by Lemma 8.22. For the [R-Cross-Roll] rule, by Proposition 8.23 (which in turn uses the Substitution lemma). All other cases are straightforward. □

Putting everything together, it follows from the proof framework of the monitor calculus (Theorem 5.19) that the two Status components in the global state are always equal. In other words, space-efficient contracts and ordinary contracts always produce equal checking results.

Chapter 9

Conclusion

Every researcher with an interest in the metatheory of contract systems has experienced the banality of proving their properties. Whether the target is correct blame 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 my dissertation is how to separate the creative from the routine; researchers should focus on the information needed for distilling the target property to an interpretation of the annotations — given a few facts about the interpretation, the rest can be abstracted away.

While my dissertation provides evidence in favor of its transition-system-based framework, it also points out next steps for realizing its vision in full. First, Chapter 7 utilizes the framework to prove the preservation of several interpretations. While this captures the correct blame property, a more satisfying result would be capturing the compete monitoring property, which needs a separate proof of progress that currently lives outside my framework. However, since homomorphisms reflect transitions, some form of progress property should be transferable across transition systems. Second, the dissertation just scratches the surface of structured information in global states. Beyond recording information about different events, such as different oper-

ations, systematically structured global states open the way for supporting expressive, stateful contracts [Dimoulas et al. 2016; Moore et al. 2016; Moy and Felleisen 2023; Moy et al. 2024]. 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 instantiations are.

References

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. https://doi.org/10.1145/3110283

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, ICFP, Article 90 (jul 2018), 30 pages. https://doi.org/10.1145/3236785

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-Scope Safe Programs and Their Proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. 195–207. https://doi.org/10.1145/3018610.3018613

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. https://doi.org/10.1145/2508168.2508171

André Arnold and Ilaria Castellani. 1996. An algebraic characterization of observational equivalence. *Theoretical Computer Science* 156, 1 (1996), 289–299. https://doi.org/10.1016/0304-3975(95)00141-7

André Arnold and Anne Dicky. 1989. An algebraic characterization of transition system equivalences. *Information and Computation* 82, 2 (1989), 198–229. https://doi.org/10.1016/0890-5401(89)90054-0

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. https://doi.org/10.1145/1016850.1016876

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

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62 (2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (apr 1986), 244–263. https://doi.org/10.1145/5397.5399

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. https://doi.org/10.1145/1328438.1328457

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. https://doi.org/10.1145/2103746.2103766

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. https://doi.org/10.1145/2039346.2039348

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. https://doi.org/10.1145/1926385.1926410

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. https://doi.org/10.1109/CSF.2014.9

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. https://doi.org/10.1145/2951913.2951930

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. https://doi.org/10.1145/1599410.1599435

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Programming Languages and Systems (ESOP '12)*. 214–233.

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. https://doi.org/10.1145/2034773.2034800

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morriset (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (feb 2018), 62–71. https://doi.org/10.1145/3127323

Matthias Felleisen and Daniel P. Friedman. 1987. A reduction semantics for imperative higher-order languages. In *PARLE Parallel Architectures and Languages Europe*. 206–223.

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. https://doi.org/10.1145/3276503

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.

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. https://doi.org/10.1145/581478.581484

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.

Ronald Garcia. 2013. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. 417–428. https://doi.org/10.1145/2500365.2500603

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. https://doi.org/10.1145/2837614.2837670

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. https://doi.org/10.1145/3649842

Abraham Ginzburg. 1968. *Algebraic Theory of Automata*. Academic Press. https://www.sciencedirect.com/science/book/9781483200132

Michael Greenberg. 2014. Space-Efficient Manifest Contracts. (2014). https://doi.org/10.48550/arXiv.1410.2813 arXiv:1410.2813 [cs.PL] Technical Report.

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. https://doi.org/10.1145/2676726.2676967

Michael Greenberg. 2016. Space-Efficient Latent Contracts. In *Trends in Functional Programming (TFP)*. 3–23.

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. https://doi.org/10.1145/1706299.1706341

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. https://doi.org/10.1145/3579833

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. https://doi.org/10.1145/3236766

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. https://doi.org/10.1145/3360548

Jessica Gronski and Cormac Flanagan. 2007. Unifying Hybrid Types and Contracts. In *Trends in Functional Programming (TFP'07)*. 54–70.

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. Logic in Computer Science [cs.LO]. NNT : 2019UNIP7110. tel-03071720. https://theses.hal.science/tel-03071720.

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.

Matthew Hennessy and Robin Milner. 1985. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM* 32, 1 (jan 1985), 137–161. https://doi.org/10.1145/2455.2460

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed Contracts for Functional Programming. In *Functional and Logic Programming (FLOPS '06)*. 208–225.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

Rodney R Howell. 2008. On Asymptotic Notation With Multiple Variables. *Department of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA, Technical Report* (2008). https://people.cs.ksu.edu/~rhowell/asymptotic.pdf

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. https://doi.org/10.1145/2784731.2784737

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (may 1994), 872–923. https://doi.org/10.1145/177492.177726

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc.

Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. 2019. Does Blame Shifting Work? *Proceedings of the ACM on Programming Languages (PACMPL)* 4, POPL, Article 65 (Dec 2019), 29 pages. https://doi.org/10.1145/3371133

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.

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. https://doi.org/10.1145/1190216.1190220

Conor McBride. 2005. Type-Preserving Renaming and Substitution. (2005). http://strictlypositive.org/ren-sub.pdf.

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. https://doi.org/10.1145/3110279

Bertrand Meyer. 1991a. Design by contract. In *Advances in Object-Oriented Software Engineering*, Dino Mandrioli and Bertrand Meyer (Eds.). Prentice Hall, 1–50.

Bertrand Meyer. 1991b. *Eiffel: The Language*. Prentice Hall.

Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

Bertrand Meyer. 2005. Standard Eiffel. (2005). http://se.ethz.ch/~meyer/publications/index_kind.html#PSTE Draft. Previously published as Meyer [1991b].

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. https://doi.org/10.1145/2983990.2984021

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. https://doi.org/10.1145/3632930

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

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. https://doi.org/10.1145/3290328

David L. Parnas. 1972. A Technique for Software Module Specification with Examples. *Commun. ACM* 15, 5 (may 1972), 330–336. https://doi.org/10.1145/355602.361309

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

Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32

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. https://doi.org/10.1145/503272.503302

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

Davide Sangiorgi. 2009. On the Origins of Bisimulation and Coinduction. *ACM Transactions on Programming Languages and Systems* 31, 4, Article 15 (may 2009), 41 pages. https://doi.org/10.1145/1516507.1516510

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. https://doi.org/10.1145/3009837.3009875

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.

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. 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. https://doi.org/10.1145/2737924.2737968

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

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. https://doi.org/10.1017/S0956796821000241

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*. 2–27.

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. https://doi.org/10.1007/978-3-319-30936-1_21

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. 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. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

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. https://doi.org/10.1145/1706299.1706342

Joseph Sifakis. 1983. Property preserving homomorphisms of transition systems. In *Logics of Programs (Lecture Notes in Computer Science)*. 458–473.

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. https://doi.org/10.1145/2518189

T. Stephen Strickland and Matthias Felleisen. 2009. Contracts for First-Class Modules. In *Proceedings of the 5th Symposium on Dynamic Languages* (Orlando, Florida, USA) *(DLS '09)*. 27–38. https://doi.org/10.1145/1640134.1640140

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

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. https://doi.org/10.1145/2784731.2784742

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. https://doi.org/10.1017/S0956796818000047

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)*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.4

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. https://doi.org/10.1145/2837614.2837630

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. https://doi.org/10.1145/2384616.2384674

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.

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

Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Programming Languages and Systems (ESOP '10)*. 550–569.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Johan van Benthem and Jan Bergstra. 1994. Logic of transition systems. *Journal of Logic, Language and Information* 3, 4 (01 Dec 1994), 247–283. https://doi.org/10.1007/BF01160018

Johan van Benthem, Jan van Eijck, and Vera Stebletsova. 1994. Modal Logic, Transition Systems and Processes. *Journal of Logic and Computation* 4, 5 (10 1994), 811–855. https://doi.org/10.1093/logcom/4.5.811

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. https://doi.org/10.1145/3009837.3009849

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. https://doi.org/10.4230/LIPIcs.SNAPL.2015.309

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems (ESOP '09)*. 1–16.

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. https://doi.org/10.1145/3276504

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. https://doi.org/10.1145/1480881.1480889