# Refinement-Based Game Semantics for Certified Components

DRAFT updated on July 16, 2020

A Dissertation Presented to the Faculty of the Graduate School of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy



by

Jérémie Koenig

Dissertation Director:

Zhong Shao

December 2020

**Abstract**

Current practices ensure software reliability through careful testing, but while testing can reveal the presence of bugs, it cannot entirely guarantee their absence. By contrast, *certified systems* come with a formal specification and a computer-checked proof of correctness, providing strong evidence that the system behaves as expected in all possible scenarios. Over the past decade, researchers have been able to build certified systems of significant size and complexity, including compilers, processor designs, operating system kernels and more. Building on these successes, the DeepSpec project proposes to assemble them as certified components to build large-scale heterogeneous certified systems.

However, by necessity, these certified components use a broad variety of semantic models and verification techniques. To connect them, we must first embed them into a common, general-purpose model. The work I present here unifies the foundations of certified abstraction layers, game semantics, algebraic effects and the refinement calculus to build models suitable for this task. We represent certified abstraction layers, interaction trees, and the certified compiler CompCert in a single framework supporting heterogeneous components, stepwise refinement, and data abstraction.

# Contents

# Chapter 1

# Introduction

## 1.1 Certified systems at scale

Certified software [Shao, 2010] is software accompanied by mechanized, machine-checkable proofs of correctness. To construct a certified program, we must not only write its code in a given programming language, but also formally specify its intended behavior and construct, using specialized tools, evidence that the program indeed conforms to the specification.

The past decade has seen an explosion in the scope and scale of practical software verification. Researchers have been able to produce certified compilers [Leroy, 2009], program logics [Appel, 2011], operating system kernels [Gu et al., 2015; Klein et al., 2009], file systems [Chen et al., 2015], processor designs [Azevedo de Amorim et al., 2014; Choi et al., 2017] and more, often introducing new techniques and mathematical models. In this context, there has been increasing interest in making these components interoperable and combining them—and their proofs of correctness—into larger certified systems.

### 1.1.1 The DeepSpec project

This is exemplified by the DeepSpec project [Appel et al., 2017], which seeks to connect various components specified and verified in the Coq proof assistant. The key idea behind DeepSpec is to use specifications as *interfaces* between components. When a component providing a certain interface has been verified, client components can rely on this for their own proofs of correctness. Standardizing this process would make it possible to construct large-scale certified systems by

assembling off-the-shelf certified components.

This approach would provides benefits beyond the potential increase in the scale of practical certified systems. As it stands, a certified system is only as trustworthy as its specification. Indeed, it is possible to prove a buggy system correct with respect to a buggy specification. If the only user of the specification is a human expert subjecting it to careful examination, these bugs could go unnoticed and persist in the final system. By contrast, if the same specification is used as a premise in the correctness proof of a client component, its deficiencies will become apparent and prevent us from carrying out this second proof. Moreover, the internal specifications used as intermediate steps in the verification of a complex system disappear from the external characterization of the system, and no longer need to be trusted. This reduces the ratio between the size of the system and the size of the specification which must be trusted and understood to establish guarantees about the overall system.

### 1.1.2 CompCert and CertiKOS

To an extent, these principles are already demonstrated in the structure of the certified C compiler CompCert [Leroy, 2009].

The introduction of CompCert a decade ago represented a breakthrough in the scale and feasability of certified software. Meanwhile, CompCert itself has been used as a platform other projects have built upon. For example, verification tools have been created with soundness proofs connecting to CompCert [Appel, 2011; Jourdan, 2016]. The composition techniques used to verify CompCert have been extended in various directions [Kang et al., 2016; Song et al., 2019; Stewart et al., 2015] and have provided a blueprint in the formalization of the *certified abstraction layers* used to verify the operating system kernel CertiKOS [Gu et al., 2015, 2016, 2018].

In CompCert, the semantics of intermediate languages serve as intermediate specifications for each compilation pass. The correctness of each pass is established by proving that the behavior of its target program refines that of its source program, which serves in this case as the specification. As passes are composed to obtain the overall C-to-assembly compiler, their correctness proofs are composed as well to construct a proof for the whole compiler. The final theorem does not mention the intermediate programs or language semantics, so that a user only needs to trust the accuracy of the C and assembly semantics, and the soundness of the proof assistant.

Likewise, the CertiKOS verification effort divided the kernel into several dozen abstraction layers which were then specified and verified individually. Layer specifications provide an abstract view of a layer's functionality, hiding the procedural details and low-level data representations involved in its implementation. Client code can be verified in terms of this abstract view in order to build higher-level layers. Certified layers with compatible interfaces can then be chained together in the way passes of a compiler can be composed when the target language of one corresponds to the source language of the other.

### 1.1.3 Challenges

While this approach is compelling, there are difficulties associated with extending it to build larger-scale certified systems by connecting disparate certified components. A key aspect enabling composition in CompCert and CertiKOS is the uniformity of the models underlying their language semantics and correctness proofs. By contrast, across certified software projects there exist a great diversity of semantic models and verification techniques. This makes it difficult to formulate interface specifications to connect specific components, let alone devise a general system to express such interfaces.

Worse yet, this diversity is not simply a historical accident. The semantic models used in the context of individual verification projects are often carefully chosen to make the verification task tractable. The semantic model used in CompCert alone has changed multiple times, addressing new requirements and techniques that were introduced alongside new compiler features and optimizations [Leroy, 2012]. Given the difficulty of verification, preserving this flexibility is essential.

Then, to make it possible to link components verified using a variety of paradigms, we need to identify a model expressive enough to embed the semantics, specifications and correctness proofs of these paradigms. To enable constructing large-scale certified systems, the model should provide high-level composition and reasoning principles.

## 1.2 General models for system behaviors

Fortunately, there is a wealth of semantics research to draw from when attempting to design models for this task.

### 1.2.1 Symmetric monoidal categories

The framework of symmetric monoidal categories, which allows components to be connected in series ($\circ$) and in parallel ($\otimes$), captures structures found in various kinds of systems and processes [Baez and Stay, 2010], and appears in different forms in many approaches to logic and programming language semantics.

### 1.2.2 Game semantics

A particularly expressive instance of this phenomenon is realized in *game semantics* [Abramsky, 2010], an approach to compositional semantics which uses two-player games to model the interaction between a component and its environment, and represents the externally observable behavior of the component as a strategy in this game. The generality of games as descriptions of the possible interactions of components makes this approach broadly applicable, and the typed aspect of the resulting models makes it ideal to the task of describing the behavior of heterogeneous systems.

However, the generality of game models often translates to a fair amount of complexity, which imposes a high barrier to entry for practitioners and makes them difficult to formalize in a proof assistant.

### 1.2.3 Algebraic effects

While more restricted, the framework of *algebraic effects* [Plotkin and Power, 2001] is sufficient for many modeling tasks, fits within the well-known monadic approach to effectful and interactive computations, and can be adapted into a particularly simple version of game semantics. Along these lines, *interaction trees* [Xia et al., 2019] have been developed for use in and across several DeepSpec projects.

### 1.2.4 The refinement calculus

Finally, while game models have been proposed for a wide variety of programming languages, there has been comparatively less focus on specifications and correctness properties. By contrast, the general approach of *stepwise refinement* suggests a uniform treatment of programs, specifications and their relationships. It has been studied extensively in the context of predicate trans-

former semantics [Dijkstra, 1975] and in the framework known as the *refinement calculus* [Back and Wright, 1998].

In refinement-based approaches, programs and specifications are expressed in a common language, and a certified program is constructed in an incremental manner, by applying a series of correctness-preserving transformation to the (abstract and declarative) specification until we obtain a (concrete and executable) program. Correctness preservation is expressed by a reflexive and transitive *refinement relation*. Language constructions are monotonic with respect to this relation, so that elementary refinement rules can be applied congruently within any program context.

## 1.3 Contributions

The central claim if this dissertation is that a synthesis of game semantics, algebraic effects, and the refinement calculus can be used to construct a hierarchy of semantic models suitable for constructing large-scale, heterogeneous certified systems. My contributions are twofold.

### 1.3.1 General-purpose models for certified systems

To provide evidence for this claim, we outline general techniques which can realize this synthesis and demonstrate their use in the context of certified abstraction layers:

- We adapt the work of Morris and Tyrrell [Morris, 2004; Morris and Tyrrell, 2008], which extends the refinement calculus to the level of terms by using *free completely distributive completions* of posets, to investigate *dual nondeterminism* in the context of game semantics and construct completely distributive lattices of *strategy specifications*, partially ordered under a form of alternating refinement [Alur et al., 1998].

- In §5, we define a version of the *free monad on an effect signature* which incorporates dual nondeterminism and refinement. The result can be used to formulate a theory of certified abstraction layers in which layer interfaces, layer implementations, and simulation relations are treated uniformly and compositionally (§3).

- In §6, we outline a more general category of games and strategy specifications; its object are effect signatures regarded as games and its morphisms specify well-bracketed strategies. The

behavior of certified abstraction layers can then be represented canonically, and reentrant layer interfaces can be modeled.

The model presented in §5 was designed to be simple but general enough to embed CompCert semantics, certified abstraction layers, and interaction trees. The main purpose for the model presented in §6 is then to hide state and characterize certified abstraction layers through their externally observable interactions only.

Note that rather than providing denotational semantics for specific programming languages, our models are intended as a coarse-grained composition "glue" between components developed and verified in their own languages, each equipped with their own internal semantics. In this context, the models' restriction to first-order computation applies only to cross-component interactions, and conforms to our interest in connecting low-level system components.

### 1.3.2 CompCertO

Given the importance of compilers in the construction of present-day computer systems, and of CompCert in the formal methods landscape, its integration to any framework attempting to tackle end-to-end verification should be a litmus test. However, this new application of CompCert also comes with its own challenges.

The second part of this thesis presents CompCertO, the first extension of CompCert suitable for these purposes:

- In §7, I summarize the approach to semantics and correctness used in CompCert, as well as the space extensions which have been proposed to make the correctness theorem of CompCert more compositional. Game semantics and dual nondeterminism provide a powerful lens through which the existing design and extensions can be examined and understood. Going beyond the idea of a *certified compiler*, I then identify a set of requirements for using CompCert as a *compiler of certified components*.

- The design of CompCertO is presented in §8. CompCertO is the first extension of CompCert which satisfies all the requrements I have identified. It generalize the CompCert semantic model to express interactions between compilation units, using *language interfaces* to describe the form of these interactions and *simulation conventions* to describe the correspon-

dance between the interfaces of source and target languages. The behavior of composite programs is specified by a *horizontal composition* operator which is shown to be correctly implemented by the existing linking operator for assembly programs. To combine and reason about simulation proofs, we introduce a rich *simulation convention algebra* and use it to derive our main compiler correctness statement.

- Compositional relational reasoning within CompCertO is explained in §9: *CompCert Kripke logical relations* unify CompCert's memory transformations as structure-preserving relations over the memory model. They can be used to define the simulation conventions used by most of the compiler's passes, and derive parametricity theorems which capture important properties of CompCert languages.

- Several passes of CompCert use typing and abstract interpretation soundness invariants. §10 explains how they fit in the simulation framework of CompCertO how the techniques used to verify these passes can be reified into a notion of *simulation modulo invariants*.

- Finally, §11 discusses the more specialized simulation conventions used for the passes of CompCert which significantly transform the shape of interactions across compilation units.

The simulation proofs for most of CompCert's passes can be updated with minimal effort. Simulation conventions can be defined which capture the internal invariants used by existing proofs, avoiding many sources of complexity found in previous work.

# Chapter 2

# Background

## 2.1 Building certified systems

The goal of certified system design is to create a formal description of the system to be constructed (the program), while ensuring through careful analysis that the system will behave properly. To this end, we assign to every system $p \in P$ a mathematical object $[\![p]\!] \in \mathbb{D}$ representing its behavior. We will call the set $\mathbb{D}$ a *semantic domain*. In this section we elucidate the structure and properties of $\mathbb{D}$ necessary to the process of builing large-scale certified systems.

### 2.1.1 Specifications and refinement

System design starts with a set of requirements on the behavior of the system to be constructed (the specification). These requirements do not capture every detail of the eventual system, but delineate a range of acceptable behaviors.

In refinement-based approaches, programs and specifications are interpreted in the same semantic domain $\mathbb{D}$, which is equipped with a *refinement* preorder $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$. The proposition $\sigma_1 \sqsubseteq \sigma_2$ asserts that $\sigma_2$ is a refinement of $\sigma_1$, and in particular a system description $p \in P$ is a correct implementation of $\sigma \in \mathbb{D}$ when $\sigma \sqsubseteq [\![p]\!]$.

### 2.1.2 Compositionality

Complex systems are built by assembling components whose behavior is understood, such that their interaction achieves a desired effect. The syntactic constructions of the language used to

describe systems correspond to the ways in which they can be composed.

To enable compositional reasoning, a suitable model must provide an account of the behavior of the composite system in terms of the behavior of its parts. For instance, if the language contains a binary operator $+ : P \times P \to P$, then the semantic domain should be equipped with a corresponding operation $\oplus : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ such that $[\![p_1 + p_2]\!] = [\![p_1]\!] \oplus [\![p_2]\!]$.

### 2.1.3 Monotonicity

Once a component has been shown to conform to a given specification, we want to abstract it as a "black box" so that further reasoning can be done in terms of the component's specification rather than its implementation details. To support this, we must establish that semantic composition operators are compatible with refinement:

$$\sigma_1 \sqsubseteq \sigma_1' \wedge \sigma_2 \sqsubseteq \sigma_2' \Rightarrow \sigma_1 \oplus \sigma_2 \sqsubseteq \sigma_1' \oplus \sigma_2' \,.$$

Suppose we have two components $p_1$ and $p_2$, where $p_2$ relies on $p_1$ for its operation, and we want to verify that their combination $p_1 + p_2$ satisfies a specification $\sigma$. Once we verify $p_1$ against its own specification $\sigma_1 \sqsubseteq [\![p_1]\!]$, by the monotonicity of $\oplus$ it is sufficient to show that $\sigma \sqsubseteq \sigma_1 \oplus [\![p_2]\!]$:

$$\sigma \;\sqsubseteq\; \sigma_1 \oplus [\![p_2]\!] \;\sqsubseteq\; [\![p_1]\!] \oplus [\![p_2]\!] \;=\; [\![p_1 + p_2]\!]$$

### 2.1.4 Abstraction

Large-scale systems operate across multiple levels of abstraction. Each level brings to its own understanding of the interaction between a component and its environment. To relate abstraction layers we need to give an explicit account of how their formulations of the interaction correspond to one another.

[XXX improve and expand: soundness relation in abstract interpretation, how it relates to the refinement relation, Galois connexions] One possibility is to define a heterogenous version of the refinement relation $\sqsubseteq_{\mathbb{R}} \subseteq \mathbb{D}_1 \times \mathbb{D}_2$ between the abstract domain $\mathbb{D}_1$ and the more concrete one $\mathbb{D}_2$, where $\mathbb{R}$ specifies a correspondance between the two views of the system.

### 2.1.5 Compilers

Abstraction is particularly relevant in the context of compilers. For example, between C and assembly interactions across compilation units are understood very differently. At the level of C, cross-module interaction is defined in terms of function calls; invoking a function consists of assigning values to the function's parameters, initializing a new stack frame, and finally executing the function's body. At the assembly level, cross-module interactions simply consist in branching to an address outside the current module with a certain register state.

In that context, the correpondance between the source and target semantic domains depends on the *calling convention* used by the compiler. The correctness property of a C-to-assembly compiler can then be stated as:

$$[\![p]\!]_s \sqsubseteq_{\mathbb{R}} [\![C(p)]\!]_t \,.$$

where $p$ is the source program, $C(p)$ the compiler's output, $[\![-]\!]_s$ gives the semantics of the source language, $[\![-]\!]_t$ gives the semantics of the target language, and $\mathbb{R}$ formalizes the C calling convention.

### 2.1.6 Embedding

[XXX: explain what it means for a model to embed in another, why we would want to do that, and return to the challenge of designing models suitable for a variety of verification paradigms]

## 2.2 The refinement calculus

Correctness properties for imperative programs are often stated as triples of the form $P\{C\}Q$ asserting that when the program $C$ is started in a state which satisfies the predicate $P$ (the *precondition*), then the state in which $C$ terminates will satisfy the predicate $Q$ (the *postcondition*). For example:

$$\mathsf{x} \text{ is odd } \{\mathsf{x} := \mathsf{x} * 2\} \, \mathsf{x} \text{ is even}$$

In the *axiomatic* approach [Hoare, 1969] to programming language semantics, inference rules corresponding to the different constructions of the language determine which triples are valid, and the meaning of a program is identified with the set of properties $P\{-\}Q$ which the program satisfies.

### 2.2.1 Dual nondeterminism

Axiomatic semantics can accommodate nondeterminism in two different ways. In the program $C_1 \sqcap C_2$, a *demon* will choose which of $C_1$ or $C_2$ is executed. The program $\mathsf{x} := 2 * \mathsf{x} \sqcap \mathsf{x} := 0$ may be executed arbitrarily as $\mathsf{x} := 2 * \mathsf{x}$ or $\mathsf{x} := 0$, with no guarantee as to which branch will be chosen. The demon works against us, so that if we want $C_1 \sqcap C_2$ to satisfy a given property, we need to make sure we can deal with either choice. This corresponds to the inference rule:

$$\frac{P\{C_1\}Q \qquad P\{C_2\}Q}{P\{C_1 \sqcap C_2\}Q}$$

Conversely, in the program $C_1 \sqcup C_2$, an *angel* will decide whether $C_1$ or $C_2$ is executed. If possible, the angel will make choices which validate the correctness property. This implies:

$$\frac{P\{C_1\}Q}{P\{C_1 \sqcup C_2\}Q} \qquad \frac{P\{C_2\}Q}{P\{C_1 \sqcup C_2\}Q}$$

The statement $\mathsf{x} := \mathsf{x} * 2 \sqcup \mathsf{x} := 0$ is more difficult to interpret than its demonic counterpart, but can be thought of as a program which magically behaves as $\mathsf{x} := \mathsf{x} * 2$ or $\mathsf{x} := 0$ depending on the needs of its user.

More generally, a triple $P\{C\}Q$ can be interpreted as a *game* between the angel and the demon [Back and Wright, 1998]. The angel resolves the $\sqcup$ choices, whereas the demon resolves $\sqcap$. The triple is valid if there is a winning strategy for the angel.

### 2.2.2 Distributivity

Note that in this setup, $\sqcap$ and $\sqcup$ distribute over each other. More precisely, for all $P, C_1, C_2, C_3, Q$:

$$P\{C_1 \sqcap (C_2 \sqcup C_3)\}Q \Leftrightarrow P\{(C_1 \sqcap C_2) \sqcup (C_1 \sqcap C_3)\}Q$$
$$P\{C_1 \sqcup (C_2 \sqcap C_3)\}Q \Leftrightarrow P\{(C_1 \sqcup C_2) \sqcap (C_1 \sqcup C_3)\}Q$$

Consider the first equivalence above. If the angel has a winning strategy for the left-hand side triple, they can win both $P\{C_1\}Q$ and either $P\{C_2\}Q$ or $P\{C_3\}Q$. Although the right-hand side triple reverses the angel and demon's choices, the angel can preemptively choose the left or right

disjunct depending on whether they can win $P\{C_2\}Q$ or $P\{C_3\}Q$. Likewise, if the angel can win the right-hand side, then they have a winning strategy for the left-hand side as well. The second equivalence corresponds to a similar situation where the angel and demon have been exchanged.

### 2.2.3 Program refinement

Instead of proving program correctness in one go, *stepwise refinement* techniques use a more incremental approach centered on the notion of program refinement. A refinement $C_1 \sqsubseteq C_2$ means that any correctness property satisfied by $C_1$ will also be satisfied by $C_2$:

$$C_1 \sqsubseteq C_2 \;:=\; \forall PQ \cdot P\{C_1\}Q \Rightarrow P\{C_2\}Q$$

We say that $C_2$ refines $C_1$ or that $C_1$ is refined by $C_2$.

Typically, under such approaches, the language will be extended with constructions allowing the user to describe abstract specifications as well as concrete programs. Then the goal is to establish a sequence of refinements $C_1 \sqsubseteq \cdots \sqsubseteq C_n$ to show that a program $C_n$ involving only executable constructions correctly implements a specification $C_1$, which may be stated in much more abstract terms.

If the language is sufficiently expressive, then a correctness property $P\{-\}Q$ can itself be encoded [Morgan, 1988] as a specification statement $\langle P, Q\rangle$ such that:

$$P\{C\}Q \;\Leftrightarrow\; \langle P, Q\rangle \sqsubseteq C\,.$$

In the context of refinement, the properties associated with demonic and angelic choice generalize as:

$$C \sqsubseteq C_1 \;\wedge\; C \sqsubseteq C_2 \;\Rightarrow\; C \sqsubseteq C_1 \sqcap C_2$$

$$C \sqsubseteq C_1 \;\vee\; C \sqsubseteq C_2 \;\Rightarrow\; C \sqsubseteq C_1 \sqcup C_2$$

Given the symmetry between the demon and angel, it is then natural to interpret demonic and angelic choices respectively as meets and joins of the refinement ordering.

### 2.2.4 Nondeterministic choice in specifications

Until this point, we have discussed demonic ($\sqcap$) and angelic ($\sqcup$) choices as implementation constructs (appearing to the right of $\sqsubseteq$), taking the point of view of a client seeking to use the program to achieve a certain goal. However, in this work they are used primarily as *specification* constructs (appearing to the left of $\sqsubseteq$), and we are interested in what it means for a system to implement them. As a specification, $C_1 \sqcap C_2$ allows the system to refine either one of $C_1$ or $C_2$, while $C_1 \sqcup C_2$ requires it to refine *both* of them:

$$C_1 \sqsubseteq C \;\vee\; C_2 \sqsubseteq C \;\Rightarrow\; C_1 \sqcap C_2 \sqsubseteq C$$
$$C_1 \sqsubseteq C \;\wedge\; C_2 \sqsubseteq C \;\Rightarrow\; C_1 \sqcup C_2 \sqsubseteq C$$

In other words, demonic choices give us more implementation freedom, whereas angelic choices make a specification stronger and more difficult to implement. Therefore we can think of demonic choices as choices of the *system*, and think of angelic choices as choices of the *environment*.

### 2.2.5 The refinement calculus

These basic ingredients have been studied systematically in the *refinement calculus*, dating back to Ralph-Johan Back's 1978 PhD thesis [Back, 1978]. In its modern incarnation [Back and Wright, 1998], the refinement calculus subsumes programs and specifications with *contracts* featuring unbounded angelic and demonic choices. These choice operators constitute a completely distributive lattice with respect to the refinement order.

Dijkstra's *predicate transformer* semantics [Dijkstra, 1975] are a natural fit for the refinement calculus, but other approaches are possible. For instance, the understanding of contracts as a game between the angel and the demon can be formalized to provide a form of game semantics for the refinement calculus.

More generally, The refinement calculus can be presented as a *hierarchy* where simpler models (state transformer functions, relations) can be embedded in more general ones (predicate transformers) in various structure-preserving ways. This makes it possible to reason about simpler components in a limited, stronger version of the framework, while retaining the possibility of em-

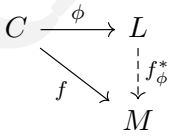
bedding them in a setting where more general constructions are available, and where they can be composed with components developed and analyzed in a different setting. This expressivity is highly desirable as a scalable approach to verifying heterogeneous systems.

However, the refinement calculus only applies to imperative programs with no side-effects beyond changes to the program state. Recent research has attempted to extend the paradigm to a broader setting, and the present work can be understood as a step in this direction as well.

### 2.2.6 Dually nondeterministic functions

Morris and Tyrrell were able to extend the lattice-theoretic approach used in the refinement calculus to functional programming [Morris, 2004; Morris and Tyrrell, 2008; Tyrrell et al., 2006]: if types are interpreted as posets, dual nondeterminism can be added at the type level using free completely distributive completions. This allows dual nondeterminism to be used in a variety of new contexts.

**Definition 1.** *A completely distributive lattice $L$ is a* free completely distributive completion *of a poset $C$ if there is a monotonic function $\phi : C \to L$ such that for any completely distributive lattice $M$ and monotonic function $f : C \to M$, there exists a unique complete homomorphism $f^*_\phi : L \to M$ such that $f^*_\phi \circ \phi = f$:*




*A free completely distributive completion of a poset always exists and it is unique up to isomorphism. We write $\mathbf{FCD}(C)$ for the free completely distributive completion of $C$.*

Morris [Morris, 2004] shows that the free completely distributive completion of $(A, \leq)$ can be constructed as one of:

$$\mathbf{FCD}(A, \leq) := \mathcal{DU}(A, \leq) \qquad \mathbf{FCD}(A, \leq) := \mathcal{UD}(A, \leq)$$
$$\phi(a) := {\downarrow}{\uparrow}a \qquad \phi(a) := {\uparrow}{\downarrow}a\,.$$

In the expressions above, $\mathcal{D}$ and $\mathcal{U}$ are themselves completions. A *downset* of a poset $(A, \leq)$ is a

subset $x \subseteq A$ satisfying:

$$\forall\, a, b \in A \cdot a \leq b \wedge b \in x \Rightarrow a \in x\,.$$

Unions and intersections preserve this property, giving rise to the *downset lattice* $\mathcal{D}(A, \leq)$, which consists of all downsets of $(A, \leq)$, ordered by set inclusion ($\subseteq$) with unions as joins and intersections as meets. The dual *upset lattice* $\mathcal{U}(A, \leq)$ is ordered by set containment ($\supseteq$) with intersections as joins and unions as meets.

Categorically speaking, $\mathbf{FCD} : \mathbf{Poset} \to \mathbf{CDLat}$ is the left adjoint to the forgetful functor $U : \mathbf{CDLat} \to \mathbf{Poset}$ from the category $\mathbf{CDLat}$ of completely distributive lattices and complete homomorphisms to the category $\mathbf{Poset}$ of partially ordered sets and monotonic functions. As such, $U \circ \mathbf{FCD}$ is a monad over $\mathbf{Poset}$, and can be used to model dual nondeterminism as an effect. In the remainder of this thesis, I will identify $\mathbf{FCD}$ with the monad $U \circ \mathbf{FCD}$, and refer to the function $U(f^*_\phi)$ as the $\mathbf{FCD}$ *extension* of $f$.

Computationally, the $\mathbf{FCD}$ monad can be used to interpret dual nondeterminism as an effect. As usual, $\phi(a) \in \mathbf{FCD}(a)$ corresponds to a computation which terminates immediately with the outcome $a \in A$. For a computation $x \in \mathbf{FCD}(A)$ and for $f : A \to \mathbf{FCD}(B)$, the computation $f^*_\phi(x) \in \mathbf{FCD}(B)$ replaces any outcome $a$ of $x$ with the computation $f(a)$. We will use the notation $a \leftarrow x; f(a)$ for $f^*_\phi(x)$, or simply $x; y$ when $f$ is constant with $f(a) = y$.

A computation $x \in \mathbf{FCD}(a)$ can be understood as a *structured* collection of possible outcomes. More precisely, each element $x \in \mathbf{FCD}(A)$ can be written as $x = \bigsqcap_{i \in I} \bigsqcup_{j \in J_i} \phi(a_{ij})$ where the index $i \in I$ ranges over possible demonic choices, the index $j \in J_i$ ranges over possible angelic choices, and $a_{ij} \in A$ is the corresponding outcome of the computation. Note that $f^*_\phi(x) = \bigsqcap_{i \in I} \bigsqcup_{j \in J_i} f(a_{ij})$.

The algebraic properties of lattices underlie the model's insensitivity to *branching*. Complete distributivity:

$$\bigsqcap_{i \in I} \bigsqcup_{j \in J_i} x_{i,j} = \bigsqcup_{f \in (\prod_i J_i)} \bigsqcap_{i \in I} x_{i,f_i}$$

further allows angelic and demonic choices to commute, and the status of $f^*_\phi$ as a complete homo-

morphism enables the following properties:

$$a \leftarrow \left(\bigsqcup_{i \in I} x_i\right); M[a] = \bigsqcup_{i \in I} (a \leftarrow x_i; M[a])$$
$$a \leftarrow \left(\bigsqcap_{i \in I} x_i\right); M[a] = \bigsqcap_{i \in I} (a \leftarrow x_i; M[a])$$
$$a \leftarrow x; b \leftarrow y; M[a, b] = b \leftarrow y; a \leftarrow x; M[a, b]$$

Finally, the least element $\bot := \bigsqcup \varnothing$, traditionally called abort, merits some discussion. As a specification construct, it places no constraint on the implementation (it is refined by every element). As an implementation construct, we use it indiscriminately to interpret failure, silent divergence, and any other behavior which we want to exclude (it refines only itself). The *assertion* $\{P\} \in \mathbf{FCD}(\mathbb{1})$ of a proposition $P$ evaluates to the unit value $\phi(*)$ when the proposition is true and to $\bot$ otherwise. We will use it to formulate guards blocking a subset of angelic choices.

## 2.3 Game semantics

Game semantics [Abramsky and McCusker, 1999; Blass, 1992] is a form of denotational semantics which incorporates some operational aspects. An early success of this approach was the formulation of the first fully abstract models of the programming language PCF [Abramsky et al., 2000; Hyland and Ong, 2000]. Typically, game semantics interpret types as two-player games and terms as strategies for these games. Games describe the form of the interaction between a program component (the *system*) and its execution context (the *environment*). Strategies specify which move the system plays for all relevant positions in the game.

Positions are usually identified with sequences of moves, and strategies with the set of positions a component can reach. This representation makes game semantics similar to trace semantics of process algebras, but it is distinguished by a strong polarization between actions of the system and the environment, and between outputs and inputs. This confers an inherent "rely-guarantee" flavor to games which facilitates compositional reasoning [Abramsky, 2010].

For example, in a simple game semantics resembling that of Idealized Algol [Abramsky and McCusker, 1997], sequences of actions corresponding to the execution of $\mathsf{x} := 2 * \mathsf{x}$ could have the

form:

$$\mathsf{run} \cdot \underline{\mathsf{rd}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{wr}_{\mathsf{x}}[2n]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}} \quad (n \in \mathbb{N})$$

The moves of the system have been underlined. The environment initiates the execution with the move run. The system move $\mathsf{rd}_{\mathsf{x}}$ requests the value of the variable x, communicated in response by the environment move $n$. The system move $\mathsf{wr}_{\mathsf{x}}[2n]$ requests storing the value $2n$ into the variable x, and is acknowledged by the environment move ok. Finally, the system move done expresses termination.

### 2.3.1 Games

A game is defined by a set of moves players will choose from, as well as a stipulation of which sequences of moves are valid. We focus on two-player, alternating games where the environment plays first and where the players each contribute every other move.

When typesetting examples, we underline the moves of the system. For chess, moves are taken in the set $\{a1 \dots h8\} \times \{a1 \dots h8\}$, and a valid sequence of moves may look like:

$$e2e4 \cdot \underline{c7c5} \cdot c2c3 \cdot \underline{d7d5} \cdots$$

The games we use to model low-level components will rely on the following constructions.

Most game semantics include additional structure in the description of games. The set of moves is usually partitioned into proponent and opponent moves ($M = M^{\mathsf{O}} \uplus M^{\mathsf{P}}$), and into questions and answers ($M = M^{\circ} \uplus M^{\bullet}$). Game models for high-order languages are often more complex, and include *justification pointers* encoding the causal structure of the interaction.

The expressive power of game semantics comes from the ways in which complex games can be derived from simple ones, and used to interpret compound types. For example, in the game $A \times B$ the environment initially chooses whether to play an instance of $A$ or an instance of $B$. The game $A \to B$ usually consists of an instance of $B$ played together with instances of $A$ started at the discretion of the system, where the roles of the players are reversed.

### 2.3.2 Strategies

The *plays* of a game are sequences of moves; they identify a position in the game and describe the succession of actions that led to it. Most game models of sequential computation use *alternating* plays, in which the system and environment each contribute every other move. It is also common to require the environment to play first and to restrict plays to even lengths, so that they specify which action the system took in response to the latest environment move. We write $P_G$ for the set of plays of the game $G$, partially ordered by the prefix relation $\sqsubseteq_{\mathsf{p}}$.

Traditionally [Abramsky and McCusker, 1999], strategies are defined as prefix-closed sets of plays, so that strategies $\sigma \in S_G$ for the game $G$ are downsets of $P_G$ satisfying certain requirements:

$$S_G \subseteq \mathcal{D}(P_G, \sqsubseteq_{\mathsf{p}})$$

A play $s \in P_G$ can be promoted to a strategy $\downarrow s \in \mathcal{D}(P_G, \sqsubseteq_{\mathsf{p}})$:

$$\downarrow s := \{t \in P_G \mid t \sqsubseteq_{\mathsf{p}} s\}$$

Set inclusion corresponds to strategy refinement, and the downset completion augments $P_G$ with angelic choices.

### 2.3.3 Nondeterminism

A common constraint is that a strategy $\sigma \in S_G$ should not contain two plays $sm_1, sm_2 \in \sigma$ where $m_1$ and $m_2$ are distinct moves of the system. This is usually understood as enforcing *system determinism*: given a set of environment choices, there is only one possible behavior for the system. Therefore, relaxing this constraint has usually been understood as the first step toward modeling nondeterministic systems [Harmer and McCusker, 1999]. However, looking at the situation through the lens of refinement and dual nondeterminism yields to a different conclusion, where this requirement corresponds to *environment determinacy*.

Angelic nondeterminism allows us to range over all possible choices of the environment and

record the resulting plays. For instance, the strategy for $\mathsf{x} := 2 * \mathsf{x}$ would be:

$$\begin{aligned}\sigma &:= \bigcup_{n\in\mathbb{N}} {\downarrow}(\mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}} \cdot n \cdot \underline{\mathsf{wr}_\mathsf{x}[2n]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}) \\ &= \{\epsilon, \quad \mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}}, \quad \mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}} \cdot n \cdot \underline{\mathsf{wr}_\mathsf{x}[2n]}, \quad \mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}} \cdot n \cdot \underline{\mathsf{wr}_\mathsf{x}[2n]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}} \mid n \in \mathbb{N}\}\end{aligned}$$

Note that this strategy admits refinements containing much more angelic nondeterminism, including with respect to moves of the system. For instance:

$$\sigma \subseteq \bigcup_{n\in\mathbb{N}} \bigcup_{-1\leq\delta\leq 1} {\downarrow}(\mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}} \cdot n \cdot \underline{\mathsf{wr}_\mathsf{x}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}})$$

These refinements do not correspond to interpretations of concrete programs, and in game models which seek to achieve definability they are usually excluded. In our context, retaining them is algebraically important, and they can in fact appear as intermediate terms in some applications. In the construction above, although $\delta$ appears in a system move, it is still associated with an *angelic* choice. This can be interpreted as a choice of the environment which is not directly observed (perhaps as a result of abstraction), but which nonetheless influences the behavior of the system.

This is quite different from allowing the *system* to choose an answer in the interval $[2n - 1, 2n + 1]$. The model and refinement lattice which we have presented so far are insufficient to express such a specification, because the downsets do not add enough meets, forcing the would-be specification to become much coarser:

$$\begin{aligned}\sigma' &:= \bigcup_{n\in\mathbb{N}} \bigcap_{-1\leq\delta\leq 1} {\downarrow}(\mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}} \cdot n \cdot \underline{\mathsf{wr}_\mathsf{x}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}) \\ &= \{\epsilon,\ \mathsf{run} \cdot \underline{\mathsf{rd}_\mathsf{x}} \mid n \in \mathbb{N}\}\,.\end{aligned}$$

We will remedy this by using a richer completion, as explained in §4.

## 2.4 Logical relations

Logical relations are structure-preserving relations in the way homomorphisms are structure-preserving maps. However, logical relations are more compositional than homomorphisms, be-

cause they do not suffer from the same problems in the presence of mixed-variance constructions like the function arrow [Hermida et al., 2014]. In the context of typed languages, this means that type-indexed logical relations can be defined by recursion over the structure of types.

Logical relations have found widespread use in programming language theory. Unary logical relations can be used to establish various properties of type systems: a type-indexed predicate expressing a property of interest is shown to be compatible with the language's reduction, and to contain all of the well-typed terms of the language. Binary logical relations can be used to capture contextual equivalence between terms, as well as notions such as non-interference or compiler correctness. Relational models of type quantification yield Reynold's well-known theory of relational parametricity, and can be used to prove *free theorems* that all terms of a given parametric type must satisfy.

### 2.4.1 Binary logical relations

Logical relations can be of any arity, but we restrict our attention to binary logical relations. Given an algebraic structure $\mathcal{S}$, a *logical relation* between two instances $S_1, S_2$ of $\mathcal{S}$ is a relation $R$ between their carrier sets, such that the corresponding operations of $S_1$ and $S_2$ take related arguments to related results. We write $R \in \mathcal{R}(S_1, S_2)$.

**Example 1.** *A monoid is a set with an associative operation $\cdot$ and an identity element $\epsilon$. A* logical relation of monoids *between $\langle A, \cdot_A, \epsilon_A \rangle$ and $\langle B, \cdot_B, \epsilon_B \rangle$ is a relation $R \subseteq A \times B$ such that:*

$$(u\ R\ u' \wedge v\ R\ v' \ \Rightarrow\ u \cdot_A v\ \ R\ \ u' \cdot_B v') \ \wedge\ \epsilon_A\ R\ \epsilon_B\,. \tag{2.1}$$

### 2.4.2 Relators

Logical relations between multisorted structures consist of one relation for each sort, between the corresponding carrier sets. In the case of structures which include type operators, we can associate to each base type $A$ a relation over its carrier set $[\![A]\!]$, and to each type operator $T(A_1, \ldots, A_n)$ a corresponding *relator*: given relations $R_1, \ldots, R_n$ over the carrier sets $[\![A_1]\!], \ldots, [\![A_n]\!]$, the relator for $T$ will construct a relation $T(R_1, \ldots, R_n)$ over $[\![T(A_1, \ldots, A_n)]\!]$. Relators for some common constructions are shown in Fig. 2.1. In this framework, the proposition (2.1) can be reformulated

$$
\begin{aligned}
x\ [R_1 \times R_2]\ y &\Leftrightarrow \pi_1(x)\ [R_1]\ \pi_1(y) \wedge \pi_2(x)\ [R_2]\ \pi_2(y) \\
x\ [R_1 + R_2]\ y &\Leftrightarrow (\exists\, x_1\, y_1 \,.\, x_1\ [R_1]\ y_1 \wedge x = i_1(x_1) \wedge y = i_1(y_1)) \\
&\quad \vee (\exists\, x_2\, y_2 \,.\, x_2\ [R_2]\ y_2 \wedge x = i_2(x_2) \wedge y = i_2(y_2)) \\
f\ [R_1 \to R_2]\ g &\Leftrightarrow \forall\, x\, y \,.\, x\ [R_1]\ y \Rightarrow f(x)\ [R_2]\ g(y) \\
A\ [\mathcal{P}^{\leq}(R)]\ B &\Leftrightarrow \forall\, x \in A \,.\, \exists\, y \in B \,.\, x\ [R]\ y
\end{aligned}
$$

Figure 2.1: A selection of relators

as:

$$\cdot_A\ [R \times R \to R]\ \cdot_B\ \wedge\ \epsilon_A\ R\ \epsilon_B\,.$$

**Example 2.** *Simulation relations are logical relations of transition systems. Consider the transition systems $\alpha : A \to \mathcal{P}(A)$ and $\beta : B \to \mathcal{P}(B)$. A simulation relation $R \in \mathcal{R}(A, B)$ satisfies:*

$$
\begin{array}{ccc}
s_1 & \xrightarrow{\ \alpha\ } & s_1' \\
{\scriptstyle R}\big| & & \vdots\ {\scriptstyle R} \\
s_2 & \xrightarrow{\ \beta\ } & s_2'
\end{array}
\qquad
\begin{aligned}
&\forall s_1\, s_2\, s_1' \,.\, \alpha(s_1) \ni s_1' \wedge s_1\ R\ s_2 \Rightarrow \\
&\exists s_2' \,.\, \beta(s_2) \ni s_2' \wedge s_1'\ R\ s_2'
\end{aligned}
$$

*Using the relators in Fig. 2.1, we can express the same property concisely and compositionally as:*

$$\alpha\ [R \to \mathcal{P}^{\leq}(R)]\ \beta\,.$$

### 2.4.3 Kripke relations

Since relations for stateful languages often depend on the current state, Kripke logical relations are parametrized over a set of state-dependent *worlds*. Components related at the same world are guaranteed to be related in compatible ways. We use the following notations.

**Definition 2.** *A* Kripke *relation is a family of relations $(R_w)_{w \in W}$. We write $R \in \mathcal{R}_W(A, B)$ for a Kripke relation between the sets $A$ and $B$. For $w \in W$ we write:*

$$[w \Vdash R]\ :=\ R_w \qquad [\Vdash R]\ :=\ \textstyle\bigcap_w R_w$$

A simple relation $R \in \mathcal{R}(A, B)$ can be promoted to a Kripke relation $\lceil R \rceil \in \mathcal{R}_W(A, B)$ by

defining $[w \Vdash \lceil R \rceil] := R$ for all $w \in W$. More generally, for an $n$-ary relator $F$ we have:

$$\frac{F : \mathcal{R}(A_1, B_1) \times \cdots \times \mathcal{R}(A_n, B_n) \to \mathcal{R}(A, B)}{\lceil F \rceil : \mathcal{R}_W(A_1, B_1) \times \cdots \times \mathcal{R}_W(A_n, B_n) \to \mathcal{R}_W(A, B)}$$

where for the Kripke relations $R_i \in \mathcal{R}_W(A_i, B_i)$:

$$[w \Vdash \lceil F \rceil(R_1, \ldots, R_n)] \;:=\; F(w \Vdash R_1, \ldots, w \Vdash R_n)$$

We use $\lceil - \rceil$ implicitly when a relator appears in a context where a Kripke logical relation is expected. Since reasoning with logical relations often involves self-relatedness, we use the notation $x :: R$ to denote $x \; R \; x$. For legibility, we will also write $w \Vdash x \; R \; y$ for $x \; [w \Vdash R] \; y$ and $\Vdash x \; R \; y$ for $x \; [\Vdash R] \; y$.

## 2.5 Algebraic effects

The framework of *algebraic effects* [Plotkin and Power, 2001] models computations as terms in an algebra whose operations represent effects: a term $m(x_1, \ldots x_n)$ represents a computation which first triggers an effect $m$, then continues as a computation derived from the subcomputations $x_1, \ldots x_n$. For example, the term:

$$\mathsf{readbit}(\mathsf{print}[\text{"Hello"}](\mathsf{done}), \mathsf{print}[\text{"World"}](\mathsf{done}))$$

could denote a computation which first reads one bit of information, then depending on the result causes the words "Hello" or "World" to be output, and finally terminates.

Note that somewhat surprisingly, the *arguments* of operations correspond to the possible *outcomes* of the associated effect. For instance the readbit operation takes two arguments. Moreover, effects such as print which take parameters are represented by *families* of operations indexed by the parameters' values, so that there is a $\mathsf{print}[s]$ operation for every $s \in \mathsf{string}$.

### 2.5.1 Effects and algebraic theories

Under this approach, effects can be described as algebraic theories: a signature describes the set of operations together with their arities, and a set of equations describes their behaviors by specifying which computations are equivalent. The example above uses a signature with the operations done of arity $0$, readbit of arity $2$, and a family of operations $(\mathsf{print}[s])_{s \in \mathsf{string}}$ of arity $1$. An equation for this signature is:

$$\mathsf{print}[s](\mathsf{print}[t](x)) = \mathsf{print}[st](x)\,,$$

which indicates that printing the string $s$ followed by printing the string $t$ is equivalent to printing the string $st$ in one go. In this work, we use effect signatures to represent the possible external interactions of a computation, but we will not use equational theories. We will however make it possible to interpret effects into another signature, modeling a limited form of *effect handlers* [Plotkin and Pretnar, 2009].

### 2.5.2 Effect signatures

**Definition 3.** *An* effect signature *is a set $E$ of operations together with a mapping* $\mathsf{ar}$*, which assigns to each $e \in E$ a set $\mathsf{ar}(e)$ called the* arity *of $e$. We will use the notation $E = \{e_1 : \mathsf{ar}(e_1),\ e_2 : \mathsf{ar}(e_2),\ \ldots\}$ to describe effect signatures.*

Note that in this definition, arities are *sets* rather than natural numbers. This allows the representation of effects with a potentially infinite number of outcomes. The examples above use effects from the following signature:

$$E_{\mathsf{io}} := \{\mathsf{readbit} : \mathbb{2},\ \mathsf{print}[s] : \mathbb{1},\ \mathsf{done} : \varnothing \mid s \in \mathsf{string}\}$$

The most direct way to interpret an effect signature is the algebraic point of view, in which it induces a set of terms built out of the signature's operations.

### 2.5.3 Monads

An effect signature can also be used for describing the interface of a monad $T$, where each effect $e \in E$ corresponds to a computation of type $T(\mathsf{ar}(e))$. Presented as a monadic expression of type

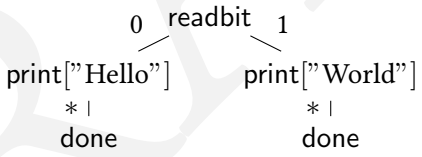
$T(\varnothing)$, the example above corresponds to:

$$b \leftarrow \mathsf{readbit}\ ;\ \mathsf{print}[s_b]\ ;\ \mathsf{done}$$

where $s_0 =$ "Hello" and $s_1 =$ "World". Monads which offer this structure include the *free monad* on the signature $E$, which leaves effects entirely uninterpreted; roughly, its computations of type $A$ correspond to terms over the signature $E \uplus \{v : \varnothing \mid v \in A\}$. The interaction specification monad $\mathcal{I}_E$ presented in §5 is a version of the free monad which combines the dual nondeterminism of **FCD** with uninterpreted effects taken in the signature $E$.

### 2.5.4 Games

Finally, an effect signature can also be seen as a particularly simple *game*, in which the proponent chooses a question $m \in E$ and the opponent responds with an answer $n \in \mathsf{ar}(m)$. Then the terms induced by the signature are *strategies* for an iterated version of this game. Indeed, the abstract syntax tree of our example term can directly be read as the strategy:




where we interpret node labels as moves of the system, and edge labels as moves of the environment.

# Part I

# Refinement-based game semantics

# Chapter 3

# Certified abstraction layers

## 3.1 Introduction

### 3.1.1 Abstraction layers

Software is constructed in layers. The basic facilities provided by the programming environment are used to implement more abstract data structures and operations. The programmer can then forget the details of their implementation, and instead think of them as *primitive* constructions when building the next layer of code.

This core principle is especially relevant in the context of system code, where abstraction layers may transform the programming model significantly. For example, at the level of the bare metal, the memory address space must be managed explicitly, but the operating system's memory management layers provide a much more convenient view of the memory for higher-level code.

This illustrates a conceptual similarity between abstraction layers and compilers. As is the case for abstraction layers, the goal of a compiler is to use a lower-level programming model (like assembly code) to provide a more abstract, higher-level one (as defined for example by the C programming language). For most abstraction layers, the transformation of client code from high- to low-level is less expansive, consisting only in adding a layer's code. nevertheless they can be understood as a specific instance of the more general phenomenon.

| CompCert | CertiKOS |
|---|---|
| Language semantics | Layer interface |
| Source language | Overlay interface |
| Compilation pass | Abstraction layer |
| Target language | Underlay interface |
| Pass correctness | Layer correctness |

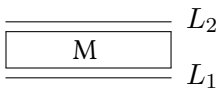
Table 3.1: Correspondance between certified compilation and certified abstraction layers

### 3.1.2 Certified abstraction layers in CertiKOS

These insights are leveraged in the construction of the certified operating system kernel CertiKOS. The verification of CertiKOS builds on CompCert in two different ways. In terms of functionality, CompCert is used to provide language semantics for C and assembly, and to transport code proofs formulated at the level of C to establish guarantees about the kernel's compiled assembly code. But CertiKOS also inherits some of the structures and techniques used in CompCert to establish the compiler's correctness theorem (Tbl. 3.1).

The kernel is divided into several dozen abstraction layers, which are then specified and verified individually. Specifications of abstraction layers are called *layer interfaces*. They extend the programming model formalized in CompCert language semantics to provide a set of *primitives*, which can be invoked in client programs as external functions. The behavior of primitives is described in terms of an *abstract state*, maintained alongside CompCert's memory state, but only updated by primitive invocations.

A layer implementation consists in client code for an *underlay* interface, which implements the primitives described by an *overlay* interface. A layer $M$ implementing the overlay interface $L_2$ on top of the underlay interface $L_1$ can be depicted as follows:

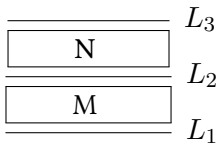



The correctness of $M$ is formulated as the contextual refinement property:

$$\forall C \,.\, [\![C]\!]_{L_2} \sqsubseteq [\![C + M]\!]_{L_1}$$

expressing that the behavior of the client code $C$ running alongside $M$ on top of the underlay

interface $L_1$ will refine the behavior of $C$ evaluated on top of the overlay interface $L_2$. Then, when the underlay interface of one layer corresponds to the overlay interface of another:




the contextual refinement properties can be composed to obtain the composite certified layer:


in the same way the passes of CompCert and their correctness properties can be composed when the target language of one corresponds to the source language of another.

### 3.1.3 Contributions

In our original design [Gu et al., 2015], the formalization of certified abstraction layers was closely tied with the semantic infrastructure of CompCert. Indeed, we introduced the variant CompCertX discussed in §7.4.2 to make it possible for language semantics to use arbitrary underlay interfaces and express the behavior of layer implementations. As we proceeded to verify the 37 layers of CertiKOS in the framework outlined above, it quickly became apparent that the common structures found in the verification of each layer were a source of excessive redundancy in our proofs. This prompted my work on the *layer calculus* of CertiKOS, which became the starting point for the research presented in this thesis.

This chapter presents a modernized version of the layer calculus decoupled from CompCert. The new version avoids the complications of the original model but captures its essential features. Through a series of embeddings, the new model can be interfaced with CompCertO (§8) to provide the capabilities of the original model and more.

## 3.2 Layer model

A layer interface $L$ has three components. First, a *signature* enumerates primitive operations together with their types, given as $\mathsf{op} : A \to B$ where $A$ and $B$ are sets. In terms of Def. 3, this

corresponds to a family $\{\mathsf{op}[a] : B \mid a \in A\}$. Second, the set $S$ contains the *abstract states* of the layer interface. Finally, for each primitive $\mathsf{op} : A \to B$, a *specification* is given as a function:

$$L.\mathsf{op} : A \times S \to \mathcal{P}^1(B \times S)\,.$$

Throughout this thesis, the notation $v@k \in V \times S$ is used for a pair containing the value $v \in V$ and the state $k \in S$.

### 3.2.1 Specification monad

In the type of $L.\mathsf{op}$ above, $\mathcal{P}^1$ corresponds to the *maybe* monad:

$$\mathcal{P}^1(A) := \{x \subseteq A : |x| \leq 1\}\,.$$

where a terminating computation producing the value $v \in A$ is represented as the singleton:

$$\eta_{\mathcal{P}}(v) := \{v\}\,,$$

and where the empty set $\varnothing \in \mathcal{P}^1(A)$ specifies an undefined computation, which is free to silently diverge, crash, or produce any possible outcome. When two computations are sequentially composed, both must be successful for the result to be defined. For $x \in \mathcal{P}(\mathcal{P}(A))$, the multiplication $\mu_{\mathcal{P}}(x) \in \mathcal{P}(A)$ is defined as:

$$\mu_{\mathcal{P}}(x) := \{v \in A \mid \exists\, y\,.\, x \ni y \ni v\}\,.$$

The notion of refinement associated with this monad is set inclusion, which allows an unspecified computation to be refined by a computation with a definite outcome:

$$\varnothing \subseteq \{v\}\,.$$

To account for state, we can combine $\mathcal{P}^1$ with the state monad transformer. For a set of states

$S$, we obtain the monad:

$$\mathcal{L}_S(A) := S \to \mathcal{P}^1(A \times S)\,,$$

equipped with the structure:

$$\eta_{\mathcal{L}}(v) := \lambda\, k\,.\,\{v@k\}$$
$$\mu_{\mathcal{L}}(x) := \lambda\, k\,.\,\{v@k'' \mid \exists\, y\, k'\,.\, x(k) \ni y@k' \wedge y(k') \ni v@k''\}\,.$$

This monad is used to specify the primitives in a layer interface using abstract states in the set $S$.

### 3.2.2 Layer interfaces

The monad $\mathcal{L}_S$ is used to define layer interfaces.

**Definition 4.** *A* layer interface *is a tuple $L = \langle E, S, \pi\rangle$, where $E$ is an effect signature, $S$ is a set of states, and $\sigma$ assigns to each operation $(m : N) \in E$ a specification $\sigma^m \in \mathcal{L}_S(N)$. We will sometimes use the notation $L.m$ to refer to $\sigma^m$.*

As a running example, we will use the certified layer described in Fig. 3.1, which implements a bounded queue with at most $N$ elements using a circular buffer. We outline the construction of the correponding layer interfaces below.

**Example 3** (Bounded queue and ring buffer interfaces)**.** *The layer interface $L_{\mathsf{rb}} = \langle E_{\mathsf{rb}}, S_{\mathsf{rb}}, \sigma_{\mathsf{rb}}\rangle$ describes a bounded queue. Its states are sequences of values, expected to contain at most $N$ elements. The two primitives* enq *and* deq *respectively add a new element to the queue and remove the oldest element. If we attempt to add an element which would overflow the queue's capacity $N$, or remove an element from an empty queue, the result is $\varnothing$ (i.e., the operation aborts).*

*I will demonstrate how the bounded queue interface can be implemented in terms of a circular buffer described by the layer interface $L_{\mathsf{bq}} = \langle E_{\mathsf{bq}}, S_{\mathsf{bq}}, \sigma_{\mathsf{bq}}\rangle$. The states of $L_{\mathsf{rb}}$ contain an array $f \in V^N$ storing $N$ values of type $V$, and two counters taking values in the interval $0 \le c_1, c_2 < N$. The array can be accessed through the primitives* get *and* set*; the primitives* $\mathsf{inc}_1$ *and* $\mathsf{inc}_2$ *increment the corresponding counter and return the counter's old value.*

$$
\begin{array}{|c|}\hline L_{\mathsf{bq}} \\ \hline \end{array}
\qquad
\begin{array}{l}
S_{\mathsf{bq}} := V^* \\
L_{\mathsf{bq}}.\mathsf{enq}(v)@\vec{q} := \{*@\vec{q}v \mid |\vec{q}| < N\} \\
L_{\mathsf{bq}}.\mathsf{deq}(*)@\vec{q} := \{v@\vec{p} \mid \vec{q} = v\vec{p}\}
\end{array}
$$

$$
\mathsf{enq} : V \to \mathbb{1} \qquad \mathsf{deq} : \mathbb{1} \to V
$$

$$
\begin{array}{|c|}\hline M_{\mathsf{bq}} \\ \hline \end{array}
\qquad
R \subseteq S_{\mathsf{bq}} \times S_{\mathsf{rb}}
$$

$$
\begin{array}{l}
M_{\mathsf{bq}}.\mathsf{enq}(v) := i \leftarrow \mathsf{inc}_2;\ \mathsf{set}(i, v) \\
M_{\mathsf{bq}}.\mathsf{deq}(*) := i \leftarrow \mathsf{inc}_1;\ \mathsf{get}(i)
\end{array}
\qquad
\begin{array}{l}
\vec{q}\ R\ (f, c_1, c_2) \Leftrightarrow (c_1 \le c_2 < N \wedge \vec{q} = f_{c_1} \cdots f_{c_2 - 1}) \vee \\
\qquad\qquad\qquad\qquad (c_2 \le c_1 < N \wedge \vec{q} = f_{c_1} \cdots f_{N-1} f_0 \cdots f_{c_2 - 1})
\end{array}
$$

$$
\begin{array}{|c|}\hline L_{\mathsf{rb}} \\ \hline \end{array}
\qquad
\begin{array}{l}
S_{\mathsf{rb}} := V^N \times \mathbb{N} \times \mathbb{N} \\
L_{\mathsf{rb}}.\mathsf{set}(i, v)@(f, c_1, c_2) := \{*@(f', c_1, c_2) \mid i < N \wedge f' = f[i := v]\} \\
L_{\mathsf{rb}}.\mathsf{get}(i)@(f, c_1, c_2) := \{f_i@(f, c_1, c_2) \mid i < N\} \\
L_{\mathsf{rb}}.\mathsf{inc}_1@(f, c_1, c_2) := \{c_1@(f, c_1', c_2) \mid c_1' = (c_1 + 1) \bmod N\} \\
L_{\mathsf{rb}}.\mathsf{inc}_2@(f, c_1, c_2) := \{c_2@(f, c_1, c_2') \mid c_2' = (c_2 + 1) \bmod N\}
\end{array}
$$

$$
\mathsf{set} : \mathbb{N} \times V \to \mathbb{1} \qquad \mathsf{get} : \mathbb{N} \to V \qquad \mathsf{inc}_1 : \mathbb{1} \to \mathbb{N} \qquad \mathsf{inc}_2 : \mathbb{1} \to \mathbb{N}
$$

Figure 3.1: A certified abstraction layer $L_{\mathsf{rb}} \vdash_R M_{\mathsf{bq}} : L_{\mathsf{bq}}$ implementing a bounded queue of size $N$ using a ring buffer. The left-hand side of the figure shows the signatures of the overlay and underlay interfaces, and the code associated with the layer. The right-hand side shows primitive specifications and the simulation relation used by the correctness proof.

### 3.2.3 Client code

Given an underlay interface $L = \langle E, S, \sigma \rangle$, we will use the *free monad* $\mathcal{F}_E$ on the signature $E$ as a general representation for client code.

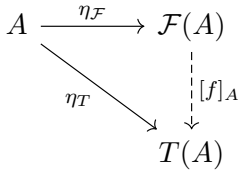
As discussed in §2.5.3, a computation $x \in \mathcal{F}_E(A)$ in the free monad is a term built out of the operations of $E$ and the elements of $A$ regarded as variables, representing a computation which proceeds inward from the top-level operation towards the leaves of the term:

$$
x \in \mathcal{F}_E(A) ::= v \mid m(x_n)_{n \in N} \qquad \text{where } v \in A, (m : N) \in E
$$

Terms of the form $v$ correspond to a computation which immediately terminates with the corresponding results. Terms of the form $m(x_i)_{i \in N}$, where $m : N$ is an operation in the signature $E$, first invoke the primitive $m$. The outcome $n \in N$ then resumes the computation as prescribed by the subterm $x_n \in \mathcal{F}_E(A)$.

The defining property of the free monad is that for any other monad $T$ equipped with an interpretation $f^m \in T(N)$ for each $(m : N) \in E$, there is a unique transformation $[f] : \mathcal{F} \to T$

which preserves the monadic structure. In particular:




I will write $x[f] \in T(A)$ for the image by $[f]$ of $x \in \mathcal{F}(A)$. This transformation uses the interpretation and the monad's multiplication to replace every operation $m$ in the term $x$ by its interpretation $f^m$, and the monad's unit every variable $v \in A$ by $\eta_T(v)$:

### 3.2.4 Layer implementations

A layer implementation provides underlay client code for each operation of the overlay interface.

**Definition 5.** *A* layer implementation *for an underlay signature $E$ and an overlay signature $F$ is a family $M \in \prod_{(m:N)\in F} \mathcal{F}_E(N)$. Then, for an underlay interface $L = \langle E, S, \sigma \rangle$, the* evaluation *of $M$ over $L$ is the layer interface:*

$$M[L] := \langle F, S, M[\sigma] \rangle \,,$$

*where $M[\sigma]^m := M^m[\sigma]$.*

**Example 4.** *The layer implementation $M_{\mathsf{bq}}$ stores the queue's elements into the array, between the indices given by the counters' values. This is expressed by the simulation relation $R$, which explains how overlay states are realized by $M_{\mathsf{bq}}$ in terms of underlay states. The code of $M_{\mathsf{bq}}$ can be interpreted in the monad $S_{\mathsf{rb}} \to \mathcal{P}^1(- \times S_{\mathsf{rb}})$, with calls to primitives of $L_{\mathsf{rb}}$ replaced by their specifications, to obtain the layer interface $M_{\mathsf{bq}}[L_{\mathsf{rb}}]$.*

### 3.2.5 Correctness

Consider an underlay interface $L_1$, an overlay interface $L_2$, and a corresponding layer implementation $M$. The layer interface $M[L_1]$ shares a signature with $L_2$, but its behavior is described in terms of the states of $L_1$. To establish correctness, we need to provide a simulation relation $R \subseteq S_2 \times S_1$ explaining the correspondance between the more abstract states of $L_2$ and the more concrete states of $L_1$. Using the relators $\to$, $\times$, $\mathcal{P}^{\leq}$ presented in §**??**, the associated simulation

property can be defined as follows.

**Definition 6** (Simulation of layer interfaces)**.** *For a signature $E$, two layer interfaces $L_1 = \langle E, S_1, \sigma_1 \rangle$ and $L_2 = \langle E, S_2, \sigma_2 \rangle$, and a relation $R \subseteq S_2 \times S_2$, we say that $L_2$ is* simulated by *$L_1$ and write $L_2 \leq_R L_1$ when the following simulation property holds for each operation $(m : N) \in E$:*

$$L_2.m \;\; [R \to \mathcal{P}^{\leq}(= \times R)] \;\; L_1.m\,. \tag{3.1}$$

Then, layer correctness is established by showing a simulation between the overlay interface and the layer implementation.

**Definition 7** (Certified abstraction layer)**.** *For an underlay interface $L_1 = \langle E_1, S_1, \sigma_1 \rangle$, an overlay interface $L_2 = \langle E_2, S_2, \sigma_2 \rangle$ and a layer implementation $M : E_1 \to E_2$, we say that $M$ is a* correct implementation of $L_2$ on top of $L_1$ *when there is a simulation relation $R \subseteq S_2 \times S_1$ such that the following property holds:*

$$L_1 \vdash_R M : L_2 \quad :\Leftrightarrow \quad L_2 \leq_R M[L_1]\,.$$

*We say that $L_1$, $M$, $L_2$ together constitute a* certified abstraction layer.

**Example 5** (XXX fill in with stuff from slides)**.**

**Theorem 1** (Soundness)**.** *For a certified abstraction layer $L_1 \vdash_R M : L_2$, the following* contextual refinement *property holds:*

$$\forall\, C \in \mathcal{F}_{E_2}(A)\,. \quad C[L_2] \;\leq_R\; (C \circ M)[L_1]$$

*Proof.* By the monotonicity of substitution and the fact that $(C \circ M)[L_1] = C[M \circ L_1]$. □

### 3.2.6 Composing certified abstraction layers

Certified abstraction layers presented in this way can be vertically composed as expected:

$$\frac{L_1 \vdash_R N : L_2 \qquad L_2 \vdash_S M : L_3}{L_1 \vdash_{R \circ S} M[N] : L_3}$$

This allows us to decompose a large system into multiple abstraction layers, verify their correctness individually, then derive a correctness property for the whole system.

Note that for every layer interface $L = \langle E, S, \sigma \rangle$, we can also derive a trivial certified abstraction layer which leaves everything unchanged:

$$L \vdash_{=} \mathbf{I}_E : L$$

In other words, we can define a category $\mathbf{CAL}$ of layer interface and certified layer implementation.

## 3.3 Horizontal composition

While vertical composition is quite useful, individual abstraction layers can themselves be quite complex and involve a significant number of independent opertations. To address this, we seek to decompose layers futher by introducing *horizontal* composition principles.

### 3.3.1 Layer implementations

As a starting point, we note that the category of effect signatures and layer implementations is equipped with a cartesian structure. For a simple signature $F = \{m : A \to B\}$, we write:

$$(m[a] \mapsto x) : E \to F\,,$$

with $a \in A$ bound in $x \in \mathcal{F}_E(B)$, for a correspondingly simple layer implementation of the signature $F$ in terms of $E$. Cartesian tupling can be used to construct more complex layer implementation from the simple ones.

**Example 6** (Decomposition of $M_{\mathsf{bq}}$)**.** *The signature $E_{\mathsf{bq}}$ and layer implementation $M_{\mathsf{bq}}$ can be decomposed into the elementary components:*

$$E_{\mathsf{enq}} := \{\mathsf{enq} : V \to \mathbb{1}\} \qquad M_{\mathsf{enq}} : E_{\mathsf{rb}} \to E_{\mathsf{enq}} := \mathsf{enq}[v] \mapsto i \leftarrow \mathsf{inc}_2\,; \mathsf{set}[i, v]$$

$$E_{\mathsf{deq}} := \{\mathsf{deq} : \mathbb{1} \to V\} \qquad M_{\mathsf{deq}} : E_{\mathsf{rb}} \to E_{\mathsf{deq}} := \mathsf{deq}[*] \mapsto i \leftarrow \mathsf{inc}_1\,; \mathsf{get}[i]$$

*Then:*

$$E_{\mathsf{bq}} = E_{\mathsf{enq}} \otimes E_{\mathsf{deq}}$$
$$M_{\mathsf{bq}} = \langle M_{\mathsf{enq}}, M_{\mathsf{deq}} \rangle .$$

### 3.3.2 Layer interfaces sharing state

When two layer interfaces $L_1 = \langle E_1, S, \sigma_1 \rangle$ and $L_2 = \langle E_2, S, \sigma_2 \rangle$ share a common set of abstract states, the specifications $\sigma_1$ and $\sigma_2$ will have compatible types, and the product of layer interfaces can be defined as:

$$L_1 \times L_2 := \langle E_1 \otimes E_2, S, (\sigma_1, \sigma_2) \rangle , \qquad \text{where} \quad (\sigma_1, \sigma_2)^{\iota_i(m)} := \sigma_i^m .$$

Layer correctness properties can be decomposed along the same lines:

$$\frac{L \vdash_R M_1 : L_1 \qquad L \vdash_R M_2 : L_2}{L \vdash_R \langle M_1, M_2 \rangle : L_1 \times L_2}$$

Conversely, the cartesian projections:

$$L_1 \times L_2 \vdash_= \pi_1 : L_1 \qquad\qquad L_1 \times L_2 \vdash_= \pi_2 : L_2$$

allow us retain only part of a layer's interface.

**Example 7** (Decomposition of $L_{\mathsf{bq}}$)**.** *The specification and verification of our bounded queue implementation can be decomposed in the following way:*

$$\frac{L_{\mathsf{rb}} \vdash_{R_{\mathsf{bq}}} M_{\mathsf{enq}} : L_{\mathsf{enq}} \qquad L_{\mathsf{rb}} \vdash_{R_{\mathsf{bq}}} M_{\mathsf{deq}} : L_{\mathsf{deq}}}{L_{\mathsf{rb}} \vdash M_{\mathsf{bq}} : L_{\mathsf{bq}}}$$

*where:*

$$L_{\mathsf{enq}} := \langle E_{\mathsf{enq}}, S_{\mathsf{bq}}, (\mathsf{enq}[v] \mapsto L_{\mathsf{bq}}.\mathsf{enq}[v]) \rangle$$
$$L_{\mathsf{deq}} := \langle E_{\mathsf{deq}}, S_{\mathsf{bq}}, (\mathsf{deq}[v] \mapsto L_{\mathsf{bq}}.\mathsf{deq}[v]) \rangle$$

The various forms of composition I have introduced so far are summarized in Fig. [XXX: building blocks figure from slides]. Together, they describe the principles of the layer calculus we used to verfiy CertiKOS. However, the form of horizontal composition we have introduced so far is largely syntactic in nature. Crucially, they rely on the fact that within an abstraction layer, components share the same low- and high-level state, and correctness proof share the same correspondance between them. In particular, not that the diagonal morphism:

$$L \vdash_{=} \Delta_L : L \times L$$

does not duplicate any of the resources implemented by $L$. That is, in the layer:

$$\frac{L_{\mathsf{rb}} \vdash_{R_{\mathsf{bq}}} M_{\mathsf{bq}} : L_{\mathsf{bq}} \qquad L_{\mathsf{rb}} \vdash_{R_{\mathsf{bq}}} M_{\mathsf{bq}} : L_{\mathsf{bq}}}{L_{\mathsf{rb}} \vdash_{R_{\mathsf{bq}}} \langle M_{\mathsf{bq}}, M_{\mathsf{bq}} \rangle : L_{\mathsf{bq}} \times L_{\mathsf{bq}}}$$

we have simply duplicated the code for each of the queue operations. But each copy uses the same underlying ring buffer, and the layer still implements a single queue.

### 3.3.3 Monoidal structure

[XXX motivation: text+figure from prospectus]

In this section, we present a more general form of horizontal composition which lets two layers operate independently of one another. The layers do not need to share state.

**Definition 8.** *The* tensor product *of two layer interfaces $L_1 = \langle E_1, S_1, \sigma_1 \rangle$ and $L_2 = \langle E_2, S_2, \sigma_2 \rangle$ is defined as:*

$$L_1 \otimes L_2 := \langle E_1 \otimes E_2, S_1 \otimes S_2, (\hat{\sigma}_1, \hat{\sigma}_2) \rangle \,,$$

*where $\hat{\sigma}_1$ and $\hat{\sigma}_2$ act on their respective halves of the state:*

$$\hat{\sigma}_1^{m@(k_1,k_2)} := \{ v@(k_1', k_2) \mid \sigma_1^{m@k_1} \ni v@k_1' \}$$
$$\hat{\sigma}_2^{m@(k_1,k_2)} := \{ v@(k_1, k_2') \mid \sigma_1^{m@k_2} \ni v@k_2' \}$$

Note that the corresponding structure is not cartesian. In particular, two certfied abstraction

layers $L \vdash_{R_1} M_1 : L_1$ and $L \vdash_{R_2} M_2 : L_2$ cannot be composed into a single one in this way:

$$L \vdash_{\langle R_1, R_2 \rangle} \langle M_1, M_2 \rangle : L_1 \otimes L_2$$

because the operations of one layer will update the underlay state in a way that may break the simulation relation for the other layer. However, layers can be composed side-by-side when the act on two independent underlay interfaces.

**Theorem 2** (Tensor product of certified abstraction layers)**.** *Two certified abstraction layers can be composed as follows:*

$$\frac{L_1 \vdash_{R_1} M_1 : L_1' \qquad L_2 \vdash_{R_2} M_2 : L_2'}{L_1 \otimes L_2 \vdash_{R_1 \times R_2} M_1 \times M_2 : L_1' \otimes L_2'}$$

Then, when doing so makes sense, we can model a situation where two independent layers access a common underlay $L$, by showing that the resources of $L$ can be multiplexed. This can be formalized by introducing a third certified abstraction layer $L \vdash_R M : L \otimes L$.

[XXX example of a memory allocator maintaining a pool of $n$ pages, we we can split as $L^{n_1+n_2} \vdash_R \Delta : L^{n_1} \otimes L^{n_2}$. Problem of modeling the nondeterminism of arbitrary page numbers being returned in the context of the limited monad $\mathcal{L}$]

## 3.4 CompCertX

[XXX give an overview of how this translates when we mix with CompCert, and the implementation of the layer library, which was also the inspiration behind CompCert Kripke logical relations.]

## 3.5 Conclusion

The new presentation of certified abstraction layers laid out in this chapter eliminates some of the complexity found in the original code by decoupling the model from CompCert semantics. In doing so, the new exposition makes precise the underlying mathematical structures and some of the challenges involved in our approach to layered verification.

In particular, the categorical structures behind our layer calculus are made explicit. The distinctions between our restricted form of product and the more general monoidal structure hints

at the connections between our framework and linear logic, where cartesian and tensor products cohabitate and work together.

# Chapter 4

# Games and dual nondeterminism

Our approach to demonic nondeterminism in game semantics will be to substitute **FCD** for $\mathcal{D}$ in the construction of strategies presented earlier. The more permissive *strategy specification* $\sigma'$ which we attempted to construct above can then be expressed as:

$$\sigma' := \bigsqcup_{n \in \mathbb{N}} \prod_{-1 \leq \delta \leq 1} \phi\big(\mathsf{run} \cdot \underline{\mathsf{rd}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{wr}_{\mathsf{x}}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}\big)$$

Because of the properties of **FCD**, the strategy specification $\sigma'$ will retain not only angelic choices, but demonic choices as well, expressing possible behaviors of the system.

For the construction $\mathbf{FCD} := \mathcal{UD}$, strategy specifications correspond to sets of traditional strategies, ordered by containment ($\supseteq$). This outer set ranges over demonic choices. Writing $s_{n,\delta} := \mathsf{run} \cdot \underline{\mathsf{rd}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{wr}_{\mathsf{x}}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}$, the strategy specification $\sigma'$ will be encoded as:

$$\sigma' = \{\sigma \in \mathcal{D}(P_G) \mid \forall n \in \mathbb{N} \cdot \exists \delta \in [-1,1] \cdot s_{n,\delta} \in \sigma\}\,.$$

Upward closure ensures that a strategy specification which contains a strategy $\sigma$ contains all of its refinements as well. For instance, the only strategy specification containing the completely undefined strategy $\varnothing$ is the maximally permissive strategy specification $\bot = \mathcal{D}(P_G)$.

# Chapter 5

# The interaction specification monad

We begin our formal development by defining the *interaction specification monad*, a variant of the free monad on an effect signature which incorporates dual nondeterminism.

## 5.1 Overview

Given an effect signature $E$, we construct a prefix-ordered set of plays $\bar{P}_E(A)$ corresponding to the possible interactions between a computation with effects in $E$ and its environment, including the computation's ultimate outcome in $A$. The interaction specification monad $\mathcal{I}_E(A)$ is then obtained as the free completely distributive completion of the poset $\bar{P}_E(A)$.

For each effect $e \in E$, the interaction specification monad has an operation $\mathbf{I}_E^e \in \mathcal{I}_E(\mathsf{ar}(e))$ which triggers an instance of $e$ and returns its outcome. Given a second effect signature $F$, a family $(f^m)_{m \in F}$ of computations $f^m \in \mathcal{I}_E(\mathsf{ar}(m))$ can be used to interpret the effects of $F$ into the signature $E$. This is achieved by a *substitution* operator $\bullet[f]$, which transforms a computation $x \in \mathcal{I}_F(A)$ into the computation $x[f] \in \mathcal{I}_E(A)$, where each occurrence of an effect $m \in F$ in $x$ is replaced by the corresponding computation $f^m$.

Effect signatures are used as simple games, and a family $(f^m)_{m \in F}$ as described above can be interpreted as a certain kind of strategy for the game $!E \multimap F$. We use this approach to define a first category of games and strategy specifications $\mathcal{G}^{ib}_{\sqsubseteq}$, where $(\mathbf{I}_E^e)_{e \in E}$ is the identity morphism for $E$ and the substitution operator is used to define composition.

## 5.2 Plays

We first introduce the partially ordered sets of plays which we use to construct the interaction specification monad. Since we intend to describe *active* computations, we use *odd*-length plays which start with *system* moves, by contrast with the more common approach presented in §**??**.

**Definition 9.** *The set $\bar{P}_E(A)$ of* interactions *for an effect signature $E$ and a set of values $A$ is defined inductively:*

$$s \in \bar{P}_E(A) ::= \underline{v} \mid \underline{m} \mid \underline{m}ns\,,$$

*where $v \in A$, $m \in E$ and $n \in \mathsf{ar}(m)$. The set $\bar{P}_E(A)$ is ordered by the prefix relation $\sqsubseteq \subseteq \bar{P}_E(A) \times \bar{P}_E(A)$, defined as the smallest relation satisfying:*

$$\underline{v} \sqsubseteq \underline{v}\,, \qquad \underline{m} \sqsubseteq \underline{m}\,, \qquad \underline{m} \sqsubseteq \underline{m}nt\,, \qquad \frac{s \sqsubseteq t}{\underline{m}ns \sqsubseteq \underline{m}nt}\,.$$

A play corresponds to a finite observation of an interaction between the system and the environment. At any point in such an interaction, the system can terminate the interaction with a given value ($v$), or it can trigger an effect $m \in E$ and wait to be resumed by an answer $n \in \mathsf{ar}(m)$ of the environment ($\underline{m}ns$). A play which concludes before the environment answers a query from the system ($\underline{m}$) denotes that no information has been observed after that point. It can be refined by a longer observation of an interaction which begin with the same sequence of questions and answers.

## 5.3 Interaction specifications

We define our monad as the free completely distributive completion of the corresponding poset of plays.

For the sake of conciseness and clarity, we will use the order embedding associated with **FCD** implicitly, so that an element of a poset $s \in P$ can also be regarded as an element of its completion $s \in \mathbf{FCD}(P)$. Likewise, for a completely distributive lattice $M$, we can implicitly promote a monotonic function $f : P \to M$ to its extension $f : \mathbf{FCD}(P) \to M$. These conventions are at work in the following definition.

**Definition 10.** *The* interaction specification monad *for an effect signature $E$ maps a set $A$ to the free completely distributive completion of the corresponding poset of plays:*

$$\mathcal{I}_E(A) := \mathbf{FCD}(\bar{P}_E(A))$$

*An element $x \in \mathcal{I}_E(A)$ is called an* interaction specification.

*The monad's action on a function $f : A \to B$ replaces the values in an interaction specification with their image by $f$:*

$$\begin{aligned}\mathcal{I}_E(f)(\underline{v}) &:= \underline{f(v)}\\ \mathcal{I}_E(f)(\underline{m}) &:= \underline{m}\\ \mathcal{I}_E(f)(\underline{m}ns) &:= \underline{m}n\,\mathcal{I}_E(f)(s)\,.\end{aligned}$$

*The monad's unit $\eta^E_A : A \to \mathcal{I}_E(A)$ is the embedding of a single play consisting only of the given value:*

$$\eta^E_A(v) := \underline{v}$$

*Finally, the multiplication $\mu^E_A : \mathcal{I}_E(\mathcal{I}_E(A)) \to \mathcal{I}_E(A)$ carries out the outer computation and sequences it with any computation it evaluates to:*

$$\begin{aligned}\mu^E_A(\underline{x}) &:= x\\ \mu^E_A(\underline{m}) &:= \underline{m}\\ \mu^E_A(\underline{m}ns) &:= \underline{m} \sqcup \underline{m}n\mu^E_A(s)\,.\end{aligned}$$

The most subtle aspect of Def. 10 is the case for $\mu^E_A(\underline{m}ns)$, which includes $\underline{m}$ as well as $\underline{m}n\mu^E_A(s)$. This is both to ensure that the effects of the first computation are preserved when the second computation is $\bot$, and to ensure the monotonicity of the underlying function used to define $\mu^E_A$. Consider for example $\underline{m} \sqsubseteq \underline{m}n\underline{\bot} \in \bar{P}_E(\mathcal{I}_E(A))$. Since $\mu^E_A(\underline{\bot}) = \bot$ and the **FCD** extension of the function $s \mapsto \underline{m}ns$ preserves $\bot$, it is not the case that $\underline{m} \sqsubseteq \underline{m}n\mu^E_A(\underline{\bot})$.

As usual, the Kleisli extension of a function $f : A \to \mathcal{I}_E(B)$ is the function $f^* = \mu^E_B \circ \mathcal{I}_E(f)$. We extend the notations used for **FCD** to the monad $\mathcal{I}_E$.

## 5.4 Interaction primitives

The operations of an effect signature $E$ can be promoted to interaction specifications of $\mathcal{I}_E$ as follows.

**Definition 11** (Interaction primitive)**.** *For an effect signature $E$ and an operation $m \in E$, the interaction specification $\mathbf{I}_E^m \in \mathcal{I}_E(\mathsf{ar}(m))$ is defined as:*

$$\mathbf{I}_E^m := \bigsqcup_{n \in \mathsf{ar}(m)} \underline{m} n \underline{n}$$

Note that in the play $\underline{m}n\underline{n}$, the first occurrence of $n$ is the environment's answer, whereas the second occurrence is the value returned by $\mathbf{I}_E^m$.

To model effect handling for a signature $F$, we use a family of interaction specifications $(f^m)_{m \in F}$ to provide an interpretation $f^m \in \mathcal{I}_E(\mathsf{ar}(m))$ of each effect $m \in F$ in terms of another effect signature $E$. This allows us to transform an interaction specification $x \in \mathcal{I}_F(A)$ into an interaction specification $x[f] \in \mathcal{I}_E(A)$, defined as follows. The constructions $\bot$ and $\{P\}$ were discussed in §**??**; they carry similar meanings in the context of the interaction specification monad.

**Definition 12** (Interaction substitution)**.** *Given the effect signatures $E, F$ and the set $A$, for an interaction specification $x \in \mathcal{I}_F(A)$ and a family $(f^m)_{m \in F}$ with $f^m \in \mathcal{I}_E(\mathsf{ar}(m))$, the* interaction substitution *$x[f] \in \mathcal{I}_E(A)$ is defined by:*

$$\begin{aligned} \underline{v}[f] &:= \underline{v} \\ \underline{m}[f] &:= r \leftarrow f^m; \bot \\ \underline{m}ns[f] &:= r \leftarrow f^m; \{r = n\}; s[f]\,. \end{aligned}$$

The outcome of the interaction specification is left unchanged, but effects are replaced by their interpretation. Whenever that interpretation produces an outcome $r$, the substitution process resumes with the remainder of any matching plays of the original computation.

## 5.5 Categorical structure

As presented so far, the interaction specification monad can be seen as an extension of the refinement calculus able to model effectful computations for a given signature. We now shift our point of view to game semantics and show how interaction substitutions can be used to define a simple category of games and strategies featuring dual nondeterminism and alternating refinement.

**Definition 13** (Morphisms)**.** *Consider the effect signatures $E$, $F$ and $G$. We will write $f : E \to F$ whenever $(f^m)_{m \in F}$ is a family of interactive computations such that $f^m \in \mathcal{I}_E(\mathsf{ar}(m))$. For $f : E \to F$ and $g : F \to G$, we define $g \circ f : E \to G$ as:*

$$(g \circ f)^m = g^m[f]\,.$$

*The completely distributive lattice structure of $\mathcal{I}_F(-)$ can be extended pointwise to morphisms, so that for a family $(f_i)_{i \in I}$ with $f_i : E \to F$, we can define $\bigsqcup_{i \in I} f_i : E \to F$ and $\bigsqcap_{i \in I} f_i : E \to F$. For $f, g : E \to F$ we define refinement as:*

$$f \sqsubseteq g \;\Leftrightarrow\; \forall m \in F \cdot f^m \sqsubseteq g^m\,.$$

A morphism $f : E \to F$ can be interpreted as a *well-bracketed* strategy for the game $!E \multimap F$. In this game, the environment first plays a move $m \in F$. The system can then ask a series of questions $q_1, \ldots q_k \in E$ to which the environment will reply with answers $r_i \in \mathsf{ar}(q_i)$, and finally produce an answer $n \in \mathsf{ar}(m)$ to the environment's initial question $m$. The plays of $!E \multimap F$ are restricted to a single top-level question $m$. In addition, the well-bracketing requirement imposes that at any point, only the most recent pending question may be answered.

Compared with the usual notion of strategy, our model introduces arbitrary demonic choices and relaxes constraints over angelic choices. The definition of $g \circ f$ given above otherwise corresponds to the traditional definition of strategy composition. The identity strategy is given by $\mathbf{I}_E : E \to E$.

**Lemma 1.** *Consider the effect signatures $E, F, G, H$ and the morphisms $f : E \to F$, $g : F \to G$*

*and $h : G \to H$. The following properties hold:*

$$\mathbf{I}_F \circ f = f \circ \mathbf{I}_E = f$$

$$h \circ (g \circ f) = (h \circ g) \circ f$$

*Composition preserves all extrema on the left, and all non-empty extrema on the right.*

*Proof.* Using properties of **FCD** and inductions on plays. □

Having established the relevant properties, we can now define our first category of games and strategies.

**Definition 14.** *The category $\mathcal{G}^{ib}_{\sqsubseteq}$ has effect signatures as objects. Morphisms, identities and composition have been defined above. The hom-sets $\mathcal{G}^{ib}_{\sqsubseteq}(E, F)$ are completely distributive lattices, with composition preserving all extrema on the left, and all non-empty extrema on the right.*

## 5.6 Products

Effect signatures can be combined in the following way.

**Definition 15.** *We define the effect signature $1 := \varnothing$. For a family of effect signatures $(E_i)_{i \in I}$, we define:*

$$\bigotimes_i E_i := \{(i, e) : \mathsf{ar}(e) \mid i \in I, e \in E_i\}$$

For example, the signature $E_{\mathsf{io}}$ above is equivalent to the following composite one:

$$\{\mathsf{readbit} : \mathbb{2}\} \otimes \{\mathsf{print}[s] : \mathbb{1} \mid s \in \mathsf{string}\} \otimes \{\mathsf{done} : \varnothing\}$$

The construction $\otimes$ gives products in the category $\mathcal{G}^{ib}_{\sqsubseteq}$, as demonstrated below.

**Theorem 3.** *The category $\mathcal{G}^{ib}_{\sqsubseteq}$ has all products. Objects are given by $\bigotimes_{i \in I} E_i$ and projection arrows are given for each $i \in I$ by the morphism:*

$$\pi_i : \bigotimes_{j \in I} E_j \to E_i \qquad \pi_i^m := (i, m)\,.$$

*Proof.* We need to show that for an effect signature $X$ and a collections of morphisms $(f_i)_{i \in I}$ with $f_i : X \to E_i$, there is a unique $\langle f_i \rangle_{i \in I} : X \to \bigotimes_{i \in I} E_i$ such that for all $i \in I$:

$$f_i = \pi_i \circ \langle f_j \rangle_{j \in I} \,.$$

Note that for $x : X \to \bigotimes_{i \in I} E_i$, $i \in I$ and $m \in E_i$, we have:

$$(\pi_i \circ x)^m = \pi_i^m[x] = (i, m)[x] = x^{(i,m)}$$

Hence, $\langle f_i \rangle_{i \in I}$ is uniquely defined as:

$$\langle f_i \rangle_{i \in I}^{(j,m)} := f_j^m \,.$$

□

## 5.7 Certified abstraction layers

Certified abstraction layers can be embedded into the category $\mathcal{G}_{\sqsubseteq}^{ib}$ as follows.

### 5.7.1 Signatures

The signature of a layer interface or implementation can be encoded as an effect signature. For example:

$$E_{\mathsf{bq}} := \{\mathsf{enq}[v] : \mathbb{1}, \mathsf{deq} : V \mid v \in V\}$$
$$E_{\mathsf{rb}} := \{\mathsf{set}[i, v] : \mathbb{1}, \mathsf{get}[i] : V, \mathsf{inc}_1 : \mathbb{N}, \mathsf{inc}_2 : \mathbb{N} \mid i \in \mathbb{N}, v \in V\}$$

A layer implementation $M$ with an underlay signature $E$ and an overlay signature $F$ can then be interpreted as a morphism $[\![M]\!] : E \to F$ in a straightforward manner, by replacing underlay operations used in the definition of $M$ with the corresponding interaction primitives:

$$[\![M]\!]^m \;:=\; (M.m)[e := \mathbf{I}_E^e]_{e \in E}$$

### 5.7.2 Interfaces

In order to handle the layer's abstract data, we can extend signatures with state in the following way:

$$E@S := \{m@k : \mathsf{ar}(m) \times S \mid m \in E, k \in S\}$$

A layer interface $L$ with a signature $E$ and states in $S$ can be interpreted as a morphism $[\![L]\!] : 1 \to E@S$ almost directly, mapping $\varnothing$ to $\bot$ in outcomes of primitive specifications:

$$[\![L]\!]^{m@k} := \bigsqcup L.m@k$$

### 5.7.3 Keeping state

For a morphism $f : E \to F$, we construct $f@S : E@S \to F@S$ which keeps updating a state $k \in S$ as it performs effects in $E@S$, then adjoins the final state to any answer returned by $f$. For a set $A$, we first define $-\#- : \bar{P}_E(A) \times S \to \mathcal{I}_{E@S}(A \times S)$:

$$\underline{v}\#k := \underline{v@k}$$

$$\underline{m}\#k := \underline{m@k}$$

$$\underline{m}ns\#k := \bigsqcup_{k' \in S} \underline{m@k}\ n@k'\ s\#k' \,,$$

and extend it to morphisms as $(f@S)^{m@k} := f^m\#k$. Then in particular, running a layer implementation $[\![M]\!] : E \to F$ on top of a layer interface $[\![L]\!] : 1 \to E@S$ yields the morphism $[\![M]\!]@S \circ [\![L]\!] : 1 \to F@S$.

### 5.7.4 Simulation relations

The most interesting aspect of our embedding is the representation of simulation relations. We will see that dual nondeterminism allows us to represent them as regular morphisms.

Recall the definition of the judgment $L_1 \vdash_R M : L_2$, which means that a layer implementation $M$ correctly implements $L_2$ on top of $L_1$ through a simulation relation $R \subseteq S_2 \times S_1$. If we write

$L_1' := [\![M]\!]@S_1 \circ [\![L_1]\!]$ for the layer interface obtained by interpreting $M$ on top of $L_1$, then:

$$L_1 \vdash_R M : L_2 \;\Leftrightarrow\; \forall m \in E_2 \cdot L_2^m\ [R \to \mathcal{P}^{\leq}(= \times R)]\ {L_1'}^m$$

We will use the families of morphisms $R_E^* : E@S_2 \to E@S_1$ and $R_*^E : E@S_1 \to E@S_2$ to encode this judgment:

$$(R_E^*)^{m@k_1} := \bigsqcup_{k_2 \in R^{-1}(k_1)} n@k_2' \leftarrow \mathbf{I}_{E@S_2}^{m@k_2};\ \bigsqcap_{k_1' \in R(k_2')} n@k_1'$$

$$(R_*^E)^{m@k_2} := \bigsqcap_{k_1 \in R(k_2)} n@k_1' \leftarrow \mathbf{I}_{E@S_1}^{m@k_1};\ \bigsqcup_{k_2' \in R^{-1}(k_1')} n@k_2'$$

They yield two equivalent ways to encode layer correctness as refinement properties.

In the first case, $R_E^*$ is intended to translate a high-level specification $\sigma$ which uses overlay states $k_2, k_2' \in S_2$ into a low-level specification $R_E^* \circ \sigma$ which uses underlay states $k_1, k_1' \in S_1$. The client calls $R_E^*$ with an underlay state $k_1$, with the expectation that if there is *any* corresponding overlay state, then $R_E^* \circ \sigma$ will behave accordingly (it is angelic with respect to its choice of $k_2$). On the other hand, $R_E^* \circ \sigma$ is free to choose any underlay representation $k_1'$ for the outcome $k_2'$ produced by $\sigma$, and the client must be ready to accept it (it is demonic with respect to its choice of $k_1'$).

In the second case, $R_*^E \circ \tau$ is the strongest high-level specification which a low-level component $\tau$ implements with respect to $R$. For an overlay state $k_2 \in S_2$, $\tau$ may behave in various ways depending on the corresponding underlay state $k_1 \in S_1$ it is invoked with, and so the specification must allow them using demonic choice. On the other hand, when $\tau$ returns with a new underlay state $k_1'$, the environment is free to choose how to interpret it as an overlay state $k_2'$.

**Theorem 4.** *For* $\sigma := [\![L_2]\!]$ *and* $\tau := [\![M]\!]@S_1 \circ [\![L_1]\!]$*:*

$$R_{E_2}^* \circ \sigma \sqsubseteq \tau \;\Leftrightarrow\; L_1 \vdash_R M : L_2 \;\Leftrightarrow\; \sigma \sqsubseteq R_*^{E_2} \circ \tau\,.$$

*Proof.* The proof is straightforward but requires Thm. 5.3 from Morris and Tyrrell [2008]. □

# Chapter 6

# Stateful and reentrant strategies

We now sketch a more general model allowing strategies to retain state across different activations. We explain how the new model $\mathcal{G}^{b}_{\sqsubseteq}$ can embed the morphisms of $\mathcal{G}^{ib}_{\sqsubseteq}$, and how it can be used to characterize certified abstraction layers independently of the states used in their description.

## 6.1 Overview

As discussed in §5.5, the morphisms of $\mathcal{G}^{ib}_{\sqsubseteq}(E, F)$ correspond to the well-bracketed strategies for the game $!E \multimap F$. As such, they can be promoted to well-bracketed strategies for the more general game $!E \multimap !F$, which allows the environment to ask multiple question of $F$ in a row, and to ask nested questions whenever it is in control.

More precisely, strategies promoted in this way correspond to the *innocent* well-bracketed strategies for $!E \multimap !F$, meaning that they will behave in the same way in response to the same question, regardless of the history of the computation. The model we introduce in this section relaxes this constraint, allowing strategies to maintain internal state.

After outlining the construction of a new category $\mathcal{G}^{b}_{\sqsubseteq}$ of games and strategies (§6.2–§6.5), we define an embedding of $\mathcal{G}^{ib}_{\sqsubseteq}$ into $\mathcal{G}^{b}_{\sqsubseteq}$ (§6.6), and show how the states used by a strategy $\sigma : E@S \to F@S$ can be internalized and hidden from its interactions (§6.7).

## 6.2 Games

To facilitate reasoning, and make it easier to describe operators on strategies in a systematic way, we describe games as a specific kind of graph where vertices represent players and edges determine which questions can be asked by one player to another. Generalizing from effect signatures, questions are assigned an arity which gives the type of the answer.

**Definition 16.** *A* game signature $\Gamma$ *is a set of players with a distinguished element* $\mathsf{O}$*, together with an effect signature* $\Gamma(u, v)$ *for all* $u, v \in \Gamma$*. The operations* $m \in \Gamma(u, v)$ *are called the* questions *of* $u$ *to* $v$*, and the elements* $n \in \mathsf{ar}(m)$ *are called* answers *to the question* $m$*.*

We depict game signatures as directed graphs whose vertices are the players and whose edges are labeled by the corresponding effect signature. Missing edges correspond to the empty signature $\varnothing$. For example, the game $!E \multimap !F$ is generated by the game signature:

$$[E, F] = \quad \mathsf{P} \underset{F}{\overset{E}{\rightleftarrows}} \mathsf{O}$$

When we consider the ways in which questions propagate through a game signature, the distinguished player $\mathsf{O}$ serves the role of both a source and sink. As such, it is visually useful to depict $\mathsf{O}$ as two nodes, one capturing the incoming edges of $\mathsf{O}$, and one capturing its outgoing edges. For example, the following game signature generates the interaction sequences used in the definition of strategy composition:

$$[E, F, G] = \quad \mathsf{O} \xleftarrow{E} \mathsf{P}_1 \xleftarrow{F} \mathsf{P}_2 \xleftarrow{G} \mathsf{O}$$

As another example of a game signature, a situation where $\sigma_1 : E_1 \to F_1$ and $\sigma_2 : E_2 \to F_2$ interact with the environment independently of one another can be described as:

$$[E_1, F_1] \vee [E_2, F_2] = \quad \begin{array}{ccccc} & & \mathsf{P}_1 & & \\ & E_1 \swarrow & & \searrow F_1 & \\ \mathsf{O} & & & & \mathsf{O} \\ & E_2 \searrow & & \swarrow F_2 & \\ & & \mathsf{P}_2 & & \end{array}$$

The signature above will be used to compute tensor products of strategies. These constructions generalize as follows.

**Definition 17** (Constructions on game signatures)**.** *For a collection of effect signature $(E_i)_{1\leq i\leq n}$ and an effect signature $F$, the game signature $[E_1, \ldots, E_n, F]$ has the players $\mathsf{O}, \mathsf{P}_1, \ldots, \mathsf{P}_n$ and the following edges:*

$$[E_1, \ldots, E_n, F] := \mathsf{O} \xleftarrow{E_1} \mathsf{P}_1 \xleftarrow{E_2} \cdots \xleftarrow{E_n} \mathsf{P}_n \xleftarrow{F} \mathsf{O}$$

*For a collection of game signatures $(\Gamma_i)_{i\in I}$, the* wedge sum *$\bigvee_{i\in I} \Gamma_i$ has the players:*

$$\{\mathsf{O}\} \cup \{(i, p) \mid i \in I \wedge p \in \Gamma_i \setminus \{\mathsf{O}\}\}$$

*For $i \in I$ and $p \in \Gamma_i$, the corresponding player in $\bigvee_{j\in I} \Gamma_j$ is:*

$$\iota_i(p) := \begin{cases} \mathsf{O} & \text{if } p = \mathsf{O} \\ (i, p) & \text{otherwise.} \end{cases}$$

*Then for each question $m : u \to v$ in $\Gamma_i$, the wedge sum has a corresponding question $\iota_i(m) : \iota_i(u) \to \iota_i(v)$.*

## 6.3 Plays and strategies

The well-bracketing requirement enforces a kind of *stack discipline* on the succession of questions and answers. A well-bracketed play can be interpreted as an activation tree, where questions are understood as function calls and answers are understood as the corresponding calls returning. At any point in a play over a signature $\Gamma$, its possible evolutions are characterized by the stack of pending questions.

**Definition 18.** *For a game signature $\Gamma$ and a player $p \in \Gamma$, a* $p$-stack *over $\Gamma$ is a path:*

$$\mathsf{O} = p_0 \xrightarrow{m_1} p_1 \xrightarrow{m_2} \cdots \xrightarrow{m_n} p_n = p$$

*where* $p_i \in \Gamma$ *and* $m_i \in \Gamma(p_{i-1}, p_i)$*. We will write this path as* $\kappa = m'_1 \cdots m'_n : \mathsf{O} \twoheadrightarrow p \in \Gamma$*.*

Such stacks can in turn be arranged in a graph $\hat{\Gamma}$ over which the game associated with $\Gamma$ will be played.

**Definition 19** (Strategy specifications)**.** *For a signature* $\Gamma$*, the graph* $\hat{\Gamma}$ *is defined as follows. The vertices of* $\hat{\Gamma}$ *are pairs* $(u, \kappa)$ *in which* $u \in \Gamma$ *and* $\kappa$ *is a* $u$*-stack. For each question* $m \in \Gamma(u, v)$ *and stack* $\kappa : \mathsf{O} \twoheadrightarrow u$*, there is an edge:*

$$m : (u, \kappa) \to (v, \kappa m) \in \hat{\Gamma}\,.$$

*In addition, for each answer* $n \in \mathsf{ar}(m)$*, there is an edge:*

$$n : (v, \kappa m) \to (u, \kappa) \in \hat{\Gamma}\,.$$

*The* plays *over* $\Gamma$ *are paths of type* $(\mathsf{O}, \epsilon) \twoheadrightarrow (\mathsf{O}, \kappa) \in \hat{\Gamma}$*, where* $\kappa : \mathsf{O} \twoheadrightarrow \mathsf{O} \in \Gamma$ *is a stack. We will write* $P\,\Gamma$ *for the poset of plays over* $\Gamma$ *under the prefix ordering. The* strategy specifications *for* $\Gamma$ *are given by the completion:*

$$S\,\Gamma := \mathbf{FCD}(P\,\Gamma)\,.$$

## 6.4 Operations on strategies

**Definition 20.** *A* transformation *from the game signature* $\Gamma_1$ *to the game signature* $\Gamma_2$ *associates to each player* $p \in \Gamma_1$ *a player* $f(p) \in \Gamma_2$ *with* $f(\mathsf{O}) = \mathsf{O}$*, and to each question* $m \in \Gamma_1(u, v)$ *a* path *of questions in* $\Gamma_2$*:*

$$f(u) = p_0 \xrightarrow{m'_1} p_1 \xrightarrow{m'_2} \cdots \xrightarrow{m'_n} p_n = f(v)\,,$$

*written as* $f(m) = m'_1 m'_2 \cdots m'_n : f(u) \twoheadrightarrow f(v)$*, and such that* $\mathsf{ar}(m'_1) = \cdots = \mathsf{ar}(m'_n) = \mathsf{ar}(m)$*. We extend* $f$ *itself to the paths in* $\Gamma_1$ *by taking the image of* $m_1 \cdots m_n : u \twoheadrightarrow v$ *to be:*

$$f(m_1 \cdots m_n) := f(m_1) \cdots f(m_n) : f(u) \twoheadrightarrow f(v)\,.$$

*Game signatures and transformations form a category.*

In other words, a transformation is a structure-preserving map on paths. Transformations can be extended to plays.

**Definition 21** (Action on plays)**.** *A transformation $f : \Gamma_1 \to \Gamma_2$ induces a monotonic function $Pf : P\,\Gamma_1 \to P\,\Gamma_2$ as follows. For $m \in \Gamma(u, v)$ and $\kappa : \mathsf{O} \twoheadrightarrow u$, the image of the move $m : (u, \kappa) \to (v, \kappa m)$ is the path:*

$$f(m) : (f(u), f(\kappa)) \twoheadrightarrow (f(v), f(\kappa)f(m))\,.$$

*For $n \in \mathsf{ar}(m)$, the image of $n : (v, \kappa m) \to (u, \kappa)$ is the path:*

$$n^{|f(m)|} : (f(v), f(\kappa)f(m)) \twoheadrightarrow (f(u), f(\kappa))\,,$$

*where $n^{|f(m)|}$ denotes a sequence $nn \cdots n$ of copies of the answer $n \in \mathsf{ar}(m)$ of same length as the path $f(m)$.*

Operators on strategies will generally be defined by a game signature of global *interaction sequences*, and will use transformations to project out the corresponding plays of the arguments and the result.

### 6.4.1 Composition

When composing the strategy specifications $\sigma \in S\,[E, F]$ and $\tau \in S\,[F, G]$ to obtain $\tau \circ \sigma \in S\,[E, G]$, we will use the transformation $\psi^c_X : [E, F, G] \to [E, G]$ to describe the externally observable behavior of interaction sequences in $[E, F, G]$:

$$\psi^c_X(\mathsf{P}_1) = \psi^c_X(\mathsf{P}_2) = \mathsf{P} \qquad \psi^c_X(\mathsf{O}) = \mathsf{O}$$

$$\psi^c_X(m) = \begin{cases} \epsilon & \text{if } m \in F \\ m & \text{otherwise} \end{cases}$$

This can be described concisely as $\psi^c_X = [1, 0, 1]$, with:

$$\psi^c_1 := [1, 1, 0] : [E, F, G] \to [E, F]$$

$$\psi^c_2 := [0, 1, 1] : [E, F, G] \to [F, G]$$

defined similarly.

We can now formulate the composition of strategy specifications as follows. The "footprint" of the plays $s_1 \in P\,[E, F]$ and $s_2 \in P\,[F, G]$ can be defined as:

$$\psi^c(s_1, s_2) := \bigsqcup_{s \in P[E,F,G]} \{\psi_1^c(s) \sqsubseteq s_1 \wedge \psi_2^c(s) \sqsubseteq s_2\}; \psi_X^c(s)\,.$$

In other words, the angel chooses a global play $s$ matching $s_1$ and $s_2$ and produces its external view. By extending $\psi^c$ to strategy specifications in the expected way, we obtain:

$$\tau \circ \sigma = s \leftarrow \sigma; t \leftarrow \tau; \psi^c(s, t)\,.$$

### 6.4.2 Identity

The strategy $\mathrm{id}_E \in S\,[E, E]$ uses the signature:

$$[E] = \quad E \circlearrowright \boxed{\mathsf{O}}$$

and the transformation:

$$\psi_X^{\mathrm{id}} := [2] : [E] \to [E, E]$$

$$\psi_X^{\mathrm{id}}(\mathsf{O}) := \mathsf{O} \qquad \psi_X^{\mathrm{id}}(m) := mm$$

Then $\mathrm{id}_E$ is defined as:

$$\mathrm{id}_E := \bigsqcup_{s \in P[E]} \psi_X^{\mathrm{id}}(s)\,.$$

### 6.4.3 Tensor

The tensor product of the strategies $\sigma_1 \in S\,[E_1, F_1]$ and $\sigma_2 \in S\,[E_2, F_2]$ is a strategy $\sigma_1 \otimes \sigma_2 \in S\,[E_1 \otimes E_2, F_1 \otimes F_2]$ defined using interaction sequences in $\Gamma = [E_1, F_1] \vee [E_2, F_2]$. The external

projection $\psi^{\otimes}_X : \Gamma \to [E_1 \otimes E_2, F_1 \otimes F_2]$ is:

$$\psi^{\otimes}_X(\mathsf{O}) = \mathsf{O} \qquad \psi^{\otimes}_X(\mathsf{P}_1) = \psi^{\otimes}_X(\mathsf{P}_2) = \mathsf{P}$$
$$\psi^{\otimes}_X(\iota_i(m)) = \iota_i(m)$$

The internal projections $\psi^{\otimes}_i : \Gamma \to [E_i, F_i]$ are given by:

$$\psi^{\otimes}_i(p) = \begin{cases} \mathsf{P} & \text{if } p = \mathsf{P}_i \\ \mathsf{O} & \text{otherwise} \end{cases}$$
$$\psi^{\otimes}_i(\iota_j(m)) = \begin{cases} m & \text{if } i = j \\ \epsilon & \text{otherwise} \end{cases}$$

The footprint of the plays $s_1 \in P[E_1, F_1]$ and $s_2 \in P[E_2, F_2]$ is:

$$\psi^{\otimes}(s_1, s_2) := \bigsqcup_{s \in P\Gamma} \{\psi^{\otimes}_1(s) \sqsubseteq s_1 \wedge \psi^{\otimes}_2(s) \sqsubseteq s_2\}; \psi^{\otimes}_X(s)$$

The tensor product can then be defined as:

$$\sigma_1 \otimes \sigma_2 := s_1 \leftarrow \sigma_1; s_2 \leftarrow \sigma_2; \psi^{\otimes}(s_1, s_2)$$

## 6.5 Category

The category $\mathcal{G}^{b}_{\sqsubseteq}$ has effect signatures as objects, and has the elements of $S\,[E, F]$ as morphisms $\sigma : E \to F$. The categorical structure is defined in the previous section.

The associator, unitor and braiding associated with $\otimes$ can be obtained by embedding the corresponding morphisms of $\mathcal{G}^{ib}_{\sqsubseteq}$ using the process outlined in §6.6 below. Note however that unlike that of $\mathcal{G}^{ib}_{\sqsubseteq}$, the symmetric monoidal structure of $\mathcal{G}^{ib}_{\sqsubseteq}$ is not cartesian, because the interactions of a strategy $\sigma : E \to F_1 \otimes F_2$ which involve only one of the games $F_1$ and $F_2$ are not sufficient to characterize the behavior of $\sigma$ in interactions that involve both of them.

## 6.6 Embedding $\mathcal{G}^{ib}_{\sqsubseteq}$

Since a morphism $f \in \mathcal{G}^{ib}_{\sqsubseteq}(E, F)$ defined using the interaction specification monad only describes the behavior of a component for a single opponent question, to construct a corresponding strategy $Wf \in \mathcal{G}^{b}_{\sqsubseteq}(E, F)$ we must duplicate the component's behavior, compounding the angelic and demonic choices of each copy.

We proceed as follows. For a stack $\kappa : \mathsf{O} \twoheadrightarrow \mathsf{O}$, the set $P^{\kappa}_{\Gamma}$ contains partial plays of type $(\mathsf{O}, \kappa) \twoheadrightarrow (\mathsf{O}, \kappa') \in \hat{\Gamma}$, and for a question $q \in F$, the set $\bar{P}^{\kappa q}_{\Gamma}$ contains partial plays of type $(\mathsf{P}, \kappa q) \twoheadrightarrow (\mathsf{O}, \kappa') \in \hat{\Gamma}$. We will define an operator:

$$\omega^{\kappa} : P^{\kappa}_{\Gamma} \to \mathbf{FCD}(P^{\kappa}_{\Gamma})$$

which *prepends* an arbitrary number of copies of $f$ to a play of $P^{\kappa}_{\Gamma}$. Starting with $\omega^{\kappa}_0(t) := t$, we construct a series of approximations:

$$\omega^{\kappa}_{i+1}(t) := t \sqcup \bigsqcup_{q \in F} q\, \bar{\omega}^{\kappa q}_i(f^q, \omega^{\kappa}_i(t))$$

The auxiliary construction:

$$\bar{\omega}^{\kappa q} : \bar{P}_E(\mathsf{ar}(q)) \times P^{\kappa}_{\Gamma} \to \mathbf{FCD}(\bar{P}^{\kappa q}_{\Gamma})$$

embeds an interaction $s \in \bar{P}_E(\mathsf{ar}(q))$, inserting reentrant calls as appropriate, and continues with the play $t$ if $s$ terminates:

$$\begin{aligned}
\bar{\omega}^{\kappa q}_i(\underline{v}, t) &= vt \\
\bar{\omega}^{\kappa q}_i(\underline{m}, t) &= m\, \omega^{\kappa q m}_i(\epsilon) \\
\bar{\omega}^{\kappa q}_i(\underline{m} n s, t) &= m\, \omega^{\kappa q m}_i(n\, \bar{\omega}^{\kappa q}_i(s, t))
\end{aligned}$$

The index $i$ limits both the number of sequential and reentrant copies of $f$ which are instantiated.

The strategy specification associated to $f$ in $\mathcal{G}^b_{\sqsubseteq}$ is:

$$Wf := \bigsqcup_{i \in \mathbb{N}} \omega_i(\epsilon)\,.$$

## 6.7 Hiding state

The functor $W : \mathcal{G}^{ib}_{\sqsubseteq} \to \mathcal{G}^b_{\sqsubseteq}$ can be used to embed the layer theory defined in §5.7 as-is. In addition, the *state* of layer interfaces can be propagated across consecutive calls and eliminated from the representation.

**Definition 22.** *The* state-free observation *at $k_0 \in S$ of a partial play $s : (O, \kappa) \twoheadrightarrow (O, \kappa')$ over the signature $[E@S, F@S]$ is written $s/k_0$ and defined recursively as:*

$$\epsilon/k_0 := \epsilon$$

$$(m@k_1\ n@k_2\ s)/k_0 := \{k_0 = k_1\}; mn(s/k_2)$$

*For $\sigma : E@S \to F@S$, the strategy $\sigma/k_0 : E \to F$ is obtained using the* **FCD** *extension of the operator above.*

When the strategy $\sigma/k_0$ is first activated, $\sigma$ is passed the initial state $k_0$. Then, whenever $\sigma$ makes a move $m@k$, $\sigma/k_0$ removes $k$ from the visible interaction, but remember it in order to adjoin it to the next incoming move.

# Part II

# CompCertO

# Chapter 7

# Semantics in CompCert

This chapter describes the semantic model of CompCert [Leroy, 2009]. There is no novelty in the technical material presented. However, the conceptual framework of dual nondeterminism and refinement is used to shine a new light on some aspects of the model and illuminate some of CompCert's more obscure definitions.

## 7.1 Whole-program semantics in CompCert

The semantics of CompCert languages are given in terms of a simple notion of process behavior. By *process*, we mean a self-contained computation which can be characterized by the sequence of system calls it performs. For a C program to be executed as a process, its translation units must be compiled to object files, then linked together into an executable binary loaded by the system.

The model used for verifying CompCert accounts for this in the way depicted in Fig. 7.1. Linking is approximated by merging programs, seen as sets of global definitions. The execution of a program composed of the translation units `M1.c` ... `Mn.c` which compile to `M1.s` ... `Mn.s` is

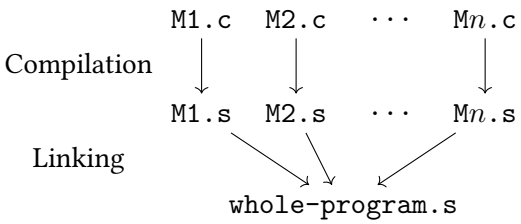



Figure 7.1: CompCert's approximation of the C toolchain

modeled as:

$$L_{\mathsf{tgt}} := \mathsf{Asm}(\texttt{M1.s} + \cdots + \texttt{M}n\texttt{.s})\,.$$

Here, $+$ denotes CompCert's linking operator and Asm maps an assembly program to its semantics. Note that the loading process is encoded as part of the definition of Asm, which constructs a global environment laying out the program's code and static data into the runtime address space, and models the conventional invocation of `main`. To formulate compiler correctness, we must also specify the behavior of the source program. To this end, CompCert defines a linking operator and semantics for the language Clight,[1] allowing the desired behavior to be specified as:

$$L_{\mathsf{src}} := \mathsf{Clight}(\texttt{M1.c} + \cdots + \texttt{M}n\texttt{.c})\,.$$

The correctness of CompCert can then be stated as $L_{\mathsf{src}} \sqsubseteq L_{\mathsf{tgt}}$.

### 7.1.1 Transition systems

Language semantics are given as labelled transition systems (LTS), which characterize a program's behavior in terms of sequences of observable events. Schematically, a CompCert LTS is a tuple $L = \langle S, I, \to, F \rangle$ consisting of a set of states $S$, a subset $I \subseteq S$ of initial states, a labelled transition relation $\to \subseteq S \times \mathbb{E}^* \times S$, and a set $F \subseteq S \times \mathsf{int}$ of final states associated with exit statuses. The relation $s \xrightarrow{t} s'$ indicates that the state $s$ may transition to the state $s'$ with the externally observable interaction $t \in \mathbb{E}^*$.

### 7.1.2 Trace semantics

The possible behaviors of CompCert LTS fall into four categories, which are used to define their trace semantics:

- An execution reaching a final state is said to *terminate*. For example, the following execution generates the event trace $t_1 t_2 \cdots t_{n-1}$ and terminates with status $r$:

$$I \ni s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s_n \; F \; r\,.$$

[1] Although CompCert features a frontend for a richer version of the C language, the simplified intermediate dialect Clight is usually used as the source language when using CompCert to build certified artifacts.

- An execution reaching an infinite sequence of $\epsilon$ transitions is said to *silently diverge*. The following execution diverges after generating the trace $t_1 t_2 \cdots t_{n-1}$:

$$I \ni s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s_n \xrightarrow{\epsilon} s_{n+1} \xrightarrow{\epsilon} \cdots$$

- By contrast, infinite executions which keep interacting are said to exhibit *reactive* behavior. The following execution is reactive if and only if $\forall i \, \exists j \, . \, i \leq j \wedge t_j \neq \epsilon$:

$$I \ni s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \xrightarrow{t_3} \cdots$$

- Finally, an execution which reaches a stuck state is said to *go wrong*. It will have the shape:

$$I \ni s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s_n \, ,$$

  with no $t, s'$ such that $s_n \xrightarrow{t} s'$. This models *undefined behavior* and can be refined by any behavior admitting $t_1 t_2 \cdots t_{n-1}$ as a prefix.

The model outlined above makes no attempt to model interactions across components, and is only ever used to describe the behavior of whole programs. This makes it challenging to reason about the behavior of individual compilation units, although there have been successful attempts in this direction [Kang et al., 2016; Song et al., 2019]. In any case, the compiler's correctness property as described in §?? only considers uses which follow the pattern approximated in Fig. 7.1.

To formulate a more fine-grained and flexible version of the correctness theorem of CompCert, we need an account of the behavior of individual translation units. The model presented in §8.2.1 achives this by recoding control transfers to and from the modeled system explicitly. These control transfer will need to expose more information about the internal structure of program states; in the following section we review the design of CompCert's memory model, which all program states are constructed from.

The construction of states in CompCert language semantics follows common patterns. In particular, all languages start with the same notion of *memory state*.

$$v \in \mathsf{val} ::= \mathsf{undef} \mid \mathsf{int}(n) \mid \mathsf{long}(n) \mid \mathsf{float}(x) \mid \mathsf{single}(x) \mid \mathsf{vptr}(b, o)$$
$$(b, o) \in \mathsf{ptr} = \mathsf{block} \times \mathbb{Z} \qquad (b, l, h) \in \mathsf{ptrrange} = \mathsf{block} \times \mathbb{Z} \times \mathbb{Z}$$

$$\begin{aligned} \mathsf{alloc} &: \mathsf{mem} \to \mathbb{Z} \to \mathbb{Z} \to \mathsf{mem} \times \mathsf{block} \\ \mathsf{free} &: \mathsf{mem} \to \mathsf{ptrrange} \to \mathsf{option}(\mathsf{mem}) \\ \mathsf{load} &: \mathsf{mem} \to \mathsf{ptr} \to \mathsf{option}(\mathsf{val}) \\ \mathsf{store} &: \mathsf{mem} \to \mathsf{ptr} \to \mathsf{val} \to \mathsf{option}(\mathsf{mem}) \end{aligned}$$

Figure 7.2: Outline of the CompCert memory model

### 7.1.3 Memory model

The CompCert memory model [Leroy et al., 2012; Leroy and Blazy, 2008] is the core algebraic structure underlying the semantics of CompCert's languages. Some of its operations are shown in Fig. 7.2. The idealized version presented here involves the type of memory states mem, the type of runtime values val, and the types of pointers ptr and address ranges ptrrange. To keep our exposition concise and clear, we gloss over the technical details associated with modular arithmetic and overflow constraints.

The memory is organized into a finite number of *blocks*. Each memory block has a unique identifier $b \in \mathsf{block}$ and is equipped with its own linear address space. Block identifiers and offsets are often manipulated together as pointers $p = (b, o) \in \mathsf{ptr} = \mathsf{block} \times \mathbb{Z}$. New blocks are created with prescribed boundaries using the primitive alloc. A runtime value $v \in \mathsf{val}$ can be stored at a given address using the primitive store, and retreived using the primitive load. Values can be integers (int, long) and floating point numbers (float, single) of different sizes, as well as pointers (vptr). The special value undef represents an undefined value. Simulation relations often allow undef to be refined into a more concrete value; we write value refinement as $\leq_{\mathsf{v}} := \{(\mathsf{undef}, v), (v, v) \mid v \in \mathsf{val}\}$.

The memory model is shared by all of the languages in CompCert. States always consist of a memory component $m \in \mathsf{mem}$, alongside language-specific components which may contain additional values (val).

## 7.2 Decomposing heterogenous systems

Existing work turns CompCert into a platform enabling compositional verification (§7.4). However, in most cases, the horizon is a completed assembly program to be run as a user-level process. This becomes a limitation in the context of heterogenous systems.

For example, consider the problem of verifying a network interface card (NIC) driver. The NIC and its driver are closely coupled, but the details of their interaction are irrelvant to the rest of the system and should not leak into our reasoning at larger scales. Instead, we wish to treat them as a unit and establish a direct relationship between calls into the driver's C interface and network communication. Together, the NIC and driver implement a specification $\sigma : \mathsf{Net} \to \mathcal{C}$ (see §6). The driver code would be specified ($\sigma_{\mathsf{drv}}$) and verified at the level of CompCert semantics, whereas device I/O primitives ($\sigma_{\mathsf{io}}$) and the NIC ($\sigma_{\mathsf{NIC}}$) would be specified as additional components:

$$\sigma_{\mathsf{NIC}} : \mathsf{Net} \to \mathsf{IO} \qquad \sigma_{\mathsf{io}} : \mathsf{IO} \to \mathcal{C} \qquad \sigma_{\mathsf{drv}} : \mathcal{C} \to \mathcal{C}$$

By reasoning about their interaction, it would be possible to establish a relationship between the overall specification $\sigma$ and the composition $\sigma_{\mathsf{drv}} \circ \sigma_{\mathsf{io}} \circ \sigma_{\mathsf{NIC}}$. Then a *compiler of certified components* would help us transport specifications and proofs obtained with respect to the driver's C code to the compiled code operating at the level of assembly ($\sigma' : \mathsf{Net} \to \mathcal{A}$).

In this paper, we show how to adapt the semantic model and correctness proofs of CompCert so that they can be used in this way. Our model is not intended to reach the level of generality required to handle all aspects of the problem above; indeed we want it to remain tailored to CompCert's verification as much as possible. Instead, we envision a hierarchy where components could be verified in suitable models, then soundly embedded in more general ones to be made interoperable. To make it possible for CompCert to be used in this context, the ability to treat the driver code as an independent component is crucial. This excludes approaches to compositional compiler correctness which are formulated in terms of completed programs.

| Variant | Semantic model | (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|---|
| (Sep)CompCert [Kang et al., 2016; Leroy, 2009] | $\chi : \mathbf{1} \twoheadrightarrow \mathcal{C} \vdash \mathbf{1} \twoheadrightarrow \mathcal{W}$ | | | | ✓ | ✓ |
| CompCertX [Gu et al., 2015] | $\chi : \mathbf{1} \twoheadrightarrow \mathcal{C} \times \mathcal{A} \vdash \mathbf{1} \twoheadrightarrow \mathcal{C} \times \mathcal{A}$ | | | ✓ | ✓ | ✓ |
| Compositional CompCert [Stewart et al., 2015] | $\mathcal{C} \twoheadrightarrow \mathcal{C}$ | ✓ | ✓ | | | |
| CompCertM [Song et al., 2019] | $\mathcal{C} \times \mathcal{A} \twoheadrightarrow \mathcal{C} \times \mathcal{A}$ | ✓ | | ✓ | ✓ | ✓ |
| CompCertO | $A \twoheadrightarrow A \quad (A \in \mathbb{L})$ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 7.1: Taxonomy of CompCert extensions in terms of the corresponding game models (§6) and the requirements they satisfy (§7.3). The parameter $\chi : \mathbf{1} \twoheadrightarrow \mathcal{C}$ pre-specifies the behavior of external functions, whereas games on the left of arrows correspond to dynamic interactions. As a distinguishing feature, CompCertO's model is parametrized by a generic notion of *language interface* $A \in \mathbb{L} \supseteq \{\mathcal{C}, \mathcal{A}\}$. See §7.4 for details.

## 7.3 Requirements for CompCert

To handle use cases like the one we have presented above, the compiler's correctness proof should satisfy the following requirements:

1. The semantics of the source and target languages should characterize the behavior of *open* components in terms of their interactions with the rest of the program.
2. The correctness theorem should go beyond refinement under a fixed notion of program context, and relate the interactions of the source and target modules directly.
3. The abstraction gap between C and assembly-level interactions should be made explicit.
4. Some form of certified *linking* should be provided as well as certified compilation.
5. To facilitate integration into the official release, changes to the existing proofs of CompCert should be minimal.

As outlined in Table 7.1, each of these requirements is fulfilled by some exisiting CompCert extension, however none satisfies them all.

This paper introduces CompCertO, the first extension of CompCert to address all of these requirements simultaneously. The key to this achievement is the expressivity of our model.

## 7.4 Related work and evaluation

A general survey, discussion and synthesis of various compositional compiler correctness results is provided by Patterson and Ahmed [2019]. We focus on CompCert extensions. Our conceptual

framework can be used to classify previous work on semantic models of CompCert. By reinterpreting these models in terms of games and strategies, we can establish the taxonomy presented in Table 7.1.

### 7.4.1 CompCert and SepCompCert

As noted in §8.1, the whole-program semantics used by CompCert express strategies for the game $\mathbf{1} \twoheadrightarrow \mathcal{W}$. CompCert's original correctness theorem stated the refinement property $\mathsf{C}_{\mathsf{wp}}(p) \sqsubseteq \mathsf{Asm}_{\mathsf{wp}}(p')$, where $\mathsf{C}_{\mathsf{wp}}$ and $\mathsf{Asm}_{\mathsf{wp}}$ denote the source and target whole-program semantics. SepCompCert [Kang et al., 2016] later introduced the linking operator $+$ and generalized the correctness theorem to the form discussed in §7.1.

Since external calls are not accounted for explicitly in this semantic model, they are interpreted using a common global parameter $\chi$ specifying their behavior. The correctness proof assumes that $\chi$ is deterministic and that it satisfies a number of healthiness requirements with respect to the memory transformations used in CompCert's correctness proof, corresponding roughly to self-simulation under the CKLRs ext and injp.

### 7.4.2 Contextual compilation

CompCertX [Gu et al., 2015] and Stack-Aware CompCert [Wang et al., 2019] generalize the incoming interface of programs from $\mathcal{W}$ to $\mathcal{C}$, and as such characterizes the behavior not only of `main` but of any function of the program, called with any argument values. This allows CompCertX and its correctness theorem to be used in the layer-based verification of the CertiKOS kernel: once the code of a given abstraction layer has been verified and compiled using CompCertX, that layer's specification can be used as the new $\chi$ when the next layer is verified. However, this approach does not support mutually recursive components, and requires the healthiness conditions on $\chi$ to be proved before the next layer is added.

### 7.4.3 Compositional CompCert

The *interaction semantics* of Compositional CompCert [Stewart et al., 2015] are closer to our own model but are limited to the language interface $\mathcal{C}$. Likewise, the simulations used in Compositional CompCert correspond to our notion of forward simulation for a single convention called *structured*

*injections*, which we will write $\mathbb{SI}$. Simulation proofs are updated to follow this model, and the *transitivity* of $\mathbb{SI}$ is established ($\mathbb{SI} \cdot \mathbb{SI} \equiv \mathbb{SI}$), so that passes can be composed to obtain a simulation for the whole compiler.

Compositional CompCert also introduced a notion of *semantic linking* similar to our horizontal composition (§8.2.1). As in our case, semantic linking is shown to preserve simulations (Thm. 5), however semantic linking is not related to syntactic linking of assembly programs, and this was later shown to be problematic [Song et al., 2019].

Another limitation of Compositional CompCert is the complexity of the theory and the proof effort required. Because of the use of a single simulation convention, many assumptions naturally expressed as relational invariants in the simulation relations of CompCert must be either captured by $\mathbb{SI}$ or handled at the level of language semantics, and simulation proofs essentially had to be rewritten and adapted to structured injections.

### 7.4.4 CompCertM

The most recent extension of CompCert is CompCertM [Song et al., 2019], which shares common themes and was developped concurrently with our work. While its correctness is ultimately stated in terms of closed semantics, CompCertM uses a notion of open semantics as an intermediate construction to enable compositional compilation and verification.

The open semantics used in CompCertM builds on interaction semantics by incorporating an assembly language interface. The resulting semantic model can be characterized as $\mathcal{C} \times \mathcal{A} \twoheadrightarrow \mathcal{C} \times \mathcal{A}$. Simulations are parametrized by Kripke relations similar to CKLRs (§9) and predicates similar to our invariants (§10). While simulations do not directly compose, a new technique called *refinement under self-related context* (RUSC) can nonetheless be used to derive a contextual refinement theorem for the whole compiler with minimal overhead.

This approach has many advantages. CompCertM avoids much of the complexity of Compositional CompCert when it comes to composing passes, and the flexibility of the simulations used makes updating the correctness proofs of passes much easier. CompCertM also charts new ground in several directions. The RUSC relation used to state the final theorem is shown to be adequate with respect to the trace semantics of closed programs. CompCertM has improved support for static variables and the verification of the assembly runtime function `utod` is demonstrated.

| Component | SLOC | |
|---|---|---|
| Semantic framework (§8.2) | +782 | (+14%) |
| Horizontal composition (§8.2.1) | 676 | |
| Simulation convention algebra (§8.3) | 1,052 | |
| CKLR theory and instances (§9) | 1,807 | |
| Clight and RTL parametricity (§9.5) | 2,741 | |
| Invariant preservation proofs (§10) | +549 | (+7%) |
| Pass correctness proofs (Tbl. 8.3) | +765 | (+2%) |
| **Total** | **8372** | |

Table 7.2: Significant lines of code in CompCertO relative to CompCert v3.6. See Table 8.3 for a per-pass breakdown of the increase in size of pass correctness proofs, and `overhead.py` in the Coq development for the list of files included in each group.

In other aspects, CompCertM inherits the limitation of previous approaches whereas CompCertO goes further. Because the compiler correctness theorem is not itself expressed as a simulation, it fails requirement #2 laid out in §7.3. While it may be possible to accomodate #3, the parametrization of simulations does not offer the same flexibility as our full-blown notion of simulation convention. As a consequence, a cascade of techniques (*repaired* interaction semantics, *enriched* memory injections, the mixed simulations of [Neis et al., 2015]) need to be deployed to enforce invariants which find a natural relational expression under our appoach.

Therefore, an interesting question for future investigation will be to determine to what extent the techniques used by CompCertM and CompCertO could be integrated to combine the strengths of both developments.

### 7.4.5 CompCertO

To give a sense of the overall complexity of CompCertO, we list in Table 7.2 the increase in significant lines of code it introduces compared to CompCert v3.6. As shown in Table 8.3, our methodology comes with a negligible increase in the complexity of most simulation proofs. Although SLOC is an imperfect measure, and a 1:1 comparison between developments which prove different things is difficult, our numbers represent a drastic improvement over Compositional CompCert, and compare favorably or are on par with the corresponding sections of CompCertM.

Our use of the simulation conventions injp, alloc, stacking and asmgen in particular underscores the benefits of our approach. The corresponding passes are the root of much complexity in Compositional CompCert, CompCertX and CompCertM. For instance, to express the require-

ment on the areas protected by inj, both Compositional CompCert and CompCertM introduce general mechanisms for tracking ownership of different regions of memory as part of an extended notion of memory injection. The approach taken here demonstrates that the requirements placed on external functions by the original CompCert are already good enough for the job! Because our framework is expressive enough to capture them, the corresponding passes barely need any modifications, and the associated issues are resolved before they even show up.

Likewise, the preservation of callee-save registers ensured by the Allocation pass, and the subtle issues associated with argument-passing in the Stacking pass have been the cause of much pain in previous CompCert extensions. The ease with which they are addressed here demonstrates the power of an explicit treatment of abstraction, made possible by our notions of language interface and simulation convention.

# Chapter 8

# CompCertO

The games we start from have a particularly simple structure. We call each one a *language interface*. Their moves are partitionned into questions and answers, where questions correspond to function invocations and answers return control to the caller.

**Definition 23.** *A* language interface *is a tuple* $A = \langle A^\circ, A^\bullet \rangle$*, where* $A^\circ$ *is a set of* questions *and* $A^\bullet$ *is a set of* answers.

We focus on games of the form $A \to B$, where $A$ and $B$ are language interfaces. In this setting, the valid positions of $A \to B$ are sequences of the form:

$$q \cdot \underline{m_1} \cdot n_1 \cdots \underline{m_k} \cdot n_k \cdot \underline{r} \in B^\circ (A^\circ A^\bullet)^* B^\bullet$$

and all their prefixes. In our context, this describes a program component responding to an incoming call $q$. The component performs a series of external calls $m_1 \dots m_k$ yielding the results $n_1 \dots n_k$, and finally returns from the top-level call with the result $r$. The arrows show the correspondance between questions and answers but are not part of the model.

**Example 8.** *We use a simplified version of C and assembly to illustrate some of the principles behind our model. Consider the program components in Fig. 8.1. The behavior of B.c as it interacts with A.c is described by plays of the form:*

$$\mathsf{sqr}(3) \cdot \underline{\mathsf{mult}(3,3)} \cdot 9 \cdot \underline{9} \tag{8.1}$$

| A.c | int mult(int n, int p) {<br>  return n * p;<br>} | A.s | mult:<br><br><br> | %eax := %ebx<br>%eax *= %ecx<br>ret |
|---|---|---|---|---|
| B.c | int sqr(int n) {<br>  return mult(n, n);<br>} | B.s | sqr:<br><br>L1: | %ecx := %ebx<br>call mult<br>ret |

Figure 8.1: Two simple C compilation units and corresponding assembly code. For this example, the calling convention stores arguments in the registers %ebx and %ecx and return values in the register %eax.

*This corresponds to the game $\tilde{\mathcal{C}} \to \tilde{\mathcal{C}}$ for a language interface $\tilde{\mathcal{C}} := \langle \mathsf{ident} \times \mathsf{val}^*, \mathsf{val} \rangle$. Questions specify the function to invoke and its arguments; answers carry the value returned by the function.*

*To describe the behavior of `A.s` and `B.s`, we use a set of registers $R := \{\mathsf{pc}, \mathsf{eax}, \mathsf{ebx}, \mathsf{ecx}\}$ together with a stack of pending return addresses (*pc *is the program counter). The corresponding language interface can be defined as $\tilde{\mathcal{A}} := \langle \mathsf{val}^R \times \mathsf{val}^*, \mathsf{val}^R \times \mathsf{val}^* \rangle$. A possible execution of `B.s` is:*

$$\left[\begin{array}{rl} \mathsf{pc} \mapsto & \mathsf{sqr} \\ \mathsf{eax} \mapsto & 42 \\ \mathsf{ebx} \mapsto & 3 \\ \mathsf{ecx} \mapsto & 7 \\ \textit{stack:} & x \cdot \vec{k} \end{array}\right] \cdot \underline{\left[\begin{array}{rl} \mathsf{pc} \mapsto & \mathsf{mult} \\ \mathsf{eax} \mapsto & 42 \\ \mathsf{ebx} \mapsto & 3 \\ \mathsf{ecx} \mapsto & 3 \\ \textit{stack:} & \mathsf{L1} \cdot x \cdot \vec{k} \end{array}\right]} \cdot \left[\begin{array}{rl} \mathsf{pc} \mapsto & \mathsf{L1} \\ \mathsf{eax} \mapsto & 9 \\ \mathsf{ebx} \mapsto & 3 \\ \mathsf{ecx} \mapsto & 3 \\ \textit{stack:} & x \cdot \vec{k} \end{array}\right] \cdot \underline{\left[\begin{array}{rl} \mathsf{pc} \mapsto & x \\ \mathsf{eax} \mapsto & 9 \\ \mathsf{ebx} \mapsto & 3 \\ \mathsf{ecx} \mapsto & 3 \\ \textit{stack:} & \vec{k} \end{array}\right]} \tag{8.2}$$

*The correspondance between (8.1) and (8.2) is determined by the C calling convention in use. We will discuss this point in more detail in §8.1.3.*

## 8.1 Overview

The semantic model of CompCert corresponds to a game $\mathcal{E} \to \mathcal{W}$. Programs are run without any parameters and produce a single integer denoting their exit status. This is described by the language interface $\mathcal{W} := \langle \mathbb{1}, \mathsf{int} \rangle$, where $\mathbb{1} = \{*\}$ is the unit set and int is the set of machine integers. Interaction with the environment is captured as a trace of events from a predefined set, each with an output and input component. These events, described by the language interface $\mathcal{E}$, correspond to system calls and accesses to volatile variables.

### 8.1.1 Semantic model

In order to model open components and cross-component interactions, we generalize CompCert's labelled transition systems to describe strategies for games of the form:

$$A \twoheadrightarrow B \;:=\; A \times \mathcal{E} \to B\,.$$

The language interface $B$ describes how a component can be activated, and the ways in which it can return control to the caller. The language interface $A$ describes the external calls that the component may perform in the course of its execution.

This flexibility allows us to treat interactions at a level of abstraction adapted to each language. For example, CompCertO's source language Clight uses the game $\mathcal{C} \twoheadrightarrow \mathcal{C}$. The questions of $\mathcal{C}$ specify a function to call, argument values, and the state of the memory at the time of invocation; the answers specify a return value and an updated memory state. On the other hand, the target language Asm uses $\mathcal{A} \twoheadrightarrow \mathcal{A}$, where $\mathcal{A}$ describes control transfers in terms of processor registers rather than function calls. The language interfaces used in CompCertO are described in §8.2.1.

### 8.1.2 Simulations

CompCert uses simulation proofs to establish a correspondence between the externally observable behaviors of the source and target programs of each compilation pass. The internal details of simulation relations have no bearing on this correspondance, so these details can remain hidden to fit a uniform and transitive notion of pass correctness. This makes it easy to derive the correctness of the whole compiler from the correctness of each pass.

To achieve compositionality across compilation units, our model must reveal details about component interactions which were previously internal. Since many passes transform these interactions in specialized ways, this breaks the uniformity of pass correctness properties.

Existing work attempts to recover this uniformity by developing a general notion of correctness covering all passes or by delaying pass composition so that it operates on closed semantics only. Unfortunately, these techniques either conflict with our requirement #2, make proofs more complex, or cascade into subtle "impedence mismatch" problems requiring their own solutions (see §7.4).

By contrast, we capture the particularities of each simulation proof by introducing a notion of *simulation convention* expressing the correspondance between source- and target-level interactions. To describe simulation conventions and reason about them, we use logical relations.

### 8.1.3 Simulation conventions

The framework of Kripke relations allows us to define simulation conventions as follows. The worlds ensure that corresponding pairs of questions and answers are related consistently.

**Definition 24.** *A* simulation convention *between the language interfaces $A_1 = \langle A_1^\circ, A_1^\bullet \rangle$ and $A_2 = \langle A_2^\circ, A_2^\bullet \rangle$ is a tuple $\mathbb{R} = \langle W, \mathbb{R}^\circ, \mathbb{R}^\bullet \rangle$ with $\mathbb{R}^\circ \in \mathcal{R}_W(A_1^\circ, A_2^\circ)$ and $\mathbb{R}^\bullet \in \mathcal{R}_W(A_1^\bullet, A_2^\bullet)$. We will write $\mathbb{R} : A_1 \Leftrightarrow A_2$. In particular, for a language interface $A$, the* identity *simulation convention is defined as $\mathsf{id}_A := \langle \mathbb{1}, =, = \rangle : A \Leftrightarrow A$. We will usually omit the subscript $A$.*

A simulation between the transition systems $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$ is then assigned a type $\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B$, where $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$ are simulation conventions relating the corresponding language interfaces. We write $L_1 \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$.

Table 8.1 presents a summary of notations. Simulation conventions will often be derived from more elementary relations, following the internal structure of questions and answers (see §9).

**Example 9.** *The calling convention we used in Example 8 can be formalized as a simulation convention $\tilde{\mathbb{C}} := \langle \mathsf{val}^*, \tilde{\mathbb{C}}^\circ, \tilde{\mathbb{C}}^\bullet \rangle : \tilde{\mathcal{C}} \Leftrightarrow \tilde{\mathcal{A}}$. We use the set of worlds $\mathsf{val}^*$ to relate the stack of assembly questions to that of the corresponding answers. The relations $\tilde{\mathbb{C}}^\circ, \tilde{\mathbb{C}}^\bullet$ are defined by:*

$$\frac{rs[\mathsf{pc}] = f \qquad \vec{v} \text{ is is contained in } rs[\mathsf{ebx}, \mathsf{ecx}]}{x\vec{k} \Vdash f(\vec{v})\ \tilde{\mathbb{C}}^\circ\ rs@x\vec{k}} \qquad \frac{rs[\mathsf{eax}] = v' \qquad rs[\mathsf{pc}] = x}{x\vec{k} \Vdash v'\ \tilde{\mathbb{C}}^\bullet\ rs@\vec{k}}$$

*For a C-level function invocation $f(\vec{v})$, we expect the register* pc *to point to the beginning of the function $f$, and the registers* ebx *and* ecx *to contain the first and second arguments (if applicable). Other registers may contain arbitrary values. The stack $x\vec{k}$ has no relationship to the C question, however the assembly answer is expected to pop the return address and branch to it, setting the program counter* pc *accordingly. In addition, the return value $v'$ must be stored in the register* eax.

The expressive power of simulation conventions makes the adaptation of existing correctness proofs for the various passes of CompCert straightforward. Instead of forcing all passes into a

| Notation | Examples | Description |
|---|---|---|
| $R \in \mathcal{R}(S_1, S_2)$ | $\leq_{\mathsf{v}}$ | Simple relation |
| $R \in \mathcal{R}_W(S_1, S_2)$ | $\hookrightarrow_{\mathsf{m}}$ | Kripke relation (Def. 2) |
| $w \Vdash R$ | | Kripke relation at world $w$ |
| $w \Vdash x\ R\ y$ | | $x$ and $y$ related at world $w$ |
| $A, B, C$ | $\mathcal{C}, \mathcal{A}, \mathbf{1}$ | Language interface (Def. 23) |
| $\mathbb{R} : A_1 \Leftrightarrow A_2$ | $\mathsf{alloc}$ | Simulation convention (Def. 24) |
| $L : A \twoheadrightarrow B$ | $\mathsf{Clight}(p)$ | LTS for $A \twoheadrightarrow B$ (Def. 25) |
| $L_1 \oplus L_2$ | | Horizontal composition (Def. 26) |
| $L_1 \leq_{\mathbb{R} \twoheadrightarrow \mathbb{S}} L_2$ | Thm. 10 | Simulation property (Def. 27) |

Table 8.1: Summary of notations

$$\mathsf{id} : A \Leftrightarrow A \qquad L \leq_{\mathsf{id} \twoheadrightarrow \mathsf{id}} L$$

$$\frac{\mathbb{R} : A_1 \Leftrightarrow A_2 \qquad \mathbb{S} : A_2 \Leftrightarrow A_3}{\mathbb{R} \cdot \mathbb{S} : A_1 \Leftrightarrow A_3}$$

$$\frac{L_1 \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2 \qquad L_2 \leq_{\mathbb{S}_A \twoheadrightarrow \mathbb{S}_B} L_3}{L_1 \leq_{\mathbb{R}_A \cdot \mathbb{S}_A \twoheadrightarrow \mathbb{R}_B \cdot \mathbb{S}_B} L_3}$$

$$\begin{array}{ccc} A_1 & \xrightarrow{L_1} & B_1 \\ {\scriptstyle \mathbb{R}_A}\Big\updownarrow & & \Big\updownarrow{\scriptstyle \mathbb{R}_B} \\ A_2 & \xrightarrow{L_2} & B_2 \\ {\scriptstyle \mathbb{S}_A}\Big\updownarrow & & \Big\updownarrow{\scriptstyle \mathbb{S}_B} \\ A_3 & \xrightarrow{L_3} & B_3 \end{array}$$

Figure 8.2: Simulation identity and vertical composition

single one-size-fits-all mold, we can choose conventions matching the simulation relation and invariants used in each pass. Simulations for each pass can then be composed in the way shown in Fig. 8.2.

### 8.1.4 Simulation convention algebra

CompCert's *injection* passes (see §9) pose a particular challenge, already encountered in the work on Compositional CompCert [Stewart et al., 2015]: injection passes make stronger assumptions on external calls than they guarantee for incoming calls. Our framework can express this situation by using different simulation conventions for external and incoming calls ($\mathsf{injp} \twoheadrightarrow \mathsf{inj}$). However, since the external calls of one component and the incoming calls of another will not be related in compatible ways, this asymmetry breaks horizontal compositionality (Thm. 5).

In CompCertO, we rectify this imbalance *outside* of the simulation proof itself. Simulations for individual passes are not always horizontally compositional, but we are able to derive a symmetric simulation convention for the compiler as a whole. Properties of the Clight and RTL languages allow us to strengthen the correctness proof. These properties are encoded as self-simulations and

| Name | Question | Answer | Description |
|---|---|---|---|
| $\mathcal{C}$ | $vf[sg](\vec{v})@m$ | $v'@m'$ | C calls |
| $\mathcal{L}$ | $vf[sg](ls)@m$ | $ls'@m'$ | Abstract locations |
| $\mathcal{M}$ | $vf(sp, ra, rs)@m$ | $rs'@m'$ | Machine registers |
| $\mathcal{A}$ | $rs@m$ | $rs'@m'$ | Arch-specific |
| $\mathbf{1}$ | n/a | n/a | Empty interface |
| $\mathcal{W}$ | * | $r$ | Whole-program |

Table 8.2: Language interfaces used in CompCertO

inserted as pseudo-passes. We can then perform algebraic manipulations on simulation statements to rewrite the overall simulation convention used by the compiler into a symmetric one.

These algebraic manipulations are based on a notion of simulation convention refinement ($\sqsubseteq$) allowing a simulation convention to replace another in all simulation statements. We construct a typed Kleene algebra [Kozen, 1998] based on this order, and use it to ensure that our compiler correctness statement is both compositional and insensitive to the inclusion of optional passes.

## 8.2 Operational semantics

### 8.2.1 Open semantics in CompCertO

The memory model also plays a central role when describing interactions between program components. In our approach, the memory state is passed alongside all control transfers.

**Language interfaces** Our models of cross-component interactions in CompCert languages are shown in Table 8.2. At the source level ($\mathcal{C}$), questions consist of the address of the function being invoked ($vf \in \mathsf{val}$), its signature ($sg \in \mathsf{signature}$), the values of its arguments ($\vec{v} \in \mathsf{val}^*$), and the state of the memory at the point of entry ($m \in \mathsf{mem}$); answers consist of the function's return value and the state of the memory at the point of exit. This language interface is used for Clight and for the majority of CompCert's intermediate languages.

As we move towards lower-level languages, this is reflected in the language interfaces we use: function arguments are first mapped into abstract locations alongside local temporary variables ($\mathcal{L}$, used by LTL and Linear). These locations are eventually split between in-memory stack slots and a fixed number of machine registers ($\mathcal{M}$, used by Mach). Finally, the target assembly language Asm stores the program counter, stack pointer, and return address into their own machine

registers, which is reflected in its interface $\mathcal{A}$.

The interface of whole-program execution can also be described in this setting: the language interface $\mathbf{1}$ contains no move; per §8.1, the interface $\mathcal{W}$ has a single trivial question $*$, and the answers $r \in \mathsf{int}$ give the exit status of a process. Hence the original CompCert semantics described in §7.1 can be seen to define strategies for $\mathbf{1} \twoheadrightarrow \mathcal{W}$: the process can only be started in a single way, cannot perform any external calls, and indicates an exit status upon termination.

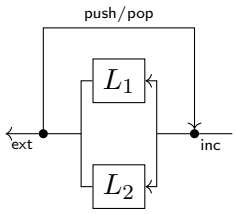
**Transition systems** To account for the cross-component interactions described by language interfaces, CompCertO extends the transition systems described in §7.1 as follows.

**Definition 25.** *Given an* incoming *language interface $B$ and an* outgoing *language interface $A$, a* labelled transition system for the game $A \twoheadrightarrow B$ *is a tuple $L = \langle S, \to, D, I, X, Y, F \rangle$. The relation $\to \subseteq S \times \mathbb{E}^* \times S$ is a* transition relation *on the set of states $S$. The set $D \subseteq B^\circ$ specifies which questions the component accepts; $I \subseteq D \times S$ then assigns to each one a set of* initial states. *$F \subseteq S \times B^\bullet$ designates* final states *together with corresponding answers. External calls are specified by $X \subseteq S \times A^\circ$, which designates* external states *together with a question of $A$, and $Y \subseteq S \times A^\bullet \times S$, which is used to select a* resumption state *to follow an external state based on the answer provided by the environment. We write $L : A \twoheadrightarrow B$ when $L$ is a labelled transition system for $A \twoheadrightarrow B$.*

We use infix notation for the various transition relations $I, X, Y, F$. In particular we write $n\ Y^s\ s'$ to denote that $n \in A^\bullet$ resumes the suspended external state $s$ to continue with state $s'$. The main reason for treating events $e \in \mathbb{E}$ and external calls $mn \in A^\circ A^\bullet$ differently is that while events are expected to be the same between the source and target programs, the form of external calls varies significantly across languages and the simulation convention they follow must be defined explicitly. In addition, while events and event traces bundle together the output and input components of the interaction, our representation of external calls separates them, which simplifies the formulation of horizontal composition and open simulations.

**Horizontal composition** To model linking, we need to express the external behavior of a collection of components in terms of the behaviors of individual components.

Consider the components $L_1, L_2 : A \twoheadrightarrow A$. When $L_1$ is running and performs an external call to one of the functions implemented by $L_2$, the execution of $L_1$ is suspended. The question of $L_1$ to




$$\frac{q \in D_i \qquad q\ I_i\ s}{q\ I\ (i,s)}\ \mathsf{inc}^\circ \qquad \frac{s \xrightarrow{t}_i s'}{(i,s)\,k \xrightarrow{t} (i,s')\,k}\ \mathsf{run} \qquad \frac{s\ F_i\ r}{(i,s)\ F\ r}\ \mathsf{inc}^\bullet$$

$$\frac{s\ X_i\ q \qquad q \in D_j \qquad q\ I_j\ s'}{(i,s)\,k \xrightarrow{\epsilon} (j,s')\,(i,s)\,k}\ \mathsf{push} \qquad \frac{s'\ F_j\ r \qquad r\ Y_i^s\ s''}{(j,s')\,(i,s)\,k \xrightarrow{\epsilon} (i,s'')\,k}\ \mathsf{pop}$$

$$\frac{s\ X_i\ q \qquad \forall j\,.\,q \notin D_j}{(i,s)\,k\ X\ q}\ \mathsf{ext}^\circ \qquad \frac{r\ Y_i^s\ s'}{r\ Y^{(i,s)\,k}\ (i,s')\,k}\ \mathsf{ext}^\bullet$$

Figure 8.3: Horizontal composition of open semantics. The state is a stack of alternating activations of the two components, initialized as a singleton by an incoming question (inc°). During normal execution (run), the top-level state is updated. Calls into the other component push a new state onto the stack (push), initialized to handle the call in question. If a final state is reached while there are suspended activations (pop), the result is used to resume the most recent one. External calls which are provided by neither component (ext°, ext•), and final states encountered at the top level (inc•), are simply passed along to the environment.

$L_2$ is used to initialize a new state for $L_2$, and $L_2$ becomes the active component. Once $L_2$ reaches a final state, the corresponding answer is used to resume the execution of $L_1$. In the process $L_2$ may itself perform cross-component calls, instantiating *new* executions of $L_1$. Therefore, in addition to the state of the active component, we need to maintain a *stack* of suspended states for component instances awaiting resumption. The corresponding transition system is described in Fig. 8.3.

**Definition 26** (Horizontal composition)**.** *For two transition systems $L_1, L_2 : A \twoheadrightarrow A$ with $L_i = \langle S_i, \to_i, D_i, I_i, X_i, Y_i, F_i \rangle$, the* horizontal composition *of $L_1$ and $L_2$ is defined as:*

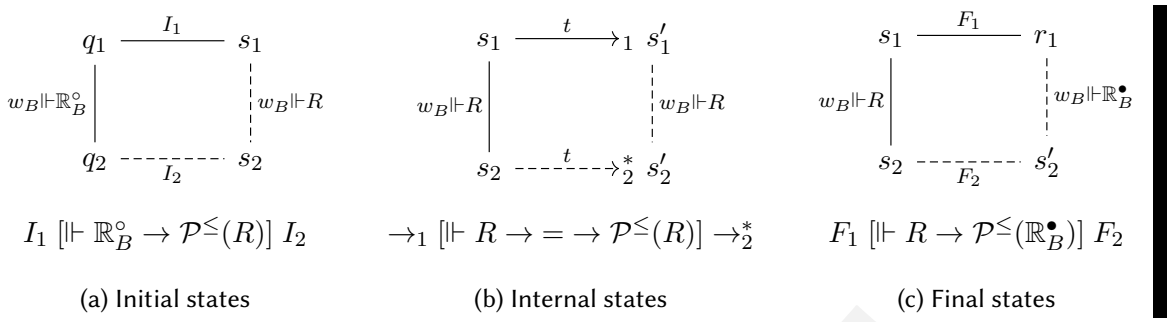
$$L_1 \oplus L_2 := \langle (S_1 + S_2)^*, \to, D_1 \cup D_2, I, X, Y, F \rangle$$

*where the components $\to$, $I$, $X$, $Y$, $F$ are defined by the rules shown in Fig. 8.3.*

### 8.2.2 Open simulations

CompCert is proved correct using a simulation between the transition semantics of the source and target programs. This *forward*[1] simulation is used to establish a *backward* simulation. Backward simulations are in turn proved to be sound with respect to trace containement. We have updated forward and backward simulations to work with CompCertO's semantic model. In this section we present forward simulations, which are used as our primary notion of refinement.

[1] In this usage, *forward* pertains to the compilation process, rather than the execution of programs.




(a) Initial states (b) Internal states (c) Final states

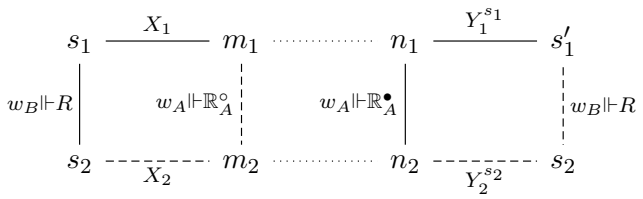
Figure 8.4: Forward simulation properties for initial, internal and final states. (a) When the execution is initiated by two related questions, any initial state of $L_1$ is matched by a related initial state of $L_2$. (b) Every internal transition of $L_1$ is then matched by a sequence of transitions of $L_2$, preserving the simulation relation. (c) When a final state is eventually reached by $L_1$, any related state in $L_2$ is final as well and produces a related answer. The indexing of the relations $\mathbb{R}_B^\circ$, $R$, $\mathbb{R}_B^\bullet$ by the Kripke world $w_B$ guarantees that the original questions and their eventual answers are related in a consistent way.

A forward simulation asserts that any transition in the source program has a corresponding transition sequence in the target. The sequence may be empty, but to ensure the preservation of silent divergence this can only happen for finitely many consecutive source transitions. This is enforced by indexing the simulation relation over a well-founded order, and requiring the index to decrease whenever an empty transition sequence is used. This mechanism is unchanged in CompCertO and is largely orthogonal to the techniques we introduce, so we omit this aspect of forward simulations in our exposition below.

**Forward simulations** Our transition systems introduce various forms of external communication, which must be taken into account by our notions of simulation. In CompCertO, a forward simulation between the small-step semantics $L_1 : A_1 \to B_1$ and $L_2 : A_2 \to B_2$ operates in the context of the simulation conventions $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ and $\mathbb{R}_B : B_1 \Leftrightarrow B_2$.

As depicted in Fig. 8.4, if questions of $B_1$ and $B_2$ respectivelly used to activate $L_1$ and $L_2$ are related by the simulation convention $\mathbb{R}_B$ at a world $w_B$, simulations guarantee that the corresponding answers will be related by $\mathbb{R}_B$ as well: the diagrams can be pasted together horizontally to follow the executions of $L_1$ and $L_2$. Note that the simulation relation $R \in \mathcal{R}_{W_B}(S_1, S_2)$ is itself indexed by $w_B$ to ensure that answers are related consistently with the corresponding questions.

The simulation convention $\mathbb{R}_A$ determines the correspondance between outgoing questions




$$
\begin{aligned}
&\forall\, w_B\, s_1\, s_2\, m_1 \,.\, w_B \Vdash s_1 \;\; R \;\; s_2 \wedge s_1 \;\; X_1 \;\; m_1 \Rightarrow \\
&\quad \exists\, w_A\, m_2 \,.\, w_A \Vdash m_1 \;\; \mathbb{R}_A^\circ \;\; m_2 \wedge s_2 \;\; X_2 \;\; m_2 \;\wedge \\
&\quad\quad \forall\, n_1\, n_2\, s_1' \,.\, w_A \Vdash n_1 \;\; \mathbb{R}_A^\bullet \;\; n_2 \wedge n_1 \;\; Y_1^{s_1} \;\; s_1' \Rightarrow \\
&\quad\quad\quad \exists\, s_2' \,.\, w_B \Vdash s_1' \;\; R \;\; s_2' \wedge n_2 \;\; Y_2^{s_2} \;\; s_2'
\end{aligned}
$$

Figure 8.5: Forward simulation property for external states. When $L_1$ assigns a question $m_1$ to an external state $s_1$, $L_2$ assigns a corresponding question $m_2$ to any related state $s_2$. The questions are related according to the simulation convention $\mathbb{R}_A$, at a world $w_A$ chosen by the simulation. When $L_1$ is resumed by an answer $n_1$, then a related answer $n_2$ also resumes $L_2$, and reestablishes the simulation relation between resulting states.

triggered by the transition systems' external states. The corresponding simulation properties are shown in Fig. 8.5. Compared with the treatment of incoming questions, the roles of the system and environment are reversed: the simulation proof can choose $w_A$ to relate the outgoing questions, and the environment guarantees that any corresponding answers will be related at that world.

**Definition 27** (Forward simulation)**.** *Given two simulation conventions* $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ *and* $\mathbb{R}_B : B_1 \Leftrightarrow B_2$*, and given the transition systems* $L_1 : A_1 \twoheadrightarrow B_1 = \langle S_1, \to_1, D_1, I_1, X_1, Y_1, F_1 \rangle$ *and* $L_2 : A_2 \twoheadrightarrow B_2 = \langle S_2, \to_2, D_2, I_2, X_2, Y_2, F_2 \rangle$*, a* forward simulation *between* $L_1$ *and* $L_2$ *consists of a relation* $R \in \mathcal{R}_{W_B}(S_1, S_2)$ *satisfying the properties shown in Fig. 8.4 and Fig. 8.5. In addition, the domains of* $L_1$ *and* $L_2$ *must satisfy:*

$$(\lambda q_1 \,.\, (q_1 \in D_1))\ [\Vdash \mathbb{R}_B^\circ \to \Leftrightarrow]\ (\lambda q_2 \,.\, (q_2 \in D_2))$$

*We will write* $L_1 \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$*.*

**Horizontal composition** The horizontal composition operator described by Def. 26 preserves simulations. Roughly speaking, whenever new component instances are created by cross-component calls, the simulation property for the new components can be stitched in-between the two halves of the callers' simulation property described in Fig. 8.5.

**Theorem 5** (Horizontal composition of simulations)**.** *For a simulation convention* $\mathbb{R} : A_1 \Leftrightarrow A_2$

*and transition systems* $L_1, L_1' : A_1 \twoheadrightarrow A_1$ *and* $L_2, L_2' : A_2 \twoheadrightarrow A_2$*, the following propery holds:*

$$\frac{L_1 \leq_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2 \qquad L_1' \leq_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2'}{L_1 \oplus L_1' \leq_{\mathbb{R}\twoheadrightarrow\mathbb{R}} L_2 \oplus L_2'}$$

*Proof.* See `common/SmallstepLinking.v` in the Coq development. □

One interesting and novel aspect of the proof is the way worlds are managed. Externally, only the worlds corresponding to incoming and outgoing questions and answers are observed. Internally, the proof of Thm. 5 maintains a *stack* of worlds to relate the corresponding stack of activations in the source and target composite semantics. See also §9.7.

Horizontal composition of simulations allows us to decompose the verification of a complex program into the verification of its parts. To establish the correctness of the linked assembly program, we can then use the following result.

**Theorem 6.** *Linking* Asm *programs yields a correct implementation of horizontal composition:*

$$\forall p_1 p_2 \,.\, \mathsf{Asm}(p_1) \oplus \mathsf{Asm}(p_2) \leq_{\mathsf{id}\twoheadrightarrow\mathsf{id}} \mathsf{Asm}(p_1 + p_2)$$

*Proof.* See `x86/AsmLinking.v` in the Coq development. □

**Vertical composition** Simulations also compose *vertically*, combining the simulation properties for successive compilation passes into a single one. The convention used by the resulting simulation can be described as follows.

**Definition 28** (Composition of simulation conventions)**.** *The composition of two Kripke relations* $R \in \mathcal{R}_{W_R}(X, Y)$ *and* $S \in \mathcal{R}_{W_S}(Y, Z)$ *is the Kripke relation* $R \cdot S \in \mathcal{R}_{W_R \times W_S}(A, C)$ *defined by:*

$$(w_R, w_S) \Vdash x\ [R \cdot S]\ z \Leftrightarrow \exists y \in Y \,.\, w_R \Vdash x\ R\ y \wedge w_S \Vdash y\ R\ z\,.$$

*Then for the simulation conventions* $\mathbb{R} : A \Leftrightarrow B$ *and* $\mathbb{S} : B \Leftrightarrow C$*, we define* $\mathbb{R} \cdot \mathbb{S} : A \Leftrightarrow C$ *as:*

$$\mathbb{R} \cdot \mathbb{S} := \langle W_{\mathbb{R}} \times W_{\mathbb{S}},\ \mathbb{R}^\circ \cdot \mathbb{S}^\circ,\ \mathbb{R}^\bullet \cdot \mathbb{S}^\bullet \rangle$$

**Theorem 7.** *Simulations compose vertically in the way depicted in Fig. 8.2.*

*Proof.* Visually speaking, the diagrams shown in Figs. 8.4 and 8.5 can be pasted vertically. For details, see the Coq proofs `identity_forward_simulation` and `compose_forward_simulations` in the file `common/Smallstep.v`. □

## 8.3 Simulation convention algebra

Now that we have described the structures we use to model and relate the execution of program components, we shift our attention to the techniques we use to derive a correctness statement for the whole compiler from the correctness properties of each pass.

### 8.3.1 Refinement of simulation conventions

As discussed in §8.1.4, the composite simulation conventions obtained when we vertically compose the passes of CompCertO are not immediately satisfactory. In the remainder of this section, we introduce the algebraic infrastructure we use to rewrite them into an acceptable form, centered around a notion of refinement for simulation conventions.

A refinement $\mathbb{R} \sqsubseteq \mathbb{S}$ captures the idea that the convention $\mathbb{S}$ is more general than $\mathbb{R}$, so that any simulation accepting $\mathbb{S}$ as its incoming convention can accept $\mathbb{R}$ as well. The shape of the symbol illustrates its meaning: related questions of $\mathbb{R}$ can be transported to related question of $\mathbb{S}$; when we get a response, the related answers of $\mathbb{S}$ can be transported back to related answers of $\mathbb{R}$.

**Definition 29** (Simulation convention refinement)**.** *Given two simulation conventions* $\mathbb{R}, \mathbb{S} : A_1 \Leftrightarrow A_2$*, we say that* $\mathbb{R}$ *refines* $\mathbb{S}$ *and write* $\mathbb{R} \sqsubseteq \mathbb{S}$ *when the following holds:*

$$\forall w\, m_1\, m_2 \,.\, w \Vdash m_1\ R^\circ\ m_2 \Rightarrow \exists\, v \,.\, (v \Vdash m_1\ S^\circ\ m_2 \wedge$$
$$\forall\, n_1\, n_2 \,.\, v \Vdash n_1\ S^\bullet\ n_2 \Rightarrow w \Vdash n_1\ R^\bullet\ n_2)\,.$$

*We write* $\mathbb{R} \equiv \mathbb{S}$ *when both* $\mathbb{R} \sqsubseteq \mathbb{S}$ *and* $\mathbb{S} \sqsubseteq \mathbb{R}$*.*

**Theorem 8.** *For* $\mathbb{R} : A_1 \Leftrightarrow A_2$*,* $\mathbb{S} : A_2 \Leftrightarrow A_3$*,* $\mathbb{T} : A_3 \Leftrightarrow A_4$*, the following properties hold:*

$$(\cdot) :: \sqsubseteq \times \sqsubseteq \to \sqsubseteq \qquad (\mathbb{R} \cdot \mathbb{S}) \cdot \mathbb{T} \equiv \mathbb{R} \cdot (\mathbb{S} \cdot \mathbb{T}) \qquad \mathbb{R} \cdot \mathsf{id} \equiv \mathsf{id} \cdot \mathbb{R} \equiv \mathbb{R}$$

*In addition, when* $\mathbb{R} \sqsubseteq \mathbb{R}' : A_1 \Leftrightarrow A_2$ *and* $\mathbb{S}' \sqsubseteq \mathbb{S} : B_1 \Leftrightarrow B_2$*, for all* $L_1 : A_1 \twoheadrightarrow B_1$ *and* $L_2 : A_2 \twoheadrightarrow B_2$*:*

$$L_1 \leq_{\mathbb{R} \to \mathbb{S}} L_2 \;\Rightarrow\; L_1 \leq_{\mathbb{R}' \to \mathbb{S}'} L_2 \,.$$

*Proof.* See `open_fsim_ccref` in `common/CallconvAlgebra.v`. □

In CompCert, the passes Cshmgen, Renumber, Linearize, CleanupLabels and Debugvar restrict the source and target memory states and values to be identical. Their simulation proofs require very few changes and can be assigned the convention id $\twoheadrightarrow$ id (see Table 8.3). The properties above ensure that these passes have no impact on the overall simulation convention of CompCertO.

### 8.3.2 Kleene algebra

Given a collection of simulation conventions, it is possible to combine them by allowing questions to be related by any one of them. This is the additive operation of our Kleene algebra. The Kleene star combines all possible finite iterations of a simulation convention.

This construction is key in our treatment of asymmetric injection passes (injp $\twoheadrightarrow$ inj). The details of inj and injp are discussed in §9, but are not an essential part of the technique. Schematically, by pre- and post-composing injection passes with self-simulations of types $\mathsf{injp}^* \twoheadrightarrow \mathsf{injp}^*$ and $\mathsf{inj} \twoheadrightarrow \mathsf{inj}$, we obtain the simulation conventions:

$$\mathsf{injp}^* \cdot \mathsf{injp} \cdot \mathsf{inj} \twoheadrightarrow \mathsf{injp}^* \cdot \mathsf{inj} \cdot \mathsf{inj}$$

The property $\mathsf{injp}^* \cdot \mathsf{injp} \sqsubseteq \mathsf{injp}^*$ on the left-hand side, and the idempotency of inj on the right-hand side (Thm. 13) allow us to rewrite the above into a symmetric simulation convention.

**Definition 30.** *Consider* $(\mathbb{R}_i)_{i \in I}$ *a family of conventions with* $\mathbb{R}_i = \langle W_i, R_i^\circ, R_i^\bullet \rangle : A_1 \Leftrightarrow A_2$ *for all* $i \in I$*. We define the simulation convention* $\sum_{i \in I} \mathbb{R}_i := \langle W, R^\circ, R^\bullet \rangle$*, where:*

$$W := \sum_{i \in I} W_i \qquad \begin{aligned} (i, w) \Vdash R^\circ &:= w \Vdash R_i^\circ \\ (i, w) \Vdash R^\bullet &:= w \Vdash R_i^\bullet \,. \end{aligned}$$

*We will write* $\mathbb{R}_1 + \cdots + \mathbb{R}_n$ *for the finitary case* $\sum_{i=1}^n \mathbb{R}_i$*. Then for* $\mathbb{R} : A_1 \Leftrightarrow A_2$*, we define*

| Language/Pass | Outgoing ↠ Incoming | SLOC | See also |
|---|---|---|---|
| **Clight** | $\mathcal{C} \twoheadrightarrow \mathcal{C}$ | +17 (+3%) | |
| *Thm. 13* | $(\mathsf{ext} + \mathsf{injp})^* \twoheadrightarrow (\mathsf{ext} + \mathsf{injp})^*$ | | §9.8 |
| SimplLocals | $\mathsf{injp} \twoheadrightarrow \mathsf{inj}$ | -4 (-1%) | §9.6 |
| Cshmgen | $\mathsf{id} \twoheadrightarrow \mathsf{id}$ | +0 (+0%) | §8.3.1 |
| **Csharpminor** | $\mathcal{C} \twoheadrightarrow \mathcal{C}$ | +15 (+4%) | |
| Cminorgen | $\mathsf{injp} \twoheadrightarrow \mathsf{inj}$ | -21 (-2%) | §9.6 |
| **Cminor** | $\mathcal{C} \twoheadrightarrow \mathcal{C}$ | +27 (+3%) | |
| Selection | $\mathsf{wt} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{wt} \cdot \mathsf{ext}$ | +43 (+0%) | §10.2 |
| **CminorSel** | $\mathcal{C} \twoheadrightarrow \mathcal{C}$ | +15 (+3%) | |
| RTLgen | $\mathsf{ext} \twoheadrightarrow \mathsf{ext}$ | +8 (+0%) | §9.5 |
| **RTL** | $\mathcal{C} \twoheadrightarrow \mathcal{C}$ | +11 (+2%) | |
| Tailcall† | $\mathsf{ext} \twoheadrightarrow \mathsf{ext}$ | -1 (-1%) | §9.5 |
| Inlining | $\mathsf{injp} \twoheadrightarrow \mathsf{inj}$ | +58 (+3%) | §9.6 |
| Renumber | $\mathsf{id} \twoheadrightarrow \mathsf{id}$ | -14 (-7%) | §8.3.1 |
| *Thm. 13* | $\mathsf{inj} \twoheadrightarrow \mathsf{inj}$ | | §9.8 |
| Constprop† | $\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$ | -17 (-2%) | §10.3 |
| CSE† | $\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$ | +3 (+0%) | §10.3 |
| Deadcode† | $\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$ | -7 (-1%) | §10.3 |
| Allocation | $\mathsf{wt} \cdot \mathsf{alloc} \twoheadrightarrow \mathsf{wt} \cdot \mathsf{alloc}$ | +13 (+0%) | §11.1 |
| **LTL** | $\mathcal{L} \twoheadrightarrow \mathcal{L}$ | +15 (+6%) | |
| Tunneling | $\mathsf{ext} \twoheadrightarrow \mathsf{ext}$ | +2 (+0%) | §9.5 |
| Linearize | $\mathsf{id} \twoheadrightarrow \mathsf{id}$ | -15 (-3%) | §8.3.1 |
| **Linear** | $\mathcal{L} \twoheadrightarrow \mathcal{L}$ | +16 (+7%) | |
| CleanupLabels | $\mathsf{id} \twoheadrightarrow \mathsf{id}$ | -10 (-4%) | §8.3.1 |
| Debugvar | $\mathsf{id} \twoheadrightarrow \mathsf{id}$ | -12 (-3%) | §8.3.1 |
| Stacking | $\mathsf{stacking} \twoheadrightarrow \mathsf{stacking}$ | +291 (+11%) | §11.2 |
| **Mach** | $\mathcal{M} \twoheadrightarrow \mathcal{M}$ | +100 (+26%) | |
| Asmgen | $\mathsf{asmgen} \twoheadrightarrow \mathsf{asmgen}$ | +179 (+6%) | §11.3 |
| **Asm** | $\mathcal{A} \twoheadrightarrow \mathcal{A}$ | +53 (+5%) | |
| | Total: | +765 (+2%) | |

Table 8.3: Languages and passes of CompCertO. Passes are grouped by their source language. † indicates an optional optimization. The simulation conventions ext, inj, injp are explained in §9. The invariants wt, va are explained in §10. The conventions alloc, stacking, asmgen are explained in §11. To handle the asymmetry of injection passes (injp ↠ inj), self-simulation properties are inserted as pseudo-passes (Thm. 13). Significant lines of code (SLOC) measured by `coqwc`, compared to CompCert v3.6.

$\mathbb{R}^* := \sum_{n\in\mathbb{N}} \mathbb{R}^n$, *where* $\mathbb{R}^0 := \mathsf{id}$ *and* $\mathbb{R}^{n+1} := \mathbb{R}^n \cdot \mathbb{R}$.

**Theorem 9** (Kleene iteration of simulations). *The constructions* $\sqsubseteq, \cdot, +, *$ *work together as a typed Kleene algebra. Moreover, the following properties hold:*

$$\frac{\forall i \,.\, L_1 \leq_{\mathbb{R}\to\mathbb{S}_i} L_2}{L_1 \leq_{\mathbb{R}\to\sum_i \mathbb{S}_i} L_2} \qquad \frac{L \leq_{\mathbb{R}\to\mathbb{S}} L}{L \leq_{\mathbb{R}^*\to\mathbb{S}^*} L}$$

*Proof.* See `cc_star_fsim` and the preceding definitions in `common/CallconvAlgebra.v`. □

### 8.3.3 Compiler correctness

We can now present our central result. The passes of CompCertO are shown in Table 8.3. The techniques outlined above allow us to formulate a simulation convention $\mathbb{C} : \mathcal{C} \Leftrightarrow \mathcal{A}$ for the compiler, and to establish the following correcntess property.

**Theorem 10** (Compositional Correctness of CompCertO). *For a* Clight *program* $p$ *and an* Asm *program* $p'$ *such that* $\mathsf{CompCert}(p) = p'$*, the following simulation holds:*

$$\mathsf{Clight}(p) \leq_{\mathbb{C}\to\mathbb{C}} \mathsf{Asm}(p')\,,$$

*where the simulation convention* $\mathbb{C}$ *is defined as:*

$$\mathbb{C} := (\mathsf{ext} + \mathsf{injp})^* \cdot \mathsf{inj} \cdot (\mathsf{va} \cdot \mathsf{ext})^3 \cdot \mathsf{wt} \cdot \mathsf{alloc} \cdot \mathsf{ext} \cdot \mathsf{stacking} \cdot \mathsf{asmgen}\,.$$

*Proof.* Use Thm. 7 to compose the correctness proofs of the compiler passes and self-simulations shown in Table 8.3. By properties of the Kleene star, the outgoing simulation convention of each of the passes SimplLocals–Renumber can be folded into $(\mathsf{ext} + \mathsf{injp})^*$ to obtain the overall outgoing convention $\mathbb{C}$. Likewise, by properties of inj and ext their incoming simulation conventions can be folded into inj to obtain the overall incoming convention $\mathbb{C}$. For details, see `driver/Compiler.v`.

□

### 8.3.4 Compositional compilation and verification

Consider the translation units $\texttt{M1.c}, \ldots, \texttt{Mn.c}$ compiled and linked to $\texttt{M1.s} + \ldots + \texttt{Mn.s} = \texttt{M.s}$. We can use Thms. 5, 6 and 10 to establish the following separate compilation property:

$$\mathsf{Clight}(\texttt{M1.c}) \oplus \cdots \oplus \mathsf{Clight}(\texttt{Mn.c}) \leq_{\mathbb{C} \to \mathbb{C}} \mathsf{Asm}(\texttt{M.s}) \tag{8.3}$$

That is, the linked Asm program `M.s` faithfully implements the horizontal composition of the source modules' behaviors, following the simulation convention $\mathbb{C}$.

Additionally, suppose we wish to verify that the overall program satisfies a specification $\Sigma$, also expressed as a transition system for $\mathcal{C} \twoheadrightarrow \mathcal{C}$. We must first decompose $\Sigma$ into:

$$\Sigma \leq_{\mathsf{id} \to \mathsf{id}} \Sigma_1 \oplus \cdots \oplus \Sigma_n \,.$$

Then for each module, we prove that $\Sigma_i \leq_{\mathsf{id} \to \mathsf{id}} \mathsf{Clight}(\texttt{Mi.c})$. This can be combined with Thm. 5 and Eqn. 8.3 to establish the correctness property $\Sigma \leq_{\mathbb{C} \to \mathbb{C}} \mathsf{Asm}(\texttt{M.s})$.

Note that Eqn. 8.3 can be established as long as the property $\mathsf{Clight}(\texttt{Mi.c}) \leq_{\mathbb{C} \to \mathbb{C}} \mathsf{Asm}(\texttt{Mi.s})$ holds independently for each module. It is possible for the different `Mi.s` to be obtained by different compilers, as long as each one satisfies a correctness property following the simulation convention in Thm. 10. Indeed this is the case for versions of CompCertO obtained by enabling different optimization passes. Moreover, if some of the `Mi.s` are hand-written assembly modules satisfying $\mathcal{C}$-style specifications, then we can prove on a case-by-case basis that $\Sigma_i \leq_{\mathbb{C} \to \mathbb{C}} \mathsf{Asm}(\texttt{Mi.s})$ and proceed with the rest of the proof as before.

Using C functions from arbitrary assembly contexts is also possible, because the compiler's simulation convention $\mathbb{C}$ captures all of the guarantees provided by CompCertO and directly specifies the behavior of the compiled assembly code. A proof involving an arbitrary assembly context which invokes a function compiled by CompCertO must establish that the call is performed according to the C calling convention. Then Thm. 10 can be used to reason about the behavior of the call in terms of the semantics of the source code or a $\mathcal{C}$ specification that it satisfies.

# Chapter 9

# Logical relations for CompCert

In the next few chapters, we examine the correctness properties and simulation conventions at the level of individual passes. In this chapter, we focus on simulations which use ext, inj and injp.

The questions, answers and states used to describe the semantics of CompCert languages all contain a memory state and surrounding runtime values. Likewise, simulation conventions and relations are constructed around *memory transformations* and relate the surrounding components in ways that are compatible with the chosen memory transformation.

## 9.1 Memory extensions

For passes where strict equality is too restrictive, but the source and target programs use similar memory layouts, CompCert uses the *memory extension* relation, which allows the values stored in the target memory state to refine the values stored in the source memory at the same location.

By analogy with the value refinement relation $v_1 \leq_{\mathsf{v}} v_2$ introduced in §7.1.3, we write $m_1 \leq_{\mathsf{m}} m_2$ to signify that the source memory $m_1$ is extended by the target memory $m_2$. Together, the relations $\leq_{\mathsf{v}}$ and $\leq_{\mathsf{m}}$ constitute a logical relation for the memory model, in the sense that loads from memory states related by extension yield values related by refinement, writing values related by refinement preserves memory extension, and similarly for the remaining memory operations.

## 9.2 Memory injections

The most complex simulation relations of CompCert allow memory blocks to be dropped, added, or mapped at a given offset within a larger block. These transformations of the memory structure are specified by partial functions of type:

$$\mathsf{meminj} := \mathsf{block} \rightharpoonup \mathsf{block} \times \mathbb{Z}$$

We will call $f \in \mathsf{meminj}$ an *injection mapping*. An entry $f(b) = (b', o)$ means that the source memory block with identifier $b$ is mapped into the target block $b'$ at offset $o$.

As with refinement and extension, an injection mapping determines both a relation on values and a relation on memory states, which work together as a logical relation for the CompCert memory model. The relation $f \Vdash v_1 \hookrightarrow_{\mathsf{v}} v_2$ allows $v_2$ to refine $v_1$, but also requires any pointer present in $v_1$ to be transformed according to $f$. The relation $f \Vdash m_1 \hookrightarrow_{\mathsf{m}} m_2$ requires that the corresponding addresses of $m_1$ and $m_2$ hold values that are related by $f \Vdash \hookrightarrow_{\mathsf{v}}$.

Note that corresponding memory allocations in the source and target states cause $f$ to evolve into a more defined injection mapping $f \subseteq f'$ relating the two newly allocated blocks. To handle this, we introduce the following constructions.

## 9.3 Modal Kripke relators

We defined general Kripke relations in §??. We add structure to sets of Kripke worlds, specifying how they can evolve.

**Definition 31.** *A* Kripke frame *is a tuple* $\langle W, \rightsquigarrow \rangle$*, where* $W$ *is a set of* possible worlds *and* $\rightsquigarrow$ *is a binary* accessibility relation *over* $W$*. Then the Kripke relator* $\Diamond$ *is defined by:*

$$w \Vdash x \; [\Diamond R] \; y \;\Leftrightarrow\; \exists\, w' \,.\, w \rightsquigarrow w' \wedge w' \Vdash x \; R \; y$$

**Example 10** (Injection simulation diagrams)**.** *Building on Ex. 2, consider once again the simple transition systems* $\alpha : A \to \mathcal{P}(A)$ *and* $\beta : B \to \mathcal{P}(B)$*. An injection-based simulation relation*

*between them will be a Kripke relation $R \in \mathcal{R}_{\mathsf{meminj}}(A, B)$ satisfying the property:*

$$\begin{array}{ccc} s_1 & \xrightarrow{\alpha} & s_1' \\ {\scriptstyle f \Vdash R}\big| & & \vdots\,{\scriptstyle f' \Vdash R} \quad (f \subseteq f') \\ s_2 & \overset{\beta}{\dashrightarrow} & s_2' \end{array} \qquad \begin{array}{l} \forall f\ s_1\ s_2\ s_1' \,.\, f \Vdash s_1\ R\ s_2 \wedge \alpha(s_1) \ni s_1' \Rightarrow \\ \qquad \exists f'\ s_2' \,.\, f \subseteq f' \wedge \beta(s_2) \ni s_2' \wedge f' \Vdash s_1'\ R\ s_2' \end{array} \tag{9.1}$$

*The new states may be related according to a new injection mapping $f'$, but in order to preserve existing relationships between any surrounding source and target pointers, the new mapping must include the original one ($f \subseteq f'$). This pattern is very common in CompCert and appears in a variety of contexts. By using $\langle \mathsf{meminj}, \subseteq \rangle$ as a Kripke frame, we can express (9.1) as:*

$$\alpha\ [\Vdash\ R \to \mathcal{P}^{\leq}(\Diamond R)]\ \beta\,.$$

## 9.4 CompCert Kripke logical relations

We can now formalize the idea that extensions and injections constitute logical relations for the CompCert memory model.

**Definition 32** (CompCert Kripke Logical Relation)**.** *For a tuple $R = (W, \rightsquigarrow, f, R^{\mathsf{mem}})$, where $\langle W, \rightsquigarrow \rangle$ is a Kripke frame, $f : W \to \mathsf{meminj}$ associates an injection mapping to each world, and where $R^{\mathsf{mem}} \in \mathcal{R}_W(\mathsf{mem})$ is a Kripke relation on memory states, the Kripke relations $R^{\mathsf{ptr}} \in \mathcal{R}_W(\mathsf{ptr})$ and $R^{\mathsf{ptrrange}} \in \mathcal{R}_W(\mathsf{ptrrange})$ are defined by the rules:*

$$\frac{f_w(b) = (b', \delta)}{w \Vdash (b, o)\ R^{\mathsf{ptr}}\ (b', o + \delta)} \qquad \frac{w \Vdash (b_1, l_1)\ R^{\mathsf{ptr}}\ (b_2, l_2) \qquad h_1 - l_1 = h_2 - l_2}{w \Vdash (b_1, l_1, h_1)\ R^{\mathsf{ptrrange}}\ (b_2, l_2, h_2)}$$

*and $R^{\mathsf{val}} \in \mathcal{R}_W(\mathsf{val})$ is the smallest Kripke relation satisfying:*

$$\forall\, v \in \mathsf{val}\,.\ \Vdash \mathsf{undef}\ R^{\mathsf{val}}\ v \qquad \mathsf{vptr} :: \Vdash R^{\mathsf{ptr}} \to R^{\mathsf{val}}$$

$$\mathsf{int}, \mathsf{long}, \mathsf{float}, \mathsf{single} :: \Vdash = \to R^{\mathsf{val}}\,.$$

*We say that $R$ is a* CompCert Kripke logical relation *if the properties shown in Fig. 9.1 are satisfied.*

$$\rightsquigarrow \text{ is reflexive and transitive}$$
$$w \rightsquigarrow w' \Rightarrow f(w) \subseteq f(w')$$

$$\begin{aligned}
\mathsf{alloc} &:: \Vdash R^{\mathsf{mem}} \to = \to = \to \Diamond(R^{\mathsf{mem}} \times R^{\mathsf{block}}) \\
\mathsf{free} &:: \Vdash R^{\mathsf{mem}} \to R^{\mathsf{ptrrange}} \to \mathsf{option}^{\leq}(\Diamond R^{\mathsf{mem}}) \\
\mathsf{load} &:: \Vdash R^{\mathsf{mem}} \to R^{\mathsf{ptr}} \to \mathsf{option}^{\leq}(R^{\mathsf{val}}) \\
\mathsf{store} &:: \Vdash R^{\mathsf{mem}} \to R^{\mathsf{ptr}} \to R^{\mathsf{val}} \to \mathsf{option}^{\leq}(\Diamond R^{\mathsf{mem}})
\end{aligned}$$

Figure 9.1: Defining properties of CKLRs. Note the correspondance with the types of operations in Fig. 7.2.

**Rationale** The relation $R^{\mathsf{mem}}$ is given. We expect $R^{\mathsf{ptr}}$ to map each source pointer to at most one target pointer and to be shift-invariant in the following sense:

$$\frac{w \Vdash (b_1, o_1)\ R^{\mathsf{ptr}}\ (b_2, o_2)}{w \Vdash (b_1, o_1 + \delta)\ R^{\mathsf{ptr}}\ (b_2, o_2 + \delta)}$$

Any such relation can be uniquely specified by the injection mapping $f$. We expect the other relations to be consistent with $R^{\mathsf{ptr}}$ and $\mathsf{undef}$ to act as a least element for $R^{\mathsf{val}}$, which determines them completely.

Note that $R^{\mathsf{mem}}$ is the central component driving world transitions, as witnessed by the uses of $\Diamond$ in Fig. 9.1. The surrounding relations are monotonic in $w$, so that any extra state constructed from pointers and runtime values will be able to "follow along" when world transitions occur.

**Theorem 11.** *Extensions and injections correspond to the CompCert Kripke logical relations:*

$$\mathsf{ext} := \langle \{*\},\ \{(*,*)\},\ * \mapsto (b \mapsto (b, 0)),\ \leq_{\mathsf{m}} \rangle \qquad \mathsf{inj} := \langle \mathsf{meminj}, \subseteq, f \mapsto f, \hookrightarrow_{\mathsf{m}} \rangle$$

*Proof.* The correspondance between $R^{\mathsf{val}}_{\mathsf{inj}}$ and $\hookrightarrow_{\mathsf{v}}$ is easily verified, as is the correspondance between $* \Vdash R^{\mathsf{val}}_{\mathsf{ext}}$ and $\leq_{\mathsf{v}}$. The properties of Fig. 9.1 reduce to well-known properties of the memory model already proven in CompCert. See `cklr/Extends.v` and `cklr/Inject.v` for details. □

## 9.5 From CKLRs to simulation conventions

In simulations, the accessibility relation allows us to update the world after each step in the program's execution. Transitivity allows us to combine multiple steps in one:

$$\overset{\curvearrowleft}{q} \cdot \overset{\curvearrowleft}{s_1} \cdot \overset{\curvearrowleft}{s_2} \cdots \overset{\curvearrowleft}{s_k} \cdot r \quad \Rightarrow \quad \overbrace{q \cdot s_1 \cdot s_2 \cdots s_k \cdot r}^{\curvearrowleft}$$

In our approach to simulation conventions, the accessibility relation is not part of the interface of simulations. Instead, a single world is used to formulate the 4-way relationship between pairs of questions and answers. As shown below, in the case of simulation conventions based on CKLRs, this relation does involve the accessibility relations which CKLRs introduce.

Given a specific language interface $\mathcal{X}$, the components of any CKLR $R = \langle W, \rightsquigarrow, f, R^{\mathsf{mem}} \rangle$ can be used to construct a simulation relation $R_{\mathcal{X}} : \mathcal{X} \Leftrightarrow \mathcal{X}$. For instance:

$$R_{\mathcal{C}} := \langle W,\ (R^{\mathsf{val}} \times = \times \vec{R}^{\mathsf{val}} \times R^{\mathsf{mem}}),\ \Diamond(R^{\mathsf{val}} \times R^{\mathsf{mem}}) \rangle\,.$$

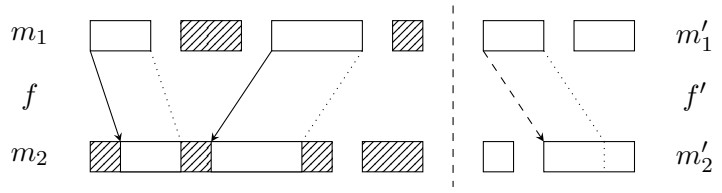
We will often implicitly promote $R$ to $R_{\mathcal{C}}$. Furthermore, since the semantics of CompCert languages are built out of the operations of the memory model, they are well-behaved with respect to CKLRs and we can prove the following parametricity theorems.

**Theorem 12** (Relational parametricity of Clight and RTL)**.** *For all programs $p$ and CKLR $R$:*

$$\mathsf{Clight}(p) \leq_{R \twoheadrightarrow R} \mathsf{Clight}(p) \qquad \mathsf{RTL}(p) \leq_{R \twoheadrightarrow R} \mathsf{RTL}(p)$$

*Proof.* See `cklr/*rel.v` in the Coq development, in particular `Clightrel.v` and `RTLrel.v`. □

The passes of CompCert which use memory extensions do not feature complex invariants which must be preserved at call sites; it is enough for external calls to preserve the memory extension. Consequently, they are not much more difficult to update than identity passes, and can be assigned the type ext $\twoheadrightarrow$ ext. By contrast, injection passes are trickier to handle.




$$(f, m_1, m_2) \rightsquigarrow_{\mathsf{injp}} (f', m'_1, m'_2)$$

Figure 9.2: External calls and memory injections. The source and target memory states are depicted at the top and bottom of the figure. Arrows describe the injection mapping. The memory block on the left of the dashed line are present at the beginning of the call. Memory blocks on the right are allocated during the call, adding a new entry to the injection mapping. The shaded areas must not be modified by the call.

## 9.6 External calls in injection passes

Passes which alter the block structure of the memory use *memory injections* (§9.2). The convention inj can be used for incoming calls, but it is insufficient for outgoing calls.

Consider the SimplLocals pass, which removes some local variables from the memory. The corresponding values are instead stored as temporaries in the target function's local environment, and the correspondance between the two is enforced by the simulation relation. To maintain it, we need to know that external calls do not modify the corresponding source memory blocks.

More generally, as depicted in Fig. 9.2, injection passes expect external calls to leave regions outside of the injection's footprint untouched. This expectation is reasonable because external calls should behave uniformly between the source and target executions. These requirements can be formalized as the following CompCert Kripke logical relation:

$$\mathsf{injp} := \langle \mathsf{meminj} \times \mathsf{mem} \times \mathsf{mem},\ \rightsquigarrow_{\mathsf{injp}},\ \pi_1,\ R^{\mathsf{mem}}_{\mathsf{injp}} \rangle \qquad \frac{f \Vdash m_1 \hookrightarrow_{\mathsf{m}} m_2}{(f, m_1, m_2) \Vdash m_1\ R^{\mathsf{mem}}_{\mathsf{injp}}\ m_2}$$

where $(f, m_1, m_2) \rightsquigarrow_{\mathsf{injp}} (f', m'_1, m'_2)$ ensures that $f \subseteq f'$ and that the memory states satisfy the constraints in Fig. 9.2 (for details, see `cc_injp` in `common/LanguageInterface.v`).

## 9.7 Discussion: world transitions and compositionality

The injp convention illustrates a key novelty in the granularity at which we deploy Kripke worlds. In previous work, Kripke worlds are usually assumed to evolve linearly with the execution. Writing

$s_i$ for internal states, this can be depicted as:

$$q \cdot s_1 \cdot s_2 \cdot m \cdot s'_1 \cdot s'_2 \cdot n \cdot s_3 \cdots s_k \cdot r$$

To enable horizontal compositionality, the challenge is then to construct worlds, accessibility relations, and simulation relations which are sophisticated enough to express ownership constraints like the ones discussed in §9.6, which evolve and shift as the execution switches between components.

In our open simulations, worlds can be deployed independently for incoming and outgoing calls, in a way which follows the structure of plays, as depicted here:

$$q \cdot m_1 \cdot n_1 \cdots m_k \cdot n_k \cdot r$$

Internal steps are not part of a component's observable behavior, and individual simulation proofs are free to use any simulation relation establishing the simulation convention at interaction sites.

Two examples illustrate this flexibility. First, as explained in §8.2.2, to handle nested cross-component calls, composite simulations use an internal stack of worlds. A situation where $m_1$ and $m_2$ are cross-component calls and $m_3$ is an external call can be described as:

$$q \cdot m_1 \cdot m_2 \cdot m_3 \cdot n_3 \cdot n_2 \cdot n_1 \cdot r$$

Second, in simulations which use CKLRs, the simulation relation can simply be qualified as $w_B \Vdash \Diamond R$ to allow the world to evolve as the execution progresses. The corresponding shape is:

$$q \cdot s_1 \cdot s_2 \cdot s_3 \cdots s_k \cdot r$$

Since $\Diamond\Diamond R = \Diamond R$, per-step world transitions can easily be folded into the overall constraint. Moreover, this allows steps which *do not* individually conform to world transitions ($\Vdash R \to \Diamond R$), but *do* maintain $\Diamond R$ with respect to the initial world ($w_B \Vdash \Diamond R \to \Diamond R$).

For instance, the simulation convention of the Stacking pass is based on $\mathsf{injp} \twoheadrightarrow \mathsf{injp}$. Stacking stores the contents of some temporaries used by the source program into *spilling locations* of the

target in-memory stack frames. To prove correctness, we must ensure that spilling locations are only accessed as intended, by enforcing their separation from the injected source memory. This property is maintained by $\rightsquigarrow_{\mathsf{injp}}$, which most internal steps and external calls conform to. On the other hand, internal steps which *do* access spilling locations in the expected way do not conform to $\rightsquigarrow_{\mathsf{injp}}$ at a granular level. However, since the stack frame is a *new* memory block allocated after a function is called, these steps do maintain $\rightsquigarrow_{\mathsf{injp}}$ with repect to the initial world. This allows us to encode CompCert's original and "instantaneous" assumptions about external calls directly, and existing simulation proofs relying on them can be updated with almost no changes.

Combined together, the two examples above are sufficient to express ownership constraints which require sophisticated permission maps in other approaches, by using conditions already present in CompCert.

## 9.8 Properties

Finally, we state some properties which are used to derive Thm. 10.

**Theorem 13.** *For all* Clight *and* RTL *programs:*

$$\forall\, p\,.\, \mathsf{Clight}(p) \leq_{(\mathsf{ext}+\mathsf{injp})^* \to (\mathsf{ext}+\mathsf{injp})^*} \mathsf{Clight}(p)$$

$$\forall\, p\,.\, \mathsf{RTL}(p) \leq_{\mathsf{inj} \to \mathsf{inj}} \mathsf{RTL}(p)$$

*In addition, the simulation relations derived from* ext *and* inj *compose in the following way:*

$$\mathsf{ext} \cdot \mathsf{inj} \equiv \mathsf{inj} \cdot \mathsf{ext} \equiv \mathsf{inj} \cdot \mathsf{inj} \equiv \mathsf{inj}\,.$$

*Proof.* The first statement is derived from Thms. 9 and 12; see `driver/Compiler.v`. For the second statement, see `ext_inj`, `inj_ext`, `inj_inj` found under `cklr/`. □

# Chapter 10

# Invariants

Several passes of CompCert rely on the preservation of invariants by their source program (this situation is illustrated in more detail in the supplementary appendix): when the semantics of a language preserves an invariant, the preservation properties can assist in proving forward simulations which use the language as their source. This allows us to decompose the simulation proof, and in the case of RTL the preservation proofs can be reused for multiple passes.

In CompCert, this technique is deployed in an ad-hoc manner: for each pass using an invariant, the simulation relation is strengthened to assert that the invariant holds on the source state, and the preservation properties for the source language are used explicitly in the simulation proof to maintain this invariant. In CompCertO, this becomes more involved, because the simulation convention must be altered to ensure that invariants are preserved by external calls.

On the other hand, our simulation infrastructure offers the opportunity to capture and reason about invariants explicitly, and to further decouple preservation and simulation proofs. In this section, we give an overview of our treatment of invariants. For details, see the supplementary appendix and `common/Invariant.v` in the Coq development.

## 10.1 Invariants and language interfaces

First, we define a sort of "invariant convention", which describes how a given invariant impacts the questions and answers of the language under consideration.

**Definition 33.** *An* invariant for a language interface $A$ *is a tuple* $\mathbb{P} = \langle W, \mathbb{P}^\circ, \mathbb{P}^\bullet \rangle$, *where* $W$ *is a*

*set of worlds and* $\mathbb{P}^\circ, \mathbb{P}^\bullet$ *are families of predicates on* $A^\circ, A^\bullet$ *indexed by* $W$*.*

**Example 11.** *Typing constraints for the language interface* $\mathcal{C}$ *can be expressed as the invariant:*

$$\mathsf{wt} := \langle \mathsf{sig}, \mathbb{P}^\circ_{\mathsf{wt}}, \mathbb{P}^\bullet_{\mathsf{wt}} \rangle \qquad \frac{\vec{v} <: sg.\mathsf{args}}{sg \Vdash vf[sg](\vec{v})@m \in \mathbb{P}^\circ_{\mathsf{wt}}} \qquad \frac{v' <: sg.\mathsf{res}}{sg \Vdash v'@m' \in \mathbb{P}^\bullet_{\mathsf{wt}}}$$

*The proposition* $\vec{v} <: sg.\mathsf{args}$ *asserts that the types of the arguments* $\vec{v}$ *match those specified by the signature* $sg$*. The proposition* $v' <: sg.\mathsf{res}$ *asserts a similar property for the return value* $v'$*.*

Invariants can be seen as a special case of simulation convention which constrain the source and target questions and answers to be equal. This can be formalized as follows.

**Definition 34** (Simulation conventions for invariants)**.** *A* $W$*-indexed predicate* $P$ *on a set* $X$ *can be promoted to a Kripke relation* $\hat{P} \in \mathcal{R}_W(X, X)$ *defined by the rule:*

$$\frac{w \Vdash x \in P}{w \Vdash x \ \hat{P} \ x}$$

*Then an invariant* $\mathbb{P} = \langle W, \mathbb{P}^\circ, \mathbb{P}^\bullet \rangle$ *can be promoted to a simulation convention:* $\hat{\mathbb{P}} := \langle W, \hat{\mathbb{P}}^\circ, \hat{\mathbb{P}}^\bullet \rangle$*.*

## 10.2 Typing invariants

The typing invariant described in Ex. 11 is used by the Selection and Allocation passes. We have updated their correctness proofs as well as the preservation proofs in `Cminortyping.v` and `RTLtyping.v` to use our framework.

The invariant wt satisfies one key property: when a simulation convention $\mathbb{R}$ consists of a sequence of CKLRs and other invariants, the following property holds:

$$\mathsf{wt} \cdot \mathbb{R} \cdot \mathsf{wt} \equiv \mathbb{R} \cdot \mathsf{wt}$$

This means CompCertO's overall simulation convention can eliminate the typing invariant for the Selection pass, retaining only that used for Allocation. In turn, this facilitates the simplification of the convention for the passes from Clight to Inlining.

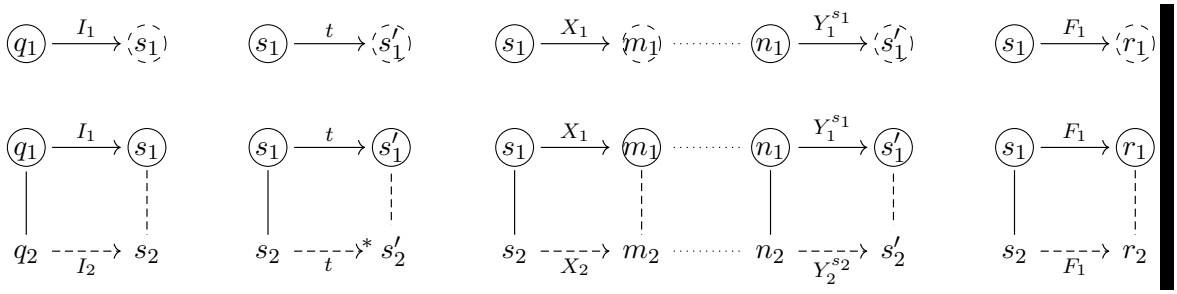



Figure 10.1: Simulation with invariants. Circles indicate questions, answers and states which satisfy the appropriate invariants. When the transition system $L_1$ preserves the invariants in the way shown in the top row, a simulation of $L_1$ by $L_2$ can be established through the weakened diagrams shown in the bottom row. The resulting simulation uses the convention $\mathbb{P}_A \cdot \mathbb{R}_A \twoheadrightarrow \mathbb{P}_B \cdot \mathbb{R}_B$, ensuring that the environment establishes and preserves the appropriate invariants on questions and answers. The simulation relation $P \cdot R$ then ensures that the strengthened assumptions used by the weakened simulation diagrams can be satisfied.

## 10.3 Value analysis

The passes Constprop, CSE and Deadcode use CompCert's value analysis framework. Abstract interpretation is performed on their source program, and the resulting information is used to carry out the corresponding optimizations. The correctness proofs for these passes then rely on the invariant va, which asserts that the concrete runtime states satisfy the constraints encoded in the corresponding abstract states.

We have updated the value analysis framework and associated pass correctness proofs to fit the invariant infrastructure described in this section. Value analysis passes use the convention $\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$. Unfortunately, because it combines constraints with mixed variance, the invariant va does not propagate in the same way as wt, so the compiler's simulation convention must retain the component $(\mathsf{va} \cdot \mathsf{ext})^3$ as-is. When some of the corresponding optimization passes are disabled, we use self-simulations of the RTL language to match this convention nonetheless.

## 10.4 Simulations modulo invariants

The top row in Fig. 10.1 illustrates the preservation of invariants by transition systems. In the context of a transition system $L_1 : A_1 \to B_1$, we consider three invariants working together:

- an invariant $\mathbb{P}_A$ for the language interface $A$;

- an invariant $\mathbb{P}_B$ for the language interface $B$;
- a $W_B$-indexed predicate $P$ on the states of $L_1$.

The preservation of $\langle \mathbb{P}_A, \mathbb{P}_B, P \rangle$ is then analogous to a unary simulation property, where $\mathbb{P}_A \twoheadrightarrow \mathbb{P}_B$ play the roles of the simulation conventions, and $P$ plays the role of the simulation relation. In fact, when $L_1$ preserves these invariants, the following property holds:

$$L_1 \leq_{\hat{\mathbb{P}}_A \twoheadrightarrow \hat{\mathbb{P}}_B} L_1$$

Once we have established that the source language preserves the invariants, we wish to use this fact to help prove the forward simulation for a given pass. To this end, we define a *strengthened* transition system $L_1^{\mathbb{P}} : A_1 \twoheadrightarrow B_1$, with the property that $L_1 \leq_{\hat{\mathbb{P}}_A \twoheadrightarrow \hat{\mathbb{P}}_B} L_1^{\mathbb{P}}$. For a target transition system $L_2 : A_2 \twoheadrightarrow B_2$, it then suffices to show that $L_1^{\mathbb{P}} \leq_{\mathbb{R}_A \twoheadrightarrow \mathbb{R}_B} L_2$ to establish:

$$L_1 \leq_{\hat{\mathbb{P}}_A \cdot \mathbb{R}_A \twoheadrightarrow \hat{\mathbb{P}}_B \cdot \mathbb{R}_B} L_2 \,.$$

Simulations from $L_1^{\mathbb{P}}$ are easier to prove, because $L_1^{\mathbb{P}}$ provides assumption that the invariants hold on all source questions, answers and states. The simulation diagrams reduce to those shown in the bottom row of Fig. 10.1. However, since they are formulated in terms of Def. 27, the standard forward simulation techniques defined by CompCert in `Smallstep.v` remain available.

# Chapter 11

# Specialized simulation conventions

For the Alloc, Stacking and Asmgen passes, we construct more specific simulation conventions which express the correspondance between the higher-level and lower-level representations of function calls and returns. These passes use identical conventions for incoming and external calls.

## 11.1 The Allocation pass

The Allocation pass from RTL to LTL is the first pass to modify the interface of function calls. LTL uses *abstract locations* which represent the stack slots and machine registers eventually used in the target assembly program. Abstract locations contain arguments, temporaries and return values. The contents are stored in a *location map*, passed across components by the interface $\mathcal{L}$ alongside memory states. The compiler also expects the values of abstract locations designated as *callee-save* to be preserved by function calls.

To express the simulation convention used by the Allocation pass, we will use the following notations. For a signature $sg$ and a location map $ls$, we write $\mathsf{args}(sg, ls)$ to represent the argument values stored in $ls$. Likewise, $\mathsf{retval}(sg, ls)$ extracts the contents of locations used to store the return value. The relation $\equiv_{\mathsf{CS}}$ asserts that two location maps agree on callee-save locations.

The simulation convention $\mathsf{alloc} : \mathcal{C} \Leftrightarrow \mathcal{L}$ uses its worlds to remember the initial location map and the signature associated with a call, and can then be defined by:

$$\mathsf{alloc} := \langle \mathsf{signature} \times \mathsf{locmap},\ R^{\circ}_{\mathsf{alloc}},\ R^{\bullet}_{\mathsf{alloc}} \rangle$$

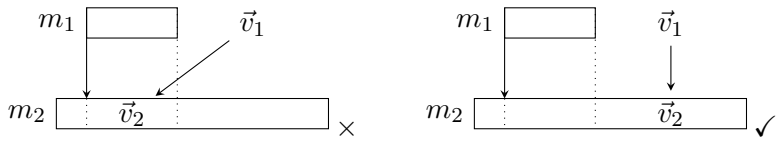



Figure 11.1: Separation of argument in the stacking simulation convention.

$$\frac{\vec{v} \leq_{\mathsf{v}} \mathsf{args}(sg, ls) \qquad m_1 \leq_{\mathsf{m}} m_2}{(sg, ls) \Vdash vf[sg](\vec{v})@m_1 \; R^{\circ}_{\mathsf{alloc}} \; vf[sg](ls)@m_2}$$

$$\frac{v' \leq_{\mathsf{v}} \mathsf{retval}(sg, ls') \qquad ls \equiv_{\mathsf{CS}} ls' \qquad m'_1 \leq_{\mathsf{m}} m'_2}{(sg, ls) \Vdash v'@m'_1 \; R^{\bullet}_{\mathsf{alloc}} \; ls'@m'_2}$$

## 11.2 The Stacking pass

The Stacking pass consolidates the information which Linear stores in abstract stack locations into in-memory stack frames. The simulation proof uses a memory injection, and involves maintaining separation properties ensuring that the source memory and the regions of stack frames introduced by Stacking occupy disjoint areas of the target memory.

With regards to the memory state, the stacking simulation convention is essentially identical to injp. Since the new regions of stack frames are outside the image of source memory, and most of them are local to function activations, the properties of injp are largely sufficient (see also §9.7).

The exception to the rule pertains to argument passing. Loads from a function's argument locations access the caller's stack frame. If the area used to store arguments overlaps with the injected source memory state, then the source program and external calls may alter them in unexpected ways. In previous CompCert extensions, sophisticated techniques were required to prevent this from happening.

In our model, we simply encode the required separation condition in the simulation convention stacking : $\mathcal{L} \Leftrightarrow \mathcal{M}$. The convention asserts that the contents of argument locations are stored into corresponding stack slots within the target memory. Additionally, it requires that the region of the target memory used to store arguments within the caller's stack must be disjoint from the injection image of the source memory (Fig. 11.1). For details, see `cc_stacking` in `backend/Mach.v`.

## 11.3 The Asmgen pass

The Asmgen pass from Mach to Asm uses a memory extension. Asm introduces explicit registers for the program counter, stack pointer and return address. The corresponding simulation convention asmgen : $\mathcal{M} \Leftrightarrow \mathcal{A}$ ensures that the appropriate components of Mach-level queries are mapped to the new registers. In addition, we must ensure that the call returns the stack pointer to its original value, and set the program counter to the return address specified by the caller.

[XXX: we don't talk about nextblock in our simplified exposition of the memory model] A more complex challenge is that the Asm language does not have a control stack, but instead executes instruction after instruction in a "flat" manner, making it difficult to distinguish final states. To address this, we keep track of the initial value $nb_0$ of the memory's nextblock counter, and use it to distinguish between inner and outer stack pointers. When $rs[\mathsf{sp}] \geq nb_0$, we interpret the ret instruction as an *internal* return and simply jump to the location pointed to by $rs[\mathsf{ra}]$. However, when $rs[\mathsf{sp}] < nb_0$, we intepret ret as a top-level return to the environment, and model its behavior as a final state rather than an internal step. In addition, the simulation convention asmgen must ensure that the initial value of the stack pointer points to a valid block of the initial memory.

This is expressed as:

$$\mathsf{asmgen} := \langle \mathsf{val} \times \mathsf{val}, R^{\circ}_{\mathsf{asmgen}}, R^{\bullet}_{\mathsf{asmgen}} \rangle$$

$$\frac{rs_1 \uplus [\mathsf{sp} := sp, \mathsf{ra} := ra, \mathsf{pc} := vf] \leq_{\mathsf{v}} rs_2 \qquad m_1 \leq_{\mathsf{m}} m_2}{(sp, ra) \Vdash vf(sp, ra, rs_1)@m_1 \; R^{\circ}_{\mathsf{asmgen}} \; rs_2@m_2}$$

$$\frac{rs'_1 \uplus [\mathsf{sp} := sp, \mathsf{pc} := ra] \leq_{\mathsf{v}} rs'_2 \qquad m'_1 \leq_{\mathsf{m}} m'_2}{(sp, ra) \Vdash rs'_1@m'_1 \; R^{\bullet}_{\mathsf{asmgen}} \; rs'_2@m'_2}$$

# Chapter 12

# The Coqrel library

# Bibliography


Samson Abramsky. 2010. From CSP to Game Semantics. In *Reflections on the Work of C.A.R. Hoare*. Springer, London, 33–45. https://doi.org/10.1007/978-1-84882-912-1_2

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470. https://doi.org/10.1006/inco.2000.2930

Samson Abramsky and Guy McCusker. 1997. Linearity, Sharing and State: A Fully Abstract Game Semantics for Idealized Algol with Active Expressions. In *Algol-like Languages*. Birkhäuser, Boston, MA, 297–329. https://doi.org/10.1007/978-1-4757-3851-3_10

Samson Abramsky and Guy McCusker. 1999. Game semantics. In *Computational logic: Proceedings of the 1997 Marktoberdorf Summer School*. Springer, Berlin, Heidelberg, 1–55. https://doi.org/10.1007/978-3-642-58622-4_1

Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. 1998. Alternating refinement relations. In *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR '98)*. Springer, Berlin, Heidelberg, 163–178. https://doi.org/10.1007/BFb0055622

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*. Springer, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Phil. Trans. R. Soc. A* 375, 2104 (2017), 20160331. https://doi.org/10.1098/rsta.2016.0331

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 165–178.

Ralph-Johan Back. 1978. *On the correctness of refinement steps in program development.* Technical Report A-1978-4. Department of Computer Science, University of Helsinky, Helsinki, Finland.

Ralph-Johan Back and Joakim von Wright. 1998. *Refinement Calculus: A Systematic Introduction.* Springer, New York. https://doi.org/10.1007/978-1-4612-1674-2

John Baez and Mike Stay. 2010. Physics, topology, logic and computation: a Rosetta Stone. In *New structures for physics*. Springer, Berlin, Heidelberg, 95–172. https://doi.org/10.1007/978-3-642-12821-9_2

Andreas Blass. 1992. A game semantics for linear logic. *Ann. Pure Appl. Log.* 56, 1–3 (1992), 183–220. https://doi.org/10.1016/0168-0072(92)90073-9

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. https://dl.acm.org/doi/10.5555/3026877.3026928

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Russell Harmer and Guy McCusker. 1999. A fully abstract game semantics for finite nondeterminism. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS '99)*. IEEE Computer Society, USA, 422–430. https://doi.org/10.1109/LICS.1999.782637

Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. 2014. Logical Relations and Parametricity - A Reynolds Programme for Category Theory and Programming Languages. *Electron. Notes Theor. Comput. Sci.* 303 (March 2014), 149–180. https://doi.org/10.1016/j.entcs.2014.02.008

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

J. M. E. Hyland and C.-H. L. Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/inco.2000.2917

Jacques-Henri Jourdan. 2016. *Verasco: a formally verified C static analyzer.* Ph.D. Dissertation.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL'16)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/2837614.2837642

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Dexter Kozen. 1998. *Typed Kleene algebra.* Technical Report TR98-1669. Cornell University. https://hdl.handle.net/1813/7323

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2012. Mechanized semantics for compiler verification. In *Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*. Springer, Berlin, Heidelberg, 386–388. https://doi.org/10.1007/978-3-642-35182-2_27

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2.* Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31.

Carroll Morgan. 1988. The specification statement. *ACM Trans. Program. Lang. Syst.* 10, 3 (July 1988), 403–419. https://doi.org/10.1145/44501.44503

Joseph M. Morris. 2004. Augmenting Types with Unbounded Demonic and Angelic Nondeterminacy. In *Proceedings of the 7th International Conference on Mathematics of Program Construction (MPC 2004)*. Springer, Berlin, Heidelberg, 274–288. https://doi.org/10.1007/978-3-540-27764-4_15

Joseph M Morris and Malcolm Tyrrell. 2008. Dually nondeterministic functions. *ACM Trans. Program. Lang. Syst.* 30, 6 (Oct. 2008), 34. https://doi.org/10.1145/1391956.1391961

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Impera-

tive Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (July 2019), 29 pages. https://doi.org/10.1145/3341689

Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2001)*. Springer, Berlin, Heidelberg, 1–24. https://doi.org/10.1007/3-540-45315-6_1

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. Springer, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Zhong Shao. 2010. Certified Software. *Commun. ACM* 53, 12 (Dec. 2010), 56–66. https://doi.org/10.1145/1859204.1859226

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 275–287. https://doi.org/10.1145/2676726.2676985

Malcolm Tyrrell, Joseph M. Morris, Andrew Butterfield, and Arthur Hughes. 2006. A Lattice-Theoretic Model for an Algebra of Communicating Sequential Processes. In *Proceedings of the Third International Colloquium on Theoretical Aspects of Computing (ICTAC 2006)*. Springer, Berlin, Heidelberg, 123–137. https://doi.org/10.1007/11921240_9

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290375

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371119