Abstract

Refinement-Based Game Semantics for Certified Components

Jérémie Koenig

2020

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 seeks 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 proposes to combine game semantics, algebraic effects and the refinement calculus to build models suitable for this task. In particular, certified abstraction layers and the certified compiler CompCert can be embedded into a single framework supporting heterogeneous components, stepwise refinement, and data abstraction.

Refinement-Based Game Semantics for Certified Components

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

Contents

List of Figures

List of Tables

Acknowledgements

I am grateful to Prof. Zhong Shao for his guidance and feedback in his role as my advisor, as well as Profs. Dana Angluin, Andrew Appel, and Ruzica Piscak for serving as members of my dissertation committee. My work also benefited from countless interactions with fellow members of the Yale FLINT group, in particular Quentin Carbonneaux, David Costanzo, Ronghui Gu, Jieung Kim, Tahina Ramananandro, Yuting Wang, Shu-Chun Weng, Pierre Wilke and Xiongnan Wu.

During my time at Yale, I have relied on the support of friends and family members including Annie Bénétreau, Anwesha Bhattacharya, Kinnari Dave, Marianne Di Tomaso, Ces Domingo Bartozzi, Mengting Gu, Judith Hoeller, Naim Goksel Karacayli, Joo Kim, Hiroko Kumaki, Jingyi Lin, Mengci Liu, Xinyue Luo, Zhenya Sakal, Mansa Saxena, Lachi Singh, Mansa Srivastav, Azusa Takeishi, Minghui Tan, Amnart Tangkiriphimarn, Sławomir Wójcik, Qing Xia, Xin Yan, Yifei Yang, Jessica Yuan, Mingshuo Zeng and many others.

Finally, it would have been impossible for me to join a PhD program without the prior support of several faculty members in the department of Mathematics and Computer Science at the University of Strasbourg. In the context of their administrative functions, Stella Marc-Zwecker and Eric Violard repeatedly allowed me to extend my enrollment as an undergraduate student, despite my complicated situation at the time. As teachers in the Master program, Nicolas Magaud, Julien Narboux and Pascal Schreck nurtured my interest in research and provided the guidance I needed to pursue it further.

The work presented in this thesis was supported in part by NSF grants 1521523, 1715154 and 1763399.

Chapter 1

Introduction

1.1 Certified systems at scale

Certified software [Shao, 2010] is software accompanied by a mechanized, machine-checkable proof 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. Then, we must use specialized tools to construct 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, with the goal of combining them—and their proofs of correctness—into larger certified systems.

1.1.1 The DeepSpec project

For example, the DeepSpec project [Appel et al., 2017] 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 promises benefits beyond the potential increase in the scale of 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 specification is only validated by 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, making it impossible to carry out this second proof.

Moreover, internal specifications used as intermediate steps in the verification of a complex system disappear from its external characterization and no longer need to be trusted. This reduces the ratio between the size of the system and the size of the trusted specification.

1.1.2 CompCert

To an extent, these principles are already demonstrated in the structure of the certified C compiler CompCert [Leroy, 2009]. CompCert can emit assembly code for PowerPC, ARM, RISC-V and x86 processors, and supports an extensive subset of ISO C99 as its source language.

Like all components used in the DeepSpec project, CompCert was specified and verified in the Coq proof assistant. As a certified compiler, CompCert includes language semantics for the source and target languages, and a proof of correctness relating the behaviors of the source and target programs. For the purposes of this work, I will consider Clight as the source language of CompCert.[1] The architecture-specific assembly language targeted by CompCert is known as Asm.

CompCert's correctness proof shows that if the compiler successfully transforms a source program p into a target program p', then the behavior of the target program *refines* that of the source program:

$$\frac{\mathsf{CompCert}(p) = p'}{\mathsf{Clight}[p] \supseteq \mathsf{Asm}[p']}$$

In the statement above, the semantics of the source and target programs are expressed as the sets of traces $\mathsf{Clight}[p]$ and $\mathsf{Asm}[p']$. Each trace records a possible execution of the corresponding program

[1] Clight is a simplified dialect of C and an early intermediate language of CompCert. The utility program `clightgen` included with CompCert can be used to convert C source code into a Coq definition of a Clight abstract syntax tree. Source-level verification tools interfaced with CompCert typically operate on this representation of the source program. Because the resulting proofs characterize the Clight code directly, `clightgen` does not need to be trusted.

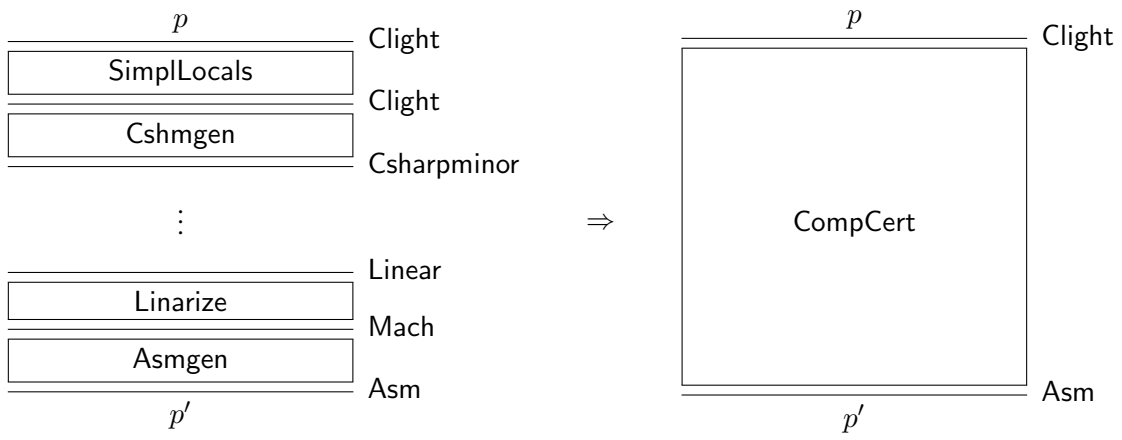


Figure 1.1: Structure of the certified compiler CompCert. The source program p is progressively transformed into the target program p' by successive compilation passes, depicted here as rectangular boxes. The horizontal line above each compilation pass depicts its source language, and the one below it depicts its target language. Two passes can be composed when the target language of the first one corresponds to the source language of the second one. See also Table 9.3.

as a sequence of events. The trace containment property $\mathsf{Clight}[p] \supseteq \mathsf{Asm}[p']$ expresses that every possible behavior of the target program p' is a possible behavior of the source program p.

The introduction of CompCert in 2008 represented a breakthrough in the scale and feasibility of certified software. The key to this achievement was the decomposition of CompCert into compilation passes, which are verified individually. When passes are composed to obtain the overall C-to-assembly compiler (Figure 1.1), their correctness proofs can be composed as well to establish the compiler's overall correctness theorem. The final theorem does not mention the intermediate programs or language semantics, so that a user only needs to trust the accuracy of the Clight and Asm semantics, and the soundness of the proof assistant.

Since its introduction, CompCert 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], and 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].

1.1.3 CertiKOS

The techniques used in CompCert also provided a blueprint for the verification of the operating system kernel CertiKOS [Gu et al., 2015, 2016, 2018] developed in the Yale FLINT group. I joined

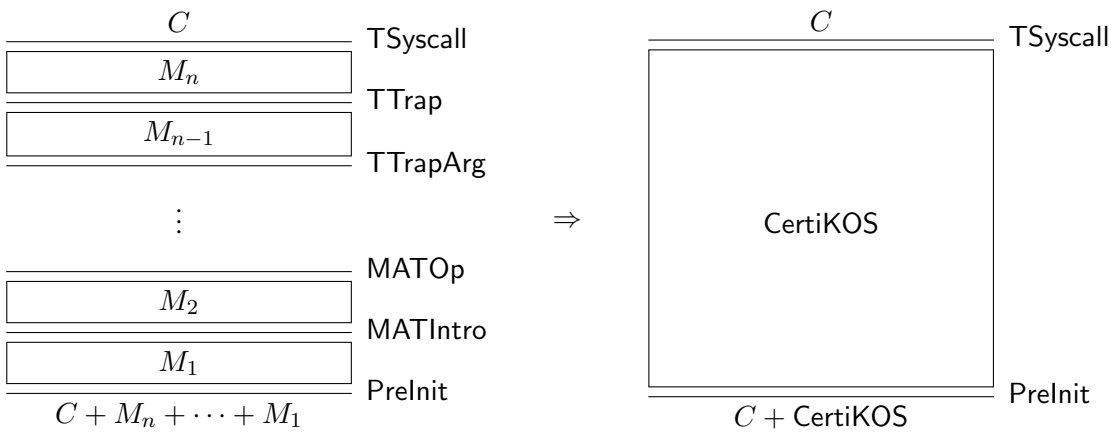


Figure 1.2: Structure of the certified OS kernel CertiKOS. Here, boxes represent certified abstraction layers and horizontal lines represent layer interfaces. The high-level client program C is transformed into the complete low-level program $C + \mathsf{CertiKOS}$ by linking it with the successive certified abstraction layers of CertiKOS. As before, the correctness properties of layers can be composed to derive a correctness property for the whole system.

the FLINT group around the time the effort to verify a complete version of CertiKOS started.

CertiKOS is divided into several dozen *certified abstraction layers*, which were specified and verified individually. *Layer interfaces* provide an abstract view of a layer's functionality, hiding the procedural details and low-level data representations involved in its implementation. A layer interface enumerates the primitives implemented by a layer and specifies their expected behavior. Client code can then be verified in terms of this abstract view in order to build higher-level layers.

To make this possible, we parameterized CompCert semantics by a layer interface: for example, the expression $\mathsf{Asm}_L[C]$ denotes the set of traces generated by the client assembly program C running on top of the layer interface L. Then a layer M implements an *overlay* interface L_2 on top of an *underlay* interface L_1 when the following *contextual refinement* property holds:

$$\forall\, C \,\cdot\, \mathsf{Asm}_{L_2}[C] \supseteq \mathsf{Asm}_{L_1}[C + M]\,.$$

Here, the execution of the client code C on top of the overlay L_2 serves as the specification. The property shows that this specification can be implemented by running C together with the layer implementation M on top of the underlay interface L_1.

Certified abstraction layers with compatible interfaces can then be chained together, in the

same way two passes of a compiler can be composed when the target language of one corresponds to the source language of the other. This allows us to derive a contextual refinement property for the whole kernel (Figure 1.2). The system call interface of CertiKOS is specified by the topmost layer interface TSyscall, and the basic facilities offered by the hardware are formalized as the base layer interface Prelnit. Then, by decomposing $\mathsf{CertiKOS} = M_n + \cdots + M_1$ into 37 certified abstraction layers, we were able to derive the overall theorem:

$$\forall\, C \cdot \mathsf{Asm}_{\mathsf{TSyscall}}[C] \supseteq \mathsf{Asm}_{\mathsf{Prelnit}}[C + \mathsf{CertiKOS}]\,.$$

1.1.4 Horizontal composition

In addition to the *vertical* composition principles outlined above, CompCert and CertiKOS provide limited forms of *horizontal* composition, which allow individual programs and layer implementations to be decomposed further.

In CompCert, this is used to model *separate compilation*. The original correctness theorem of CompCert only characterized the compilation of a whole program, but in practice C programs usually consist of several `.c` files, known as *compilation units*. These components are compiled independently, and the results are combined through *linking* to build an executable image. To reflect this, Kang et al. [2016] introduced a notion of program linking ($+$), and generalized the correctness theorem of CompCert to the separate compilation property:

$$\frac{\forall\, 1 \le i \le k \cdot \mathsf{CompCert}(p_i) = p_i'}{\mathsf{Clight}[p_1 + \cdots + p_k] \supseteq \mathsf{Asm}[p_1' + \cdots + p_k']}$$

Likewise, in CertiKOS the verification of a given layer can be decomposed into the verification of the individual functions which constitute its implementation. This is achieved through the *layer calculus* presented in Chapter 4.

1.1.5 Challenges

The vertical and horizontal composition principles used in CompCert and CertiKOS enable the construction of certified systems of significant size. However, extending them to build large-scale certified systems out of disparate certified components is difficult. A key aspect enabling com-

position in CompCert and CertiKOS is the uniformity of the models underlying their language semantics and correctness proofs. By contrast, across certified software projects there exists a great diversity of semantic models and verification techniques. This makes it difficult to formulate specifications for interfacing specific projects, let alone devise a general method for connecting certified components.

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 which were introduced alongside new compiler features and optimizations [Leroy, 2012].

Given the difficulty of verification, it is essential to preserve this flexibility in the choice of models used to verify individual components. To make it possible to link components verified using a variety of paradigms, we must then identify a model expressive enough to embed the semantics, specifications and correctness proofs of all these paradigms. The model should provide high-level composition and reasoning principles, allowing us to construct large-scale certified systems.

1.2 General models for system behaviors

Fortunately, there is a wealth of semantics research to draw from as we attempt to design general models for certified components. I outline some of it below.

1.2.1 Symmetric monoidal categories

As a whole, category theory provides a general taxonomy of compositional structures found across mathematics. In a category, two components (morphisms) can be chained together when the interface which the first one provides (its target object) matches the interface which the second one relies on (its source object). For example, as described above, compilation passes and certified abstraction layers both constitute categories.

Categories can be equipped with additional structure. In particular, *symmetric monoidal categories* allow components to be connected not only in series ($\circ$) but also in parallel ($\otimes$), as illustrated

by the following rules:

$$\frac{f : A \to B \qquad g : B \to C}{g \circ f : A \to C} \qquad\qquad \frac{f_1 : A_1 \to B_1 \qquad f_2 : A_2 \to B_2}{f_1 \otimes f_2 : A_1 \otimes A_2 \to B_1 \otimes B_2}$$

Many kinds of systems and processes exhibit this structure [Baez and Stay, 2010]. The basic setup can be refined by introducing additional constructions, for instance modeling feedback loops (traced monoidal categories) or allowing wires to run in both directions ($*$-autonomous categories).

Symmetric monoidal categories appear in different forms in many approaches to logic and programming language semantics. For example, *cartesian closed* categories correspond to models of the simply typed lambda calculus, and are a special case of symmetric monoidal categories. However, symmetric monoidal categories in general do not require multiple components to be able to connect to the same interface ($\Delta : A \to A \otimes A$), giving us more flexibility when modeling resources and stateful components. In the same way the simply-typed lambda calculus provides an internal language for cartesian closed categories, various fragments of *linear logic* provide internal languages for various kinds of symmetric monoidal categories.

In general, the theory of symmetric monoidal categories constitutes a repository of algebraic structures formalizing the compositional aspects of all kinds of systems, and can guide the design of general models.

1.2.2 Game semantics

A specific way to construct symmetric monoidal categories is *game semantics* [Abramsky, 2010], which uses two-player *games* to describe the possible interactions between program components and their environment, and uses *strategies* for these games to represent the externally observable behavior of components.

Game semantics is a synthesis of various approaches to programming language semantics. It is a form of *denotational semantics*, defining the behavior of complex programs in terms of the behavior of their components. However, since it models components by describing their interactions across time, game semantics also exhibits a strong *operational* flavor. Finally, the usual construction of strategies used in game semantics is a variation on the *trace semantics* used in the context of process algebras and concurrent systems.

An early success of game semantics was the formulation of the first satisfactory fully complete models for the programming language PCF [Abramsky et al., 2000; Hyland and Ong, 2000]. Since then, game semantics have provided fully complete models for a wide variety of languages, incorporating features such as state [Abramsky and McCusker, 1997], control [Laird, 1997], general references [Abramsky et al., 1998], concurrency [Ghica and Murawski, 2004] and more.

There are deep connections between game semantics and linear logic [Blass, 1993], and hence symmetric monoidal categories, hinting at its promise as a general approach for modeling various kinds of systems and processes. In particular, the typed aspect of many game models makes them well-suited for 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 seen as a particularly simple version of game semantics.

In this framework, computations are modeled as *terms* in an algebra whose operations correspond to the available effects. The computation proceeds inward, with each function symbol representing an occurrence of an effect, and each argument representing a possible way for the computation to be continued after the effect has occurred.

An advantage of this approach is that the methods and results of universal algebra become available to reason about effectful computations. Equational theories can be used to characterize the behavior of effects, and interpretations of one effect algebra into another model *effect handlers* [Plotkin and Pretnar, 2009]. The *free monad* associated to an algebra can be used to recover the more familiar monadic model of computational effects [Moggi, 1991]. Along these lines, *interaction trees* [Xia et al., 2019] have been developed and formalized in the Coq proof assistant for use in and across several DeepSpec projects.

Algebraic effects can also be understood as a limited form of game semantics: signatures can be interpreted as simple games, and the abstract syntax trees of terms can be interpreted as strategies. However, their restriction to first-order computation make them easier to formalize and reason

about than more general notions of games.

1.2.4 The refinement calculus

While game semantics and effect systems 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 and specifications. It has been studied extensively in the context of Dijkstra's predicate transformer 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.

To make it possible to express specifications, the language is extended with non-executable constructions. In particular, the refinement calculus includes infinitary versions of both *angelic* and *demonic* nondeterministic choice operators. In its modern presentation, the refinement calculus is formulated in a lattice-theoretic framework where joins ($\sqcup$) and meets ($\sqcap$) correspond respectively to angelic and demonic choices. The resulting language is remarkably expressive and requires very few additional primitive constructions. The duality inherent in this approach also lends itself to game-theoretic interpretations, and indeed the semantics of the refinement calculus can be expressed as a two-player game between the angel and the demon.

1.3 Compiling certified components

There are two ways to look at CompCert in the context of software verification: as a certified system with an interesting structure, or as a tool we can use to build certified programs.

1.3.1 CompCert as a certified system

Until now, I have mostly discussed CompCert as an example of certified system, describing the compositional structures used in its construction as a precedent for the development of certified systems of significant size.

From this point of view, the language semantics formalized alongside the compiler's code are components of its specification, used to express the behavior expected of a C compiler and to formulate the compiler's correctness theorem. The long-standing effort to refine and generalize these semantics and proofs can be understood as an attempt to model real-world compiler usage more accurately, preventing bugs in the compiler from making it through the verification process due to simplifications in its specification.

Connecting CompCert with other components could mean combining the correctness proof of CompCert with that of a certified assembler and certified linker, perhaps even with a certified processor design. By verifying a larger portion of the toolchain end-to-end, this would provide a stronger guarantee that the source programs written by the user will ultimately be executed in a way that conforms to the C standard. Work in this direction includes Wang et al. [2019].

1.3.2 CompCert as a tool for building certified systems

CompCert is also used as a tool for constructing other certified systems. For example, CertiKOS uses the assembly semantics formalized in CompCert to model the execution of the kernel's code and to express the correctness property which it must satisfy. In addition, CompCert is used to compile the portions of the kernel which are written in C, and the compiler's correctness theorem allows us to use code proofs carried out with respect to the source code to prove the compiled assembly code correct.

CompCert serves in this case as a rudimentary *framework* for the construction of certified C and assembly programs in the Coq proof assistant. From this point of view, efforts to increase the precision and flexibility of the compiler's correctness theorem have made this framework more general and compositional.

1.3.3 Integrating CompCert into a general-purpose model

Given the specialized nature of CompCert's semantics and proofs, it is difficult to imagine that "CompCert as a framework" itself could be extended to allow the construction of general certified systems spanning a wide range of abstraction levels. Still, given the importance of compilers in the construction of present-day computer systems, and the importance of CompCert in the formal methods landscape, its integration into any framework attempting to tackle end-to-end verification should be a litmus test, and would provide immediate benefits:

- CompCert uses transition systems to define language semantics. Embedding these transition systems into a general model would immediately augment that model with a semantics for C and assembly programs. If we use a version of CompCert transition systems which can express the behavior of individual translation units, this would also give us the ability to formally interface, at a granular level, program components with components of different kinds, for example file systems, network services, or hardware devices.

- Then, in conjunction with a sufficiently precise correctness theorem, CompCert would not simply provide a certified compiler, but in fact a *compiler of certified program components*. Correctness proofs characterizing the interactions of a component's source code with the environment could be transferred in a systematic way to the compiled code.

Unfortunately, despite the abundance of research extending CompCert in this direction, no current extension is flexible enough to be used in this way (§8.3). As noted by Patterson and Ahmed [2019], the problem of certified compositional compilation is extremely complex and challenging, lacking even a commonly accepted definition. In fact, as illustrated by the connections between certified abstraction layers and certified compilation, the very extensive form of compositional certified compilation which I describe above is almost as challenging to address as the broader problem of modeling heterogeneous certified components.

On the other hand, this means that the techniques and conceptual framework developed in this thesis are well-suited for understanding and tackling this problem, and indeed the culmination of my work is a version of CompCert which addresses this challenge.

1.4 Contributions

The central claim of 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:

- Part I presents some relevant background.
- Part II introduces the general approach of *refinement-based game semantics*. A reexamination of the fundamentals of game semantics under the lens of dual nondeterminism invites us to regard plays as *elementary specifications* for the behaviors of interactive systems. The completion of plays with arbitrary angelic and demonic choices yields a notion of *strategy specification*, which provides support for stepwise refinement and data abstraction in the context of game semantics.

  I demonstrate use of this approach in the context of certified abstraction layers. Starting from the layer calculus of CertiKOS, I construct increasingly expressive models where layer interfaces, layer implementations and simulation relations can be treated in a uniform and compositional way.
- Part III presents my work on the certified compiler CompCert. I examine existing models and techniques under the lens of refinement-based game semantics, and articulate why none of the existing CompCert extensions can be integrated within the framework in a completely satisfactory way. I then introduce CompCertO, the first extension of CompCert suitable for this use, which nevertheless avoids much of the complexity found in previous approaches.

Most of this work has been formalized in the Coq proof assistant. As I am writing this, the beginning of a Coq library for refinement-based game semantics is available at:

`https://github.com/CertiKOS/rbgs/`

A complete implementation of CompCertO can also be found at the following address:

`https://github.com/CertiKOS/compcert/tree/compcerto`

Part I

Preliminaries

Chapter 2

Background

Refinement-based game semantics combines several lines of research. This chapter aims to provide a short introduction to each one, and to give the reader a sense of how they relate to one another.

I begin by outlining in §2.1 general principles which can be used in the construction of certified systems. In the following sections, I present the *refinement calculus* (§2.2), *logical relations* (§2.3), *game semantics* (§2.4), *algebraic effects* (§2.5) and *monads* (§2.6).

2.1 Building certified systems

2.1.1 Semantic domains

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. I will call the set $\mathbb{D}$ a *semantic domain*.

Example 2.1 (Trace semantics)**.** *In CompCert, the behaviors of programs are ultimately expressed as sets of traces. Each trace records a possible execution of the program, as a finite or infinite sequence of externally observable events taken from a set $\mathbb{E}$. The program may then terminate with an exit status $r \in \mathsf{int}$, exhibit an undefined behavior ($\lightning$) or silently diverge ($\Uparrow$). The corresponding semantic domain can be defined as:*

$$\mathbb{D}_{\mathsf{CompCert}} := \mathcal{P}(\mathbb{E}^*\mathsf{int} + \mathbb{E}^*\lightning + \mathbb{E}^*\Uparrow + \mathbb{E}^\omega)\,.$$

In the remainder of this section I elucidate the structure and properties of $\mathbb{D}$ necessary to the process of building large-scale certified systems.

2.1.2 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 σ_1 is refined by σ_2. The correctness of a system description $p \in P$ with respect to a specification $\sigma \in \mathbb{D}$ can then be formulated as $\sigma \sqsubseteq [\![p]\!]$.

Refinement is intended to be a correctness-preserving relation. For a set $\mathbb{P} \subseteq \mathcal{P}(\mathbb{D})$ of properties of interest and for two semantic objects $\sigma_1, \sigma_2 \in \mathbb{D}$, the following property should hold:

$$\sigma_1 \sqsubseteq \sigma_2 \;\Rightarrow\; \forall P \in \mathbb{P} \cdot P(\sigma_1) \Rightarrow P(\sigma_2)\,.$$

Note that in particular, the property $P_\sigma(x) \;:=\; \sigma \sqsubseteq x$ is always preserved in this way.

Example 2.2 (Trace containment)**.** *Continuing Example 2.1, in the trace semantics of CompCert, the set of traces associated with the source program is understood as a set of* permissible *behaviors for the target program. As a first approximation, the corresponding notion of refinement is* trace containment*, expressed by the superset relation $\supseteq$, so that for example $\mathsf{Clight}[p] \in \mathbb{D}_{\mathsf{CompCert}}$ is refined by $\mathsf{Asm}[p'] \in \mathbb{D}_{\mathsf{CompCert}}$ when $\mathsf{Clight}[p] \supseteq \mathsf{Asm}[p']$. Trace containment preserves any property $P(\sigma)$ asserting that* all *traces $t \in \sigma \in \mathbb{D}_{\mathsf{CompCert}}$ satisfy a given condition.*

In fact, CompCert also allows undefined behaviors to be refined by more specific ones, which makes the refinement relation slightly more sophisticated. See Chapter 8 for details.

2.1.3 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 a 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 $\oplus : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$. Ideally, the operation $\oplus$ will characterize $+$ exactly:

$$[\![p_1]\!] \oplus [\![p_2]\!] = [\![p_1 + p_2]\!]\,. \tag{2.1}$$

Example 2.3. *Denotational semantics are compositional by nature. The behavior of program components is defined recursively on their structure, so that (2.1) holds* by definition.

In the context of refinement-based verification, we can regard $\oplus$ as a *specification* for $+$, and it is sufficient to establish that:

$$[\![p_1]\!] \oplus [\![p_2]\!] \sqsubseteq [\![p_1 + p_2]\!]\,. \tag{2.2}$$

Example 2.4 (Linking)**.** *CompCert approximates* linking *as an operator $+$ which merges programs. In my variant CompCertO presented in Chapter 9, the semantic model is equipped with a* horizontal composition *operation $\oplus$, which provides a specification for the linking operator. In particular, the correctness of assembly linking is established in Thm. 9.9 as $\mathsf{Asm}(p_1) \oplus \mathsf{Asm}(p_2) \sqsubseteq \mathsf{Asm}(p_1 + p_2)$.*

2.1.4 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:

$$\frac{\sigma_1 \sqsubseteq \sigma_1' \qquad \sigma_2 \sqsubseteq \sigma_2'}{\sigma_1 \oplus \sigma_2 \sqsubseteq \sigma_1' \oplus \sigma_2'}$$

For example, 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 σ. 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 $\sigma \sqsubseteq \sigma_1 \oplus [\![p_2]\!]$:

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

2.1.5 Types

In many cases, systems are built out of components of various types which can only be composed when their types are compatible. Different semantic domains can then be used to interpret components of different types.

As discussed in Chapter 3, categories offer a systematic framework to realize this separation. Instead of a single set $\mathbb{D}$, we use a category $\mathbf{D}$. The objects $A \in \mathbf{D}$ correspond to the possible types or interfaces alongside which components can be connected. A component which uses an interface A to implement an interface B can be modeled as a morphism of type $A \to B$. Categorical constructions can then be used to formulate and characterize typed composition principles, and the theory of *enriched categories* can be used to explore how these composition principles interact with the structure of the semantic domains $\mathbf{D}(A, B)$.

Example 2.5 (Signatures)**.** *The models introduced in this thesis use* signatures*, discussed in §2.5, to enumerate the operations used and provided by program components. Consequently, the semantics of components are often expressed in categories whose objects are signatures. Morphisms of type $E \to F$ describe the behavior of components which use the operations of the signature E to implement the operations of the signature F. Categorical products and other monoidal structures can then be used to describe various horizontal composition principles.*

2.1.6 Abstraction

Large-scale systems operate across multiple levels of abstraction. Each abstraction level brings its own understanding of the interaction between a component and its environment. To reason across abstraction layers we need to give an explicit account of how these different views of a component's behavior correspond to one another.

This is studied extensively in the context of abstract interpretation [Cousot and Cousot, 1992]. Suppose $\mathbb{D}_1$ is the concrete domain and $\mathbb{D}_2$ is the abstract one. To establish a correspondence between them, the most general approach is to define a *soundness relation* $\rho \subseteq \mathbb{D}_2 \times \mathbb{D}_1$. We expect ρ to be compatible with refinement:

$$\frac{\sigma_2' \sqsubseteq_2 \sigma_2 \qquad \sigma_2 \ \rho \ \sigma_1 \qquad \sigma_1 \sqsubseteq_1 \sigma_1'}{\sigma_2' \ \rho \ \sigma_1'}$$

In other words, if σ_2 is an abstraction which soundly approximates a concrete semantic object σ_1, we can make σ_2 less precise or σ_1 more precise and preserve this relationship.

Often there will be a more elementary description of the correspondence, based on the construction of the semantic domains, as illustrated by the following example.

Example 2.6. *In the model of* certified abstraction layers *presented in Chapter 4, the semantic domain used to model a layer interface is built from a set of abstract states S. To compare a layer interface formulated in terms of a more concrete set of states S_1 with a layer interface formulated in terms of a more abstract set of states S_2, we will use a relation $R \subseteq S_2 \times S_1$. This relation can then be extended to the level of layer interfaces as a soundness relation $\leq_R \subseteq \mathbb{D}[S_2] \times \mathbb{D}[S_1]$, asserting that R is a simulation relation between a layer interface in $\mathbb{D}[S_2]$ and a layer interface in $\mathbb{D}[S_1]$.*

In the case of categorical semantics where $\mathbb{D}_1 = \mathbf{D}(A_1, B_1)$ and $\mathbb{D}_2 = \mathbf{D}(A_2, B_2)$ are homsets, the constituents of the soundness relation may themselves be organized as a category with the same objects as $\mathbf{D}$. This induces a double category structure, where horizontal morphisms are the semantic objects, vertical morphisms are abstraction relationships, and 2-cells correspond to the soundness relation.

Example 2.7 (Simulation conventions in CompCertO)**.** *Interactions between C compilation units are understood very differently from interactions between assembly-language components. At the level of C, cross-module interactions are defined in terms of function calls; invoking a function means 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 of the current module with a certain register state. The* calling convention *used by a compiler specifies the correspondence between the two.*

In the model used in CompCertO, language interfaces *describe the form of cross-component interactions for a class of languages. They play the role of objects, and semantic domains are assigned a type $A \to B$ where A and B are the language interfaces respectively used by outgoing and incoming calls. In addition, a* simulation convention $\mathbb{R}_A : A_1 \Leftrightarrow A_2$ *is used to formulate the correspondence between the high-level language interface A_1 and the low-level language interface A_2. A simulation*

property can then be depicted as the 2-cell:

$$\begin{array}{ccc} A_1 & \xrightarrow{L_1} & B_1 \\ \mathbb{R}_A \Updownarrow & & \Updownarrow \mathbb{R}_B \\ A_2 & \xrightarrow[L_2]{} & B_2 \end{array}$$

When semantic domains are rich enough, there may be a most precise abstraction $\sigma_2 \in \mathbb{D}_2$ for each concrete semantic object $\sigma_1 \in \mathbb{D}_1$, defining an abstraction function $\alpha : \mathbb{D}_1 \to \mathbb{D}_2$ such that:

$$\sigma_2' \ \rho \ \sigma_1 \ \Leftrightarrow \ \sigma_2' \sqsubseteq_2 \alpha(\sigma_1)$$

Conversely, there may be a most general σ_1 corresponding to each σ_2, defining a concretization function $\gamma : \mathbb{D}_2 \to \mathbb{D}_1$ such that:

$$\gamma(\sigma_2) \sqsubseteq_1 \sigma_1' \ \Leftrightarrow \ \sigma_2 \ \rho \ \sigma_1'$$

When both an abstraction and a concretization function exist, they are related by the property:

$$\gamma(\sigma_2) \sqsubseteq_1 \sigma_1 \ \Leftrightarrow \ \sigma_2 \sqsubseteq_2 \alpha(\sigma_1)$$

This corresponds to the original formulation of abstract interpretation [Cousot and Cousot, 1977] in terms of Galois connections [Erné et al., 1993].

2.1.7 Embedding

It is often useful to first interpret the semantics of a component in a restricted domain, then embed this domain into a more general one:

$$p \in P \xrightarrow{[\![-]\!]} \sigma \in \mathbb{D} \xhookrightarrow{|-|} |\sigma| \in \mathbb{D}'$$

The elements of the restricted domain $\mathbb{D}$ will often have stronger properties, or will be expressed in a way which makes certain proof methods more tractable. The more general domain $\mathbb{D}'$ can

then provide more expressivity and additional algebraic structures, and may embed the behaviors of different kinds of components, but may be less amenable to domain-specific reasoning.

In this situation, we will want the embedding $|-| : \mathbb{D} \hookrightarrow \mathbb{D}'$ to preserve any relevant structure present in both $\mathbb{D}$ and $\mathbb{D}'$. For example, in the case of categorical semantics, this embedding will be a (faithful) functor of the appropriate kind.

Example 2.8 (Transition systems and trace semantics in CompCert)**.** *As mentioned in Example 2.1, CompCert semantics ultimately characterize the behavior of programs in terms of traces. However, they are first defined as labelled transition systems, and the correctness properties of compilation passes are first proved as simulations. An embedding then assigns to each transition system L the set of traces $|L|$ characterizing its observable behavior, and a soundness proof shows that under certain conditions, simulation properties $L_1 \leq L_2$ imply trace containment $|L_1| \supseteq |L_2|$.*

Example 2.9 (Certified abstraction layers)**.** *The theory of certified abstraction layers developed in Chapter 4 describes deterministic layer interfaces and implementations. Abstraction is expressed using simulation relations. Layer interfaces, layer implementations and simulation relations are represented as three different kinds of objects.*

In Chapter 6, I show how this theory of certified abstraction layers can be embedded into a more general framework which supports dual nondeterminism, allowing layer interfaces, layer implementations and simulation relations to be represented as morphisms in a uniform way, at the cost of a more sophisticated notion of refinement.

Finally in Chapter 7, I outline a framework where the layers' states can be encapsulated and hidden from the descriptions of the layers' behaviors, and again provide an embedding of the previous theory into this more general one. Layer correctness can be expressed without simulation relations, but the cartesian products of layers sharing states (§4.3.3) are no longer available.

Each of these embeddings preserves categorical composition, refinement, and tensor products.

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 to programming language semantics [Hoare, 1969], 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 it 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. For example, 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 correctness property, we need to make sure we can deal with either choice:

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

Conversely, in the program $C_1 \sqcup C_2$, an *angel* will decide whether C_1 or C_2 is executed. The statement $\mathsf{x} := \mathsf{x} * 2 \;\sqcup\; \mathsf{x} := 0$ is more difficult to interpret than its demonic counterpart, but roughly speaking it magically behaves as $\mathsf{x} := \mathsf{x} * 2$ or $\mathsf{x} := 0$ depending on the needs of its user. If possible, the angel will make choices which validate the correctness property. Therefore:

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

More generally, a triple $P\{C\}Q$ can be interpreted as a *game* between the angel and the demon [Back and Wright, 1998, Chapter 14]. The angel resolves the $\sqcup$ choices, whereas the demon resolves the $\sqcap$ choices. The triple is valid if there is a strategy for the angel to validate the correctness property.

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. For the angel to have a winning strategy for the left-hand side triple, they must be able to 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 order between the angel and the 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 is 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 correctly implements a specification C_1. The specification may be formulated in abstract, declarative terms, but the program should only use executable constructions.

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 become:

$$\frac{C \sqsubseteq C_1 \qquad C \sqsubseteq C_2}{C \sqsubseteq C_1 \sqcap C_2} \qquad \frac{C \sqsubseteq C_1}{C \sqsubseteq C_1 \sqcup C_2} \qquad \frac{C \sqsubseteq C_2}{C \sqsubseteq C_1 \sqcup C_2}$$

Given the duality 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, I 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:

$$\frac{C_1 \sqsubseteq C}{C_1 \sqcap C_2 \sqsubseteq C} \qquad \frac{C_2 \sqsubseteq C}{C_1 \sqcap C_2 \sqsubseteq C} \qquad \frac{C_1 \sqsubseteq C \qquad C_2 \sqsubseteq C}{C_1 \sqcup C_2 \sqsubseteq C}$$

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*.

Remark 2.10. *Since the work presented in this thesis borrows from various lines of research, one difficulty is that the conventions and notations they use are often different and sometimes inconsistent.*

As the discussion above illustrates, in the literature on Hoare logic, the reader is often invited to identify with a user *of the program (the environment) trying to ascertain the program's properties, and the terminology of "angelic" and "demonic" is used accordingly. By contrast, descriptions of game semantics are often narrated from the point of view of the* system*, and in that context the terminology associated with dual nondeterminism can be counter-intuitive at first.*

(a) Angelic choice (b) Demonic choice

Figure 2.1: Angelic and demonic choices in specifications, from the perspective of the implementer. Life is hard when you're expected to behave like an angel!

2.2.5 The refinement calculus

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

Dijkstra's *predicate transformer* semantics [Dijkstra, 1975] is a natural fit for the refinement calculus, but other approaches are possible. For instance, as mentioned above, 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.

In fact, the refinement calculus can be presented as a *hierarchy* along the lines discussed in §2.1.7, whereby simpler models (state transformer functions, relations) can be embedded into 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 embedding 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.

However, in its traditional formulation, the refinement calculus only models 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 my work can be understood as a step in this

direction as well.

2.3 Logical relations

Logical relations are structure-preserving relations in the way homomorphisms are structure-preserving maps. However, logical relations are more compositional than homomorphisms, because 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. For example, 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 a 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 Reynolds' well-known theory of relational parametricity, and can be used to prove *free theorems* that all terms of a given generic type must satisfy.

For our purposes, logical relations will provide a language to formulate relationships between high-level and low-level behaviors, or between specifications and implementations, in a uniform and compositional way.

2.3.1 Binary logical relations

Logical relations can be of any arity, but I will focus on binary logical relations. Consider 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 2.11. *A monoid is a set with an associative operation $\cdot$ and an identity element ϵ. 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:*

$$(u\ R\ u' \wedge v\ R\ v' \ \Rightarrow\ u \cdot_A v\ \ R\ \ u' \cdot_B v') \ \wedge\ \epsilon_A\ R\ \epsilon_B\,. \tag{2.3}$$

Logical relations used to reason about contextual equivalence are often partial equivalence relations (PER). By contrast, in the context of refinement, most of the relations will not be symmetric.

2.3.2 Relators

Logical relations between multi-sorted 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 Figure 2.2. Using them, the proposition (2.3) can be reformulated as:

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

Example 2.12. *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:*

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

Using the relators in Figure 2.2, we can express the same property concisely and compositionally as:

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

2.3.3 Kripke relations

Relations for stateful languages often depend on the current state. To address this, Kripke logical relations are parameterized 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 2.13. A *Kripke* relation is a family of relations $(R_w)_{w \in W}$. I will write $R \in \mathcal{R}_W(A, B)$

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

Figure 2.2: A selection of relators. The relators $\times$, $+$, $\to$ are standard. $\mathcal{P}^{\leq}$ and $\mathcal{P}^{\geq}$ are asymmetric relators for the powerset type operator $\mathcal{P}$, which can be used to formulate simulations.

for a Kripke relation between the sets A and B. For $w \in W$, I will use the following notations:

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

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 $R_i \in \mathcal{R}_W(A_i, B_i)$:

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

In the following, I will 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, I will use the notation $x :: R$ to denote $x\ R\ x$. For legibility, I also write $w \Vdash x\ R\ y$ for $x\ [w \Vdash R]\ y$ and $\Vdash x\ R\ y$ for $x\ [\Vdash R]\ y$.

2.3.4 Modal relators

Kripke logical relations are connected to the Kripke semantics of modal logic. In that context, the set of worlds is used to index a model, and the satisfaction of formulas may depend on which world is considered. The modality $\Diamond$ is then interpreted in terms of an *accessibility* relation, which

I will write as $\rightsquigarrow$. The formula $\Diamond P$ is satisfied in a world w if there exists $w \rightsquigarrow w'$ such that P is satisfied in w'. Dually, the formula $\Box P \equiv \neg(\Diamond \neg P)$ is satisfied in a world w if P is satisfied in *all* worlds w' accessible from w.

The accessibility relation essentially defines a directed graph on the set of worlds, and formulas constructed with $\Diamond$ and $\Box$ can explore the neighborhood of a given node. A wide variety of complex data structures can be seen as graphs; modal logics and Kripke semantics can be used to reason about them, with countless applications [Blackburn et al., 2001].

Translating this approach to the setting of logical relations, the modalities become relators.

Definition 2.14. 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 $\Diamond : \mathcal{R}_W(A, B) \to \mathcal{R}_W(A, B)$ is defined by:

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

2.4 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 moves corresponding to the execution of $\mathsf{x} := 2 * \mathsf{x}$ have the form:

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

The moves of the system have been underlined. The environment initiates the execution with the move run. The system move $\underline{\mathsf{read}_\mathsf{x}}$ then requests the value of the variable x, communicated in response by the environment move n. The system move $\underline{\mathsf{write}_\mathsf{x}[2n]}$ requests storing the value $2n$ into the variable x, and is acknowledged by the environment move ok. Finally, the system move done expresses termination. The gray arrows show the relationships between questions and their corresponding answers, but in the simple game models that I will consider they are not part of the formalism.

2.4.1 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. As above, when typesetting examples, we underline the moves of the system.

Example 2.15. *In the game of chess, moves are taken in the set* $\{a1 \dots h8\} \times \{a1 \dots h8\}$. *From the perspective of the player with black pieces, a valid sequence of moves may look like:*

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

Most game semantics include additional structure in the description of games. The set of moves is usually partitioned into environment and system 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 compositionality 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 where the roles

of the players are reversed and started at the discretion of the system.

2.4.2 Strategies

The *plays* of a game are sequences of moves; they both 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:

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

Prefix closure makes it possible both to represent partially defined strategies, and to represent infinite computations using their finite prefixes. Additional constraints are carefully chosen to construct strategy models with the right properties for a given application.

2.4.3 Determinism

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 *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]. We will see in Chapter 5 that by approaching the question from the point of view of dual nondeterminism, we are led to a different interpretation and a different approach.

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

$$w := \mathsf{readbit}(\mathsf{print}[\text{“Hello”}](\mathsf{done}), \mathsf{print}[\text{“World”}](\mathsf{done}))$$

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. I will use the term w in several examples below.

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 $\mathsf{print}[u]$ operation for every $u \in \mathsf{string}$.

2.5.1 Effects 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 $(\mathsf{print}[u])_{u \in \mathsf{string}}$ of arity 1. An equation for this signature is:

$$\mathsf{print}[u](\mathsf{print}[v](x)) = \mathsf{print}[uv](x)\,,$$

which indicates that printing the string u followed by printing the string v is equivalent to printing the string uv in one go.

In this work, I use effect signatures to represent the possible external interactions of a computation, but I will not use equational theories. It will however be possible to interpret effects into another signature, modeling a limited form of *effect handlers* [Plotkin and Pretnar, 2009].

2.5.2 Effect signatures

Definition 2.16. An *effect signature* is a set E of operations together with a mapping ar, which assigns to each operation $m \in E$ a set $\mathsf{ar}(m)$ called the *arity* of m. I will describe effect signatures using the notation $E = \{m_1{:}N_1, m_2{:}N_2, \ldots\}$, where $N_i = \mathsf{ar}(m_i)$ is the arity of the corresponding operation m_i.

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 signature

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

where $\mathbb{1} = \{*\}$ and $\mathbb{2} := \{\mathsf{tt}, \mathsf{ff}\}$ are finite sets of the expected size.

Since the construction $\{m[x] : B \mid x \in A\}$ is used extensively, I will represent it using the syntactic sugar $\{m : A \to B\}$ so that for example the signature above can be described as:

$$E_{\mathsf{io}} = \{\mathsf{readbit} : \mathbb{2},\ \mathsf{print} : \mathsf{string} \to \mathbb{1},\ \mathsf{done} : \varnothing\}$$

2.5.3 Computations as terms

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. A term represents a computation which proceeds inward from the top-level operation towards the leaves of the term.

The leaves are the constants $(c : \varnothing) \in E$, and in terms representing partial computations they may be variables as well. In that case, the variables may be thought of as placeholders, each one representing a possible intermediate outcome.

Terms are defined below. Since we are using infinite arities, the argument tuple for an operation $m : N$ will often be given as a family $(t_n)_{n \in N}$ indexed by N. When it seems helpful, I will use underlining to prevent any confusion between the operation $(m : N) \in E$ itself and the term constructor $\underline{m} : T_E(X)^N \to T_E(X)$ associated with it, and to prevent confusion between an element $v \in X$ of the set of variables and the corresponding term $\underline{v} \in T_E(X)$.

Definition 2.17 (Terms over a signature)**.** The set of *terms* associated with a signature E and a set of variables X is defined by the grammar:

$$t \in T_E(X) ::= \underline{m}(t_n)_{n \in N} \mid \underline{v} \qquad \text{where } (m : N) \in E,\ v \in X$$

Consider an operation $m : N$ in the signature E. A term of the form $\underline{m}(t_n)_{n \in N}$ first triggers the corresponding effect. The effect's outcome $n \in N$ then resumes the computation as prescribed by the subterm $t_n \in T_E(X)$. On the other hand, terms of the form $\underline{v}$ correspond to computations which immediately terminate with the outcome $v \in X$. A variable substitution $f : X \to T_E(Y)$ can then specify how the computation is to be continued.

Definition 2.18 (Variable substitution)**.** A substitution $f : X \to T_E(Y)$ can be *applied* to a term $t \in T_E(X)$, yielding the term $tf \in T_E(Y)$ defined recursively by:

$$\underline{v}\, f := f(v) \qquad v \in X$$

$$\underline{m}(t_n)_{n \in N}\, f := \underline{m}(t_n f)_{n \in N} \qquad (m : N) \in E$$

Example 2.19. *The computation w given above can be decomposed into the term $t \in T_{E_{\mathsf{io}}}(\{x, y\})$ and the substitution $f : \{x, y\} \to \varnothing$ defined as:*

$$t := \underline{\mathsf{readbit}}(\underline{x}, \underline{y})$$

$$f := \{x \mapsto \underline{\mathsf{print}[\text{“Hello”}]}(\underline{\mathsf{done}}),\ y \mapsto \underline{\mathsf{print}[\text{“World”}]}(\underline{\mathsf{done}})\}\,.$$

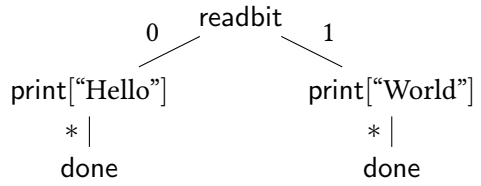
Then the term $tf \in T_{E_{\mathsf{io}}}(\varnothing)$ obtained applying f to t is again:

$$tf = \mathsf{readbit}(\mathsf{print}[\text{“Hello”}](\mathsf{done}), \mathsf{print}[\text{“World”}](\mathsf{done})) = w\,.$$

2.5.4 Terms as strategies

An effect signature can also be seen as a particularly simple *game*, in which the system chooses a question $(m : N) \in E$ and the environment responds with an answer $n \in N$. Then the terms induced by the signature are *strategies* for an iterated version of this game.

For example, the abstract syntax tree of the term $w \in T_{E_{\mathsf{io}}}(\varnothing)$ can be read as the strategy:



In this tree, the nodes are labeled with operations and can be interpreted as moves of the systems. The edges are labeled with elements of the arity sets and can be interpreted as moves of the environment. Represented as a set of plays, the same strategy could be written as:

$$\omega = \{ \quad \underline{\mathsf{readbit}} \cdot 0 \cdot \underline{\mathsf{print}[\text{“Hello”}]} \cdot * \cdot \underline{\mathsf{done}}, \\ \underline{\mathsf{readbit}} \cdot 1 \cdot \underline{\mathsf{print}[\text{“World”}]} \cdot * \cdot \underline{\mathsf{done}} \quad \}.$$

2.5.5 Interpreting effects

To assign a semantics to the effects of E, we can interpret the operations of the signature in a domain A by defining for each $(m : N) \in E$ a corresponding function $\alpha^m : A^N \to A$. When equational theories are used, we need to make sure that the corresponding equations hold. In our limited setting we can use the simple definition below.

Definition 2.20. An *algebra* for the effect signature E is a *carrier* set A together with a function $\alpha^m : A^N \to A$ for each operation $(m : N) \in E$. We can then interpret a term $t \in T_E(A)$ as an element $t^\alpha \in A$ of the carrier set, defined recursively by:

$$\underline{v}^\alpha := v \qquad v \in A$$

$$\underline{m}(t_n)^\alpha_{n \in N} := \alpha^m(t^\alpha_n)_{n \in N} \qquad (m : N) \in E$$

Example 2.21 (Trace semantics for E_{io})**.** *The operations of E_{io} can be interpreted using traces in the language $P = (\mathbb{2} \cup \mathsf{string})^*$. A trace $s \in P$ records a possible execution, and sets of traces are used to*

characterize computations. Hence the algebra π uses the carrier $\mathcal{P}(P)$ and the following operations:

$$\begin{aligned}\pi^{\mathsf{readbit}}(\sigma_0, \sigma_1) &:= \{\mathsf{ff} \cdot s_0 \mid s_0 \in \sigma_0\} \cup \{\mathsf{tt} \cdot s_1 \mid s_1 \in \sigma_1\} \\ \pi^{\mathsf{print}[u]}(\sigma) &:= \{u \cdot s \mid s \in \sigma\} \\ \pi^{\mathsf{done}} &:= \{\epsilon\}\end{aligned}$$

For instance, the recurring example w yields the set:

$$w^\pi = \{0 \cdot \text{“Hello”},\ 1 \cdot \text{“World”}\}\,.$$

Note that as presented, the algebra does not respect the equation:

$$\mathsf{print}[u](\mathsf{print}[v](x)) = \mathsf{print}[uv](x)\,,$$

since the left-hand side will add two events $u \cdot v$ at the beginning of the traces, while the right-hand side will add a single event uv.

We can define an algebra for E on the set of terms $T_E(X)$ itself. Each operation $(m : N) \in E$ is interpreted by the term constructor $c^m_X : T_E(X)^N \to T_E(X)$ defined as:

$$c^m_X(t_n)_{n\in N} := \underline{m}(t_n)_{n\in N}\,.$$

Then the interpretation of a term $t \in T_E(T_E(X))$ will be a “flattened” term $t^c \in T_E(X)$ where any variable occurrence $\underline{v}$ is replaced by the term $v \in T_E(X)$.

2.6 Monads

Semantics of effectful computations are often formulated using *monads*. There are deep connections between monadic and algebraic effects, which mirror the way monads have long been used both for modeling effects and in the categorical treatment of universal algebra. This is explained in more detail in §3.3.6. Below, I give a brief introduction free of categorical jargon.

2.6.1 Motivation

The notion of algebra given in Definition 2.20 does not provide a very sophisticated account of terms with variables (partial computations). We did allow the interpreted terms to include "variables" taken directly from the algebra's underlying set A. In fact, it is possible to interpret terms with variables from an arbitrary set X if we provide an assignment $\rho : X \to A$, by constructing a substitution $\hat{\rho}(x) := \underline{\rho(x)}$ and interpreting $t \in T_E(X)$ as $(t\hat{\rho})^\alpha \in A$.

However, to be able to interpret open terms independently of any assignment, we need a notion of algebra which is *parametric* in the set of variables we use. Instead of using a single carrier set, we will specify for every possible set X of variables:

- A carrier set $T(X)$ for the corresponding algebra.
- A function $\eta_X : X \to T(X)$ providing the interpretations of variables;
- For every "semantic substitution" $f : X \to T(Y)$, a map $f^\dagger : T(X) \to T(Y)$ applying the substitution to the elements of $T(X)$.

These data can be interpreted in computational terms as follows:

- A semantic object $\tau \in T(X)$ is a partial computation with an intermediate result in X.
- The computation $\eta_X(v) \in T(X)$ immediately yields the intermediate result $v \in X$.
- A *continuation* $f : X \to T(Y)$ is activated with a value $v \in X$ to produce a computation with outcomes in Y. Its extension $f^\dagger(\tau)$ is the sequential composition of $\tau \in T(X)$ with the continuation f, sometimes written $v \leftarrow \tau \,;\, f(v)$.

When they behave consistently with each other, these constructions define a monad.

Definition 2.22. A *monad* $\langle T, \eta, (-)^\dagger \rangle$ is given as above, and must satisfy for all $f : X \to T(Y)$ and $g : Y \to T(Z)$ the following properties:

$$\eta_Y^\dagger \circ f \;=\; f^\dagger \circ \eta_X \;=\; f \qquad\qquad (g^\dagger \circ f)^\dagger \;=\; g^\dagger \circ f^\dagger \,.$$

The family η is called the monad's *unit*, and $f^\dagger$ is called the *Kleisli extension* of f.

Written in a more computational style, the monad laws given above can be reformulated as:

$$\begin{aligned} v \leftarrow \tau \,;\, \eta_X(v) &= \tau \\ v \leftarrow \eta_X(x) \,;\, f(v) &= f(x) \\ v \leftarrow (u \leftarrow \tau \,;\, f(u)) \,;\, g(v) &= u \leftarrow \tau \,;\, v \leftarrow f(u) \,;\, g(v) \end{aligned}$$

2.6.2 Interpreting effects

To assign a meaning to the operations of a signature E in the context of a monad T, we assign to each operation $(m : N) \in E$ a computation $\sigma^m \in T(N)$. Then for every set X of variables, we can define a corresponding algebra $\langle T(X), \sigma_X \rangle$ where $\sigma^m_X : T(X)^N \to T(X)$ is defined as:

$$\sigma^m_X(\tau_n)_{n\in N} := n \leftarrow \sigma^m \,;\, \tau_n \,.$$

Note that this family of algebras is compatible with the monadic structure, in the sense that substituting the variables within a computation before or after interpreting its outermost operation $m : N$ yields the same result. In other words, for a family of arguments $(\tau_n)_{n\in N}$ taken in $T(X)$ and for a substitution $f : X \to T(Y)$, the following property holds:

$$f^\dagger(\sigma^m_X(\tau_n)_{n\in N}) = \sigma^m_Y(f^\dagger(\tau_n))_{n\in N} \tag{2.4}$$

In fact, every set-indexed family of algebras which satisfies this property can be specified in the computational style we started from. Specifically, given the family of algebras $(\sigma^m_X)^{(m:N)\in E}_X$, we can define $\sigma^m = \sigma^m_N(\eta_N(n))_{n\in N}$ as the computation associated with $m : N$ and rederive the same algebras:

$$n \leftarrow \sigma^N_m(\eta_N(n))_{n\in N} \,;\, \tau_n \;=\; \sigma^m_X(n' \leftarrow \eta_N(n) \,;\, \tau_{n'})_{n\in N} \;=\; \sigma^m_X(\tau_n)_{n\in N}$$

Definition 2.23 (Interpretation into a monad)**.** An *interpretation* of the signature E into the monad T is a family $(\sigma^m)_{(m:N)\in E}$ with $\sigma^m \in T(N)$ for all $(m : N) \in E$. Then the interpretation

$t[\sigma] \in T(X)$ of a term $t \in T_E(X)$ can be recursively defined as:

$$\begin{aligned} \underline{m}(t_n)_{n\in N}[\sigma] &:= n \leftarrow \sigma^m \,;\, t_n[\sigma] && (m : N) \in E \\ \underline{v}[\sigma] &:= \eta_X(v) && v \in X \end{aligned}$$

Note that the functions $(-)[\sigma] : T_E(X) \to T_E(X)$ map variables $\underline{v} \in T_E(X)$ to $\eta^T_X(v) \in T(X)$, and syntactic substitutions tf to the "semantic substitutions" expressed by $(-)^\dagger$ in T:

$$\underline{v}[\sigma] = \eta^T_X(v) \qquad\qquad (tf)[\sigma] = v \leftarrow t[\sigma] \,;\, f(v)[\sigma] \tag{2.5}$$

In that case $(-)[\sigma]$ is called a *monad homomorphisms* (see Definition 2.26 below).

Remark 2.24. *In the context of universal algebra, monads are usually understood as algebraic theories themselves rather than as a kind of model, although we may perhaps interpret a theory into another using a monad homomorphism. We will see in the remainder of this section that the* free monad *for a signature bridges the gap between these two views.*

For my purposes it is useful to retain signatures as the starting point for constructing models, in particular in view of their interpretations as games outlined in §2.5.4. For example, Chapter 6 describes a monad which adds a lattice structure to signatures in a systematic way, constructed by borrowing ideas from game semantics. The presentation above also reveals more clearly the connections between effect signatures and the practical use of monads in functional programming, where the additional operations associated with a specific monad are often given in the style I have used.

2.6.3 Free monad

The terms generated by a signature E can themselves be presented as a monad equipped with an "identity" interpretation of E.

Definition 2.25 (Free monad for a signature)**.** The *free monad* for a signature E is given by the triple $\langle T_E, \eta^E, (-)^\dagger \rangle$ whose components are defined as follows:

$$\begin{aligned} \eta^E_X(v) &:= \underline{v} && v \in X \\ f^\dagger(t) &:= tf && t \in T_E(X),\ f : X \to T_E(Y) \end{aligned}$$

The canonical interpretation of E into T_E is given by the family $(c^m)_{(m:N)\in E}$, where for each operation $m : N$ in the signature E, the elementary term $c^m \in T_E(N)$ is defined by:

$$c^m := \underline{m}(\underline{n})_{n\in N}\,.$$

Note that in the associated algebras, $c^m_X : T_E(X)^N \to T_E(X)$ interprets the operation $m : N$ as the corresponding term constructor; hence $c^m_X = \underline{m}$ and $t[c] = t$.

2.6.4 Monad homomorphisms

The free monad gives us a different way to present monadic interpretations of effect signatures. Note that for an interpretation σ of the signature E into the monad T, the family of functions $(-)[\sigma] : T_E(X) \to T(X)$ preserves the monadic structure in the following way:

$$\begin{aligned} \eta^E_X(v)\,[\sigma] &= \eta^T_X(v) && v \in X \\ (v \leftarrow t\,;\, f(v))[\sigma] &= v \leftarrow t[\sigma]\,;\, f(v)[\sigma] && t \in T_E(X),\ f : X \to T_E(Y) \end{aligned}$$

In other words, it is a monad homomorphism in the following sense.

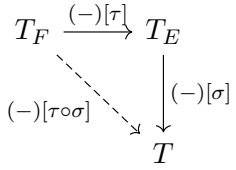
Definition 2.26. A *monad homomorphism* between from $\langle T, \eta^T, (-)^\dagger\rangle$ to $\langle U, \eta^U, (-)^\star\rangle$ is a family of functions $\phi_X : T(X) \to U(X)$ satisfying for all $f : X \to T(Y)$ the following properties:

$$\phi_X \circ \eta^T_X = \eta^U_X \qquad\qquad \phi_Y \circ f^\dagger = (\phi_Y \circ f)^\star \circ \phi_X$$

Conversely, any monad homomorphism $\phi : T_E \to T$ out of the free monad T_E is uniquely defined by an interpretation of E into the monad T, which uses $\phi^m := \phi_N(c^m) \in T(N)$ to interpret each operation $(m : N) \in E$. By induction on $t \in T_E(X)$:

$$\begin{aligned} \underline{v}[\phi] &= \eta^T_X(v) \\ &= \phi_X(\underline{v}) \end{aligned} \qquad\qquad \begin{aligned} \underline{m}(t_n)_{n\in N}[\phi] &= n \leftarrow \phi_N(c^m)\,;\, \phi_X(t_n) \\ &= \phi_X(n \leftarrow c^m\,;\, t_n) \\ &= \phi_X(\underline{m}(t_n)_{n\in N}) \end{aligned}$$

Since monad homomorphisms compose, this means that an interpretation of F into a free monad T_E can be combined with an interpretation of E into an arbitrary monad T to obtain an interpretation of F into T.



This makes interpretations into free monads special.

Definition 2.27 (Interpretation into another signature)**.** An interpretation τ of the signature F into the free monad T_E is also called an interpretation of F into the signature E, and labeled as $\tau : E \to F$. The interpretation τ can be composed with an interpretation σ of E into an arbitrary monad T to yield the interpretation $\tau \circ \sigma$ of F into T defined in the following way:

$$(\tau \circ \sigma)^m := \tau^m[\sigma] \qquad\qquad (m : N) \in F$$

The interpretation τ *uses* terms over the signature E to *provide* an implementation for the operations of F. When T is itself of the form T_D for a signature D so that $\sigma : D \to E$, the composite has type $\tau \circ \sigma : D \to F$. This justifies the choice of the notation $\circ$ which is somewhat abusive in the general case. Note that $t[\tau \circ \sigma] = t[\tau][\sigma]$ for all terms $t \in T_F(X)$.

2.6.5 Interpretations as strategies

As discussed in §2.5.4, signatures can be understood as simple games, and terms in a signature can be understood as a certain kind of strategy for the associated game.

Likewise, an interpretation $\sigma : E \to F$ defines a strategy in a version of the game $E \to F$. The plays for this game are of the form

$$m \cdot \underbrace{\underline{m_1} \cdot n_1 \cdots \underline{m_j} \cdot n_j \cdot \underline{n}}\,.$$

The environment opens with a question $(m : N) \in F$. The system then plays according to the term $\sigma^m \in T_E(N)$, asking a series of questions $m_1 \dots m_j$ in E which the environment answers with $n_1 \dots n_j$. If a variable $\underline{n}$ is reached within σ^m, the corresponding value $n \in N$ is used to

answer the environment's initial question m.

Chapter 3

A refresher on category theory

In general, to keep the exposition as accessible as possible, I will avoid relying on category theory to describe the constructions and methods I use. A reader unfamiliar with category theory should be able to skip this chapter and understand the work presented in the remainder of this thesis.

Nevertheless, category theory allows us to understand in a common language the high-level structures exhibited by various theories. Given the unifying ambition behind refinement-based game semantics, this is a valuable resource and can help guide the design of general-purpose models. Therefore, whenever possible I will mention the categorical structures underlying the constructions I describe.

This chapter is a brief summary of the concepts and definitions of category theory I will use for this purpose, but is not a self-contained introduction. For a more complete and careful treatment, you may want to use following resources:

- a short introduction to category theory for computer scientists is given in Pierce [1991];
- a modern textbook covering the basics is provided by Awodey [2010];
- a standard reference is Mac Lane [1978].

3.1 Motivation

Ultimately, refinement-based game semantics seeks to provide general methods for constructing heterogeneous certified systems, by integrating a wide range of semantic models and verification

techniques. Category theory allows us to understand the commonalities and differences between these models in a unified and systematic way. By describing the compositional structure of a model in categorical terms, we can step back from the details of its construction and focus instead on the abstract, high-level facilities which the model provides. Formulated in the universal language of categories, they can be readily compared with those of similar models, or reveal connections with seemingly unrelated phenomena across distant fields of mathematics.

In many cases, a condensed description of a model's high-level categorical properties will be enough to characterize it up to isomorphism, without reference to the details of its construction. For mechanization in a proof assistant, this can be a very useful *proof engineering* device. In addition, a characterization along these lines will provide evidence that the model is in fact the most general one exhibiting a certain structure, and demonstrate that its construction is free of arbitrary choices which may prove inadvisable at a later point.

Formalizing category theory itself in a proof assistant like Coq can be useful [Spitters and van der Weegen, 2011], but it is a challenging undertaking involving sophisticated techniques, and can steepen the learning curve for the users of a code base. A more mundane approach is to simply spell out the categorical characterization of a given structure. This will give a compact specification which we can nonetheless trust to be complete, with universal properties providing representation-independent reasoning principles for the structure of interest.

Regardless, category theory allows us to develop an understanding of abstract compositional structures in and of themselves, independently of the context in which they may show up. This makes it possible to transfer intuition and apply general, abstract forms of reasoning across a variety of mathematical settings.

3.2 Basic definitions

3.2.1 Categories

Definition 3.1. A *category* $\mathbf{C}$ is a collection of *objects* $A \in \mathbf{C}$ together with a collection of *morphisms* $\mathbf{C}(A, B)$ between any two objects $A, B \in \mathbf{C}$. We write $f : A \to B$ whenever $f \in \mathbf{C}(A, B)$ is a morphism from A to B. For every object $A \in \mathbf{C}$ there is an *identity* morphism $\mathsf{id}_A : A \to A$, and whenever $f : A \to B$ and $g : B \to C$ there is a *composite* morphism $g \circ f : A \to C$

Composition is associative and admits identities as units. In other words, for all morphisms $f : A \to B,\ g : B \to C,\ h : C \to D$, the following properties hold:

$$\mathsf{id}_B \circ f = f \circ \mathsf{id}_A = f \qquad (h \circ g) \circ f = h \circ (g \circ f)\,.$$

Categories are often named after their objects and morphisms, and assigned a short name based on their objects, as in "the category **Vect** of vector spaces and linear maps". The prototypical example is the category **Set** of sets and functions.

Example 3.2. **Set** *is the category whose objects are sets in a given universe, and whose morphisms of type $A \to B$ are the functions from A to B. The identity for $A \in$* **Set** *is the function $\mathsf{id}_A : A \to A$ defined by $\mathsf{id}_A(a) := a$. The composite of the functions $f : A \to B$ and $g : B \to C$ is the function $g \circ f : A \to C$ defined by $(g \circ f)(a) := g(f(a))$.*

Many categories use sets equipped with some additional structure as objects, and structure-preserving functions as morphisms. Categories of this form are known as *concrete categories*.

Example 3.3. *In the category* **Mon** *of monoids and monoid homomorphisms:*

- *The objects are* monoids*, in other words tuples $\langle A, \cdot, \epsilon \rangle$ where A is a set and where the binary operation $\cdot : A \times A \to A$ is associative and admits $\epsilon \in A$ as a unit.*
- *The morphisms are* monoid homomorphisms*. A monoid homomorphism from $\langle A, \cdot_A, \epsilon_A \rangle$ to $\langle B, \cdot_B, \epsilon_B \rangle$ is a function $f : A \to B$ such that $f(x \cdot_A y) = f(x) \cdot_B f(y)$ and $f(\epsilon_A) = \epsilon_B$.*

It is easy to verify that the identity function is a monoid homomorphism and that the composition of two monoid homomorphisms is again a monoid homomorphism.

Table 3.1 lists some categories I will use together with some of their properties.

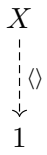
3.2.2 Products

Definition 3.4. A *product* of a collection $(A_i)_{i \in I}$ of objects $A_i \in \mathbf{C}$ is an object $A \in \mathbf{C}$ together with a collection of morphisms $(\pi_i : A \to A_i)_{i \in I}$ satisfying the following property: for any $X \in \mathbf{C}$ and collection of morphisms $(f_i : X \to A_i)_{i \in I}$, there exists a unique morphism $\langle f_j \rangle_{j \in I} : X \to A$ such that $f_i = \pi_i \circ \langle f_j \rangle_{j \in I}$ for all $i \in I$. A category with all finite products is called *cartesian*.

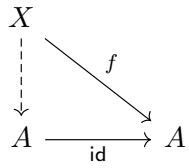
Category	Objects	Morphisms	Mon.		See also
Set	Sets	Functions	$[\times]$		Ex. 3.2
Pos	Partially ordered sets	Monotonic functions	$[\times]$		
Mon	Monoids	Monoid homomorphisms	$\times$		Ex. 3.3
Sup	Complete lattices	Preserve all sups	$\times$	$[\otimes]$	§3.4.2
CDLat	Completely distributive lattices	Complete homomorphisms	$\times$	$\otimes$	
CAL	Layer interfaces	Certified abstraction layers		$\otimes$	Chap. 4
$\mathcal{G}_{\sqsubseteq}^{ib}$	Effect signatures	Innocent strategies	$\times$		Chap. 6
$\mathcal{G}_{\sqsubseteq}^{b}$	Effect signatures	Reentrant strategies		$\otimes$	Chap. 7

Table 3.1: A selection of categories relevant to my work. Categories listed in the upper part of the table are standard; the ones in the lower part are described in following chapters. A cartesian category is indicated by $\times$, with cartesian closure labeled $[\times]$. Categories with tensor products are indicated by $\otimes$, or $[\otimes]$ when I know them to be monoidal closed with respect to the tensor structure.

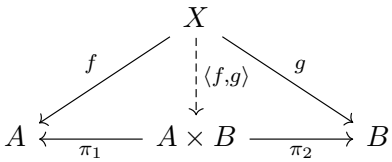
A nullary product is called a *terminal object* and written 1. The definition boils down to the existence of a unique morphism $\langle\rangle : X \to 1$ for each object $X \in \mathbf{C}$:



The unary product of an object A is just itself, with the morphism id_A as the sole "projection":



Binary products are written $A \times B$, and satisfy the following property:



More generally, we can write a finite product $\prod_{1 \le i \le n} A_i$ as $A_1 \times \cdots \times A_n$.

When they exist, products are unique up to isomorphism, so we can talk about *the* product of $(A_i)_{i \in I}$ and refer to the corresponding object as $\prod_{i \in I} A_i$, or in the finitary case use the notations

mentioned above. Note that:

$$1 \times X \cong X \times 1 \cong X \qquad X \times Y \cong Y \times X \qquad (X \times Y) \times Z \cong X \times (Y \times Z)$$

In other words, $\langle \mathbf{C}/\cong, \times, 1 \rangle$ behaves like a commutative monoid, so we do not need to be too careful about parentheses.

Example 3.5 (Products in **Set** and **Mon**). *In* **Set**, *singletons like the unit set* $\mathbb{1} = \{*\}$ *are terminal objects, and the morphism* $\langle\rangle : A \to \mathbb{1}$ *is the only possible function into* $\mathbb{1}$, *defined by* $\langle\rangle(a) := *$. *Binary products are given by sets of pairs:*

$$A \times B := \{(a, b) \mid a \in A \wedge b \in B\} \qquad \pi_1(a, b) := a \qquad \pi_2(a, b) := b$$

Products in **Mon** *extend the definitions above to provide a monoid structure on the underlying sets. The terminal monoid uses* $\mathbb{1}$ *as its underlying set with the monoid structure defined by* $* \cdot_1 * := *$ *and* $\epsilon_1 := *$. *Similarly, the monoid structure on* $A \times B$ *is defined component-wise as:*

$$(x_1, x_2) \cdot_{A \times B} (y_1, y_2) := (x_1 \cdot_A y_1,\ x_2 \cdot_B y_2) \qquad \epsilon_{A \times B} := (\epsilon_A, \epsilon_B)$$

A similar approach can be used for products of arbitrary arities.

The categories listed in the upper part of Table 3.1 are all cartesian. The construction of their products is similar to the one used for **Mon**: we start with the product of the underlying sets, and define the appropriate structure component-wise.

3.2.3 Generalized elements

Terminal objects are useful for representing elements as morphisms. For example in **Set**, there is a one-to-one correspondence between the morphism $e : 1 \to A$ and the element $e(*) \in A$. Combining this with the terminal morphism $\langle\rangle : X \to 1$, we can define the constant function $e \circ \langle\rangle : X \to A$ mapping all elements of a domain set X to the element $e(*)$ of A. More generally, products can be used to represent functions of n arguments as morphisms of type $f : A_1 \times \cdots \times A_n \to B$. We can then compose f with a family $(f_i : X \to A_i)_{1 \leq i \leq n}$ of functions producing arguments of

f as $f \circ \langle f_1, \ldots, f_n \rangle : X \to B$.

In a category with a terminal object 1, the morphisms of type $1 \to A$ are known as the *global elements* of A, but in fact we can extend this intuition to *any* morphism $e : X \to A$ by viewing it as a *generalized* element of A in the context of X. This point of view is used quite literally in categorical models of type theories.

Example 3.6 (Categorical semantics)**.** *Consider a simple type theory, for example the simply-typed lambda calculus, built around a typing judgement of the form*

$$x_1 : A_1,\ \ldots,\ x_n : A_n \vdash M : B$$

which asserts that the term M has type B in a context where each variable x_i has type A_i.

To interpret a type theory of this kind in a cartesian category $\mathbf{C}$, we can assign to each type A an object $[\![A]\!] \in \mathbf{C}$, and to each well-typed term a morphism of type:

$$[\![x_1 : A_1,\ \ldots,\ x_n : A_n \vdash M : B]\!] \ :\ [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![B]\!]\,.$$

The syntactic constructions of the language can then be interpreted in terms of categorical constructions. In particular, given the well-typed terms:

$$\Gamma,\ x : A \vdash M : B \qquad\qquad \Gamma \vdash N : A\,,$$

the substitution $M[x/N]$ of x by N in M should be interpreted as:

$$[\![\Gamma \vdash M[x/N] : B]\!] : [\![\Gamma]\!] \to [\![B]\!] \ =\ [\![\Gamma, x : A \vdash M : A]\!] \circ \langle \mathsf{id}_\Gamma, [\![\Gamma \vdash N : A]\!] \rangle\,.$$

In addition, the language may contain product types, usually defined by the typing rules:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}(M) : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}(M) : B}$$

and the reduction rules:

$$\mathsf{fst}(x, y) \rightsquigarrow x \qquad \mathsf{snd}(x, y) \rightsquigarrow y \qquad (\mathsf{fst}(z), \mathsf{snd}(z)) \rightsquigarrow z$$

Since these rules correspond to the categorical notion of product, we can use the interpretation:

$$\begin{aligned}
[\![A \times B]\!] &:= [\![A]\!] \times [\![B]\!] \\
[\![\Gamma \vdash (M, N) : A \times B]\!] &:= \langle\, [\![\Gamma \vdash M : A]\!],\ [\![\Gamma \vdash N : B]\!] \,\rangle \\
[\![\Gamma \vdash \mathsf{fst}(M) : A]\!] &:= \pi_1 \circ [\![\Gamma \vdash M : A \times B]\!] \\
[\![\Gamma \vdash \mathsf{snd}(M) : B]\!] &:= \pi_2 \circ [\![\Gamma \vdash M : A \times B]\!] \,.
\end{aligned}$$

Defining categorical semantics for programming languages in this way establishes general principles for interpreting a language in any category with the required structure. For example, the κ-calculus [Hasegawa, 1995] can be interpreted along the lines of Example 3.6 in any cartesian category. As such it gives a computational, variable-based syntax which we can use to define morphisms in $\mathbf{Set}$, $\mathbf{Mon}$, and many other categories.

3.3 Adjunctions

3.3.1 Functors

Categories are mathematical structures in their own right, and come with a natural notion of structure-preserving maps. These "homomorphisms of categories" are known as *functors*.

Definition 3.7 (Functor)**.** For two categories $\mathbf{C}$ and $\mathbf{D}$, a *functor* F from $\mathbf{C}$ to $\mathbf{D}$ associates:

- to each object $X \in \mathbf{C}$ an object $FX \in \mathbf{D}$;
- to each morphism $f : A \to B$ in $\mathbf{C}$, a morphism $Ff : FA \to FB$ in $\mathbf{D}$.

Functors must preserve identity and composition, so that for $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$:

$$F(\mathsf{id}_A) = \mathsf{id}_{FA} \qquad F(g \circ f) = Fg \circ Ff \,.$$

I will write $F : \mathbf{C} \to \mathbf{D}$ when F is a functor from $\mathbf{C}$ to $\mathbf{D}$.

The following constructions on categories are useful to define functors in several variables, as well as contravariant functors.

Definition 3.8 (Product and opposite categories)**.** The *product* of the categories $\mathbf{C}$ and $\mathbf{D}$ is the category $\mathbf{C} \times \mathbf{D}$. Its objects are pairs (X, Y) with $X \in \mathbf{C}$ and $Y \in \mathbf{D}$. The morphisms from (A, B) to (C, D) are pairs (f, g) with $f \in \mathbf{C}(A, C)$ and $g \in \mathbf{D}(B, D)$. The composites and identities are defined component-wise.

The *opposite* of a category $\mathbf{C}$ is the category $\mathbf{C}^{\mathsf{op}}$ where the morphisms are reversed: $\mathbf{C}^{\mathsf{op}}$ has the same objects as $\mathbf{C}$ and a morphism $f^{\mathsf{op}} \in \mathbf{C}^{\mathsf{op}}(B, A)$ for every morphism $f \in \mathbf{C}(A, B)$. The identity for $A \in \mathbf{C}^{\mathsf{op}}$ is $\mathsf{id}_A^{\mathsf{op}} : A \to A$, and the composite of $f^{\mathsf{op}} : A \to B$ and $g^{\mathsf{op}} : B \to C$ is the morphism $(f \circ g)^{\mathsf{op}} : A \to C$.

Remark 3.9. *Using these constructions, we can define the* homset *functor for a category* $\mathbf{C}$*, which consists of a functor* $\mathbf{C}(-,-) : \mathbf{C}^{\mathsf{op}} \times \mathbf{C} \to \mathbf{Set}$ *mapping:*

- *an object* $(X, Y) \in \mathbf{C}^{\mathsf{op}} \times \mathbf{C}$ *to the homset* $\mathbf{C}(X, Y) \in \mathbf{Set}$*, and*
- *a morphism* $(f^{\mathsf{op}}, g) : (A, B) \to (C, D)$ *to the function* $\mathbf{C}(f^{\mathsf{op}}, g) : \mathbf{C}(A, B) \to \mathbf{C}(C, D)$ *which takes* $h \in \mathbf{C}(A, B)$ *to* $g \circ h \circ f \in \mathbf{C}(C, D)$*.*

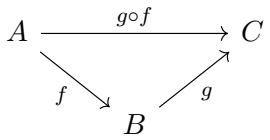
With some care, it is possible to define a category $\mathbf{Cat}$ whose objects are categories and whose morphisms are the functors between two categories. We can in fact go one step further, and define a notion of *natural transformation* between two functors of the same type.

3.3.2 Natural transformations

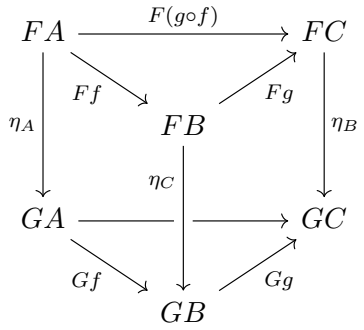
One way to think about functors is to consider the source category as a "shape", and to think of the functor as selecting an instance of this shape in the target category. This point of view is used for example to formalize commutative diagrams as functors, and in the context of other constructions such as limits and colimits.

Under this interpretation, two functors $F, G : \mathbf{C} \to \mathbf{D}$ give instances of the same shape $\mathbf{C}$ in the target category $\mathbf{D}$. A natural transformation $\eta : F \to G$ defines edges between corresponding target objects, connecting the two shapes as the two bases of a prism, and requiring the diagrams

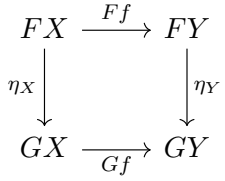
formed by each face to commute. For example, suppose the category $\mathbf{C}$ has three objects, and the following morphisms in addition to identities:



Then a natural transformation $\eta : F \to G$ will induce the following commutative diagram in the category $\mathbf{D}$:



Definition 3.10. A *natural transformation* $\eta : F \to G$ between the functors $F, G : \mathbf{C} \to \mathbf{D}$ is a family of morphisms $\eta_X : FX \to GX$ in $\mathbf{D}$, indexed by $X \in \mathbf{C}$, such that for all morphisms $f : X \to Y$ in $\mathbf{C}$, the *naturality* condition $\eta_Y \circ Ff = Gf \circ \eta_X$ holds:



Another way to gain intuition about natural transformations, especially relevant in the context of programming languages, is to think of them as parametric functions. This relies on the correspondence summarized in Table 3.2. If we think of $\mathbf{D}$ as a category of types and functions, then a functor $F : \mathbf{C} \to \mathbf{D}$, can be thought of as a *generic type* parameterized by the objects of $\mathbf{C}$. A natural transformation $\eta : F \to G$ corresponds to a generic *function* of type:

$$\eta : \forall X \cdot FX \to GX \,,$$

where the naturality condition enforces a form of *parametricity*.

Category theory	Programming
Object	Type
Morphism	Function
Functor	Generic type
Natural transformation	Parametric function
Adjunction	Introduction and elimination rules

Table 3.2: Informal correspondence between category theory and programming concepts

Example 3.11 (Lists)**.** *Consider the case where* $\mathbf{C} = \mathbf{D} = \mathbf{Set}$*. The functor* $(-)^* : \mathbf{Set} \to \mathbf{Set}$ *maps a set* A *to the corresponding set of lists* A^**. Its action on a function* $f : A \to B$ *yields the function* $f^* : A^* \to B^*$ *which applies* f *independently to every element in a list. In other words:*

$$f^*(\vec{a}) := \mathsf{map}(f, \vec{a})\,.$$

The generic function $\mathsf{length} : \forall\, A \cdot A^* \to \mathbb{N}$ *corresponds to a natural transformation* $\eta : (-)^* \to \mathbb{N}$*, where* $\mathbb{N}$ *is understood as the constant functor mapping all objects to* $\mathbb{N}$ *and all functions to* $\mathsf{id}_{\mathbb{N}}$*. Then the naturality condition:*

$$\begin{array}{ccc} A^* & \xrightarrow{f^*} & B^* \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ \mathbb{N} & \xrightarrow{\mathsf{id}} & \mathbb{N} \end{array}$$

expresses that length*, and all parametric functions of the same type, satisfy the property:*

$$\mathsf{length}(\mathsf{map}(f, \vec{x})) = \mathsf{length}(\vec{x})\,.$$

3.3.3 Hom-set adjunction

An adjunction between the categories $\mathbf{C}$ and $\mathbf{D}$ establishes a correspondence between morphisms *out of* certain objects of $\mathbf{C}$ and morphisms *into* certain objects of $\mathbf{D}$. More precisely:

- A functor $F : \mathbf{D} \to \mathbf{C}$ picks out the objects of interest in $\mathbf{C}$;
- A functor $G : \mathbf{C} \to \mathbf{D}$ picks out the objects of interest in $\mathbf{D}$.

The correspondence takes the form of a natural bijection which can be described informally as:

$$\forall XY \cdot \mathbf{C}(FX, Y) \cong \mathbf{D}(X, GY)\,.$$

The notation $F \dashv G$ is often used when F and G are *adjoint functors* in the way outlined above.

Example 3.12 (Cartesian closure in $\mathbf{Set}$)**.** *Given two sets $A, B \in \mathbf{Set}$, the function space B^A is itself a set; in other words, B^A is an object in $\mathbf{Set}$. In fact, there exists a one-to-one correspondence between the morphisms of type $A \to B$ and the global elements of B^A:*

$$\mathbf{Set}(A, B) \;\cong\; \mathbf{Set}(1, B^A)$$

More generally, there is a correspondence between the morphisms of type $\Gamma \times A \to B$ and the generalized elements $\Gamma \to A^B$:

$$\mathbf{Set}(\Gamma \times A,\; B) \;\cong\; \mathbf{Set}(\Gamma, A^B)$$

This correspondence is known as currying*, and is in fact an adjunction $- \times A \;\dashv\; (-)^A$ between:*

- *the functor $- \times A$, which maps an object X to the product $X \times A$, and $f : X \to Y$ to the morphism $f \times \mathsf{id}_A : X \times A \to Y \times A$;*
- *the functor $(-)^A$, which maps an object X to the function space X^A and $f : X \to Y$ to the morphism $f^A : X^A \to Y^A$ which maps $(x_i)_{i \in A}$ to $(y_i)_{i \in A}$ where $y_i := f(x_i)$ for all $i \in A$.*

When they exist, adjoint functors determine each other. F is called the left adjoint of G and conversely G is called the right adjoint of F. If many constructions found across mathematics can be characterized succinctly in the language of category theory, many construction in category theory can in turn be characterized succinctly in terms of adjunctions.

While the definition of adjunction sketched above is fairly compact, much structure can be extracted from it, as summarized in Figure 3.1. This allows us to look at adjunctions from different points of view, which give rise to various alternative definitions.

3.3.4 Universal morphisms

I have mentioned in passing the *naturality* of the bijection associated with an adjunction $F \dashv G$, defined for all $X \in \mathbf{D}$ and $Y \in \mathbf{C}$ as a function:

$$\phi_{X,Y} : \mathbf{C}(FX, Y) \to \mathbf{D}(X, GY)$$

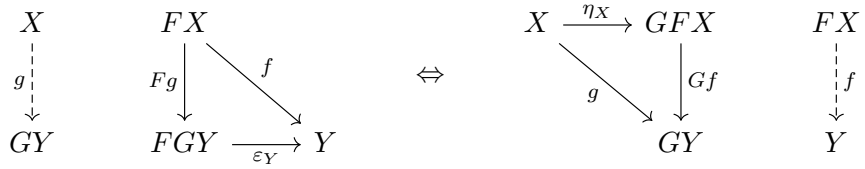


Figure 3.1: Adjunctions in terms of universal morphisms. In each pair of diagram, the left-hand side is a diagram in $\mathbf{D}$ and the right-hand side is a diagram in $\mathbf{C}$.

Once we spell out the details, naturality in this case boils down to the following property:

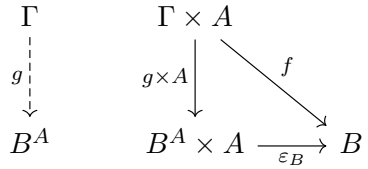
$$\frac{x : X' \to X \in \mathbf{D} \qquad f : FX \to Y \in \mathbf{C} \qquad y : Y \to Y' \in \mathbf{C}}{\phi_{X',Y'}(y \circ f \circ Fx) = Gy \circ \phi_{X,Y}(f) \circ x}$$

This means in particular that:

$$g = \phi_{X,Y}(f) = Gf \circ \phi_{X,FX}(\mathsf{id}_{FX}) \qquad f = \phi^{-1}_{X,Y}(g) = \phi^{-1}_{GY,Y}(\mathsf{id}_{GY}) \circ Fg$$

The morphism $\eta_X := \phi_{X,FX}(\mathsf{id}_{FX}) : X \to GFX$ defines a natural transformation called the adjunction's *unit*, and the morphism $\varepsilon_Y := \phi^{-1}_{GY,Y}(\mathsf{id}_{GY}) : FGY \to Y$ defines the *counit*. They satisfy the universal properties shown in Figure 3.1.

Example 3.13 (Universal property of exponentials)**.** *In the case of the adjunction $- \times A \dashv (-)^A$, the universal property of the counit is particularly interesting:*

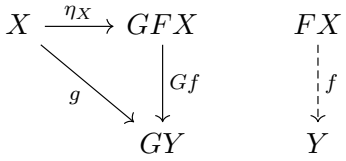


The counit $\varepsilon_B : B^A \times A \to B$ is called an evaluation map*. It applies a function in B^A to an element of A and returns the result in B.*

The universal properties of the unit and counit each suffice to characterize the adjunction. We give a definition below in terms of the universal property of the unit.

Definition 3.14 (Adjoint functors)**.** The functors $F : \mathbf{D} \to \mathbf{C}$ and $G : \mathbf{C} \to \mathbf{D}$ define an adjunction when there is a natural transformation $\eta : \mathsf{id}_{\mathbf{D}} \to GF$ with the following property:

for all $g : X \to GY$ in $\mathbf{D}$, there is a unique $f : FX \to Y$ in $\mathbf{C}$ such that $g = Gf \circ \eta_X$:



This definition is particularly useful to characterize *free* functors, which are left adjoints of *forgetful* functors.

Example 3.15 (Free objects)**.** *Consider the functor* $U : \mathbf{Mon} \to \mathbf{Set}$ *which forgets the structure of a monoid to retain only the underlying set. Likewise,* U *forgets the properties of a monoid homomorphism to retain only the underlying function. Given explicitly,*

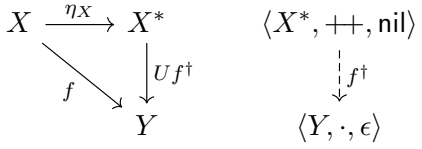
$$U\langle Y, \cdot, \epsilon\rangle := Y \qquad\qquad Uf := f\,.$$

This functor has a left adjoint $F : \mathbf{Set} \to \mathbf{Mon}$*, which for a set* X *constructs the corresponding monoid of lists. The action of* F *on a function* $f : X \to Y$ *is given by the* map *operation, which applies* f *independently to each element of the list.*

$$FX := \langle X^*, +\!\!+, \mathsf{nil}\rangle \qquad\qquad Ff := \lambda\vec{x} \cdot \mathsf{map}(f, \vec{x})\,.$$

The monoid FX *is the* free monoid *generated by the elements of* X*. Lists correspond to arbitrary monoid expressions built from these generators, identified up to the associativity and unit laws which monoids must obey. This makes* FA *unique: it is in a sense the most general monoid containing* A*.*

The unit $\eta_X : X \to X^*$ *embeds the generator* $x \in X$ *as the single-element list* $\eta_X(x) = x :: \mathsf{nil}$*. The defining property of the free monoid is that a function* $f : X \to U\langle Y, \cdot, \epsilon\rangle$ *can be uniquely extended to a monoid homomorphism* $f^\dagger : FX \to \langle Y, \cdot, \epsilon\rangle$ *such that* $Uf^\dagger \circ \eta_X = f$*, namely:*



$$f^\dagger(x :: \vec{y}) := f(x) \cdot f^\dagger(\vec{y}) \qquad f^\dagger(\mathsf{nil}) := \epsilon$$

3.3.5 Cartesian closure

Generally speaking, a category $\mathbf{C}$ is *closed* when its collections of morphisms $\mathbf{C}(X, Y)$ can be represented internally as an object of $\mathbf{C}$ itself. In *cartesian closed* categories, this correspondence is expressed using products. I discussed cartesian closure somewhat informally in Examples 3.12 and 3.13. Armed with Definition 3.14 we can formulate the following characterization.

Definition 3.16 (Cartesian closed category)**.** An object A in a cartesian category $\mathbf{C}$ is called *exponentiable* if the functor $- \times A$ has a right adjoint $(-)^A$. A cartesian category where all objects are exponentiable is called *cartesian closed*.

Cartesian closure provides a categorical notion of high-order functions which can be introduced by currying and eliminated through uncurrying. Cartesian closed categories are the models of the simply typed λ-calculus.

Example 3.17 (Simply-typed λ-calculus)**.** *In addition to the structures mentioned in Example 3.6, the simply typed λ-calculus has types of the form $A \to B$ with the associated rules:*

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \cdot M : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash f\, x : B}$$

and reductions:

$$(\lambda x \cdot M)\, N \rightsquigarrow M[x/N] \qquad (\lambda x \cdot f\, x) \rightsquigarrow f$$

In a cartesian closed category, we can give these constructions the following interpretation:

$$\begin{aligned} [\![A \to B]\!] &:= B^A \\ [\![\Gamma \vdash \lambda x \cdot M : A \to B]\!] &:= [\![\Gamma, x : A \vdash M : B]\!]^\dagger \\ [\![\Gamma \vdash f\, x : B]\!] &:= \varepsilon_B \circ \langle\, [\![\Gamma \vdash f : A \to B]\!],\ [\![\Gamma \vdash x : A]\!] \,\rangle\,. \end{aligned}$$

3.3.6 Monads

I have given in §2.6 an overview of monads in $\mathbf{Set}$ and their connection with algebraic effects. The general categorical formulation is the following.

Definition 3.18 (Monad)**.** A monad in $\mathbf{C}$ is a triple $\langle T, \eta, \mu \rangle$ consisting of a functor $T : \mathbf{C} \to \mathbf{C}$,

a natural transformations $\eta : \mathsf{id}_{\mathbf{C}} \to T$ called the monad's *unit*, and a natural transformation $\mu : TT \to T$ called its *multiplication*, which satisfy for all $X \in \mathbf{C}$ the following properties:

$$\begin{array}{ccc} T^3X & \xrightarrow{\mu_{TX}} & T^2X \\ {\scriptstyle T\mu_X}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ T^2X & \xrightarrow[\mu_X]{} & TX \end{array} \qquad \begin{array}{ccc} TX & \xrightarrow{\eta_{TX}} & T^2X \\ {\scriptstyle T\eta_X}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ T^2X & \xrightarrow[\mu_X]{} & TX \end{array}$$

$$\mu_X \circ T\mu_X = \mu_X \circ \mu_{TX} \qquad \mu_X \circ T\eta_X = \mu_X \circ \eta_{TX} = \mathsf{id}_{TX}\,.$$

Note that a monad in the sense of Definition 2.22 corresponds to a monad in **Set** in the sense of the definition above:

- The action of T on a function $f : X \to Y$ can be defined as $Tf := (\eta_Y \circ f)^\dagger$.
- The multiplication is given as $\mu_X := \mathsf{id}^\dagger_{TX}$.

Conversely, for any monad T and morphism $f : X \to TY$, its Kleisli extension $f^\dagger : TX \to TY$ can be defined as $f^\dagger := \mu \circ Tf$.

Monads are in a sense "flattened adjunctions". In particular, every adjunction $F \dashv G$ gives rise to a monad $\langle GF, \eta, \mu\rangle$ where μ is defined from ε as $\mu_X := G\varepsilon_{FX}$. A number of familiar monads can be obtained in this way from the adjunctions that I have presented. The adjunction associated with the cartesian closure of **Set** discussed in Example 3.12 gives rise to the *state* monad $(- \times S)^S$. The free monoid adjunction gives rise to the *list* monad $(-)^*$. In the next section, I discuss how the free monad T_E presented in §2.6.3 arises from an adjunction between the category **Set** and the category of algebras for the signature E.

3.3.7 Categorical algebra

I discussed in §2.5 some basic concepts from universal algebra, and their use in the context of algebraic effects. These concepts admit an elegant categorical formulation [Trnková et al., 1975], allowing us to generalize them to categories other than **Set**, and to establish a formal connection with the monadic approach to modeling effectful computations.

Recall that an algebra for the signature E is a set A together with a function $\alpha_m : A^N \to N$

for each operation $(m : N) \in E$. Equivalently, we can give a single function

$$\alpha \;:\; \left(\sum_{(m:N)\in E} A^N \right) \;\to\; A\,.$$

For example, an algebra for the signature $E_{\mathsf{io}} = \{\mathsf{readbit} : \mathbb{2},\ \mathsf{print} : \mathsf{string} \to \mathbb{1},\ \mathsf{done} : \varnothing\}$ will have the following type:

$$\alpha \;:\; A \times A \;+\; \left(\sum_{s\in\mathsf{string}} A \right) \;+\; 1 \;\to\; A\,,$$

where in the domain of α:

- The first summand corresponds to the operation $\mathsf{readbit} : \mathbb{2}$.
- The second one corresponds to the family of operations $\mathsf{print} : \mathsf{string} \to \mathbb{1}$.
- The last one corresponds to $\mathsf{done} : \varnothing$.

In fact, the "shape" of algebras for E_{io} can be captured by a functor $\hat{E}_{\mathsf{io}} : \mathbf{Set} \to \mathbf{Set}$ defined as:

$$\hat{E}_{\mathsf{io}}X \;:=\; X \times X \;+\; \left(\sum_{s\in\mathsf{string}} X \right) \;+\; 1$$

An algebra for E_{io} is then a set A with a function $\alpha : \hat{E}_{\mathsf{io}}A \to A$. This generalizes as follows.

Definition 3.19. The functor associated to an effect signature E is $\hat{E} : \mathbf{Set} \to \mathbf{Set}$, defined as:

$$\hat{E}X := \sum_{(m:N)\in E} X^N\,.$$

Once a signature has been encoded as a functor, we can use the following notion of algebra to represent the associated structure.

Definition 3.20 (F-algebras)**.** An algebra for the functor $F : \mathbf{C} \to \mathbf{C}$ or *F-algebra* is an object $A \in \mathbf{C}$ together with a morphism $\alpha : FA \to A$. F-algebras form a category $F\mathbf{Alg}$ in the following way. An object of $F\mathbf{Alg}$ is a pair $\langle A, \alpha\rangle$ as above. A *homomorphism* of F-algebras from

$\langle A, \alpha\rangle$ to $\langle B, \beta\rangle$ is a morphism $f : A \to B$ of $\mathbf{C}$ such that $f \circ \alpha = \beta \circ Ff$.

$$\begin{array}{ccc} \hat{E}A & \xrightarrow{\alpha} & A \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ \hat{E}B & \xrightarrow[\beta]{} & B \end{array}$$

The identities of $\mathbf{C}$ are identity homomorphisms of F-algebras. Homomorphisms of F-algebras likewise compose as expected.

For a signature E, an algebra homomorphisms between the $\hat{E}$-algebras $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ is a function $f : A \to B$ such that for all operations $(m : N) \in E$ and argument tuples $x \in A^N$:

$$f(\alpha_m(x_n)_{n\in N}) = \beta_m(f(x_n))_{n\in N}\,.$$

Example 3.21 (Algebra of closed terms)**.** *The set of terms for a signature E can be used to define an algebra $\langle T_E(\varnothing), e\rangle$, where for all $(m : N) \in E$ and family of closed terms $(t_n)_{n\in N}$, we define:*

$$e_m(t_n)_{n\in N} := \underline{m}(t_n)_{n\in N}\,.$$

Terms are interpreted as themselves; in other words, $t^e = t$ for all $t \in T_E(\varnothing)$. This algebra is in fact the initial object *in the category $\hat{E}\mathbf{Alg}$: for any other $\langle A, \alpha\rangle \in \hat{E}\mathbf{Alg}$, the interpretation function $(-)^\alpha : T_E(\varnothing) \to A$ defines the unique $\hat{E}$-algebra homomorphism from $\langle T_E(\varnothing), e\rangle$ to $\langle A, \alpha\rangle$.*

More generally, note that for any functor $F : \mathbf{C} \to \mathbf{C}$, we can define a forgetful functor $U_F : F\mathbf{Alg} \to \mathbf{C}$. Its action on algebras is $U_F\langle A, \alpha\rangle := A$, retaining only the underlying object of an F-algebra. Likewise, its action on algebra homomorphisms retains only the underlying morphism of $\mathbf{C}$. In the case of $\hat{E}$-algebras, T_E can be characterized as the left adjoint of U_E.

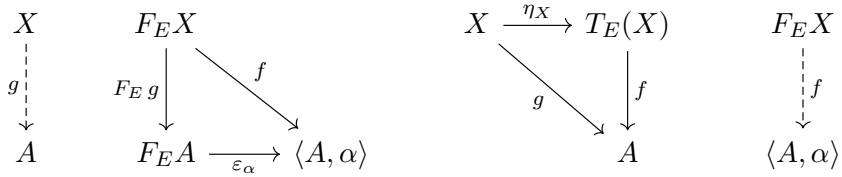
Example 3.22 (T_E as a left adjoint)**.** *The set $T_E(X)$ of terms on a signature E with variables in X can be used to define a functor $F_E : \mathbf{Set} \to \hat{E}\mathbf{Alg}$ with $F_E X := \langle T_E(X), e\rangle$, where the function $e : \hat{E}\,T_E(X) \to T_E(X)$ is defined as above by:*

$$e_m(t_n)_{n\in N} := \underline{m}(t_n)_{n\in N} \qquad (m : N) \in E\,.$$

The action of F_E on a function $g : X \to Y$ gives a $\hat{E}$-algebra homomorphism $\hat{g} : F_E X \to F_E Y$ which applies g to all variables in a term:

$$\hat{g}(\underline{m}(t_n)_{n\in N}) := \underline{m}(\hat{g}(t_n))_{n\in N} \qquad \hat{g}(\underline{x}) := \underline{g(x)}$$

There is an adjunction $F_E \dashv U_E$ with the following components:



In other words, there is a one-to-one correspondence between the variable assignments $g : X \to A$ and algebra homomorphisms of the form $f : F_E X \to \langle A, \alpha\rangle$ which map open terms with variables in X to the carrier set A and satisfy the following homomorphism property:

$$f(\underline{m}(t_n)_{n\in N}) = \alpha_m(f(t_n))_{n\in N}\,.$$

This property determines the behavior of f on operations. The behavior on variables is defined by the property $f(\underline{v}) = g(v)$. The adjunction's unit maps $v \in X$ to $\underline{v} \in T_E(X)$, and the counit maps a term $t \in T_E(A)$ to its interpretation $t^\alpha \in A$.

3.4 Monoidal structures

3.4.1 Motivation

In categories with products, systems which use and provide multiple interfaces can be represented as morphisms of type:

$$f : A_1 \times \cdots \times A_n \to B_1 \times \cdots \times B_m\,.$$

In particular, two morphisms $f_1 : A_1 \to B_1$ and $f_2 : A_2 \to B_2$ can be combined side-by-side as:

$$f_1 \times f_2 := \langle f_1 \circ \pi_1, f_2 \circ \pi_2\rangle : A_1 \times A_2 \to B_1 \times B_2\,.$$

As noted in §3.2.2, products essentially behave as a commutative monoid on the objects of a category. In the "systems" interpretation, this means that a multiset of interfaces can be combined to define a larger interface. The properties of products correspond to that of "wiring diagrams", allowing systems to be combined in series through the usual composition of morphisms, and in parallel through the operation $\times$ shown above.

However, products also come with additional structure and properties which are not always appropriate when modeling systems. For example, for every object $A \in \mathbf{C}$, there is a morphism:

$$\Delta_A := \langle \mathsf{id}_A, \mathsf{id}_A \rangle : A \to A \times A$$

with the property that $\pi_1 \circ \Delta_A = \pi_2 \circ \Delta_A = \mathsf{id}_A$. This means that every interface can be duplicated and shared between several clients with no interference. This is appropriate for some simple systems; for instance the value produced by a function can be duplicated or discarded at will and used as an input by an arbitrary number of other functions. However, systems which are stateful or manipulate resources do not usually behave in this way. For instance, invoking the primitives of an abstraction layer will transform its state. Two clients accessing the same layer interface may interfere with each other; because of this it is not possible in general to duplicate a layer interface in a transparent way (see also §4.3.3 and §4.3.4).

3.4.2 Example: nondeterministic functions

Below I illustrate how these phenomena are expressed in the context of the category **Sup** of suplattices and their homomorphisms. This category is particularly relevant to the work presented in this thesis because it provides a possible model of unbounded nondeterminism and nondeterministic functions.

Definition 3.23. A *sup-lattice* is a poset L with all least upper bounds. In other words, for every family $(x_i)_{i \in I}$ of elements of L, there is an element $\bigvee_{i \in I} x_i \in L$ with the property that:

$$\forall\, i \in I \cdot x_i \leq y \quad \Leftrightarrow \quad \bigvee_{i \in I} x_i \;\leq\; y$$

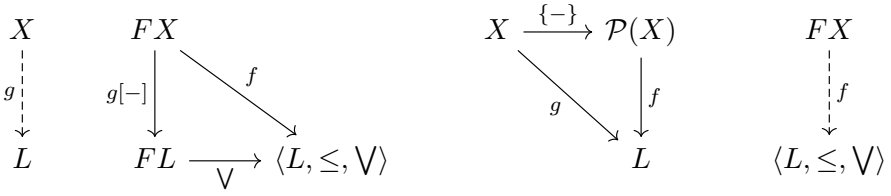


Figure 3.2: Adjunction for the free sup-lattice $FX = \langle \mathcal{P}(X), \subseteq, \bigcup \rangle$. The unit maps $x \in X$ to the singleton set $\{x\} \in \mathcal{P}(X)$. The counit "evaluates" a subset $A \subseteq L$ of the sup-lattice $\langle L, \leq, \bigvee \rangle$ to its least upper bound $\bigvee A \in L$.

A sup-lattice homomorphism from $\langle L, \leq, \bigvee \rangle$ to $\langle M, \sqsubseteq, \bigsqcup \rangle$ is a function $f : L \to M$ such that:

$$f\left(\bigvee_i x_i\right) = \bigsqcup_i f(x_i)\,.$$

I will write **Sup** for the category of sup-lattices and their homomorphisms.

In any sup-lattice $\langle L, \leq, \bigvee \rangle$, we can compute the least upper bound of a subset $X \subseteq L$ as the supremum $\bigvee X = \bigvee_{x \in X} x$. Binary joins are computed as $x \vee y = \bigvee \{x, y\}$. Every sup-lattice L has a least element $\bot = \bigvee \varnothing$ and a greatest element $\top = \bigvee L$. Note that sup-lattice homomorphisms preserve binary joins and $\bot$, but do not necessarily preserve $\top$ because the image of L may not be the entire target lattice. Since $x \leq y \;\Leftrightarrow\; x \vee y = y$, sup-lattice homomorphisms are always monotone.

Sup-lattices are in fact *complete lattices*, since arbitrary meets can be computed in any sup-lattice as $\bigwedge_{i \in I} x_i = \bigvee \{x \mid \forall i \in I \cdot x \leq x_i\}$. However, complete lattices are usually associated with *complete homomorphisms*, which preserve both arbitrary joins and arbitrary meets, whereas viewing complete lattices as a sup-lattices yields the weaker notion of sup-lattice homomorphism used to define the category **Sup**.

Free sup-lattice and powerset monad There is a forgetful functor $U : \mathbf{Sup} \to \mathbf{Set}$ which maps a sup-lattice $\langle L, \leq, \bigvee \rangle$ to its underlying set L. This functor has a left adjoint $F : \mathbf{Set} \to \mathbf{Sup}$ which maps a set X to the sup-lattice $FX = \langle \mathcal{P}(X), \subseteq, \bigcup \rangle$ of subsets of X ordered by inclusion. The action of F on a function $f : X \to Y$ is defined by the image $f[A] = \{f(a) \mid a \in A\}$ of a subset $A \in \mathcal{P}(X)$ under f. The components of the adjunction are summarized in Figure 3.2.

Cartesian structure The category **Sup** has all products $\prod_{i\in I}\langle L_i, \leq_i, \bigvee\rangle$. The underlying set of a product is the cartesian product $\prod_{i\in I} L_i$ of underlying sets, so that its elements have the form $(x_i)_{i\in I}$ with $x_i \in L_i$. Their ordering is defined component-wise by

$$(x_i)_{i\in I} \leq (y_i)_{i\in I} \;\Leftrightarrow\; \forall i \in I \cdot x_i \leq_i y_i \,,$$

and joins can likewise be computed as follows:

$$\bigvee_{j\in J} (x_i^j)_{i\in I} \;=\; \left(\bigvee_{j\in J} x_i^j\right)_{i\in I}$$

In particular, the bottom element of a product is the tuple $\bot = (\bot_i)_{i\in I}$. The terminal object of **Sup** is the trivial sup-lattice $1 = \{\bot\}$, and the terminal sup-lattice homomorphism $\langle\rangle_L : L \to 1$ maps every element of L to $\bot$.

Products in **Sup** behave in a rather unexpected way. For example, a global element $e : 1 \to L$ carries no information, because as a sup-lattice homomorphism e can only map $\bot \in 1$ to $\bot \in L$. A related phenomenon is that within the sup-lattice $L_1 \times L_2$, the tupling operation $(-,-)$ is only a sup-lattice homomorphism jointly in the two variables: although $(\bot, \bot) = \bot$, in general $(\bot, x_2) \neq \bot$ and $(x_1, \bot) \neq \bot$. If the global elements behaved in the usual way, this would be a paradox because it would be possible to construct a morphism $\mathsf{id} \times e_2 : L_1 \times 1 \to L_1 \times L_2$ which would not be a sup-lattice homomorphism. But since $e_2 : 1 \to L_2$ can only represent the element $\bot \in L_2$ this is not a problem.

The following, related phenomenon also occurs. The hom-sets $\mathbf{Sup}(X, Y)$ can be equipped with a sup-lattice structure, where the ordering and joins are defined pointwise as:

$$f \leq g \Leftrightarrow \forall x \in X \cdot f(x) \leq g(x) \qquad \left(\bigvee_{i\in I} f_i\right)(x) = \bigvee_{i\in I} f_i(x)$$

Therefore, we expect **Sup** to be *closed* in some way, with $[X, Y] = \langle \mathbf{Sup}(X, Y), \leq, \bigvee\rangle$ as defined above representing the hom-set $\mathbf{Sup}(X, Y)$ as an object in the category **Sup** itself. However, **Sup** cannot be *cartesian* closed with respect to $[X, Y]$. For example there cannot be in general a correspondence between the single morphism of type $1 \to [X, Y]$ and the potentially many

morphisms of type $X \to Y$.

While the cartesian structure of **Sup** does not support elements and closure in the expected way, it turns out we can recover them by using alternative constructions which behave similarly in some aspects. This alternative *monoidal* structure on **Sup** can be defined using the *tensor product* of sup-lattices [Joyal and Tierney, 1984, Chapter I].

Tensor product Tensor products were first studied in the context of linear algebra, but they can be defined for many structures other than vector spaces, and they admit a general categorical description. Tensor products of vector spaces emerge from the notion of *bilinear* maps. Likewise, the more general notion of tensor product emerges from a notion of *bimorphism* [Banaschewski and Nelson, 1976].

In the context of **Sup**, a *bimorphism* from the sup-lattices L_1 and L_2 to the sup-lattice M is a function $f : UL_1 \times UL_2 \to UM$ in **Set**, which satisfies the following properties:

$$\begin{aligned} f(\bigvee_{i\in I} x_i,\, y) &= \bigvee_{i\in I} f(x_i, y) \\ f(x,\, \bigvee_{i\in I} y_i) &= \bigvee_{i\in I} f(x, y_i)\,, \end{aligned}$$

Note that f is not in general a sup-lattice homomorphism. The defining property of the tensor product $L_1 \otimes L_2$ is the existence of a bimorphism $h : UL_1 \times UL_2 \to U(L_1 \otimes L_2)$, universal in the sense that all bimorphisms f out of L_1 and L_2 factor through h in the following way:

$$\begin{array}{ccc} UL_1 \times UL_2 & \xrightarrow{\;h\;} & U(L_1 \otimes L_2) \\ & \searrow^{f} & \downarrow{\scriptstyle Uf^\dagger} \\ & & UM \end{array} \qquad\qquad \begin{array}{c} L_1 \otimes L_2 \\ \vdots \\ \downarrow{\scriptstyle f^\dagger} \\ M \end{array} \tag{3.1}$$

This means in particular that bimorphisms are in one-to-one correspondence with homomorphisms out of the corresponding tensor object.

One way to construct the tensor product is to start from the free sup-lattice $F(UL_1 \times UL_2)$ with $h(x, y) = \{(x, y)\}$, then identify elements of $L_1 \otimes L_2$ to make h into a bimorphism. Often, the notation $\otimes$ is used for elements of the tensor product as well as the object, in other words

$h(x, y)$ is written as $x \otimes y \in L_1 \otimes L_2$. A sup-lattice homomorphism $f^\dagger$ out of the tensor product $L_1 \otimes L_2$ can be defined by giving its action on elements of the form $x \otimes y$, and verifying that $f^\dagger(- \otimes y)$ and $f^\dagger(x \otimes -)$ are sup-lattice homomorphisms.

Monoidal structure The tensor product is in fact a *functor* $\otimes : \mathbf{Sup} \times \mathbf{Sup} \to \mathbf{Sup}$. To define its action $f \otimes g : L_1 \otimes L_2 \to M_1 \otimes M_2$ on the morphisms $f : L_1 \to M_1$ and $g : L_2 \to M_2$, it suffices to specify

$$(f \otimes g)(x \otimes y) := f(x) \otimes g(x)\,.$$

Note that since f and g are sup-lattice homomorphisms and $\otimes$ is a bimorphism, $f(x) \otimes g(y)$ is a sup-lattice homomorphism separately in x and y.

The tensor product $\otimes$ shares various characteristics with the cartesian product $\times$ besides its functoriality. In particular, the sup-lattice $I = \{\bot, *\}$ serves as a unit for $\otimes$, which satisfies the following isomorphisms:

$$I \otimes X \cong X \otimes I \cong X \qquad X \otimes Y \cong Y \otimes X \qquad (X \otimes Y) \otimes Z \cong X \otimes (Y \otimes Z)$$

Finally, the tensor product in $\mathbf{Sup}$ has some of the characteristics which the cartesian product failed to deliver. For example, the generalized elements $e : I \to L$ are in one-to-one correspondence with the elements $e(*) \in L$. Moreover, there is an adjunction $- \otimes A \dashv [A, -]$ which allows us to recover a tensor version of the cartesian closed structure present for example in $\mathbf{Set}$. This also provides an alternative characterization of the tensor product.

The properties of $\mathbf{Sup}$ illustrate a more general phenomenon. In categories like $\mathbf{Set}$ and $\mathbf{Pos}$, all morphisms out of a product $A \times B$ are "bifunctions" or "bimonotonic functions". Because of this, the notion of tensor product collapses into the usual cartesian product. By contrast, in $\mathbf{Sup}$ morphisms and bimorphisms are two very different things, giving rise to a richer setting where these two structures interact in interesting ways. These structures mirror those found in linear algebra and are captured by various fragments of linear logic.

3.4.3 Symmetric monoidal categories

Generalizing from the cartesian and tensor products, the notion of symmetric monoidal category is defined as follows.

Definition 3.24. A *symmetric monoidal* category $\mathbf{C}$ is equipped with a functor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, a *unit object* $I \in \mathbf{C}$ and the following natural isomorphisms:

- a *left unitor* $\lambda_X : I \otimes X \cong X$ and *right unitor* $\rho_X : X \otimes I \cong X$;
- a *braiding* $\gamma_{X,Y} : X \otimes Y \cong Y \otimes X$ such that $\gamma_{Y,X} = \gamma_{X,Y}^{-1}$;
- an *associator* $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \cong X \otimes (Y \otimes Z)$.

The unitors and braiding must be compatible in the sense that $\rho_X \circ \gamma_{I,X} = \lambda_X$.

$$\begin{array}{ccc} I \otimes X & \xrightarrow{\gamma} & X \otimes I \\ {\scriptstyle\lambda}\downarrow & & \downarrow{\scriptstyle\rho} \\ X & = & X \end{array}$$

The associator must obey $(\mathsf{id}_W \otimes \alpha_{X,Y,Z}) \circ \alpha_{W,X\otimes Y,Z} \circ (\alpha_{W,X,Y} \otimes \mathsf{id}_Z) = \alpha_{W,X,Y\otimes Z} \circ \alpha_{W\otimes X,Y,Z}$.

$$\begin{array}{ccccc} ((W \otimes X) \otimes Y) \otimes Z & \xrightarrow{\alpha} & (W \otimes X) \otimes (Y \otimes Z) & \xrightarrow{\alpha} & W \otimes (X \otimes (Y \otimes Z)) \\ {\scriptstyle\alpha\otimes\mathsf{id}}\downarrow & & & & \uparrow{\scriptstyle\mathsf{id}\otimes\alpha} \\ (W \otimes (X \otimes Y)) \otimes Z & & \xrightarrow{\qquad\qquad\alpha\qquad\qquad} & & W \otimes ((X \otimes Y) \otimes X) \end{array}$$

Finally, the braiding and associator must be compatible in the sense that

$$(\mathsf{id}_Y \otimes \gamma_{X,Z}) \circ \alpha_{Y,X,Z} \circ (\gamma_{X,Y} \otimes \mathsf{id}_Z) = \alpha_{Y,Z,X} \circ \gamma_{X,Y\otimes Z} \circ \alpha_{X,Y,Z}\,.$$

$$\begin{array}{ccccc} (X \otimes Y) \otimes Z & \xrightarrow{\alpha} & X \otimes (Y \otimes Z) & \xrightarrow{\gamma} & (Y \otimes Z) \otimes X \\ {\scriptstyle\gamma\otimes\mathsf{id}}\downarrow & & & & \downarrow{\scriptstyle\alpha} \\ (Y \otimes X) \otimes Z & \xrightarrow{\alpha} & Y \otimes (X \otimes Z) & \xrightarrow{\mathsf{id}\otimes\gamma} & Y \otimes (Z \otimes X) \end{array}$$

3.4.4 Monoidal closure

In the context of symmetric monoidal categories, Definition 3.16 generalizes as follows.

Definition 3.25 (Monoidal closed category)**.** A symmetric monoidal category $\mathbf{C}$ is *closed* when for every object $A \in \mathbf{C}$ the functor $- \otimes A$ has a right adjoint $[A, -]$.

Part II

Refinement-based game semantics

Chapter 4

Certified abstraction layers

4.1 Introduction

This chapter describes the theory of certified abstraction layers used in the verification of the certified operating system kernel CertiKOS.

4.1.1 Abstraction layers

Software is constructed in layers. The basic, concrete facilities provided by the programming environment are used to implement abstract data structures and operations. The programmer can then forget the implementation details of these data structures and operations, and instead think of them as *primitives* 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 manipulated explicitly. For higher-level code, the operating system's memory management layers abstract away the hardware's low-level details and provide a more convenient view of the memory.

This illustrates a conceptual similarity between abstraction layers and compilers. As is the case for abstraction layers, the purpose of a compiler is to use a lower-level programming model (like assembly code) to provide a more abstract, higher-level model (for example the C programming language). For abstraction layers, the transformation of client code is less expansive, consisting only of linking with a layer's code. Nevertheless compilers and abstraction layers can be under-

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

Table 4.1: Correspondence between certified compilation and certified abstraction layers

stood as specific instances of a more general phenomenon.

4.1.2 Certified abstraction layers in CertiKOS

The analogy between compilers and abstraction layers can be extended to cover correctness and verification techniques. This is illustrated by the *certified abstraction layers* of CertiKOS and their connection to CompCert. The verification of CertiKOS builds on CompCert in two different ways:

- In terms of functionality, CompCert provides C and assembly semantics mechanized in the Coq proof assistant, and its correctness theorem allows us to use proofs about the kernel's C code to verify the compiled assembly code.
- Additionally, many of the patterns and techniques used to verify CompCert serve as a starting point for the formulation of certified abstraction layers (Table 4.1).

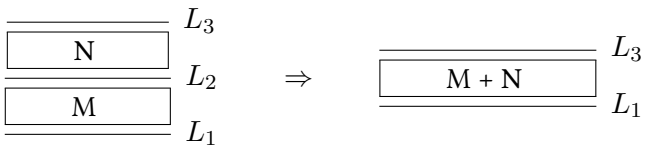
The CertiKOS kernel is divided into several dozen abstraction layers, which are specified and verified individually. Specifications of abstraction layers are called *layer interfaces*. They extend CompCert semantics to provide a set of *primitives*, which can be invoked as external functions. The behavior 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 realizes the primitives described by an *overlay* interface as client code for an *underlay* 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 refines 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 contextual refinement properties can be combined to obtain the composite certified layer:



This is similar to the way the passes of CompCert can be composed when the target language of one corresponds to the source language of another.

4.1.3 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 §4.4 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 could be interfaced with CompCertO (§9) to provide the capabilities of the original model and more.

4.2 Layer model

A layer interface L has three components. First, a *signature* E enumerates primitive operations together with their types (see Def. 2.16). Second, the set S contains the *abstract states* of the layer interface. Finally, for each operation $m : N$ in the signature E, a *specification* is given as a function:

$$L.m : S \to \mathcal{P}^1(N \times S)\,.$$

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 $k \in S$.

4.2.1 Specification monad

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

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

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

$$\eta^{\mathcal{P}}_X(v) := \{v\}\,,$$

and where the empty set $\varnothing \in \mathcal{P}^1(X)$ 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 $x \in \mathcal{P}^1(\mathcal{P}^1(X))$, the multiplication $\mu^{\mathcal{P}}(x) \in \mathcal{P}^1(X)$ is defined by:

$$\mu^{\mathcal{P}}_X(x) := \{v \in X \mid \exists\, y \in \mathcal{P}^1(X) \cdot x \ni y \ni v\} = \bigcup x\,.$$

Equivalently, for $x \in \mathcal{P}^1(X)$ and $f : X \to \mathcal{P}^1(Y)$, the Kleisli extension $f^\dagger : \mathcal{P}^1(X) \to \mathcal{P}^1(Y)$ is defined by:

$$f^\dagger(x) = \{v \in Y \mid \exists\, u \in X \cdot x \ni u \ \wedge\ f(u) \ni v\} = \bigcup_{u \in x} f(u)\,.$$

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 combine $\mathcal{P}^1$ with the state monad transformer. For a set of states S, we obtain the monad:

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

equipped with the structure:

$$\eta^{\mathcal{L}}(v) := \lambda\, k \cdot \{v@k\}$$
$$\mu^{\mathcal{L}}(x) := \lambda\, k \cdot \{v@k'' \mid \exists\, y\, k' \cdot x(k) \ni y@k' \wedge y(k') \ni v@k''\}\,.$$

Then in a layer interface L using abstract states in the set S, the primitive $m : N$ will be given a specification $L.m \in \mathcal{L}_S(N)$.

Remark 4.1. *Our version of the* maybe *monad $\mathcal{P}^1$ arises from an adjunction $\mathcal{P}^1 \dashv U$ between the category $\mathbf{Pos}_\perp$ of lower-bounded posets and the category $\mathbf{Set}$, enriching our model with a notion of failure $\perp$ and an associated partial order. Likewise, the state monad arises from the adjunction $- \times S \dashv (-)^S$ introduced in Example 3.12. The left adjoint allows computations to produce a state, and the right adjoint allows them to consume one. The monad $\mathcal{L}_S$ corresponds to the composition:*

$$\mathbf{Pos}_\perp \underset{U}{\overset{\mathcal{P}^1}{\leftrightarrows}} \mathbf{Set} \underset{(-)^S}{\overset{-\times S}{\leftrightarrows}} \mathbf{Set}\quad.$$

Since $\mathbf{Pos}_\perp$ is monoidal closed, these two aspects of $\mathcal{L}_S$ could be combined in the opposite way:

$$\mathbf{Pos}_\perp \underset{[S,-]}{\overset{-\otimes S}{\leftrightarrows}} \mathbf{Pos}_\perp \underset{U}{\overset{\mathcal{P}^1}{\leftrightarrows}} \mathbf{Set}\quad.$$

Under this approach, from a lower-bounded, partially ordered *set of states S, we would construct a*

$$
\begin{array}{rl}
\boxed{L_{\mathsf{bq}}} & S_{\mathsf{bq}} := V^* \\
\mathsf{enq} : V \to \mathbb{1} & L_{\mathsf{bq}}.\mathsf{enq}[v] := \lambda\, \vec{q} \cdot \{*@\vec{q}v \mid |\vec{q}| < N\} \\
\mathsf{deq} : \mathbb{1} \to V & L_{\mathsf{bq}}.\mathsf{deq} := \lambda\, \vec{q} \cdot \{v@\vec{p} \mid \vec{q} = v\vec{p}\}
\end{array}
$$

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

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

Figure 4.1: 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.

monad $\mathcal{L}'$ consisting of the monotonic, $\bot$-preserving *functions:*

$$\mathcal{L}'(A) := S \xrightarrow{\mathbf{Pos}_\bot} \mathcal{P}^1(A) \otimes S$$

Since the tensor product $\mathcal{P}^1(A) \otimes S$ identifies the posets' lower bounds, $\mathcal{L}'$ is largely equivalent to $\mathcal{L}$. However, $\mathcal{L}'$ can use an arbitrary bounded poset of states containing states with partial information.

4.2.2 Layer interfaces

Definition 4.2. A *layer interface* is a tuple $L = \langle E, S, \sigma \rangle$, where E is an effect signature, S is a set of states, and σ assigns to each operation $(m : N) \in E$ a specification $\sigma^m \in \mathcal{L}_S(N)$. I will sometimes use the notation $L.m$ to refer to σ^m.

As a running example, I will use the certified layer described in Figure 4.1, which implements a bounded queue with at most N elements using a circular buffer. We outline the construction of the corresponding layer interfaces below.

Example 4.3 (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_{rb} *contain an array* $f \in V^N$ *storing* N *values of type* V*, and two counters taking values in the interval* $0 \leq c_1, c_2 < N$*. The array can be accessed through the primitives* get *and* set*; the primitives* inc_1 *and* inc_2 *increment the corresponding counter and return the counter's old value.*

4.2.3 Client code

We can use the free monad T_E on the signature E as a general representation for client code for an underlay interface $L = \langle E, S, \sigma \rangle$. The monad homomorphism $(-)[\sigma] : T_E \to \mathcal{L}_S$ can then be used to evaluate client programs.

More explicitly, as discussed in §2.6, a client program $C \in T_E(X)$ is a term on the signature E with variables in X, or equivalently a monadic expression built using the operations in E with a result in X. A client program $C \in T_E(A)$ can be interpreted as a computation $C[L] \in \mathcal{L}_S(A)$ in the following way:

$$\underline{m}(t_n)_{n \in N}[L] \;=\; (n \leftarrow m\,;\, t_n)[L] \;=\; n \leftarrow \sigma^m\,;\, t_n[L] \qquad (m : N) \in E$$
$$\underline{v}[L] \;=\; \eta_A^E(v)[L] \;=\; \eta_A^{\mathcal{L}}(v) \qquad v \in A$$

In other words, in $C[L] \in \mathcal{L}_S(A)$, occurrences of the operations of E have been replaced by their specification given in σ to interpret the program in terms of state transitions.

Example 4.4 (Queue rotation)**.** *The following client program for the signature* E_{bq} *rotates a queue and returns the rotated value:*

$$C_{\mathsf{rot}} := v \leftarrow \mathsf{deq}\,;\, \mathsf{enq}[v]\,;\, \eta(v)$$

Using L_{bq} *as an underlay interface, we can evaluate it to*

$$C_{\mathsf{rot}}[L_{\mathsf{bq}}] := \lambda\, \vec{q} \cdot \{\vec{p}v@v \mid \vec{q} = v\vec{p} \wedge |\vec{p}| < N\}\,.$$

4.2.4 Layer implementations

A layer implementation provides underlay client code for each operation of the overlay interface. In other words, it is an interpretation of the overlay's signature into the underlay's signature.

Definition 4.5. A *layer implementation* for an underlay signature E and an overlay signature F is a family $(M^q)_{(q:R)\in F}$ with $M^q \in T_E(R)$. As before I will write $M : E \to F$, and sometimes use the notation $M.q$ to refer to M^q. Then, for an underlay interface $L = \langle E, S, \sigma\rangle$, the *evaluation* of M over L is the layer interface $M \circ L := \langle F, S, \tau\rangle$ with $\tau^q := M^q[L]$.

Similarly to layer interfaces, a layer implementation $M : E \to F$ is associated with a monad homomorphism $(-)[M] : T_F \to T_E$. The monad homomorphism associated with the evaluated layer interface $M \circ L$ is then the composite $(-)[L] \circ (-)[M] : T_F \to \mathcal{L}_S$.

Example 4.6. *The layer implementation M_{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 given in Figure 4.1, which explains how overlay states are realized by M_{bq} in terms of underlay states.*

The code of M_{bq} can be interpreted in the monad $S_{\mathsf{rb}} \to \mathcal{P}^1(- \times S_{\mathsf{rb}})$, with calls to primitives of L_{rb} replaced by their specifications, to obtain the layer interface $M_{\mathsf{rb}}[L_{\mathsf{bq}}]$. Concretely, the behaviors of $M_{\mathsf{bq}}.\mathsf{enq}[v]$ and $M_{\mathsf{bq}}.\mathsf{deq}$ running on the underlay interface L_{rb} evaluate to:

$$\begin{aligned}(M_{\mathsf{bq}} \circ L_{\mathsf{rb}}).\mathsf{enq}[v] &= i \leftarrow L_{\mathsf{rb}}.\mathsf{inc}_2 \,;\, L_{\mathsf{rb}}.\mathsf{set}[i, v] \\ &= \lambda\,(f, c_1, c_2) \cdot L_{\mathsf{rb}}.\mathsf{set}[c_2, v](f, c_1, (c_2 + 1) \bmod N) \\ &= \lambda\,(f, c_1, c_2) \cdot \{*@(f[c_2 := v], c_1, (c_2 + 1) \bmod N) \mid c_2 < N\}\end{aligned}$$

$$\begin{aligned}(M_{\mathsf{bq}} \circ L_{\mathsf{rb}}).\mathsf{deq} &= i \leftarrow L_{\mathsf{rb}}.\mathsf{inc}_1 \,;\, L_{\mathsf{rb}}.\mathsf{get}[i] \\ &= \lambda\,(f, c_1, c_2) \cdot L_{\mathsf{rb}}.\mathsf{get}[c_1](f, (c_1 + 1) \bmod N, c_2) \\ &= \lambda\,(f, c_1, c_2) \cdot \{f_{c_1}@(f, (c_1 + 1) \bmod N, c_2) \mid c_1 < N\}\end{aligned}$$

4.2.5 Correctness

Consider an underlay interface L_1, an overlay interface L_2, and a corresponding layer implementation M. The layer interface $M \circ L_1$ 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 correspondence between the more abstract states of L_2 and the more concrete states of L_1. Using the relators $\rightarrow$, $\times$, $\mathcal{P}^{\leq}$ presented in §2.3.2, the associated simulation property can be defined as follows.

Definition 4.7 (Simulation of layer interfaces)**.** For a relation $R \subseteq S_2 \times S_1$ and two computations $\tau_2 \in \mathcal{L}_{S_2}(A)$ and $\tau_1 \in \mathcal{L}_{S_1}(A)$, we say that τ_2 is *simulated by* τ_1 and write $\tau_2 \leq_R \tau_1$ when:

$$\tau_2 \ \ [R \rightarrow \mathcal{P}^{\leq}(= \times R)] \ \ \tau_1 \tag{4.1}$$

Likewise, given two layer interfaces $L_1 = \langle E, S_1, \sigma_1 \rangle$ and $L_2 = \langle E, S_2, \sigma_2 \rangle$ sharing a signature, 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 $(m : N) \in E$:

$$L_2.m \leq_R L_1.m\,. \tag{4.2}$$

The properties (4.1) and (4.2) correspond to the following simulation diagram, which asserts that if the computation $L_2.m$ started in a state k_1 succeeds, then the computation $L_1.m$ started in a related state k_2 must produce the same result while maintaining the relation between states:

$$\begin{array}{ccc} k_1 & \xrightarrow{L_2.m} & v@k_1' \\ {\scriptstyle R}\,\big| & & \vdots\,{\scriptstyle =\times R} \\ k_2 & \dashrightarrow^{L_1.m} & v@k_2' \end{array}$$

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

Definition 4.8 (Certified abstraction layer)**.** For an underlay interface $L_1 = \langle E_1, S_1, \sigma_1 \rangle$, a layer implementation $M : E_1 \rightarrow E_2$ and an overlay interface $L_2 = \langle E_2, S_2, \sigma_2 \rangle$, we say that M *correctly implements* L_2 *on top of* L_1 when there is a simulation relation $R \subseteq S_2 \times S_1$ such that:

$$L_2 \leq_R M \circ L_1\,.$$

We say that L_1, M, L_2 together constitute a *certified abstraction layer* and write $L_1 \vdash_R M : L_2$.

Example 4.9 (Correctness of the bounded queue implementation)**.** *I will prove the correctness of the layer implementation M_{bq} with respect to the relation R shown in Figure 4.1.*

First, we can write $[f]_{c_1}^{c_2}$ for the set of possible sequences of values read between indices c_1 and c_2 within the array f. More precisely:

$$[f]_{c_1}^{c_2} := \{f_{c_1} \dots f_{c_2-1} \mid c_1 \le c_2 < N\} \cup \{f_{c_1} \dots f_{N-1} f_0 \dots f_{c_2} \mid c_2 \le c_1 < N\}\,.$$

Then the simulation relation can be formulated as:

$$\vec{q}\; R\; (f, c_1, c_2) \;\Leftrightarrow\; \vec{q} \in [f]_{c_1}^{c_2}\,.$$

To show the correctness property $L_{\mathsf{rb}} \vdash_R M_{\mathsf{bq}} : L_{\mathsf{bq}}$, we need to prove that:

$$L_{\mathsf{bq}} \le_R M_{\mathsf{bq}} \circ L_{\mathsf{rb}}\,.$$

To this end, consider the related states $\vec{q}\; R\; (f, c_1, c_2)$. We know in particular that $c_1, c_2 < N$. Hence, from the calculation carried out in Example 4.6:

$$(M_{\mathsf{bq}} \circ L_{\mathsf{rb}}).\mathsf{enq}[v](f, c_1, c_2) = \{*@(f[c_2 := v],\ c_1,\ (c_2 + 1) \bmod N)\}$$
$$(M_{\mathsf{bq}} \circ L_{\mathsf{rb}}).\mathsf{deq}(f, c_1, c_2) = \{f_{c_1}@(f,\ (c_1 + 1) \bmod N,\ c_2)\}$$

Since the operations are specified as:

$$L_{\mathsf{bq}}.\mathsf{enq}[v](\vec{q}) = \{*@\vec{q}v \mid |\vec{q}| < N\}$$
$$L_{\mathsf{bq}}.\mathsf{deq}(\vec{q}) = \{v@\vec{p} \mid \vec{q} = v\vec{p}\}\,,$$

we need to show that whenever $|\vec{q}| < N$, the following relations hold:

$$* = * \;\wedge\; \vec{q}v\; R\; (f[c_2 := v], c_1, (c_2 + 1) \bmod N)\,,$$

and that whenever $\vec{q} = v\vec{p}$, the following relations hold:

$$v = f_{c_1} \ \wedge \ \vec{p}\, R\, (f,\ (c_1 + 1) \bmod N,\ c_2)\,.$$

This is straightforward to check, using the constraints on the length of the queue to verify that:

$$\vec{q} \in [f]_{c_1}^{c_2} \ \Rightarrow \ \vec{q}v \in [f[c_2 := v]]_{c_1}^{(c_2+1) \bmod N}$$
$$v\vec{p} \in [f]_{c_1}^{c_2} \ \Rightarrow \ \vec{p} \in [f]_{(c_1+1) \bmod N}^{c_2}\,.$$

Remark 4.10. *Note that we place no restriction on the relations we use beyond the simulation property. In particular, in the example above R is neither one-to-many nor many-to-one.*

On one hand, there are many low-level ring buffer states corresponding to a given high-level queue state: the implementation is free to choose the position c_1 where the queue's first element is stored, and arbitrary values for the $N - |\vec{q}|$ array locations which are not in use.

On the other hand, a low-level state where $c_1 = c_2$ can be interpreted in two different ways, as the empty queue ϵ or as the full queue $f_{c_1} \ldots f_{N-1} f_0 \ldots f_{c_2-1}$. This would make it impossible to implement an operation specified as $L_{\mathsf{bq}}.\mathsf{len}(\vec{q}) := \{|\vec{q}|@\vec{q}\}$ because when $c_1 = c_2$, the implementation would have to simultaneously return 0 and N. However, this is not an issue for enq *and* deq *because for both interpretations, one of the specifications is undefined:*

$$L_{\mathsf{bq}}.\mathsf{enq}[v](f_{c_1} \ldots f_{N-1} f_0 \ldots f_{c_2-1}) = \varnothing \qquad L_{\mathsf{bq}}.\mathsf{deq}(\epsilon) = \varnothing\,.$$

The implementation can satisfy both specifications under both interpretations, because in each case one of the specification does not actually place any constraints on the implementation. This allows $M_{\mathsf{bq}}.\mathsf{enq}$ to be correct by using the "empty" interpretation, and $M_{\mathsf{bq}}.\mathsf{deq}$ to be correct by using the "full" interpretation.

Once we have shown an abstraction layer correct, we will be able to reason about any client code in terms of the layer's overlay interface rather than its low-level implementation. This is enabled by the following property.

Theorem 4.11 (Soundness)**.** *For a certified abstraction layer $L_1 \vdash_R M : L_2$ and a client program*

C of L_2, the following contextual refinement *property holds:*

$$C[L_2] \leq_R C[M \circ L_1]$$

Proof. We need to show that $C[-]$ preserves $\leq_R$. In essence, as C executes successive operations of the two layer interfaces, we can follow along with the corresponding simulation diagrams, and paste them horizontally to derive a simulation for the whole execution. More formally, we can proceed by structural induction on the term C:

- If the term C is of the form $\underline{v} = \eta^E(v)$, then it is interpreted as $\lambda k \cdot \{v@k\}$ in any layer, and we get the simulation property:

$$\begin{array}{ccc} k_1 & \xrightarrow{C[L_2]} & v@k_1 \\ {\scriptstyle R}\big| & & \vdots {\scriptstyle =\times R} \\ k_2 & \xrightarrow{C[M\circ L_1]} & v@k_2 \end{array}$$

- Now suppose C is of the form $\underline{m}(C_n)_{n\in N} = n \leftarrow m\,;\, C_n$. We know from the correctness property of M and from the induction hypothesis that:

$$\begin{array}{ccc} k_1 & \xrightarrow{L_2.m} & n@k_1' \\ {\scriptstyle R}\big| & & \vdots {\scriptstyle =\times R} \\ k_2 & \xrightarrow{(M\circ L_1).m} & n@k_2' \end{array} \qquad \begin{array}{ccc} k_1' & \xrightarrow{C_n[L_2]} & v@k_1'' \\ {\scriptstyle R}\big| & & \vdots {\scriptstyle =\times R} \\ k_2' & \xrightarrow{C_n[M\circ L_1]} & v@k_2'' \end{array}$$

  By pasting the two diagrams horizontally, we can conclude that:

$$\begin{array}{ccc} k_1 & \xrightarrow{C[L_2]} & v@k_1'' \\ {\scriptstyle R}\big| & & \vdots {\scriptstyle =\times R} \\ k_2 & \xrightarrow{C[M\circ L_1]} & v@k_2'' \end{array}$$

□

4.2.6 Composing certified abstraction layers

Layer implementations can be composed as explained in §2.6.4. Given the layer implementations $M : E_1 \to E_2$ and $N : E_2 \to E_3$, the layer implementation $N \circ M$ replaces the operations of E_2 in N by their interpretation given by M:

$$(N \circ M)^m := N^m[M]\,.$$

The soundness theorem for the certified abstraction layer $L_1 \vdash_R M : L_2$ can be easily extended from individual client programs to complete layer implementations, so that:

$$\frac{L_1 \vdash_R M : L_2}{N \circ L_2 \leq_R N \circ M \circ L_1}$$

More generally, if N defines a certified abstraction layer with underlay L_2, then its correctness property can be combined with that of M to build a composite certified abstraction layer.

Theorem 4.12 (Vertical composition)**.** *Certified abstraction layers compose in the following way:*

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

Proof. Simulation properties can be composed by vertically pasting the simulations diagrams:

$$\frac{\tau_3 \leq_S \tau_2 \qquad \tau_2 \leq_R \tau_1}{\tau_3 \leq_{R \circ S} \tau_2}$$

This can be easily extended to layer interfaces. Then, the theorem can be proved by combining this property with the soundness theorem for M as follows:

$$\frac{L_3 \leq_S N \circ L_2 \qquad N \circ L_2 \leq_R N \circ M \circ L_1}{L_3 \leq_{R \circ S} N \circ M \circ L_1}$$

□

Theorem 4.12 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 $L = \langle E, S, \sigma \rangle$, the identity interpretation $e : E \to E$ is a correct imple-

mentation $L \vdash_{=} e : L$ of the layer interface L itself. In other words, layer interfaces and certified layer implementation define a category $\mathbf{CAL}$.

4.3 Horizontal composition

While vertical composition is quite useful, individual abstraction layers can themselves be fairly complex and involve a significant number of operations. We would like to verify them one by one, and build layers out of elementary components in a systematic way.

Moreover, abstraction layers do not always depend on each other in a purely linear fashion; often two independent subsystems will rely on the same underlay. We would like to be able to verify them in isolation but retain the ability to build higher-level code based on their combined functionality.

In this section I present *horizontal* composition principles which we can use to achieve this. It will then be possible to combine layers side-by-side in addition to stacking them on top of one another.

4.3.1 Signatures

To describe horizontal compositions of layers we must first combine signatures. When layers are placed side-by-side, a client program will have access to the operations of both. The result is essentially the disjoint union of the two layers' signatures, and can be constructed as follows.

Definition 4.13. The *sum* of a family of effect signatures $(E_i)_{i \in I}$ is defined as:

$$\bigoplus_{i \in I} E_i := \{\iota_i(m) : N \mid i \in I, (m : N) \in E_i\}$$

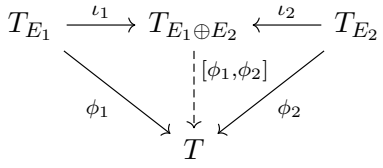
The finitary case $\bigoplus_{1 \leq i \leq n} E_i$ can be written as $E_1 \oplus \cdots \oplus E_n$.

On one hand, this allows us to compute the operations offered by a collection of layers. For instance, if we combine an implementation of L_{bq} with an implementation of the I/O operations used an example in §2.5, the signature of the result will be $E_{\mathsf{bq}} \oplus E_{\mathsf{io}}$. Conversely, this also allows us to decompose the signature of a single layer into signatures associated with individual operations,

for instance:

$$E_{\mathsf{bq}} \cong \{\mathsf{deq} : V\} \oplus \{\mathsf{enq} : V \to \mathbb{1}\}\,.$$

Remark 4.14. *In the categories of monads and monad homomorphisms, the signature $E_1 \oplus E_2$ defines the coproduct of the free monads T_{E_1} and T_{E_2}. This means that interpreting $T_{E \oplus F}$ into a monad T is the same thing as giving independent interpretations of T_E and T_F:*



In terms of interpretations, the combined monad homomorphism $[\phi_1, \phi_2]$ can be described as:

$$[\phi_1, \phi_2]^{\iota_1(m)} = \phi_1^m \in T(N) \qquad (m : N) \in E_1$$
$$[\phi_1, \phi_2]^{\iota_2(m)} = \phi_2^m \in T(N) \qquad (m : N) \in E_2$$

This immediately suggests composition principles for layer interfaces and layer implementations:

- *When T is of the form $\mathcal{L}_S$, this characterizes horizontal composition for layer interfaces expressed in terms of a common set of abstract states S.*
- *When T is itself a free monad of the form T_E, this gives a horizontal composition principle for layer implementations.*

4.3.2 Layer implementations

It is straightforward to combine layer implementations of the individual signatures E_1 and E_2 into an implementation of $E_1 \oplus E_2$.

Definition 4.15. The layer implementations $M_1 : E \to E_1$ and $M_2 : E \to E_2$ can be combined into $\langle M_1, M_2 \rangle : E \to E_1 \oplus E_2$ as follows:

$$\langle M_1, M_2 \rangle^{\iota_1(m)} := M_1^m \in T_E(N) \qquad (m : N) \in E_1$$
$$\langle M_1, M_2 \rangle^{\iota_2(m)} := M_2^m \in T_E(N) \qquad (m : N) \in E_2$$

In addition, we can define the layer implementations:

$$\pi_1 : E_1 \oplus E_2 \to E_1 \qquad \pi_1^m := e^{\iota_1(m)} \in T_{E_1 \oplus E_2}(N) \qquad (m : M) \in E_1$$
$$\pi_2 : E_1 \oplus E_2 \to E_2 \qquad \pi_2^m := e^{\iota_2(m)} \in T_{E_1 \oplus E_2}(N) \qquad (m : M) \in E_2$$

which discard the operations of one of the summands.

Given a client program formulated in terms of the operations of E_1, the projections allow $C \in T_{E_1}(A)$ to be lifted to $C[\pi_1] \in T_{E_1 \oplus E_2}(A)$, so that it can be connected to underlay interfaces and implementations which provide the larger signature. A layer implementation $M : E_1 \to F$ can likewise be lifted to $M \circ \pi_1 : E_1 \oplus E_2 \to F$.

Conversely, tupling allows us to construct a layer implementations for the signature $F_1 \oplus F_2$ out of components implementing F_1 and F_2. Combined with projections, this means in particular that two layer implementations $M_1 : E_1 \to F_1$ and $M_2 : E_2 \to F_2$ can be composed side-by-side as $M_1 \oplus M_2 : E_1 \oplus E_2 \to F_1 \oplus F_2$, where $M_1 \oplus M_2 := \langle M_1 \circ \pi_1, M_2 \circ \pi_2 \rangle$.

Moreover, we can break down any signature $F = \sum_i F_i$ into elementary components of the form $F_i = \{m_i : A_i \to B_i\}$ containing a single A_i-indexed family of operations of arity B_i. Likewise, we can break down any layer implementation into elementary components of type $E \to F_i$. If the operation $m_i[a]$ is interpreted by $C_a \in T_E(B_i)$ for all $a \in A$, we can write

$$(m_i[a] \mapsto C_a) : E \to F_i$$

for the corresponding elementary layer implementation. Tupling can be used to reconstruct complete layer implementations from these elementary components.

Example 4.16 (Decomposition of M_{bq})**.** *The signature E_{bq} and layer implementation M_{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:

$$E_{\mathsf{bq}} \cong E_{\mathsf{enq}} \oplus E_{\mathsf{deq}} \qquad\qquad M_{\mathsf{bq}} \cong \langle M_{\mathsf{enq}}, M_{\mathsf{deq}} \rangle .$$

Since layer implementations are defined operation-wise, there is not much substance to this reshuffling. However, it will allows us to verify certified abstraction layers by verifying their elementary components one by one.

4.3.3 Layer interfaces sharing state

Layer interfaces can likewise be built from elementary components. For a signature $\{m : A \to B\}$, a set of states S, and a specification $\sigma_a \in \mathcal{L}_S(B)$ for all $a \in A$, we can write:

$$(m[a] \mapsto \sigma_a) := \langle \{m : A \to B\},\ S,\ \hat{\sigma} \rangle \qquad\qquad \hat{\sigma}^{m[a]} := \sigma_a$$

Going one step further, we can build elementary certified abstraction layers by using the property:

$$\frac{\forall a \in A \cdot \sigma_a \leq_R C_a[L]}{L \vdash_R (m[a] \mapsto C_a) : (m[a] \mapsto \sigma_a)}$$

These elementary layers can then be composed in the following way.

Definition 4.17. 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, their product can be defined as:

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

Theorem 4.18. *When the product of layer interfaces is defined, the tupling of layer implementations preserves correctness in the following sense:*

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

Moreover, the projections satisfy the following correctness properties:

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

Proof. From the reflexivity of $\leq_=$ and the operation-wise definition of correctness. □

Example 4.19 (Decomposition of L_{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:

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

To summarize, the constructions I have presented so far provide a language to build complex certified abstraction layers by verifying elementary components. Together with vertical composition, this is the essence of the layer calculus we used in our initial verification of CertiKOS [Gu et al., 2015].

However, under this approach the components of a given layers must still be designed around a common notion of abstract state, and it is not possible to horizontally compose layers which have been designed and verified independently. A related phenomenon is that the diagonal morphism:

$$L \vdash_= \Delta : L \times L$$

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.

4.3.4 Tensor product

In this section, I present a more general form of horizontal composition which lets two layers operate independently of one another. In particular, the layers do not share state.

Definition 4.20. The *tensor product* of $L_1 = \langle E_1, S_1, \sigma_1 \rangle$ and $L_2 = \langle E_2, S_2, \sigma_2 \rangle$ is defined as:

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

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

$$\hat{\sigma}_1^{m@(k_1,k_2)} := \{v@(k_1', k_2) \mid \sigma_1^{m@k_1} \ni v@k_1'\}$$
$$\hat{\sigma}_2^{m@(k_1,k_2)} := \{v@(k_1, k_2') \mid \sigma_2^{m@k_2} \ni v@k_2'\}$$

The corresponding monoidal structure is not cartesian. In particular, two certified abstraction layers $L \vdash_{R_1} M_1 : L_1$ and $L \vdash_{R_2} M_2 : L_2$ cannot be composed as

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

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

Theorem 4.21 (Tensor product of certified abstraction layers)**.** *Two certified abstraction layers can be composed as follows:*

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

Proof. Each abstraction layer will act on its half of the state with no interference from the other. Hence the correctness properties of the premises can be used almost directly to establish those of the composite layer. □

To enable two independent certified abstraction layers to rely on a common underlay L, we must first show that the resources of L can be multiplexed by introducing a third component $L \vdash_R M : L \otimes L$.

Example 4.22 (Multiplexing a memory allocator)**.** *Suppose the layer interface* L_{mm} *offers an operation* alloc : $\mathbb{1} \to$ pageno *which allocates a page of memory from a pool and returns its identifier. In a practical system, memory allocators can be used concurrently by any number of subsystems, which will only access pages that have been allocated to them. To reflect this, we need to introduce a component*

$$L_{\mathsf{mm}} \vdash_R \Delta : L_{\mathsf{mm}} \otimes L_{\mathsf{mm}} \,,$$

where the simulation relation R *explicates how the* client views *of the allocator's state add up consistently to make up the global state, such that the clients won't interfere with each other when they execute operations based on their view. The layer implementation* $\Delta = \langle \pi_1, \pi_2 \rangle$ *simply passes through the calls of each client unchanged to the underlay.*

This presents several challenges. For instance, unless the allocator's interface describes an idealized allocator which always succeeds (a convenient but unimplementable abstraction), the state will need to keep track of a finite number of available pages. Since R *cannot duplicate these pages, we may need to add a parameter* n *to the layer interface* L^n_{mm} *indicating the total number of available pages. Then the multiplexing component could be weakened to take the form:*

$$L^{n_1+n_2}_{\mathsf{mm}} \vdash_R \Delta : L^{n_1}_{\mathsf{mm}} \otimes L^{n_2}_{\mathsf{mm}} \,.$$

Another issue is that the model presented in this chapter can only express deterministic *interfaces. In the case of a memory allocator, this means the allocator's abstract state should contain enough details to predict the specific identity of every page handed out by the allocator. This is possible in the global view, but when multiple clients are accessing the allocator, the sequences of page identifiers handed out to each client will depend on how their requests interleave with that of other clients. This limitation, among others, is addressed by the models presented in Chapters 6 and 7*

4.4 CompCertX

Our original definition of certified abstraction layers was tightly coupled with the semantic model of CompCert [Gu et al., 2015]. In that context, layer specifications are defined to operate on a CompCert memory state component as well as the layer-specific abstract state.

The overall structure of the layer calculus used in CertiKOS is very similar to the one I have described so far. In this section, I briefly touch on some of the differences between the two systems, and the ways in which the system is integrated with CompCert.

The version of CompCert we used for this purpose is called CompCertX. It was designed to support certified abstraction layers, and allow CompCert programs to play the role of layer implementations. CompCertX itself was largely designed and implemented by Tahina Ramananandro during his time in the Yale FLINT group.

4.4.1 Layer interfaces

Roughly speaking, *external calls* in CompCert can be understood as effects in a signature

$$\tilde{\mathcal{C}}_D = \{ f : \mathsf{val}^* \to \mathsf{val} \mid f \in D \subseteq \mathsf{ident} \} .$$

In an external call $f[\vec{v}]$, the identifier f designates the function to be called and $\vec{v} \in \mathsf{val}^*$ gives its actual parameters; the corresponding outcome $v' \in \mathsf{val}$ is the value returned by the function.

In CertiKOS, we use these signatures to construct layer interfaces of the form

$$L = \langle \tilde{\mathcal{C}}_D,\ S \times \mathsf{mem},\ \sigma \rangle .$$

The set D is the layer's *domain*, in other words the set of primitives which it implements. The set S is a layer-specific set of abstract states, and is combined with the set mem of memory states used to define the language semantics of CompCert.

4.4.2 Client code

Traditional CompCert semantics use a parameter χ common to all languages, which describes the behavior of external functions. Schematically, $\mathsf{Clight}_\chi[p]$ describes the semantics of the Clight program p, where $\chi^{f[\vec{v}]} \in \mathcal{L}_{\mathsf{mem}}(\mathsf{val})$ describes the behavior of the external call $f[\vec{v}]$. This can be understood as a rudimentary underlay interface $\langle \tilde{C}_{\mathsf{ident}}, \mathsf{mem}, \chi \rangle$ on top of which Clight programs are evaluated.

To enable the use of richer underlay interfaces, we extend this facility as follows. Given a

CertiKOS layer interface $L = \langle \mathcal{C}_D, S \times \mathsf{mem}, \sigma \rangle$, the semantics $\mathsf{Clight}_L[p]$ will maintain an abstract state $k \in S$ alongside the memory state, and use the family of specifications $\sigma^{f[\vec{v}]} \in \mathcal{L}_{S \times \mathsf{mem}}(\mathsf{val})$ to interpret external calls. This allows us to evaluate client code with L as underlay.

4.4.3 Layer implementations

To make it possible to evaluate CompCert modules as complete layer *implementations*, CompCertX also extends the *incoming* interface of language semantics. Traditional CompCert semantics only model the top-level invocation of $\mathsf{main}[\epsilon]$. In CompCertX, we make it possible to evaluate any incoming call $f[\vec{v}]$ and describe its action in terms of the combined abstract and memory states:

$$\mathsf{Clight}_{\mathsf{L}}[M]^{f[\vec{v}]} \in \mathcal{L}_{S \times \mathsf{mem}}(\mathsf{val})\,.$$

Then evaluating M on the underlay $L = \langle \tilde{\mathcal{C}}_D,\ S \times \mathsf{mem},\ \sigma \rangle$ yields the layer interface:

$$M[L] := \langle \tilde{\mathcal{C}}_{\mathsf{dom}(M)},\ S \times \mathsf{mem},\ \mathsf{Clight}_L[M] \rangle\,.$$

4.4.4 Simulation relations

Using CompCertX to compile layer implementations to assembly introduces *memory injections*, written $\iota \Vdash m_1 \hookrightarrow m_2$, which reflect the change in structure between a source memory state m_1 and a corresponding target memory state m_2. Moreover, the certified abstraction layers used in CertiKOS follow a common pattern, where abstract data is progressively realized in lower layers as global variables stored in the CompCert memory state.

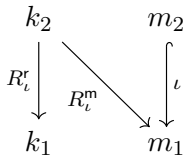
To make our certified abstraction layers compatible with injections and to capture this pattern, the simulation relations are specified by two components indexed by a memory injection ι:

$$R^{\mathsf{r}}_{\iota} \subseteq S_2 \times S_1$$
$$R^{\mathsf{m}}_{\iota} \subseteq S_2 \times \mathsf{mem}\,.$$

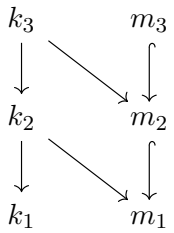
The overlay abstract states of S_2 will retain some of the underlay's abstract data in S_1. This is formalized by the component R^{m}. On the other hand, S_2 may also contain newly abstracted data,

stored in the underlay as part of the memory states. This is formalized by the component R^{r}. From this description, we can derive the layer's simulation relation as:

$$(k_2, m_2)\ R\ (k_1, m_1) \ \Leftrightarrow\ \exists\, \iota\ \cdot\ k_2\ R^{\mathsf{r}}_{\iota}\ k_1\ \wedge\ k_2\ R^{\mathsf{m}}_{\iota}\ m_1\ \wedge\ \iota \Vdash m_2 \hookrightarrow m_1$$



Describing simulation relations in this way allows us to take advantage of the pattern when we develop proof principles for individual layers, especially when it comes to making sure our simulation relations are compatible with compilation by CompCertX. Unfortunately, simulation relations described in this way do not directly compose. To see why, consider the following diagram:



Injections compose as expected, in the sense that $(\iota \circ \iota' \Vdash \hookrightarrow) = (\iota \Vdash \hookrightarrow) \circ (\iota' \Vdash \hookrightarrow)$. We also place restrictions on the component relations which guarantee that they composes with memory injections in the appropriate way. However, to formulate the composite R^{m}, we must existentially quantify over the intermediate memory state. Because there is no way to guarantee consistency between the m_2 involved in $(\iota \Vdash \hookrightarrow) \circ (\iota \Vdash \hookrightarrow)$ and in the composite R^{m}, the composition does not behave as expected.

To work around this issue, we can use formal sequences of $(R^{\mathsf{r}}, R^{\mathsf{m}})$ pairs to describe the simulation relation used by vertically composable CertiKOS layers. We can then compose the overall simulation relations derived from individual components when we formulate the soundness proof. Another approach is to generalize our pattern so that it still provides the same advantages but behaves in a better way with respect to composition. This was the original motivation for my work on CompCert Kripke logical relations (§10.1).

4.5 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 the layer calculus are made explicit. The distinctions between our restricted form of product and the more general monoidal structure hints at connections between certified abstraction layers and other models which incorporate structures found in fragments of linear logic.

Chapter 5

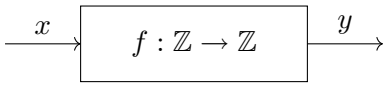
Games and dual nondeterminism

This chapter presents my general approach to dual nondeterminism in game semantics, leading to the construction of *strategy specifications* featuring dual nondeterminism and a form of alternating refinement. This approach decouples the nature of nondeterminism from the structure of plays, leading to a more uniform and tractable theory.

I outline the issues in §5.1 in the context of a simple model of *function specifications*, then articulate in §5.2 how they carry over to the context of games and strategies. In §5.3 I introduce the construction used by Morris [2004] to extend the refinement calculus to functional programming, and show in §5.4 that it can be used to construct strategy specifications.

5.1 Example: function specifications

To illustrate the use of dual nondeterminism in the context of specifications and refinement, I will use functions on integers. A function $f : \mathbb{Z} \to \mathbb{Z}$ can be seen as a simple system, which accepts a single input and produces a single output:



	$-1 \mapsto 0$	$0 \mapsto 0$	$0 \mapsto 1$	$1 \mapsto 2$
$f(x) := 2x$		✓		✓
$g(x) := x + 1$	✓		✓	✓
$h(x) := 2\lceil x/2 \rceil$	✓	✓		✓

Table 5.1: Some functions and elementary specifications used as examples in §5.1

5.1.1 Elementary function specifications

To constrain the behavior of such a system, we can use an elementary specification of the form:

$$x \mapsto y$$

The specification above asserts that whenever the function's input is $x \in \mathbb{Z}$, it should produce the output $y \in \mathbb{Z}$; in other words, $f(x) = y$. Table 5.1 shows some simple functions and elementary specifications which I will use for illustration. For example, the function defined by $f(x) := 2x$ satisfies the specifications $0 \mapsto 0$ and $1 \mapsto 2$ but does not satisfy $-1 \mapsto 0$ or $0 \mapsto 1$.

On their own, elementary specifications carry limited information about the functions they attempt to characterize. Like the plays of a game, they only give a snapshot of a single interaction between the system and its environment. However, by incorporating angelic and demonic nondeterminism, the model can be made much more expressive.

5.1.2 Angelic and demonic choices

Angelic choice allows us to range over various possible choices of the environment, placing more constraints on the function and making the specification stronger. For example, the specification:

$$-1 \mapsto 0 \sqcup 1 \mapsto 2$$

is satisfied by both of the functions $g(x) := x + 1$ and $h(x) := 2\lceil x/2 \rceil$, but not by f which only satisfies the right-hand side component. Infinite choice allows us to formulate richer specifications. For example, the following specification characterizes the function f exactly:

$$\hat{f} := \bigsqcup_{x \in \mathbb{Z}} x \mapsto 2x$$

This allows f itself to be represented as the specification $\hat{f}$. The fact that f satisfies $1 \mapsto 2$ can then be expressed as:

$$1 \mapsto 2 \sqsubseteq \bigsqcup_{x \in \mathbb{Z}} x \mapsto 2x = \hat{f}\,.$$

This also illustrates that a specification can be made stronger and more precise by adding more angelic choices, although if we go too far it may not be possible to implement it; for example, no function satisfies the following specification:

$$0 \mapsto 0 \sqcup 0 \mapsto 1\,.$$

Conversely, demonic choices make specifications weaker, allowing us to express implementation freedom. For example, the following specification:

$$0 \mapsto 0 \sqcap 0 \mapsto 1$$

is satisfied the each one of the functions f, g, h. Using infinite choice, the specification:

$$\bigsqcap_{x \in \mathbb{Z}} x \mapsto x$$

expresses that there must be *at least one* x which the function maps to itself. It can be refined by narrowing down the range of demonic choices, for example:

$$\bigsqcap_{x \in \mathbb{Z}} x \mapsto x + 1 \sqsubseteq -1 \mapsto -1 \sqcap 0 \mapsto 0 \sqcap 1 \mapsto 1\,.$$

This specification is satisfied by f and h, but not by g.

Angelic and demonic choices become even more powerful when they are used together. For example the specification

$$\bigsqcup_{x \text{ odd}} \bigsqcap_{y \text{ even}} x \mapsto y$$

expresses the constraint that *all* odd inputs must be mapped to *some* even output. All of the functions f, g, h satisfy this specification. A counter-example is the constant function $u(x) := 1$.

To summarize, adding arbitrary angelic and demonic choices to a specification framework

allows us to gain a significant amount of expressivity. When used in specifications, angelic choice corresponds to the logical conjunction and the *for all* quantifier, while demonic choice corresponds to disjunction and *there exists*.

5.1.3 Data abstraction

To illustrate the expressivity of the resulting model, consider the usual construction of integers as pairs of natural numbers.

An integer is represented by a pair $n = (n_1, n_2) \in \mathbb{N} \times \mathbb{N}$, where n_1 is understood as the positive component, n_2 is understood as the negative component, and two integers $n = (n_1, n_2)$ and $m = (m_1, m_2)$ in this form are considered equal when $n_1 + m_2 = n_2 + m_1$. The canonical embedding of natural numbers into integers can be realized as $n \mapsto (n, 0)$, and the standard arithmetic operations can be defined as:

$$
\begin{aligned}
n + m &:= (n_1 + m_1,\ n_2 + m_2) \\
n - m &:= (n_1 + m_2,\ n_2 + m_1) \\
n \times m &:= (n_1 \times m_1 + n_2 \times m_2,\ n_1 \times m_2 + n_2 \times m_1)\,.
\end{aligned}
$$

This "low-level" representation can be connected to the more usual and abstract notion of integer by the relation $\rho \subseteq \mathbb{Z} \times (\mathbb{N} \times \mathbb{N})$ defined by:

$$
x\ \rho\ (n_1, n_2) \ \Leftrightarrow\ x = n_1 - n_2\,.
$$

Then a natural question is: given a function $g : \mathbb{N}\times\mathbb{N} \to \mathbb{N}\times\mathbb{N}$ on the concrete representation, and a function $f : \mathbb{Z} \to \mathbb{Z}$ expressed in the more abstract representation, what does it mean for g to be a correct implementation of f? This situation is depicted by the simulation diagram below. Given related inputs $x\ \rho\ n$, the functions f and g are expected to produce related outputs $f(x)\ \rho\ g(n)$:

$$
\begin{array}{cccc}
(\forall) & \begin{array}{ccc} x & \xrightarrow{\ f\ } & f(x) \\ {\scriptstyle\rho}\big| & & \vdots{\scriptstyle\rho} \\ n & \xrightarrow[\ g\ ]{} & g(n) \end{array} & (\exists) & f\ [\rho \to \rho]\ g
\end{array}
$$

However, since this involves objects of two different types, it is not immediately obvious how this property can be formulated in the context of refinement. As it turns out, dual nondeterminism makes it possible to define the concretization and abstraction functions:

$$\gamma(f) := \bigsqcup_{n \in \mathbb{N} \times \mathbb{N}} \; \bigsqcup_{x \in \mathbb{Z} \mid x \,\rho\, n} \; \bigsqcap_{m \in \mathbb{N} \times \mathbb{N} \mid f(x) \,\rho\, m} n \mapsto m$$

$$\alpha(g) := \bigsqcup_{x \in \mathbb{Z}} \; \bigsqcap_{n \in \mathbb{N} \times \mathbb{N} \mid x \,\rho\, n} \; \bigsqcup_{y \in \mathbb{Z} \mid y \,\rho\, g(n)} x \mapsto y$$

such that the correspondence of f and g can be stated as:

$$\gamma(f) \sqsubseteq \hat{g} \quad \Leftrightarrow \quad \hat{f} \sqsubseteq \alpha(g)\,.$$

5.2 Refinement in game semantics

Game semantics represents strategies as sets of plays. Each play records a possible interaction between the system and the environment. For instance, if we interpret the elementary function specification $x \mapsto y$ as a simple play containing one move of the environment (x) followed by one move of the system (y), the strategy associated to $f : \mathbb{Z} \to \mathbb{Z}$ can be represented as:

$$\sigma := \{x \mapsto f(x) \mid x \in \mathbb{Z}\}\,.$$

5.2.1 Strategies and refinement

In most game models, there are restrictions on the contents of σ. One common restriction is the exclusion of nondeterministic choice involving moves of the system. In the context of function specifications, this translates to the following property:

$$(x \mapsto y_1) \in \sigma \;\wedge\; (x \mapsto y_2) \in \sigma \quad \Rightarrow \quad y_1 = y_2 \tag{5.1}$$

Another common requirement is for σ to range over all possible behaviors of the environment:

$$\forall\, x \in \mathbb{Z} \cdot \exists\, y \in \mathbb{Z} \cdot (x \mapsto y) \in \sigma\,. \tag{5.2}$$

In our setting, these constraints correspond to strategies which precisely describe functions.

To obtain a richer model, we can relax these restrictions, allowing strategies such as:

$$\sigma' := \{0 \mapsto 0,\ 1 \mapsto 1,\ 1 \mapsto -1\}\,.$$

On one hand, the relaxation of (5.1) permits some form of nondeterminism, for example between the elementary specifications $1 \mapsto 1$ and $1 \mapsto -1$ in the example above. Indeed this is the approach taken in early work on nondeterminism in game semantics [Harmer and McCusker, 1999]. On the other hand, the relaxation of (5.2) allows strategies which which do not cover the entire domain.

Yet even after these constraints have been relaxed, we need to choose how this nondeterminism should be interpreted, which will in particular determine how strategy refinement works:

- We can use an *angelic* interpretation of nondeterminism and accordingly define refinement as set inclusion ($\subseteq$). This fits with the original formulation of strategies and allows us to choose which elements of the domain to constrain. However, under this approach it is not possible to express a specification allowing different outputs for a given input. For example, the subset $\{1 \mapsto 1,\ 1 \mapsto -1\} \subseteq \sigma'$ results in an unimplementable specification, since it requires both $f(1) = 1$ and $f(1) = -1$.

- If we use a *demonic* interpretation and define refinement as set containment ($\supseteq$), we cannot formulate any specification stronger than a single $x \mapsto y$. In particular, under this approach a function can no longer be completely characterized. A function f which satisfies *any one* of the conditions $f(0) = 0$, $f(1) = 1$ or $f(1) = -1$ will match σ'.

- To address these issues, we can use an *alternating* interpretation [Alur et al., 1998], which takes into account the polarity of moves, so that σ' above is interpreted as:

  $$0 \mapsto 0 \sqcup (1 \mapsto 1 \sqcap 1 \mapsto -1)\,,$$

  requiring that the function map 0 to 0 and map 1 to either -1 or 1.

The alternating interpretation of strategy nondeterminism is the one most in line with our goals. In the context of games, choices of the system and the environment must be distinguished,

and it is natural to use the polarity of moves to make this distinction. However, the resulting refinement ordering is complex and unintuitive. For example, in the case of function specifications, we get the following ordering:

$$\{0 \mapsto 0\} \sqsubseteq \{0 \mapsto 0,\ 1 \mapsto 1,\ 1 \mapsto -1\} \sqsubseteq \{0 \mapsto 0,\ 1 \mapsto 1\}$$

In the more general case where plays consist of an unbounded alternation of input and outputs, the refinement ordering becomes even more complex to describe.

Another issue with the alternating interpretation of nondeterminism is its incompleteness. For example, the nondeterministic choices:

$$\{0 \mapsto 1\} \sqcup \{0 \mapsto -1\} \qquad \{1 \mapsto 0\} \sqcap \{-1 \mapsto 0\}$$

cannot be represented in the model proposed above. This restriction may at first seem harmless. After all, the first specification above is unimplementable, and the second one is so weak that it is unclear whether it could ever find any practical use. However, from an algebraic point of view the loss of completeness is a significant shortcoming. It breaks the uniformity of the model and imposes a proliferation of side-conditions which the user must establish.

Consider for example data abstraction as discussed in §5.1.3. The completely distributive lattice structure associated with dual nondeterminism allows us to use arbitrary abstraction relations. If the relation makes a high-level specification impossible to implement, for instance because it involves abstract data with no concrete representation, or concrete data with an ambiguous high-level interpretation, the associated concretization function γ will simply generate an unimplementable low-level specification.

If this is not possible because unimplementable low-level specifications cannot be represented, we need instead to formulate, prove and track various restrictions on the abstraction relation or the specification of interest. This is a particular concern in the context of formal proofs mechanized in a proof assistant, where even simple and intuitive side-conditions may interfere with usability.

5.2.2 Strategy specifications

Since formulating an appropriate notion of refinement for strategies is challenging, I will use a different approach. Instead of retrofitting nondeterminism and refinement onto the usual notion of strategy, we can reexamine the construction altogether:

- Individual plays are understood as elementary specifications. Like the elementary function specifications of §5.1, they indicate how the system may react to a specific behavior of the environment. The prefix relation on plays can be understood as a notion of refinement, with longer plays placing more precise constraints on the system.
- Sets of plays (strategies) allow us to range over the possible choices of the *environment*, understood as angelic nondeterminism. They are ordered by inclusion ($\subseteq$). By requiring them to be prefix-closed, we can make sure they incorporate and preserve the refinement ordering of plays.
- Sets of strategies (strategy specifications) permit choices of the *system* as well, understood as demonic nondeterminism. They are ordered by containment ($\supseteq$). By requiring them to be upward closed with respect to strategy inclusion, we can once again incorporate strategy refinement into this ordering.

Under this approach, dual nondeterminism is built into the construction of strategy specifications, allowing us to avoid the pitfalls of previous approaches. The model outlined above is complete and general in a precise way: this construction corresponds to the free completely distributive lattice generated by the underlying poset of plays.

5.3 Free completely distributive completions

Historically, dual nondeterminism in the refinement calculus only operates at the level of statements, in the context of imperative programming. More recently, 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].

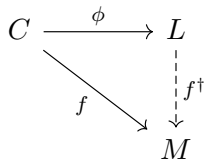
In essence, their approach works by capturing dual nondeterminism as a *monad* constructing free completely distributive lattices. Importantly, this monad operates on *partially ordered sets*

and can incorporate a "ground-level" notion of refinement. This allows dual nondeterminism to be used in a variety of new contexts.

Definition 5.1 (Free completely distributive completion)**.** A complete lattice is called *completely distributive* when it satisfies the following property for all families $(x_{i,j})_{i\in I, j\in J_i}$ of elements:

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

A completely distributive complete 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 complete lattice M and monotonic function $f : C \to M$, there exists a unique complete homomorphism $f^\dagger : L \to M$ such that $f^\dagger \circ \phi = f$:



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

5.3.1 Construction

Morris [2004] gives the following constructions for the free completely distributive completion of a partially ordered set $(A, \leq)$:

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

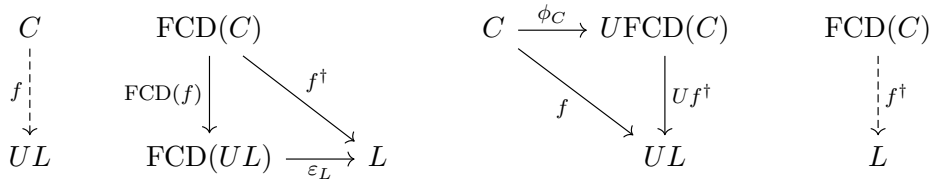
In the expressions above, $\mathcal{D}$ and $\mathcal{U}$ are themselves completions. A *downset* of a poset $(A, \leq)$ is a subset $x \subseteq A$ satisfying the *downward closure* property:

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

Unions and intersections preserve downward closure, giving rise to the *downset lattice* $\mathcal{D}(A, \leq)$. Its elements are the 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.

5.3.2 Categorical characterization

Categorically speaking, the functor $\mathrm{FCD} : \mathbf{Pos} \to \mathbf{CDLat}$ is the left adjoint to the forgetful functor $U : \mathbf{CDLat} \to \mathbf{Pos}$ from the category $\mathbf{CDLat}$ of completely distributive complete lattices and complete homomorphisms to the category $\mathbf{Pos}$ of posets and monotonic functions.



In other words, there is a correspondence:

$$\forall CM \cdot \mathbf{CDLat}(\mathrm{FCD}(C), M) \cong \mathbf{Pos}(C, UM)$$

whereby complete homomorphisms out of a free completely distributive complete lattice are characterized by the image of its generators.

The adjunction's unit corresponds to the function ϕ in Def. 5.1, and embeds the poset C of generators into its free completely distributive completion $\mathrm{FCD}(C)$. The counit ε_L flattens an element of $\mathrm{FCD}(UL)$ by using the inner joins and meets of L to interpret the outer joins and meets added by FCD.

Since $\mathrm{FCD} \dashv U$ are adjoint functors, the composite $U\mathrm{FCD}$ is a monad in $\mathbf{Pos}$, which can be used to model dual nondeterminism as an effect. The monad's unit is given by ϕ, its Kleisli extension is given by $(-)^\dagger$, and its multiplication can be described as $\mu_C := U\varepsilon_{\mathrm{FCD}(C)}$. In the following, I will often identify FCD with the monad $U\mathrm{FCD}$, and refer to the complete homomorphism $f^\dagger$ as the FCD extension of f.

5.3.3 Computational interpretation

Computationally, the FCD monad can be used to model dual nondeterminism as an effect. The computation $\phi(a) \in \mathrm{FCD}(A)$ terminates immediately with the outcome $a \in A$. For a computation $x \in \mathrm{FCD}(A)$ and for $f : A \to \mathrm{FCD}(B)$, the computation $f^\dagger(x) \in \mathrm{FCD}(B)$ replaces any outcome a of x with the computation $f(a)$. As with other monads, I will use the notation $a \leftarrow x; f(a)$ for $f^\dagger(x)$, or simply $x; y$ when f is constant with $f(a) = y$.

A computation $x \in \mathrm{FCD}(A)$ can be understood as a *structured* collection of possible outcomes. More precisely, each element $x \in \mathrm{FCD}(A)$ 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 the 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 $f^\dagger(x) = \bigsqcap_{i \in I} \bigsqcup_{j \in J_i} f(a_{ij})$.

The commutativity and associativity of meets and joins mean that the model is insensitive to *branching*. Complete distributivity further allows angelic and demonic choices to commute, and the status of $f^\dagger$ as a complete homomorphism enables the following properties, where M is any expression monotonic in the variables bound by the left arrows:

$$
\begin{aligned}
a \leftarrow \left(\bigsqcup_{i \in I} x_i \right) ; M \;&=\; \bigsqcup_{i \in I} (a \leftarrow x_i \,;\, M) \\
a \leftarrow \left(\bigsqcap_{i \in I} x_i \right) ; M \;&=\; \bigsqcap_{i \in I} (a \leftarrow x_i \,;\, M) \\
a \leftarrow x \,;\, b \leftarrow y \,;\, M \;&=\; b \leftarrow y \,;\, a \leftarrow x \,;\, M
\end{aligned}
$$

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).

Writing again $\mathbb{1} = \{*\}$ for the unit set, the *assertion* $\{P\} \in \mathrm{FCD}(\mathbb{1})$ of a proposition P evaluates to the unit value $\phi(*)$ when the proposition is true and to $\bot$ otherwise. I will use it to formulate guards blocking a subset of angelic choices.

Note that the complete homomorphism properties of $f^\dagger$ outlined above mean that failure as

denoted by $\bot$ is *global*, in the sense that a computation that fails at any point simply fails overall without yielding any partial result. This is an important point to consider when using FCD to construct models for interactive computations.

5.4 Dually nondeterministic strategies

5.4.1 Angelic choices

As mentioned in §2.4.2, in game semantics strategies are usually constructed as prefix-closed sets of plays. In other words, they are downsets over a set of plays P partially ordered under the prefix relation $\sqsubseteq_{\mathsf{p}}$. From the perspective outlined in §5.2.2, this corresponds to the angelic interpretation of strategies. A play $s \in P$ can be promoted to an elementary strategy $\downarrow s \in \mathcal{D}(P, \sqsubseteq_{\mathsf{p}})$:

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

Set inclusion ($\subseteq$) corresponds to strategy refinement, and the downset completion augments P with arbitrary angelic choices ($\cup$) allowing us to describe more precise behaviors.

Angelic nondeterminism allows us to range over all possible choices of the environment and record the resulting plays. Recall the game interpretation of the statement $\mathsf{x} := 2 * \mathsf{x}$ mentioned at the beginning of §2.4. The strategy associated with this statement is the following prefix-closed set of even-length plays:

$$\begin{aligned}
\sigma \;:=\; [\![\mathsf{x} := 2 * \mathsf{x}]\!] \;&=\; \bigcup_{n \in \mathbb{N}} \downarrow(\mathsf{run} \cdot \underline{\mathsf{read}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{write}_{\mathsf{x}}[2n]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}) \\
&=\; \{\epsilon,\ \mathsf{run} \cdot \underline{\mathsf{read}_{\mathsf{x}}}, \\
&\qquad \mathsf{run} \cdot \underline{\mathsf{read}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{write}_{\mathsf{x}}[2n]}, \\
&\qquad \mathsf{run} \cdot \underline{\mathsf{read}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{write}_{\mathsf{x}}[2n]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}} \;\mid\; n \in \mathbb{N}\}
\end{aligned}$$

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

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

These refinements do not correspond to interpretations of concrete programs, and in game models which seek to achieve definability they are usually excluded. For our purposes, retaining them is algebraically important, and they can in fact appear as intermediate terms in some applications. In the construction above, although δ appears in a system move, it is still associated with an *angelic* choice, in other words 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.

Remark 5.2 (System nondeterminism *vs* environment determinacy)**.** *As mentioned earlier, the situation described above is usually excluded by requiring that strategies do not contain two plays sm_1 and sm_2 where m_1 and m_2 are distinct moves of the system. This is usually understood as enforcing* system determinism*. However, in the present context it corresponds instead to* environment determinacy*: all choices of the environment should be directly observed and recorded in plays.*

5.4.2 Demonic choices

If we wish to allow the *system* to choose an answer in the interval $[2n-1,\ 2n+1]$, we must use a demonic choice instead. The model and refinement lattice presented above are insufficient to express such a specification, because downsets do not add enough meets, forcing the would-be demonic specification to become much coarser:

$$\bigcup_{n\in\mathbb{N}} \bigcap_{-1\leq\delta\leq 1} {\downarrow}(\mathsf{run}\cdot \underline{\mathsf{read}_{\mathsf{x}}}\cdot n\cdot \underline{\mathsf{write}_{\mathsf{x}}[2n+\delta]}\cdot \mathsf{ok}\cdot \underline{\mathsf{done}}) = \{\epsilon,\ \mathsf{run}\cdot \underline{\mathsf{read}_{\mathsf{x}}}\}\,.$$

My approach to dual nondeterminism in game semantics will be to replace $\mathcal{D}$ with FCD in the construction of strategies presented earlier. The permissive *strategy specification* which we attempted to construct above can then be expressed precisely as:

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

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

For the construction $\mathrm{FCD} := \mathcal{UD}$, strategy specifications correspond to sets of traditional strategies, ordered by containment ($\supseteq$). This outer set ranges over demonic choices. Writing

$s_{n,\delta} := \mathsf{run} \cdot \underline{\mathsf{read}_{\mathsf{x}}} \cdot n \cdot \underline{\mathsf{write}_{\mathsf{x}}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}$, the strategy specification $\tilde{\sigma}$ will be encoded as:

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

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

Chapter 6

The interaction specification monad

6.1 Introduction

Chapter 4 presented the theory of certified abstraction layers used to verify CertiKOS, articulating in particular how various components of the theory can be understood in terms of monads and monad homomorphisms:

- client code is expressed in a monad T_E;
- layer specifications are expressed in a monad $\mathcal{L}_S$;
- layer implementations define monad homomorphisms from T_F to T_E;
- layer interfaces define monad homomorphism from T_E to $\mathcal{L}_S$.

In this chapter, I switch to the point of view of refinement-based game semantics to construct the *interaction specification* monad $\mathcal{I}_E$, a variation on the free monad for the effect signature E which incorporates dual nondeterminism and refinement.

The correspondence between the primarily monadic view taken in Chapter 4 and the game semantics view taken in this chapter follows the pattern suggested in §2.6. The interaction specification monad generalizes the various monads used in Chapter 4 along the following lines:

- As a richer version of the free monad, $\mathcal{I}_E$ can represent both client code and layer implementations in a straightforward way.

- The refinement ordering induced by the component $\mathcal{P}^1$ of the monad $\mathcal{L}_S$ is generalized by a completely distributive lattice structure on $\mathcal{I}_E(X)$.
- The state component of $\mathcal{L}_S$ is addressed by extending the signature E to a signature $E@S$ where every question and answer is annotated by a state component in S.

The resulting model is remarkably expressive. It can be used to construct a category of games and strategies along the lines suggested in §2.6.5, and to formulate a theory of certified abstraction layers in which layer interfaces, layer implementations, and simulation relations are treated uniformly and compositionally. In addition, it can embed the *interaction trees* used in various DeepSpec projects [Xia et al., 2019] and can be used to formulate game semantics for the compiler CompCertO presented in Part III of this thesis.

Remark 6.1. *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, but individual components can still make use of high-order languages and reasoning techniques [Mansky et al., 2020].*

6.2 Overview

I follow the general approach to refinement-based game semantics outlined in Chapter 5. Given an effect signature E, we can construct a prefix-ordered set of plays $\bar{P}_E(X)$ corresponding to the possible interactions between a computation with effects in E and its environment. These interactions include the computation's ultimate outcome in X. The interaction specification monad $\mathcal{I}_E(X)$ is then obtained as the free completely distributive completion of the poset $\bar{P}_E(X)$.

For each effect $(m : N) \in E$, the interaction specification monad has an elementary operation $\mathbf{I}_E^m \in \mathcal{I}_E(N)$ which triggers an instance of m and returns its outcome. Given a second effect signature F, a family $(f^m)_{(m:N)\in F}$ of computations $f^m \in \mathcal{I}_E(N)$ can be used to interpret the effects of F into the signature E. This is achieved by a *substitution* operator $(-)[f]$, 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

$(m : N) \in F$ is replaced by the corresponding computation $f^m \in \mathcal{I}_E(N)$.

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

6.3 Plays

I first introduce the partially ordered sets of plays which we use to construct the interaction specification monad. Since the monad is intended to describe *active* computations, I use *odd*-length plays which start with *system* moves, by contrast with the more common approach presented in §2.4.2.

Definition 6.2. 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 $v \in A$, $(m : N) \in E$ and $n \in N$. 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 ($\underline{v}$), or it can trigger an effect $(m : N) \in E$ and wait to be resumed by an answer $n \in N$ of the environment ($\underline{m}ns$). A play which ends 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 begins with the same sequence of questions and answers.

6.4 Interaction specifications

The interaction specification monad can now be defined as the free completely distributive completion of the corresponding poset of plays. For the sake of conciseness and clarity, I 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 $s \in \mathrm{FCD}(P)$. Likewise, for a completely distributive lattice M, we can implicitly promote a monotonic function $f : P \to M$ to its extension $f : \mathrm{FCD}(P) \to M$. These conventions are at work in the following definition.

Definition 6.3. 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:

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

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 Definition 6.3 is the case for $\mu_A^E(\underline{m}ns)$, which includes $\underline{m}$ as well as $\underline{m}n\mu_A^E(s)$. 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 μ_A^E. Consider for example $\underline{m} \sqsubseteq \underline{m}n\underline{\bot} \in \bar{P}_E(\mathcal{I}_E(A))$. Since $\mu_A^E(\underline{\bot}) = \bot$ and the FCD extension of the function $s \mapsto \underline{m}ns$ preserves $\bot$, it is not the case that $\mu_A^E(\underline{m}) = \underline{m} \sqsubseteq \underline{m}n\mu_A^E(\underline{\bot})$.

As usual, the Kleisli extension of a function $f : A \to \mathcal{I}_E(B)$ is the function $f^\dagger = \mu_B^E \circ \mathcal{I}_E(f)$. It can also be given directly as:

$$f^\dagger(\underline{v}) := f(v)$$
$$f^\dagger(\underline{m}) := \underline{m}$$
$$f^\dagger(\underline{m}ns) := \underline{m} \sqcup \underline{m}nf^\dagger(s)$$

I will extend the notations used for FCD to the monad $\mathcal{I}_E$.

6.5 Interaction primitives

The operations of an effect signature E can be promoted to elementary interaction specifications in $\mathcal{I}_E$ in the following way.

Definition 6.4 (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 can use a family of interaction specifications $(f^m)_{(m:N)\in F}$ providing an interpretation $f^m \in \mathcal{I}_E(N)$ of each effect $(m : N) \in F$ 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 §2.2 in the context of free completely distributive lattices; they carry similar

meanings in the context of the interaction specification monad.

Definition 6.5 (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 $(f^m)_{(m:N)\in F}$ with $f^m \in \mathcal{I}_E(N)$, 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.

6.6 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 6.6 (Morphisms)**.** Consider the effect signatures E, F and G. I will write $f : E \to F$ whenever $(f^m)_{(m:N)\in F}$ is a family of interactive computations such that $f^m \in \mathcal{I}_E(N)$. For the morphisms $f : E \to F$ and $g : F \to G$, their composite $g \circ f : E \to G$ can be defined as:

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

The completely distributive lattice structure of $\mathcal{I}_F(-)$ can be extended pointwise to morphisms. For a family $(f_i)_{i\in I}$ with $f_i : E \to F$, the morphisms $\left(\bigsqcup_{i\in I} f_i\right) : E \to F$ and $\left(\bigsqcap_{i\in I} f_i\right) : E \to F$ are defined as:

$$\left(\bigsqcup_{i\in I} f_i\right)^m := \bigsqcup_{i\in I} f_i^m \qquad \left(\bigsqcap_{i\in I} f_i\right)^m := \bigsqcap_{i\in I} f_i^m\,,$$

For $f, g : E \to F$ refinement is defined 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, the model presented here introduces arbitrary demonic choices and relaxes all 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 6.7. *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 define a category of games and strategies.

Definition 6.8. 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.

6.7 Products

Definition 4.13 in §4.3.1 introduces the *sum* $\bigoplus_{i\in I} E_i$ of a family of signatures $(E_i)_{i\in I}$. As demonstrated below, this construction can be used to define products in the category $\mathcal{G}^{ib}_{\sqsubseteq}$.

Theorem 6.9. *The category $\mathcal{G}^{ib}_{\sqsubseteq}$ has all products. Objects are given by the sum of signatures, so that*

$$\prod_{i\in I} E_i \;:=\; \bigoplus_{i\in I} E_i\,.$$

Projection arrows are given for each $i \in I$ by the morphism defined as:

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

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 $\langle f_i\rangle_{i\in I} : X \to \prod i \in I E_i$ such that for all $i \in I$:

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

Note that for $x : X \to \prod i \in I E_i$, $i \in I$ 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:

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

□

Note in particular that the empty signature $\varnothing$ is the terminal object. Hence, I will write 1 when I refer to the empty signature as an object of $\mathcal{G}^{ib}_{\sqsubseteq}$.

6.8 Certified abstraction layers

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

6.8.1 Layer implementations

A layer implementation M with an underlay signature E and an overlay signature F can be interpreted as a strategy $[\![M]\!] : E \to F$ in $\mathcal{G}^{ib}_{\sqsubseteq}$ in a straightforward manner:

$$[\![M]\!]^m \;:=\; M.m[\mathbf{I}_E]$$

Expanding slightly, the computation $M.m \in T_E(N)$ is mapped to $(M.m)[\mathbf{I}_E] \in \mathcal{I}_E$ using the monad homomorphism $(-)[\mathbf{I}_E] : T_E \to \mathcal{I}_E$ described by:

$$\underline{m}(t_n)_{n\in N}[\mathbf{I}_E] := n \leftarrow \mathbf{I}_E^m \,;\, t_n[\mathbf{I}_E] \in \mathcal{I}_E(N) \qquad\qquad (m : N) \in E$$

Remark 6.10. *A strategy $f : E \to F$ in $\mathcal{G}^{ib}_{\sqsubseteq}$ corresponds to a monad homomorphism $T_F \to \mathcal{I}_E$. Above, the strategy $[\![M]\!]$ is obtained as the composite monad homomorphism $\mathbf{I}_E \circ M$, where $\mathbf{I}_E$ is seen as a monad homomorphism of type $T_E \to \mathcal{I}_E$ and M is seen as $T_F \to T_E$.*

6.8.2 Layer interfaces

To embed layer interfaces, we need to make their abstract states visible in the interaction by extending signatures in the following way:

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

Since the specifications in a layer interface $L = \langle E, S, \sigma\rangle$ have the type $\sigma^m : S \to \mathcal{P}^1(N \times S)$, they then be interpreted as a morphism $[\![L]\!] : 1 \to E@S$ almost directly, using the join operation in $\mathcal{I}_\varnothing(N \times S)$ to interpret $\varnothing$ as $\bot$ and a singleton $\{n@k'\}$ as the corresponding outcome:

$$[\![L]\!]^{m@k} := \bigsqcup_{n@k' \in \sigma^m(k)} n@k'$$

6.8.3 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 $[\![M]\!]@S \circ [\![L]\!] : 1 \to F@S$.

Note that $(g \circ f)@S = g@S \circ f@S$ and $\mathbf{I}_E@S = \mathbf{I}_{E@S}$, so that $-@S : \mathcal{G}^{ib}_{\sqsubseteq} \to \mathcal{G}^{ib}_{\sqsubseteq}$ is in fact a functor.

6.8.4 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:

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

I 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:

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

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

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 σ 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 into 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 σ, 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 τ implements with respect to R. For an overlay state $k_2 \in S_2$, τ 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 τ 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 6.11. *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 7

Stateful and reentrant strategies

The model presented in Chapter 6 is designed to be simple but general enough to embed CompCert semantics, certified abstraction layers, and interaction trees. I now sketch a more general model which allows strategies to retain state across successive activations. I 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.

7.1 Overview

As discussed in §6.6, 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 I 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 (§7.2–§7.5), I define an embedding of $\mathcal{G}^{ib}_{\sqsubseteq}$ into $\mathcal{G}^{b}_{\sqsubseteq}$ (§7.6), and show how the states used by a strategy $\sigma : E@S \to F@S$ can be internalized and hidden from its interactions (§7.7).

7.2 Games

To facilitate reasoning, and make it easier to describe operators on strategies in a systematic way, I will 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 7.1. A *game signature* Γ is a set of players with a distinguished element O, together with an effect signature $\Gamma(u, v)$ for all $u, v \in \Gamma$. The operations $(m : N) \in \Gamma(u, v)$ are called the *questions* of u to v, and the elements $n \in N$ are called *answers* to the question m.

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

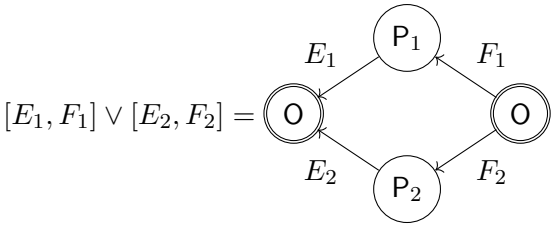
$$[E, F] = \mathsf{P} \underset{F}{\overset{E}{\rightleftarrows}} \mathsf{O}$$

When we consider the ways in which questions propagate through a game signature, the distinguished player O serves the role of both a source and sink. As such, it is visually useful to depict O as two nodes, one capturing the incoming edges of O, and one capturing its outgoing edges. For example, the following game signature generates the global interaction sequences used to define the composition of the strategies $\sigma : E \to F$ and $\tau : F \to G$. The player P_1 corresponds to σ, the player P_2 corresponds to τ, and the strategies are made to interact over the intermediate game F:

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

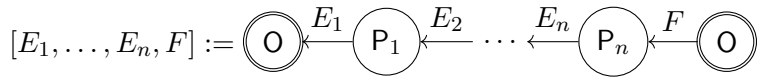
As another example of a game signature, a situation where $\sigma_1 : E_1 \to F_1$ and $\sigma_2 : E_2 \to F_2$

interact with the environment independently of one another can be described as:



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

Definition 7.2 (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:



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 Γ_i, the wedge sum has a corresponding question $\iota_i(m) : \iota_i(u) \to \iota_i(v)$.

7.3 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 corresponding returns. At any point

in a play over a signature Γ, its possible evolutions are characterized by the stack of pending questions.

Definition 7.3. For a game signature Γ and a player $p \in \Gamma$, a *p-stack* over Γ 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 $m_i \in \Gamma(p_{i-1}, p_i)$. I 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 Γ will be played.

Definition 7.4 (Strategy specifications)**.** For a signature Γ, the graph $\hat{\Gamma}$ is defined as follows. The vertices of $\hat{\Gamma}$ are pairs (u, κ) in which $u \in \Gamma$ and κ is a u-stack. For each question $(m{:}N) \in \Gamma(u, v)$ and stack $\kappa : \mathsf{O} \twoheadrightarrow u$, there is an edge:

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

Then, for each answer $n \in N$ there is an edge:

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

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

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

7.4 Operations on strategies

Definition 7.5. A *transformation* from the game signature Γ_1 to the game signature Γ_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 Γ_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 can extend f itself to the paths in Γ_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 7.6 (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 each operation $(m : N) \in \Gamma(u, v)$ and for a stack $\kappa : \mathsf{O} \twoheadrightarrow u$, the image of the move $m : (u, \kappa) \to (v, \kappa m)$ is the path:

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

Then for $n \in N$, 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 the notation $n^{|f(m)|}$ corresponds to a sequence $nn \cdots n$ of copies of the answer $n \in N$ 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.

7.4.1 Composition

When composing the strategy specifications $\sigma \in S\,[E, F]$ and $\tau \in S\,[F, G]$ to obtain $\tau \circ \sigma \in S\,[E, G]$, the transformation $\psi^{\mathsf{c}}_X : [E, F, G] \to [E, G]$ can be used 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:

$$\psi^c(s_1, s_2) := \bigsqcup_{s \in P[E,F,G]} \{\psi^c_1(s) \sqsubseteq s_1 \wedge \psi^c_2(s) \sqsubseteq s_2\}; \psi^c_X(s)\,.$$

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

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

7.4.2 Identity

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

$$[E] = \; E \circlearrowright \mathsf{O}$$

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 id_E is defined as:

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

7.4.3 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 $\sigma_1 \otimes \sigma_2 \in S\,[E_1 \oplus E_2,\ F_1 \oplus F_2]$ defined using interaction sequences in $\Gamma = [E_1, F_1] \vee [E_2, F_2]$. The external projection $\psi_X^{\otimes} : \Gamma \to [E_1 \oplus E_2,\ F_1 \oplus F_2]$ is:

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

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

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

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

$$\psi^{\otimes}(s_1, s_2) := \bigsqcup_{s \in P\Gamma} \{\psi_1^{\otimes}(s) \sqsubseteq s_1 \wedge \psi_2^{\otimes}(s) \sqsubseteq s_2\}; \psi_X^{\otimes}(s)$$

The tensor product can then be defined as:

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

7.5 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 §7.6 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 σ in interactions that involve both of them.

7.6 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 environment question, to construct a corresponding strategy $W\,f \in \mathcal{G}^b_{\sqsubseteq}(E, F)$ 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^κ_Γ 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

$$\omega^\kappa : P^\kappa_\Gamma \to \mathrm{FCD}(P^\kappa_\Gamma)$$

which *prepends* an arbitrary number of copies of f to a play of P^κ_Γ. Starting with $\omega^\kappa_0(t) := t$, we construct a series of approximations:

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

For $(q : R) \in F$, the auxiliary construction $\bar{\omega}^{\kappa q}$ of type

$$\bar{\omega}^{\kappa q} : \bar{P}_E(R) \times P^\kappa_\Gamma \to \mathrm{FCD}(\bar{P}^{\kappa q}_\Gamma)$$

embeds an interaction $s \in \bar{P}_E(R)$, inserting reentrant calls as appropriate, and continues with the

play t if s terminates:

$$\bar{\omega}_i^{\kappa q}(\underline{v}, t) = vt$$
$$\bar{\omega}_i^{\kappa q}(\underline{m}, t) = m\,\omega_i^{\kappa q m}(\epsilon)$$
$$\bar{\omega}_i^{\kappa q}(\underline{m}ns, t) = m\,\omega_i^{\kappa q m}(n\,\bar{\omega}_i^{\kappa q}(s, t))$$

The index i limits both the number of sequential and reentrant copies of f which are instantiated. The strategy specification associated to f in $\mathcal{G}_{\sqsubseteq}^{b}$ is:

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

7.7 Hiding state

The functor $W : \mathcal{G}_{\sqsubseteq}^{ib} \to \mathcal{G}_{\sqsubseteq}^{b}$ can be used to embed the layer theory defined in §6.8 as-is. In addition, the *state* of layer interfaces can be propagated across consecutive calls and eliminated from the representation.

Definition 7.7. The *state-free observation* at $k_0 \in S$ of a partial play $s : (O, \kappa) \nrightarrow (O, \kappa')$ 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 σ/k_0 is first activated, σ is passed the initial state k_0. Then, whenever σ makes a move $m@k$, σ/k_0 removes k from the visible interaction, but remember it in order to adjoin it to the next incoming move.

Part III

Compiling certified open components

Chapter 8

Semantics in CompCert

This chapter describes the semantic model used in CompCert [Leroy, 2009], as well as extensions which have been proposed to make it more compositional. There is no novelty in the technical material presented, however game semantics and dual nondeterminism provide a powerful lens through which the existing design and extensions can be examined and their more sophisticated aspects 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*.

8.1 Whole-program semantics

The semantics of CompCert languages are given in terms of a simple notion of process behavior. By *process*, I 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.

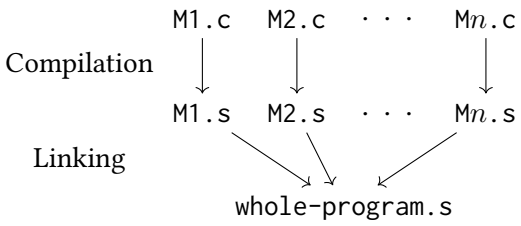


Figure 8.1: CompCert's approximation of the C toolchain

The model used for verifying CompCert accounts for this in the way depicted in Figure 8.1. 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 modeled as:

$$L_{\mathsf{tgt}} := \mathsf{Asm}[\texttt{M1.s} + \cdots + \texttt{Mn.s}] .$$

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:

$$L_{\mathsf{src}} := \mathsf{Clight}[\texttt{M1.c} + \cdots + \texttt{Mn.c}] .$$

Given an appropriate notion of refinement, the correctness of CompCert can then be stated as:

$$L_{\mathsf{src}} \sqsubseteq L_{\mathsf{tgt}} .$$

8.1.1 Transition systems

Language semantics are given as labelled transition systems (LTS), which characterize a program's behavior in terms of sequences of observable events taken from a fixed set $\mathbb{E}$.

Schematically, a CompCert LTS is a tuple $L = \langle S, I, \rightarrow, F \rangle$ consisting of a set of states S, a subset $I \subseteq S$ of initial states, a labelled transition relation $\rightarrow \subseteq S \times \mathbb{E}^* \times S$, 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 $t \in \mathbb{E}^*$.

The execution of L starts by selecting an initial state s_0, then uses the transition relation to

[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.

repeatedly update the state, until a final state is reached:

$$I \ni s_0 \xrightarrow{t_1} s_1 \xrightarrow{\epsilon} s_2 \xrightarrow{t_2} s_3 \; F \; r$$

Roughly speaking, the externally observable behavior of the program consists of the sequence of events generated by transitions ($t_1 t_2$ in the execution above), together with the exit status of the process (r in the execution above). See the next section for a more detailed account.

Note that nondeterministic choices are potentially involved at every step of the execution: the selection of an initial state from the set I, the selection of a possible transition $\{(t, s') \mid s \xrightarrow{t} s'\}$, and the selection of an outcome $\{r \mid s \; F \; r\}$. If there is a possible transition out of a final state s, there will also be a choice of whether the execution should terminate or continue.

The angelic, demonic, or mixed nature of these choices is key to understanding CompCert's notions of *determinacy* and *receptiveness* of transition systems, and the distinction and correspondence between CompCert's notions of *forward* and *backward* simulations.

8.1.2 Behaviors

I outlined above the execution of a labelled transition system in terms of terminating traces. The model used in CompCert is more general, and defines four kinds of behaviors:

- As before, 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:

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

  I will write this behavior as

  $$t_1 t_2 \cdots t_{n-1} \Downarrow r \, .$$

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

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

I will write the corresponding behavior as

$$t_1 t_2 \cdots t_{n-1} \Uparrow .$$

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

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

  Then the behavior of the transition system is represented by the infinite sequence

$$t_1 t_2 t_3 \cdots .$$

- 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 $s_n \xrightarrow{t} s'$ and no r such that $s_n \ F \ r$. The corresponding, partially defined behavior is written

$$t_1 t_2 \cdots t_{n-1} \lightning .$$

  It can be refined by any behavior admitting $t_1 t_2 \cdots t_{n-1}$ as a prefix.

A transition system with no initial state ($I = \varnothing$) immediately goes wrong. In this case, the transition system will admit the single undefined behavior $\epsilon \lightning$.

Summing up, the traces used by CompCert to define the possible behaviors of transition systems are taken from the following language:

$$b \in \mathbb{B} = \mathbb{E}^* \{ \Downarrow r, \Uparrow, \lightning \mid r \in \mathsf{int} \} \cup \mathbb{E}^\omega .$$

I will write $\sqsubseteq_{\lightning} \subseteq \mathbb{B} \times \mathbb{B}$ for CompCert's notion of *behavior improvement*, which allows behaviors

which go wrong to be refined by more defined ones. It can be defined as

$$\sqsubseteq_{\lightning} := \{(b, b) \mid b \in \mathbb{B}\} \cup \{(t\lightning, tb) \mid t \in \mathbb{E}^*, b \in \mathbb{B}\}.$$

Then the overall external behavior of a transition system L is characterized as a set $\mathsf{Beh}(L) \subseteq \mathbb{B}$ of possible individual behaviors. In the remainder of this section I will describe the techniques used in CompCert to define and establish corresponding notions of refinement.

8.1.3 Events

The events used in the traces above are taken from the following set:

$$e \in \mathbb{E} ::= \mathsf{syscall}[u, \vec{v}]/v \mid \mathsf{vload}[i, o]/v \mid \mathsf{vstore}[i, o, v] \mid \mathsf{annot}[u, \vec{v}]$$

They correspond to system calls (syscall), volatile loads and stores (vload, vstore), and annotations (annot). The data involved are strings ($u \in \mathsf{string}$), global identifiers ($i \in \mathsf{ident}$), pointer offsets ($o \in \mathbb{Z}$) and event values ($v \in \mathsf{eval}$, $\vec{v} \in \mathsf{eval}^*$). Crucially, these data are invariant between the source and target programs, so that the corresponding traces can be compared directly. For the purposes of this discussion, the exact nature and use of these events are not important, but they correspond to the external interactions which need to be preserved between the execution of the source and target program.

Events involve both an output component, chosen by the program and written here to the left of the oblique bar (/), and a (possibly trivial) input component, chosen by the environment and written to the right of the oblique. In CompCert, this distinction is formulated as a "match_traces" relation which identifies events that have the same output component. It can be described as the smallest equivalence relation $\simeq$ containing:

$$\mathsf{syscall}[u, \vec{v}]/v_1' \simeq \mathsf{syscall}[u, \vec{v}]/v_2' \qquad \mathsf{vload}[i, o]/v_1 \simeq \mathsf{vload}[i, o]/v_2$$

Equivalently, the set of events can be described as an effect signature and used to interpret the behaviors of transition systems in terms of strategies.

Nondeterminism	Transition systems	Sets of traces
Angelic interpretation	Forward simulations ($\leq$)	Trace inclusion ($\mathcal{P}^{\leq}(\sqsubseteq_{\not\downarrow})$)
Demonic interpretation	Backward simulations ($\geq$)	Trace containment ($\mathcal{P}^{\geq}(\sqsubseteq_{\not\downarrow})$)

Table 8.1: Notions of refinement in CompCert semantics

8.1.4 Games

Although CompCert semantics are not formulated explicitly as game semantics, understanding them in that framework will be useful when we consider the extensions I present in §8.2 and in Chapter 9. Event traces correspond to an interaction over the signature

$$\begin{aligned}\mathcal{E} := \{&\mathsf{syscall} : \mathsf{string} \times \mathsf{eval}^* \to \mathsf{eval},\\ &\mathsf{vload} : \mathsf{ident} \times \mathbb{Z} \to \mathsf{eval},\\ &\mathsf{vstore} : \mathsf{ident} \times \mathbb{Z} \times \mathsf{eval} \to \mathbb{1},\\ &\mathsf{annot} : \mathsf{string} \times \mathsf{eval}^* \to \mathbb{1}\}\,.\end{aligned}$$

Likewise, the initial invocation of a program and its ultimate outcome can be understood as an interaction over the signature $\mathcal{W} := \{* : \{\Downarrow r, \Uparrow \mid r \in \mathsf{int}\}\}$. Overall, the external behavior of a CompCert transition system corresponds to a strategy for the game $\mathcal{E} \to \mathcal{W}$.

However, the precise behavior of a transition system as a strategy of this kind depends on how nondeterminism is interpreted:

- The angelic interpretation is used implicitly in the context of *forward simulations*, which entail a *trace inclusion* property for the corresponding sets of behaviors.
- The demonic interpretation is used implicitly in the context of *backward simulations*, which entail under some *receptiveness* assumptions a corresponding *trace containment* property.

This is outlined in Table 8.1 and explained in the following sections.

8.1.5 Angelic interpretation

Under an angelic interpretation of nondeterminism, we look at the choice between two possible transitions $s \xrightarrow{t_1} s_1$ and $s \xrightarrow{t_2} s_2$ purely as a choice of the *environment*. In terms of sets of

behaviors, the corresponding notion of refinement is the trace inclusion relation $\mathcal{P}^{\leq}(\sqsubseteq_{\lightning})$, which guarantees that all choices available to the environment in the source program are available in the target program as well.

Formally speaking, the angelic treatment of nondeterminism yields a more homogeneous theory than its demonic counterpart. A behavior $b \in \mathbb{B}$ can be interpreted in $\mathcal{I}_{\mathcal{E}}$ as:

$$\llbracket \Downarrow r \rrbracket := \eta(\Downarrow r) \qquad \llbracket \lightning \rrbracket := \bot$$
$$\llbracket \Uparrow \rrbracket := \eta(\Uparrow) \qquad \llbracket m/n \cdot b \rrbracket := mn \llbracket b \rrbracket$$

Then $\mathsf{Beh}(L)$ can be interpreted as a strategy in $\mathcal{E} \to \mathcal{W}$ given by:

$$\llbracket \mathsf{Beh}(L) \rrbracket^* := \bigsqcup_{b \in \mathsf{Beh}(L)} \llbracket b \rrbracket ,$$

where $* \in \mathcal{W}^\circ$ denotes the only question in the signature $\mathcal{W}$. Trace inclusion and refinement then coincide in the following way:

$$\mathsf{Beh}(L_1) \; [\mathcal{P}^{\leq}(\sqsubseteq_{\lightning})] \; \mathsf{Beh}(L_2) \;\Leftrightarrow\; \llbracket \mathsf{Beh}(L_1) \rrbracket \sqsubseteq \llbracket \mathsf{Beh}(L_2) \rrbracket .$$

Often, the angelic interpretation of nondeterminism is also the most convenient one when it comes to proving refinement between two transition systems, as in the case of correctness proofs for a specific compiler pass. For transition systems, correctness can be proved using simulations of the following kind, which assert that every transition of the source program is matched by a corresponding transition sequence in the target program. Note that the target sequence may be empty, but in order to ensure the preservation of silent divergence, we must ensure this happens for at most finitely many steps in the source program.

Definition 8.1. A *forward simulation* of a transition system $L_1 = \langle S_1, I_1, \rightarrow_1, F_1 \rangle$ by a transition system $L_2 = \langle S_2, I_2, \rightarrow_2, F_2 \rangle$ is given by a relation $R \subseteq W \times S_1 \times S_2$ indexed by a well-founded order $(W, <)$ such that the following properties hold:

- For all $s_1 \in I_1$, there exist $i \in W$ and $s_2 \in I_2$ such that $(i, s_1, s_2) \in R$.
- For all $(i, s_1, s_2) \in R$ and for every transition $s_1 \xrightarrow{t}_1 s_1'$, there exist $i' \in W$ and $s_2' \in S_2$

such that $(i', s_1', s_2') \in R$ and either $s_2 \xrightarrow{t}_2^+ s_2'$ or $s_2 = s_2' \wedge i' < i$.

- For all $(i, s_1, s_2) \in R$ and every result r such that $s_1\ F_1\ r$, there exists s_2' such that $s_2 \xrightarrow{\epsilon}_2^* s_2'$ and $s_2'\ F_2\ r$.

I will write $L_1 \leq L_2$.

CompCert also includes in `common/Smallstep.v` a number of simplified formulations which can be used to establish forward simulations in simpler situations. Forward simulations imply trace inclusion:

$$L_1 \leq L_2 \ \Rightarrow\ \mathsf{Beh}(L_1)\ [\mathcal{P}^{\leq}(\sqsubseteq_t)]\ \mathsf{Beh}(L_2)$$

This is proved as `forward_simulation_behavior_improves` in `common/Behaviors.v`.

While the angelic interpretation of nondeterminism is the most convenient one to work with, it does not correspond to the intended interpretation of CompCert transition systems, detailed in the next section, where multiple possible transitions correspond primarily to choices of the system. However, under some conditions the two interpretations coincide. This makes it possible to use forward simulations to prove semantic preservation properties for most of CompCert's compilation passes.

8.1.6 Demonic interpretation

Under a demonic interpretation of nondeterminism, we look at the choice between two possible transitions $s \xrightarrow{t_1} s_1$ and $s \xrightarrow{t_2} s_2$ as a choice of the *system*. In terms of traces, the corresponding notion of refinement is the *trace containment* relation $\mathcal{P}^{\geq}(\sqsubseteq_t)$, which guarantees that all possible behaviors of the target program are permitted by the specification (source program).

This elegant story is complicated by two phenomena involving angelic as well as demonic nondeterminism:

- While the choice between multiple possible transitions should be interpreted as demonic, stuck states should still denote undefined behaviors. In other words, when there are *no* possible transitions, the "choice" between them should be interpreted angelically as $\bot$, rather than demonically as $\top$.

- When interactions with the environment are involved in the form of non-empty event traces, *outputs* should be interpreted as demonic, but *inputs* should still correspond to choices of the environment.

The first phenomenon creates a discontinuity in the interpretation of sets of possible actions, as illustrated by the following example.

Example 8.2. *Consider the following family of transition systems, where the initial state* i *may admit zero, one, or more transitions to the final state* f*:*

$$
\begin{array}{lll}
S := \{\mathsf{i}, \mathsf{f}\} & L_0 := \{S, I, \to_0, F\} & \to_0 := \varnothing \\
I := \{\mathsf{i}\} & L_1 := \{S, I, \to_1, F\} & \to_1 := \to_0 \cup \{(\mathsf{i}, e_1, \mathsf{f})\} \\
F := \{(\mathsf{f}, 42)\} & L_2 := \{S, I, \to_2, F\} & \to_2 := \to_1 \cup \{(\mathsf{i}, e_2, \mathsf{f})\} \\
 & \vdots & \vdots
\end{array}
$$

The corresponding sets of behaviors are as follows:

$$
\begin{aligned}
&\mathsf{Beh}(\mathsf{L}_0) = \{\lightning\} \\
&\mathsf{Beh}(\mathsf{L}_1) = \{e_1 \Downarrow 42\} \\
&\mathsf{Beh}(\mathsf{L}_2) = \{e_1 \Downarrow 42,\ e_2 \Downarrow 42\} \\
&\quad\vdots \qquad\qquad\qquad\qquad \ddots
\end{aligned}
$$

As a consequence of the discontinuity introduced by $\lightning$*, we obtain the ordering*

$$L_0 \sqsubseteq \cdots \sqsubseteq L_3 \sqsubseteq L_2 \sqsubseteq L_1$$

where L_0 *is pushed to the bottom.*

While so far I have only mentioned choices between transitions introduced by the step relation $\to$, initial and final states also participate in this phenomenon. In general, we can consider the way CompCert interprets nondeterminism in a set of possible actions A, depending on the set's

cardinality. When $|A| \leq 1$, nondeterminism is interpreted as angelic:

$$\bigsqcup A$$

When $|A| \geq 1$, nondeterminism is interpreted as demonic:

$$\bigsqcap A$$

In other words, CompCert uses the following form of *discontinuous choice*:

$$\bigoplus A := \begin{cases} \bot & \text{if } A = \varnothing \\ \bigsqcap A & \text{otherwise} \end{cases}$$

Concretely, this manifests in the form of additional *safety* clauses in the definition of backward simulations as given below. A state $s \in S$ in a transition system $\langle S, I, \to, F \rangle$ is called *safe* if it does not go wrong. A state $s \in S$ *goes wrong* if there exists $s \xrightarrow{\epsilon}{}^{*} s'$ such that s' is stuck (it has no successor and is not a final state).

Definition 8.3. A *backward simulation* of a transition system $L_1 = \langle S_1, I_1, \to_1, F_1 \rangle$ by a transition system $L_2 = \langle S_2, I_2, \to_2, F_2 \rangle$ is given by a relation $R \subseteq W \times S_1 \times S_2$ indexed by a well-founded order $(W, <)$ such that the following properties hold:

- If $I_1 \neq \varnothing$, then $I_2 \neq \varnothing$.
- If $I_1 \neq \varnothing$ and $s_2 \in I_2$, there exist $i \in W$ and $s_1 \in I_1$ such that $(i, s_1, s_2) \in R$.
- For all $(i, s_1, s_2) \in R$ with s_1 a *safe* state, s_2 either has a successor or is a final state.
- For all $(i, s_1, s_2) \in R$ with s_1 a *safe* state, and for every transition $s_2 \xrightarrow{t}_2 s_2'$, there exist $i' \in W$ and $s_1' \in S_1$ such that $(i', s_1', s_2') \in R$ and either $s_1 \xrightarrow{t}_1^{+} s_1'$ or $s_1 = s_1' \wedge i' < i$.
- For all $(i, s_1, s_2) \in R$ with s_1 a *safe* state, and every result r such that $s_2 \ F_2 \ r$, there exists s_1' such that $s_1 \xrightarrow{\epsilon}_1^{*} s_1'$ and $s_1' \ F_1 \ r$.

I will write $L_1 \geq L_2$.

As with forward simulations, stronger but simplified versions of these criteria are provided in `common/Smallstep.v`. Backward simulations imply trace containment:

$$L_1 \geq L_2 \;\Rightarrow\; \mathsf{Beh}(L_1)\ [\mathcal{P}^{\geq}(\sqsubseteq_{\not t})]\ \mathsf{Beh}(L_2)$$

This is proved as `backward_simulation_behavior_improves` in `common/Behaviors.v`.

8.1.7 Dual interpretation

To formulate a proper game semantics for CompCert transition systems, we must also address the distinction between outputs and inputs mentioned at the beginning of the previous section. In fact, this underscores that the notions of refinement formulated in CompCert are only satisfactory under some conditions:

- The trace inclusion property $\mathsf{Beh}(L_1)\ \mathcal{P}^{\leq}(\sqsubseteq_{\not t})\ \mathsf{Beh}(L_2)$ is only appropriate when the target transition system L_2 is *determinate*, in other words *only* contains nondeterminism over environment inputs.
- The trace containment property $\mathsf{Beh}(L_1)\ \mathcal{P}^{\geq}(\sqsubseteq_{\not t})\ \mathsf{Beh}(L_2)$ is only appropriate when the target transition system L_2 is *receptive*, in other words is *maximally* nondeterministic over environment inputs.

By eliminating variation in one kind of nondeterminism, these conditions allow trace refinement properties to focus on the other kind and to work in one direction only.

Expressing these properties at the level of individual states is made complicated when transitions can include traces of variable and mismatching lengths. For this reason the corresponding definitions below require transitions to have at most one event.

Definition 8.4 (Determinacy and receptiveness)**.** A transition system $\langle S, I, \rightarrow, F\rangle$ has *single-event transitions* when $|t| \leq 1$ for all $s \xrightarrow{t} s'$.

A single-event transition system is called *determinate* if whenever $s \xrightarrow{e_1} s_1$ and $s \xrightarrow{e_2} s_2$ the properties $e_1 \simeq e_2$ and $e_1 = e_2 \Rightarrow s_1 = s_2$ hold.

A single-event transition system is called *receptive* if for all transitions $s \xrightarrow{e_1} s_1$ and all events $e_2 \simeq e_1$, there exists s_2 such that $s \xrightarrow{e_2} s_2$.

Game semantics For a single-event transition system $L = \langle S, \rightarrow, I, F \rangle$, the associated strategy can be formulated as follows. First, to compute the behavior $[\![s]\!] \in \mathcal{I}_{\mathcal{E}}(\mathcal{W}^\bullet)$ of a state $s \in S$, we examine the sets of possible actions which can be taken by L for the state s, then perform a discontinuous choice between them. This is formalized as the least $[\![s]\!]$ satisfying the equations:

$$
\begin{aligned}
[\![s]\!] &= \bigoplus \left(A_\epsilon(s) \cup A_{\mathsf{int}}(s) \cup A_{\Downarrow}(s) \cup A_{\Uparrow}(s) \right) \\
A_\epsilon(s) &= \{ [\![s']\!] \mid s \xrightarrow{\epsilon} s' \} \\
A_{\mathsf{int}}(s) &= \left\{ n \leftarrow m \,;\, \prod_{s \xrightarrow{m/n} s'} [\![s']\!] \;\middle|\; m \in \mathcal{E}^\circ \wedge \exists n\, s' \cdot s \xrightarrow{m/n} s' \right\} \\
A_{\Downarrow}(s) &= \{ \eta(\Downarrow r) \mid r \in \mathsf{int} \wedge s\ F\ r \} \\
A_{\Uparrow}(s) &= \{ \Uparrow \mid s \text{ silently diverges} \}
\end{aligned}
$$

Note that since $\mathcal{I}_{\mathcal{E}}(\mathcal{W}^\bullet)$ is a complete lattice, we can compute the applicable fixpoint without difficulty. Then, the overall behavior $[\![L]\!] : \mathcal{E} \rightarrow \mathcal{W}$ can be obtained by defining

$$
[\![L]\!]^* = \bigoplus_{s \in I} [\![s]\!] \,.
$$

Again, here $*$ denotes the only question in the signature $\mathcal{W}$.

Converting forward to backward simulations When L_1 is receptive and L_2 is determinate, a forward simulation $L_1 \leq L_2$ can be converted into a backward simulation $L_1 \geq L_2$. This is proved in CompCert as `forward_to_backward_simulation` in the file `common/Smallstep.v`. Note that in this case, the states of L_2 which correspond to safe states of L_1 will be receptive as well, so that the backward simulation will entail refinement:

$$
L_1 \leq L_2 \;\Rightarrow\; L_1 \geq L_2 \;\Rightarrow\; [\![L_1]\!] \sqsubseteq [\![L_2]\!]
$$

8.1.8 Memory model

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 8.2: Outline of the CompCert 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 Figure 8.2. 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 retrieved 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).

8.2 Compositional extensions

The model outlined above describes only the behavior of whole programs, and makes no attempt to describe interactions across components. Likewise, the compiler correctness property given

in §8.1 only considers uses which follow the pattern approximated in Figure 8.1. This makes it challenging to reason about the behavior of individual compilation units. 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 I introduce in Chapter 9 achieves this by modeling control transfers explicitly as part of the external behavior of components, and serves as the basis for a new extension of CompCert named CompCertO. However, there already exists multiple extensions to the semantic model and correctness theorem of CompCert which address some of limitations in the original approach. I present some of them in this section. The conceptual framework of game semantics can be used to classify these extensions. By reinterpreting these models in terms of games and strategies, we can establish the taxonomy presented in Table 8.2.

While I focus here on CompCert, a more general survey, discussion and synthesis of various compositional compiler correctness results is provided by Patterson and Ahmed [2019].

8.2.1 CompCert and SepCompCert

As noted in §9.1, the whole-program semantics used by CompCert can be interpreted in terms of strategies for the game $\mathbf{1} \twoheadrightarrow \mathcal{W}$. CompCert's original correctness theorem stated the refinement property $\mathsf{C}[p] \sqsubseteq \mathsf{Asm}[p']$, where $\mathsf{C}[-]$ and $\mathsf{Asm}[-]$ 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 §8.1.

Since external calls are not accounted for explicitly in this semantic model, they are interpreted using a common global parameter χ specifying their behavior. The correctness proof assumes that χ is deterministic and that it satisfies a number of healthiness requirements with respect to the memory transformations used in CompCert's correctness proof.

8.2.2 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. As discussed in §4.4, 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 χ when the next layer is verified. However, this approach does not support mutually recursive components, and requires the healthiness conditions on χ to be proved before the next layer is added.

8.2.3 Compositional CompCert

The *interaction semantics* of Compositional CompCert [Stewart et al., 2015] provide a form of game semantics for program components, by extending the transition systems of CompCert to describe cross-component interactions, and adapting forward simulations to function with the new model.

Compositional CompCert introduces a notion of *semantic linking* similar to the horizontal composition operator which I will present in §9.2.1. Semantic linking is shown to preserve simulations, but 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. To take into account cross-components interactions, the notions of memory injections and simulations used in Compositional CompCert are significantly more complex than the original ones, and simulation proofs essentially had to be rewritten and adapted to fit the new framework.

8.2.4 CompCertM

The most recent extension of CompCert is CompCertM [Song et al., 2019], which shares common themes and was developed concurrently with my 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 parameterized by Kripke relations similar to CKLRs (§10.1) and predicates similar to my invariants (§10.2). 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.

Nevertheless, because the correctness theorem is only expressed in terms of closed semantics, it is difficult to deploy CompCertM in the context of an open-ended framework based on game semantics. The following section articulates some of these obstacles.

8.3 Limitations

8.3.1 Decomposing heterogeneous systems

The work summarized above turns CompCert into a platform enabling compositional verification. 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 heterogeneous 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 irrelevant 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 §7). The driver code would be specified (σ_{drv}) and verified at the level of CompCert semantics, whereas device I/O primitives (σ_{io}) and the NIC (σ_{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 σ 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 ($\sigma' : \mathsf{Net} \to \mathcal{A}$).

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 8.2: Taxonomy of CompCert extensions in terms of the corresponding game models (§7) and the requirements they satisfy (§8.3.2). 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 parameterized by a generic notion of *language interface* $A \in \mathbb{L} \supseteq \{\mathcal{C}, \mathcal{A}\}$. See §8.2 for details.

I will show in the next chapter how to adapt the semantic model and correctness proofs of CompCert so that they can be used in this way. My model is not intended to reach the level of generality required to handle all aspects of the problem above; indeed it should remain tailored to CompCert's verification as much as possible. Instead, it suffices to provide a model which can be soundly embedded into more general ones, where CompCert components can be made interoperable with components of other kinds and high-level reasoning can be carried out.

Yet, 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.

8.3.2 Requirements

To handle use cases like the one I 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 8.2, each of these requirements is fulfilled by some existing CompCert extension, however none satisfies them all.

The next chapter introduces CompCertO, the first extension of CompCert to address all of these requirements simultaneously.

Chapter 9

CompCertO

This chapter presents the high-level design of CompCertO, the first extension of CompCert which satisfies all of the requirements I have identified in §8.3.2. The key to this achievement is the expressivity of the semantic model, based on game semantics and certified abstraction layers.

9.1 Overview

CompCertO generalizes the semantic model of CompCert to express interactions between compilation units, using *language interfaces* to describe the form of these interactions and *simulation conventions* to describe the correspondence 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.

9.1.1 Language interfaces

The games used in CompCertO have a particularly simple structure. I will call each one a *language interface*. Their moves are partitioned into questions and answers, where questions correspond to function invocations and answers return control to the caller.

Definition 9.1. A *language interface* is a tuple $A = \langle A^\circ, A^\bullet \rangle$, where A° is a set of *questions* and $A^\bullet$ is a set of *answers*.

A.c	int mult(int n, int p) {
	  return n * p;
	}
		A.s	mult:	%eax := %ebx
				%eax *= %ecx
				ret

B.c	int sqr(int n) {
	  return mult(n, n);
	}
		B.s	sqr:
			
			L1:
				%ecx := %ebx
				call mult
				ret

Figure 9.1: 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.

CompCertO uses 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:

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

and all their prefixes. 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 correspondence between questions and answers but are not part of the model.

Example 9.2. *I will use a simplified version of C and assembly to illustrate some of the principles behind this model. Consider the program components in Figure 9.1. The behavior of B.c as it interacts with A.c is described by plays of the form:*

$$\mathsf{sqr}(3) \cdot \underline{\mathsf{mult}(3,3)} \cdot 9 \cdot \underline{9} \tag{9.1}$$

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

The behavior of A.s and B.s, is described using a set of registers $R := \{\mathsf{pc}, \mathsf{eax}, \mathsf{ebx}, \mathsf{ecx}\}$ which include a program counter pc*, together with a stack of pending return addresses. 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:*

$$\left[\begin{array}{rl} \mathsf{pc} & \mapsto \mathsf{sqr} \\ \mathsf{eax} & \mapsto 42 \\ \mathsf{ebx} & \mapsto 3 \\ \mathsf{ecx} & \mapsto 7 \\ \textit{stack:} & x \cdot \vec{k} \end{array}\right] \cdot \underline{\left[\begin{array}{rl} \mathsf{pc} & \mapsto \mathsf{mult} \\ \mathsf{eax} & \mapsto 42 \\ \mathsf{ebx} & \mapsto 3 \\ \mathsf{ecx} & \mapsto 3 \\ \textit{stack:} & \mathsf{L1} \cdot x \cdot \vec{k} \end{array}\right]} \cdot \left[\begin{array}{rl} \mathsf{pc} & \mapsto \mathsf{L1} \\ \mathsf{eax} & \mapsto 9 \\ \mathsf{ebx} & \mapsto 3 \\ \mathsf{ecx} & \mapsto 3 \\ \textit{stack:} & x \cdot \vec{k} \end{array}\right] \cdot \underline{\left[\begin{array}{rl} \mathsf{pc} & \mapsto x \\ \mathsf{eax} & \mapsto 9 \\ \mathsf{ebx} & \mapsto 3 \\ \mathsf{ecx} & \mapsto 3 \\ \textit{stack:} & \vec{k} \end{array}\right]} \tag{9.2}$$

The correspondence between (9.1) and (9.2) is determined by the C calling convention in use. I discuss this point in more detail in §9.1.4.

As mentioned in §8.1.4, 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 game $\mathcal{E}$, correspond to system calls and accesses to volatile variables.

9.1.2 Semantic model

In order to model open components and cross-component interactions, I 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 §9.2.1.

9.1.3 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 correspondence, 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, the 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 devising 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 requirement #2, make proofs more complex, or cascade into subtle "impedance mismatch" problems requiring their own solutions (see §8.2).

By contrast, CompCertO captures the particularities of each simulation proof by introducing a notion of *simulation convention* expressing the correspondence between source- and target-level interactions. To describe simulation conventions and reason about them compositionally, I will use logical relations.

9.1.4 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 9.3. 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 $\mathbb{R}^\bullet \in \mathcal{R}_W(A_1^\bullet, A_2^\bullet)$. I will write $\mathbb{R} : A_1 \Leftrightarrow A_2$. In particular, for a language interface A, the *identity* simulation convention is defined as $\mathsf{id}_A := \langle \mathbb{1}, =, = \rangle : A \Leftrightarrow A$. The subscript A will usually be omitted.

A simulation between the transition systems $L_1 : A_1 \twoheadrightarrow B_1$ and $L_2 : A_2 \twoheadrightarrow B_2$ is then

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. 2.13)
$w \Vdash R$		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. 9.1)
$\mathbb{R} : A_1 \Leftrightarrow A_2$	alloc	Simulation convention (Def. 9.3)
$L : A \twoheadrightarrow B$	$\mathsf{Clight}(p)$	LTS for $A \twoheadrightarrow B$ (Def. 9.5)
$L_1 \oplus L_2$		Horizontal composition (Def. 9.6)
$L_1 \leq_{\mathbb{R} \twoheadrightarrow \mathbb{S}} L_2$	Thm. 9.16	Simulation property (Def. 9.7)

Table 9.1: Summary of notations

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 9.1 presents a summary of notations. Simulation conventions will often be derived from more elementary relations, following the internal structure of questions and answers (see §10.1).

Example 9.4. *The calling convention we used in Example 9.2 can be formalized as a simulation convention* $\tilde{\mathbb{C}} := \langle \mathsf{val}^*, \tilde{\mathbb{C}}^\circ, \tilde{\mathbb{C}}^\bullet \rangle : \tilde{\mathcal{C}} \Leftrightarrow \tilde{\mathcal{A}}$. *The set of worlds* val^* *is used to relate the stack in assembly questions to the stack in the corresponding answers. The relations* $\tilde{\mathbb{C}}^\circ, \tilde{\mathbb{C}}^\bullet$ *are defined by:*

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

For a C-level function invocation $f(\vec{v})$*, the register* pc *is expected 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 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 Figure 9.2.

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

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

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

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

Figure 9.2: Simulation identity and vertical composition

9.1.5 Simulation convention algebra

CompCert's *injection* passes (see §10.1) 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. In CompCertO, this situation can be expressed by using different simulation conventions for external and incoming calls ($\mathsf{injp} \to \mathsf{inj}$). 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. 9.8).

I rectify this imbalance *outside* of the simulation proof itself. Simulations for individual passes are not always horizontally compositional, but we can 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 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 ($\succeq$) 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.

9.2 Operational semantics

9.2.1 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.

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 9.2: Language interfaces used in CompCertO

Language interfaces Our models of cross-component interactions in CompCert languages are shown in Table 9.2. 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 language interfaces: 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 §9.1, 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 §8.1 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 §8.1 as follows.

Definition 9.5. 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 $A \twoheadrightarrow B$.

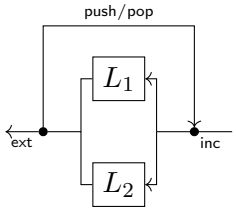
I will use infix notation for the various transition relations I, X, Y, F. In particular I write $n\ Y^s\ s'$ to denote that $n \in A^\bullet$ resumes the suspended external state s to continue with state s'. The interpretation of the transition systems described above follows the one given in §8.1, with interactions over the game A as a new source of observable actions. 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 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 Figure 9.3.

Definition 9.6 (Horizontal composition)**.** For two transition systems $L_1, L_2 : A \twoheadrightarrow A$ with $L_i = \langle S_i, \rightarrow_i, D_i, I_i, X_i, Y_i, F_i \rangle$, the *horizontal composition* of L_1 and L_2 is defined as:

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



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

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

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

Figure 9.3: 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$^\circ$). 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$^\circ$, ext$^\bullet$), and final states encountered at the top level (inc$^\bullet$), are simply passed along to the environment.

where the components $\rightarrow$, I, X, Y, F are defined by the rules shown in Figure 9.3.

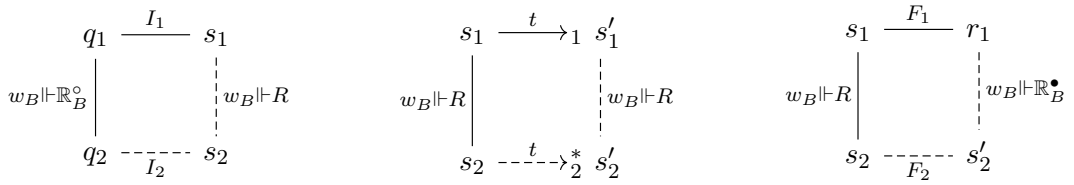
9.2.2 Open simulations

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 containment. I have updated forward and backward simulations to work with CompCertO's semantic model. In this section I present forward simulations, which are used as the primary notion of refinement.

Forward simulations As explained in §8.1.5, 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 I introduce, so I omit this aspect of forward simulations in the exposition below.

The transition systems of CompCertO introduce various forms of external communication,

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



$I_1 \ [\Vdash \mathbb{R}_B^\circ \to \mathcal{P}^{\leq}(R)] \ I_2$ $\quad \to_1 \ [\Vdash R \to = \to \mathcal{P}^{\leq}(R)] \ \to_2^*$ $\quad F_1 \ [\Vdash R \to \mathcal{P}^{\leq}(\mathbb{R}_B^\bullet)] \ F_2$

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

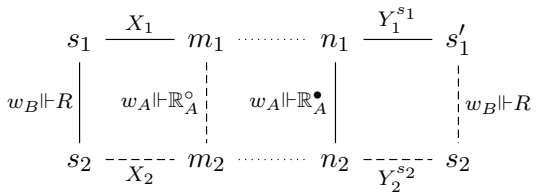
Figure 9.4: Forward simulation properties for initial, internal and final states. (a) When the execution is initiated by two related questions, any initial state of L_1 is matched by a related initial state of L_2. (b) Every internal transition of L_1 is then matched by a sequence of transitions of L_2, 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.

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 Figure 9.4, if questions of B_1 and B_2 respectively 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 correspondence between outgoing questions triggered by the transition systems' external states. The corresponding simulation properties are shown in Figure 9.5. 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 9.7 (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



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

Figure 9.5: 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.

of a relation $R \in \mathcal{R}_{W_B}(S_1, S_2)$ satisfying the properties shown in Figure 9.4 and Figure 9.5. In addition, the domains of L_1 and L_2 must satisfy:

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

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

Horizontal composition The horizontal composition operator described by Def. 9.6 preserves simulations. Informally, 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 Figure 9.5.

Theorem 9.8 (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 property 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. 9.8 maintains a *stack* of worlds to relate the corresponding stack of activations in the source and target composite semantics. See also §10.1.7.

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 9.9. *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 9.10 (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 9.11. *Simulations compose vertically in the way depicted in Figure 9.2.*

Proof. Visually speaking, the diagrams shown in Figs. 9.4 and 9.5 can be pasted vertically. For details, see the Coq proofs `identity_forward_simulation` and `compose_forward_simulations` in the file `common/Smallstep.v`. □

9.3 Simulation convention algebra

Now that I have described the structures used in CompCertO to model and relate the execution of program components, I will explain how to derive a correctness statement for the whole compiler from the correctness properties of each pass.

9.3.1 Refinement of simulation conventions

As discussed in §9.1.5, the composite simulation conventions obtained when we vertically compose the passes of CompCertO are not immediately satisfactory. In the remainder of this section, I describe the algebraic infrastructure used to rewrite them into an acceptable form, centered around a notion of refinement for simulation conventions.

A refinement $\mathbb{R} \succeq \mathbb{S}$ 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 9.12 (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 $\mathbb{R} \succeq \mathbb{S}$ 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 $\mathbb{R} \succeq \mathbb{S}$ and $\mathbb{S} \succeq \mathbb{R}$.

Theorem 9.13. *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:*

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

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

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

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 $\mathsf{id} \twoheadrightarrow \mathsf{id}$ (see Table 9.3). The properties above ensure that these passes have no impact on the overall simulation convention of CompCertO.

9.3.2 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 to the treatment of injection passes ($\mathsf{injp} \twoheadrightarrow \mathsf{inj}$). The details of inj and injp are discussed in §10.1, 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 $\mathsf{injp}^* \cdot \mathsf{injp} \succeq \mathsf{injp}^*$ on the left-hand side, and the idempotency of inj on the right-hand side (Thm. 10.5) allow us to rewrite the above into a symmetric simulation convention.

Definition 9.14. 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 $i \in I$. The simulation convention $\sum_{i \in I} \mathbb{R}_i := \langle W, R^\circ, R^\bullet \rangle$ is defined by:

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

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

Theorem 9.15 (Kleene iteration of simulations)**.** *The constructions $\succeq, \cdot, +, *$ work together as a typed Kleene algebra. Moreover, the following properties hold:*

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

Language/Pass	Outgoing ↠ Incoming	SLOC		See also
Clight	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	+17	(+3%)	
Thm. 10.5	$(\mathsf{ext} + \mathsf{injp})^* \twoheadrightarrow (\mathsf{ext} + \mathsf{injp})^*$			§10.1.8
SimplLocals	$\mathsf{injp} \twoheadrightarrow \mathsf{inj}$	-4	(-1%)	§10.1.6
Cshmgen	$\mathsf{id} \twoheadrightarrow \mathsf{id}$	+0	(+0%)	§9.3.1
Csharpminor	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	+15	(+4%)	
Cminorgen	$\mathsf{injp} \twoheadrightarrow \mathsf{inj}$	-21	(-2%)	§10.1.6
Cminor	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	+27	(+3%)	
Selection	$\mathsf{wt} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{wt} \cdot \mathsf{ext}$	+43	(+0%)	§10.2.3
CminorSel	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	+15	(+3%)	
RTLgen	$\mathsf{ext} \twoheadrightarrow \mathsf{ext}$	+8	(+0%)	§10.1.5
RTL	$\mathcal{C} \twoheadrightarrow \mathcal{C}$	+11	(+2%)	
Tailcall†	$\mathsf{ext} \twoheadrightarrow \mathsf{ext}$	-1	(-1%)	§10.1.5
Inlining	$\mathsf{injp} \twoheadrightarrow \mathsf{inj}$	+58	(+3%)	§10.1.6
Renumber	$\mathsf{id} \twoheadrightarrow \mathsf{id}$	-14	(-7%)	§9.3.1
Thm. 10.5	$\mathsf{inj} \twoheadrightarrow \mathsf{inj}$			§10.1.8
Constprop†	$\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$	-17	(-2%)	§10.2.4
CSE†	$\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$	+3	(+0%)	§10.2.4
Deadcode†	$\mathsf{va} \cdot \mathsf{ext} \twoheadrightarrow \mathsf{va} \cdot \mathsf{ext}$	-7	(-1%)	§10.2.4
Allocation	$\mathsf{wt} \cdot \mathsf{alloc} \twoheadrightarrow \mathsf{wt} \cdot \mathsf{alloc}$	+13	(+0%)	§10.3.1
LTL	$\mathcal{L} \twoheadrightarrow \mathcal{L}$	+15	(+6%)	
Tunneling	$\mathsf{ext} \twoheadrightarrow \mathsf{ext}$	+2	(+0%)	§10.1.5
Linearize	$\mathsf{id} \twoheadrightarrow \mathsf{id}$	-15	(-3%)	§9.3.1
Linear	$\mathcal{L} \twoheadrightarrow \mathcal{L}$	+16	(+7%)	
CleanupLabels	$\mathsf{id} \twoheadrightarrow \mathsf{id}$	-10	(-4%)	§9.3.1
Debugvar	$\mathsf{id} \twoheadrightarrow \mathsf{id}$	-12	(-3%)	§9.3.1
Stacking	$\mathsf{stacking} \twoheadrightarrow \mathsf{stacking}$	+291	(+11%)	§10.3.2
Mach	$\mathcal{M} \twoheadrightarrow \mathcal{M}$	+100	(+26%)	
Asmgen	$\mathsf{asmgen} \twoheadrightarrow \mathsf{asmgen}$	+179	(+6%)	§10.3.3
Asm	$\mathcal{A} \twoheadrightarrow \mathcal{A}$	+53	(+5%)	
	Total:	+765	(+2%)	

Table 9.3: Languages and passes of CompCertO. Passes are grouped by their source language. † indicates an optional optimization pass. The simulation conventions ext, inj, injp are explained in §10.1. The invariants wt, va are explained in §10.2. The conventions alloc, stacking, asmgen are explained in §10.3. To handle the asymmetry of injection passes (injp ↠ inj), self-simulation properties are inserted as pseudo-passes (Thm. 10.5). Significant lines of code (SLOC) measured by coqwc, compared to CompCert v3.6.

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

9.3.3 Compiler correctness

I will now present the central result. The passes of CompCertO are shown in Table 9.3. The techniques outlined above make it possible to formulate a simulation convention $\mathbb{C} : \mathcal{C} \Leftrightarrow \mathcal{A}$ for the whole compiler, and to establish the following correctness property.

Theorem 9.16 (Compositional Correctness of CompCertO)**.** *For a* Clight *program* p *and an* Asm *program* p' *such that* $\mathsf{CompCert}(p) = p'$*, the following simulation holds:*

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

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. 9.11 to compose the correctness proofs of the compiler passes and self-simulations shown in Table 9.3. 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`.

□

9.3.4 Compositional compilation and verification

Consider the translation units `M1.c`, . . . , `Mn.c` compiled and linked to `M1.s` + . . . + `Mn.s` = `M.s`. We can use Thms. 9.8, 9.9 and 9.16 to establish the following separate compilation property:

$$\mathsf{Clight}(\texttt{M1.c}) \oplus \cdots \oplus \mathsf{Clight}(\texttt{Mn.c}) \leq_{\mathbb{C} \twoheadrightarrow \mathbb{C}} \mathsf{Asm}(\texttt{M.s}) \tag{9.3}$$

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 Σ, also expressed as a transition system for $\mathcal{C} \twoheadrightarrow \mathcal{C}$. We must first decompose Σ 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. 9.8 and Eqn. 9.3 to establish the correctness property $\Sigma \leq_{\mathbb{C}\to\mathbb{C}} \mathsf{Asm}(\texttt{M.s})$.

Note that Eqn. 9.3 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. 9.16. 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. 9.16 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.

9.4 Evaluation

To give a sense of the overall complexity of CompCertO, I list in Table 9.4 the increase in significant lines of code it introduces compared to CompCert v3.6. As shown in Table 9.3, the 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.

The use of the simulation conventions inj, alloc, stacking and asmgen in particular under-

Component	SLOC	
Semantic framework (§9.2)	+782	(+14%)
Horizontal composition (§9.2.1)	676	
Simulation convention algebra (§9.3)	1,052	
CKLR theory and instances (§10.1)	1,807	
Clight and RTL parametricity (§10.1.5)	2,741	
Invariant preservation proofs (§10.2)	+549	(+7%)
Pass correctness proofs (Tbl. 9.3)	+765	(+2%)
Total	**8372**	

Table 9.4: Significant lines of code in CompCertO relative to CompCert v3.6. See Table 9.3 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.

scores the benefits of the approach. The corresponding passes are the root of much complexity in Compositional CompCert, CompCertX and CompCertM. For instance, to express the requirement on the areas protected by inj*p*, 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 the 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 10

Passes of CompCertO

Having described the overall design of CompCertO in Chapter 9, I turn to the details of the simulation conventions and techniques I used to update the correctness proofs of compilation passes:

- Compositional relational reasoning within CompCertO is explained in §10.1: *CompCert Kripke logical relations* unify CompCert's memory transformations as structure-preserving relations over the memory model. They can be used to define simulation conventions for most of the compiler's passes, and to derive parametricity theorems which capture important properties of CompCert languages.
- Several passes of CompCert use typing and soundness invariants. §10.2 explains how these invariants fit into the simulation framework of CompCertO and how the techniques used to verify these passes can be reified into a notion of *simulation modulo invariants*.
- Finally, §10.3 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. In the case of more complex passes, simulation conventions can be defined which capture the internal invariants used by existing proofs, avoiding many sources of complexity found in previous work.

10.1 Logical relations for the CompCert memory model

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.

10.1.1 Memory extensions

For passes where strict equality is too restrictive, but where 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 §8.1.8, I will 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.

10.1.2 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:

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

I 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.

10.1.3 World transitions

The definition of simulation relations uses Kripke worlds to ensure that questions and answers are related consistently. To formulate CompCert Kripke logical relations, I will also use the notion of Kripke frame and the relator $\Diamond$ defined in §2.3.4. The following example illustrates their use in the context of memory injections.

Example 10.1 (Injection simulation diagrams)**.** *Building on Ex. 2.12, 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:*

$$\begin{array}{ccc} s_1 & \xrightarrow{\alpha} & s_1' \\ {\scriptstyle f \Vdash R}\big| & & \vdots\,{\scriptstyle f' \Vdash R} \quad (f \subseteq f') \\ s_2 & \xrightarrow{\beta} & s_2' \end{array} \qquad \begin{array}{l} \forall f\, s_1\, s_2\, s_1' \,.\, f \Vdash s_1\ R\ s_2 \wedge \alpha(s_1) \ni s_1' \Rightarrow \\ \qquad \exists f'\, s_2' \,.\, f \subseteq f' \wedge \beta(s_2) \ni s_2' \wedge f' \Vdash s_1'\ R\ s_2' \end{array} \tag{10.1}$$

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 (10.1) as:

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

10.1.4 CompCert Kripke logical relations

The idea that extensions and injections constitute logical relations for the CompCert memory model can be formalized in the following way.

Definition 10.2 (CompCert Kripke Logical Relation)**.** For a tuple $R = (W, \rightsquigarrow, f, R^{\mathsf{mem}})$, where

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

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

Figure 10.1: Defining properties of CKLRs. Note the correspondence with the types of operations in Figure 8.2.

$\langle W, \rightsquigarrow \rangle$ 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:

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

I will say that R is a *CompCert Kripke logical relation* when the properties shown in Figure 10.1 are satisfied.

Rationale The relation R^{mem} is given as a component of R. We expect R^{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^{ptr} and undef to act as a least element for R^{val}, which determines

them completely.

Note that R^{mem} is the central component driving world transitions, as witnessed by the uses of $\Diamond$ in Figure 10.1. 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 10.3. *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 correspondence between $R^{\mathsf{val}}_{\mathsf{inj}}$ and $\hookrightarrow_{\mathsf{v}}$ is easily verified, as is the correspondence between $* \Vdash R^{\mathsf{val}}_{\mathsf{ext}}$ and $\leq_{\mathsf{v}}$. The properties of Figure 10.1 reduce to well-known properties of the memory model already proven in CompCert. See `cklr/Extends.v` and `cklr/Inject.v` for details. □

10.1.5 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:

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

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:

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

I 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 10.4 (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.

10.1.6 External calls in injection passes

Passes which alter the block structure of the memory use *memory injections* (§10.1.2). 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 correspondence 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.

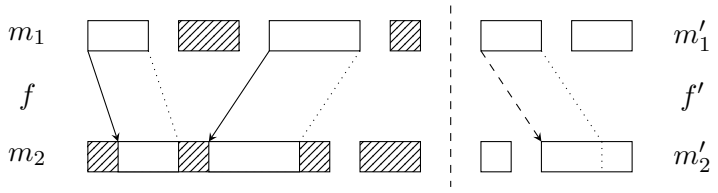
More generally, as depicted in Figure 10.2, injection passes expect external calls to leave regions outside of the injection's footprint untouched. This expectation is reasonable because external calls should behave uniformly between the source and target executions. These requirements can be formalized 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 Figure 10.2 (for details, see `cc_injp` in `common/LanguageInterface.v`).

10.1.7 Discussion: world transitions and compositionality

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



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

Figure 10.2: 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.

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

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

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 §10.1.6, 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:

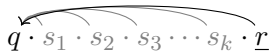
$$\overset{\curvearrowleft}{q \cdot \underline{m_1} \cdot n_1 \cdots \underline{m_k} \cdot n_k \cdot \underline{r}}$$

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 §9.2.2, to handle nested cross-component calls, composite simulations use an internal stack of worlds. A situation where m_1 and m_2 are nested cross-component calls and m_3 is an external call can be described as:

$$\overset{\curvearrowleft}{q \cdot m_1 \cdot m_2 \cdot \underline{m_3} \cdot n_3 \cdot n_2 \cdot n_1 \cdot \underline{r}}$$

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



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 $\mathsf{inj} \twoheadrightarrow \mathsf{inj}$. 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 respect 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.

10.1.8 Properties

Finally, I state some properties which are used to derive Thm. 9.16.

Theorem 10.5. *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. 9.15 and 10.4; see `driver/Compiler.v`. For the second statement, see `ext_inj`, `inj_ext`, `inj_inj` found under `cklr/`. □

10.2 Invariants

Several passes of CompCert rely on the preservation of invariants by their source program: 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 makes it possible to decompose invariant preservation from 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, I give an overview of the treatment of invariants. For details, see `common/Invariant.v` in the Coq development.

10.2.1 Invariants and language interfaces

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

Definition 10.6. 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.

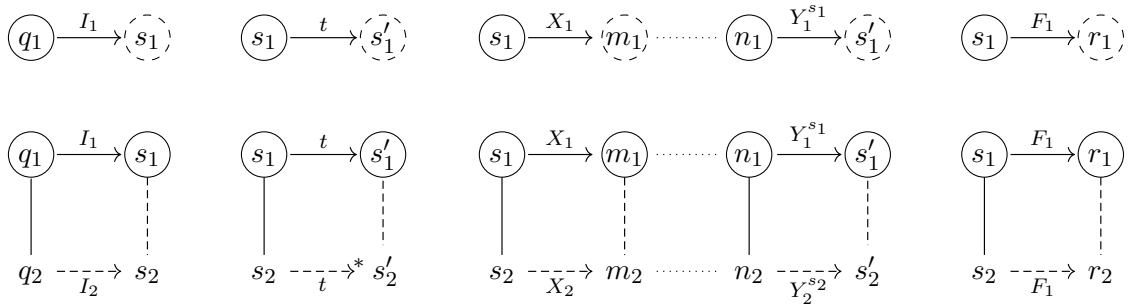


Figure 10.3: Simulation with invariants. Circles indicate questions, answers and states which satisfy the appropriate invariants. When the transition system L_1 preserves the invariants in the way shown in the top row, a simulation of L_1 by L_2 can be established through the weakened diagrams shown 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.

Example 10.7. *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 10.8 (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: $\hat{\mathbb{P}} := \langle W, \hat{\mathbb{P}}^\circ, \hat{\mathbb{P}}^\bullet \rangle$.

10.2.2 Simulations modulo invariants

The top row in Figure 10.3 illustrates the preservation of invariants by transition systems. In the context of a transition system $L_1 : A_1 \to B_1$, we must consider three invariants working together:

- an invariant $\mathbb{P}_A$ for the language interface A_1;
- an invariant $\mathbb{P}_B$ for the language interface B_1;
- 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 can 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 of $L_1^{\mathbb{P}}$ 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 Figure 10.3. However, since they are formulated in terms of Definition 9.7, the standard forward simulation techniques defined by CompCert in `Smallstep.v` remain available.

10.2.3 Typing invariants

The typing invariant described in Example 10.7 is used by the Selection and Allocation passes. I 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.

10.2.4 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.

I 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, self-simulations of the RTL language are used to match this convention nonetheless.

10.3 Specialized simulation conventions

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

10.3.1 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, I 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:

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

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

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

10.3.2 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 §10.1.7).

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 CompCertO, the required separation condition can be encoded in the simulation convention $\mathsf{stacking} : \mathcal{L} \Leftrightarrow \mathcal{M}$. 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

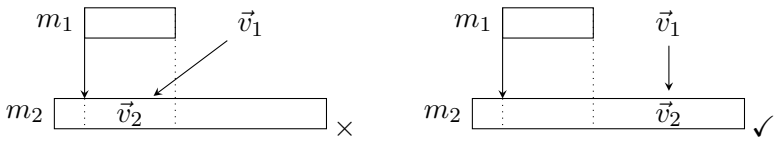


Figure 10.4: Separation of arguments in the stacking simulation convention

image of the source memory (Figure 10.4). For details, see `cc_stacking` in `backend/Mach.v`.

10.3.3 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 $\mathsf{asmgen} : \mathcal{M} \Leftrightarrow \mathcal{A}$ 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.

This is expressed as:

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

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

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

Part IV

Conclusion

Chapter 11

Conclusion and future work

The goal of this thesis is to demonstrate that the concepts and formalisms which underpin game semantics, the refinement calculus, algebraic effects and other areas of programming language theory can be combined in a fruitful manner. Their synthesis, which I call *refinement-based game semantics*, constitutes a plausible approach to the end-to-end verification of large-scale, heterogeneous computer systems.

Taking first steps in this direction, I have shown how a simple theory of certified abstraction layers can be embedded into increasingly expressive models, and how the conceptual framework of refinement-based game semantics can be used to incorporate CompCert into this hierarchy with minimal effort. A preliminary formalization in the Coq proof assistant of the game models presented in Part II demonstrates the advantages of the techniques I have used. In particular, decoupling angelic and demonic nondeterminism from the structure of plays makes the definitions simpler and facilitates proofs. The work on CompCertO presented in Part III has been fully formalized in the Coq proof assistant.

In addition, the general approach I have outlined suggests several directions for future research.

11.1 CertiKOS

A combination of refinement-based game semantics and CompCertO should serve as a new basis for the verification of CertiKOS. This will allow us to extend the capabilities of certified abstraction

layers in various directions.

First, generalizing layer specifications from the type $1 \to F$ to $E \to F$ will allow us to model *upcalls*: calls from the code of a lower-level abstraction layer into a higher-level abstraction layer. This is a very common pattern in system code. For example, in a typical network stack, incoming packet are first processed by lower-level layers, which in turn invoke callback functions in higher-level layers to handle the more abstract aspects of network communication. However, the limitations of the verification techniques used in the current version of CertiKOS have so far prevented us from adopting this technique.

The new framework should also offer a better separation of concerns. As explained in §4.4, our existing model is intimately tied with the semantic model of CompCert. Conversely, the CompCertX extension used for CertiKOS must maintain the abstract state used in certified abstraction layers as part of its memory model. Using refinement-based game semantics, it is possible to formulate layer interfaces which will only deal with abstract state, formulate CompCert semantics which will only deal with memory states, and to interface them through their interactions. The powerful abstraction mechanisms introduced by dual nondeterminism also allows us a great deal of flexibility to translate these interactions themselves between representations at various levels of abstraction.

By providing a more uniform algebraic framework, refinement-based game semantics could also facilitate the formulation of complex semantic transformations. For example, an important primitive of operating system kernels is *context switching*, which saves the current register state, loads the saved state of a different process, then returns from that process' invocation of cswitch. This primitive must be written in assembly and can be described in terms of the language interface $\mathcal{A}$, but admits no $\mathcal{C}$-style specification. However, it may be possible to account for calls to cswitch at the $\mathcal{C}$ level by giving a *transformation* of C semantics keeping track of the execution of multiple processes and interpreting $\mathcal{C}$ calls to cswitch in that context.

11.2 Richer game models

In addition, there exists a wealth of game semantics research we can draw from to build more general models for refinement-based game semantics. For example, modeling concurrency is an

important next step. Because of their order-theoretic formulation, *concurrent* and *asynchronous* games [Abramsky and Melliès, 1999; Mellies, 2004] in particular seem amenable to dually nondeterministic extensions along the lines of Chapter 5. The more recently proposed *template* games [Melliès, 2019] are similar in some aspects to the approach proposed in Chapter 7 and could constitute the basis of a more principled and flexible approach to concurrency and state.

11.3 CompCertO and compositional compiler correctness

The main goal of CompCertO is to provide a version of the semantic model and correctness theorem of CompCert which can be readily embedded into refinement-based game semantics, while providing a granular view of program components. By contrast, existing work on compositional compiler correctness usually focuses on the proof principles enabled by a given compiler correctness theorem rather than the expressivity of the semantics as an end in itself.

Consequently, while §9.3.4 outlines some possible uses of CompCertO as a compositional compiler, its capabilities in that context remain limited. To an extent, this is a conscious design choice. In the context of CertiKOS, CompCertX has served us well despite (and perhaps owing to) its simplicity. This is in part because certified abstraction layers "step in" to compensate for its limitations, enabling for example the encapsulation of state. Although CompCertO aims to solve a much more general problem compared to CompCertX, its design follows a similarly minimalistic bent. Nevertheless, I believe the semantics and correctness theorem formulated in CompCertO can be used to derive much more powerful reasoning principles, perhaps by adapting existing work.

For example, a distinguishing feature of CompCertM is the cohabitation of open and closed semantics. In CompCertO, we could adapt this idea by providing *loaders* for various language interfaces. Given a transition system $L : \mathcal{X} \twoheadrightarrow \mathcal{X}$, we can derive the corresponding closed semantics $\mathsf{load}_{\mathcal{X}} : 1 \twoheadrightarrow \mathcal{W}$ by constructing an initial memory state and modeling the conventional invocation of `main` in the context of the language interface $\mathcal{X}$. This corresponds roughly to a component $\iota_{\mathcal{X}} : \mathcal{X} \to \mathcal{W}$. On the other hand, external calls could be interpreted using CompCert's original parameter $\chi : 1 \to \mathcal{C}$, as long as the loader translates between the language interfaces $\mathcal{C}$ and $\mathcal{X}$. This would allow us to recover the original closed semantics defined in CompCert. The constructions ι and χ would be compatible with all components of the simulation convention $\mathbb{C}$ used in

CompCertO, in the sense that for $R \in \{\mathsf{ext}, \mathsf{inj}, \mathsf{injp}, \mathsf{wt}, \mathsf{va}\}$, the following properties would hold:

$$L_1 \leq_R L_2 \Rightarrow \mathsf{load}_{\mathcal{C}}(L_1) \leq_{\mathsf{id}} \mathsf{load}_{\mathcal{C}}(L_2)$$

$$L_1 \leq_{\mathsf{alloc}} L_2 \Rightarrow \mathsf{load}_{\mathcal{C}}(L_1) \leq_{\mathsf{id}} \mathsf{load}_{\mathcal{L}}(L_2)$$

$$L_1 \leq_{\mathsf{stacking}} L_2 \Rightarrow \mathsf{load}_{\mathcal{L}}(L_1) \leq_{\mathsf{id}} \mathsf{load}_{\mathcal{M}}(L_2)$$

$$L_1 \leq_{\mathsf{asmgen}} L_2 \Rightarrow \mathsf{load}_{\mathcal{M}}(L_1) \leq_{\mathsf{id}} \mathsf{load}_{\mathcal{A}}(L_2)$$

This would allow us to derive a correctness theorem for the closed semantics as well.

More generally, to facilitate horizontal composition and compositional verification, it may be advantageous to reformulate $\mathbb{C}$ to take a simpler form. This should be possible without modification to the existing correctness proof, by leveraging the simulation convention algebra and the properties of the simulation conventions involved to give a simpler but equivalent overall simulation convention $\mathbb{C}' \equiv \mathbb{C}$.

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, Kohei Honda, and Guy Mccusker. 1998. A Fully Abstract Game Semantics for General References. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*. Society Press, 334–344.

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

Samson Abramsky and Paul-André Melliès. 1999. Concurrent Games and Full Completeness. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS '99)*. IEEE Computer Society, USA, 431–442. https://doi.org/10.1109/LICS.1999.782638

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

Steve Awodey. 2010. *Category theory*. Oxford university press.

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 Helsinki, 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

Bernhard Banaschewski and Evelyn Nelson. 1976. Tensor products and bimorphisms. *Canad. Math. Bull.* 19, 4 (1976), 385–402. https://doi.org/10.4153/CMB-1976-060-2

Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal Logic.* Cambridge University Press. https://doi.org/10.1017/CBO9781107050884

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

Andreas Blass. 1993. Is game semantics necessary?. In *International Workshop on Computer Science Logic.* Springer, 66–77.

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

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547.

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

M. Erné, J. Koslowski, A. Melton, and G. E. Strecker. 1993. A Primer on Galois Connections. *Ann. N. Y. Acad. Sci.* 704, 1 (1993), 103–125. https://doi.org/10.1111/j.1749-6632.1993.tb52513.x

Dan R Ghica and Andrzej S Murawski. 2004. Angelic semantics of fine-grained concurrency. In *International Conference on Foundations of Software Science and Computation Structures.* Springer, 211–225.

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

Masahito Hasegawa. 1995. Decomposing typed lambda calculus into a couple of categorical programming languages. In *Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS '95)*. Springer, Berlin, Heidelberg, 200–219. https://doi.org/10.1007/3-540-60164-3_28

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.

André Joyal and Myles Tierney. 1984. *An extension of the Galois theory of Grothendieck.* Number 309 in Memoirs of the American Mathematical Society. American Mathematical Society.

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

James Laird. 1997. Full abstraction for functional languages with control. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, USA, 58–67. https://doi.org/10.1109/LICS.1997.614931

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.

Saunders Mac Lane. 1978. *Categories for the working mathematician*. Springer Science & Business Media. https://doi.org/10.1007/978-1-4757-4721-8

William Mansky, Wolf Honoré, and Andrew W Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *Proceedings of the 29th European Symposium on Programming (ESOP '20)*. Springer, Cham, 428–455.

Paul-André Mellies. 2004. Asynchronous games 2: the true concurrency of innocence. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004)*. Springer, Berlin, Heidelberg, 448–465.

Paul-André Melliès. 2019. Categorical combinatorics of scheduling and synchronization in game semantics. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

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

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

Benjamin C Pierce. 1991. *Basic category theory for computer scientists*. MIT press.

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

Bas Spitters and Eelis van der Weegen. 2011. Type Classes for Mathematics in Type Theory. arXiv:1102.1323 [cs.LO]

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

Věra Trnková, Jiří Adámek, Václav Koubek, and Jan Reiterman. 1975. Free algebras, input processes and free monads. *Commentationes Mathematicae Universitatis Carolinae* 16, 2 (1975), 339–351.

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