Refinement-Based Game Semantics for Certified Abstraction Layers

Jérémie Koenig
Yale University
jeremie.koenig@yale.edu

Zhong Shao
Yale University
zhong.shao@yale.edu

Abstract

Formal methods have advanced to the point where the functional correctness of various large system components has been mechanically verified. However, the diversity of semantic models used across projects makes it difficult to connect these component to build larger certified systems. Given this, we seek to embed these models and proofs into a general-purpose framework where they could interact. We believe that a synthesis of game semantics, the refinement calculus, and algebraic effects can provide such a framework.

To combine game semantics and refinement, we replace the downset completion typically used to construct strategies from posets of plays. Using the *free completely distributive completion*, we construct *strategy specifications* equipped with arbitrary angelic and demonic choices and ordered by a generalization of alternating refinement. This provides a novel approach to nondeterminism in game semantics.

Connecting algebraic effects and game semantics, we interpret effect signatures as games and define two categories of effect signatures and strategy specifications. The resulting models are sufficient to represent the behaviors of a variety of low-level components, including the *certified abstraction layers* used to verify the operating system kernel CertiKOS.

CCS Concepts: • **Theory of computation → Logic and verification**; **Program specifications**; **Program verification**; • **Software and its engineering → Formal software verification**; **Functionality**; **Abstraction, modeling and modularity**.

Keywords: certified abstraction layers; dual nondeterminism; game semantics; strategy specification; program refinement; interaction specification; algebraic effects

ACM Reference Format:
Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20), July 8–11, 2020, Saarbrücken, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373718.3394799

1 Introduction

Certified software [43] is software accompanied by mechanized, machine-checkable proofs of correctness. To construct a certified program, we must not only write its code in a given programming language, but also formally specify its intended behavior and construct, using specialized tools, evidence that the program indeed conforms to the specification.

1.1 Certified systems at scale

The past decade has seen an explosion in the scope and scale of practical software verification. Researchers have been able to produce certified compilers [30], program logics [9], operating system kernels [22, 28], file systems [16] and more, often introducing new techniques and mathematical models. In this context, there has been increasing interest in making these components interoperable and combining them—and their proofs of correctness—into larger certified systems.

This is exemplified by the DeepSpec project [10], which seeks to connect various components specified and verified in the Coq proof assistant. The key idea behind DeepSpec is to interpret 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.

To an extent, these principles are already demonstrated in the structure of the certified C compiler CompCert [30], where the semantics of intermediate languages serve as intermediate specifications for each compilation pass. The correctness of each pass is established by proving that the behavior of its target program refines that of its source program. As passes are composed to obtain the overall C-to-assembly compiler, the correctness proofs are composed as well to construct a correctness proof for the whole compiler. The final theorem does not mention the intermediate programs or language semantics, so that a user only needs to trust the accuracy of the C and assembly semantics, and the soundness of the proof assistant.

Building on this precedent, the CertiKOS verification effort [22–24] divided the kernel into several dozen abstraction layers which were then specified and verified individually. Layer specifications provide an abstract view of a layer's functionality, hiding the procedural details and low-level data representations involved in its implementation. Client code can be verified in terms of this abstract view in order to build higher-level layers. Certified layers with compatible interfaces can then be chained together in the way passes of a compiler can be composed when the target language of one corresponds to the source language of the other.

1.2 Semantic models for verification

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

Worse yet, this diversity is not simply a historical accident. The semantic models used in the context of individual verification projects are often carefully chosen to make the verification task tractable. The semantic model used in CompCert alone has changed multiple times, addressing new requirements and techniques that were introduced alongside new compiler features and optimizations [31]. Given the difficulty of verification, preserving this flexibility is essential.

Then, to make it possible to link components verified using a variety of paradigms, we need to identify a model expressive enough to embed the semantics, specifications and correctness proofs of a variety of paradigms.

1.3 General models for system behaviors

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

The framework of symmetric monoidal categories, which allows components to be connected in series ($\circ$) and in parallel ($\otimes$), captures structures found in various kinds of systems and processes [13], and appears in different forms in many approaches to logic and programming language semantics.

A particularly expressive instance of this phenomenon is realized in *game semantics* [1], an approach to compositional semantics which uses two-player games to model the interaction between a component and its environment, and represents the externally observable behavior of the component as a strategy in this game. The generality of games as descriptions of the possible interactions of components makes this approach broadly applicable, and the typed aspect of the resulting models makes it ideal to the task of describing the behavior of heterogeneous systems.

However, the generality of game models often translates to a fair amount of complexity, which imposes a high barrier to entry for practitioners and makes them difficult to formalize in a proof assistant. While more restricted, the framework of *algebraic effects* [40] is sufficient for many modeling tasks, fits within the well-known monadic approach to effectful and interactive computations, and can be adapted into a particularly simple version of game semantics. Along these lines, *interaction trees* [46] have been developed for use in and across several DeepSpec projects.

Finally, while game models have been proposed for a wide variety of programming languages, there has been comparatively less focus on specifications and correctness properties. By contrast, the general approach of *stepwise refinement* suggests a uniform treatment of programs, specifications and their relationships. It has been studied extensively in the context of predicate transformer semantics [20] and in the framework known as the *refinement calculus* [12].

1.4 Contributions

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

- We adapt the work of Morris and Tyrrell [35, 36], which extends the refinement calculus to the level of terms by using *free completely distributive completions* of posets, to investigate *dual nondeterminism* in the context of game semantics and construct completely distributive lattices of *strategy specifications*, partially ordered under a form of alternating refinement [8].
- In §3, we define a version of the *free monad on an effect signature* which incorporates dual nondeterminism and refinement. The result can be used to formulate a theory of certified abstraction layers in which layer interfaces, layer implementations, and simulation relations are treated uniformly and compositionally.
- In §4, we outline a more general category of games and strategy specifications; its object are effect signatures regarded as games and its morphisms specify well-bracketed strategies. The behavior of certified abstraction layers can then be represented canonically, and reentrant layer interfaces can be modeled.

The model presented in §3 was designed to be simple but general enough to embed CompCert semantics, certified abstraction layers, and interaction trees. The main purpose for the model presented in §4 is then to hide state and characterize certified abstraction layers through their externally observable interactions only.

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

2 Background and approach

The work presented in this paper draws from a broad range of existing research. This section summarizes the relevant aspects of these various lines of work, and outlines how we combine them to develop refinement-based game semantics.

2.1 Dual nondeterminism and refinement

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

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

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

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

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

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

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

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

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

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

If the language is sufficiently expressive, then a correctness property $P\{-\}Q$ can itself be encoded [34] as a specification statement $\langle P, Q\rangle$ such that:

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

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

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

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

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

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

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

The refinement calculus. These basic ingredients have been studied systematically in the *refinement calculus*, dating back to Ralph-Johan Back's 1978 PhD thesis [11]. In its modern incarnation [12], the refinement calculus subsumes programs and specifications with *contracts* featuring unbounded angelic and demonic choices.

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

Dually nondeterministic functions. Morris and Tyrrell were able to extend the lattice-theoretic approach used in the refinement calculus to functional programming [35, 36, 45]: if types are interpreted as posets, dual nondeterminism can be added at the type level using *free completely distributive completions*. This allows dual nondeterminism to be used in a variety of new contexts.

Definition 2.1. A completely distributive lattice L is a free completely distributive completion of a poset C if there is a monotonic function $\phi : C \to L$ such that for any completely distributive lattice M and monotonic function $f : C \to M$, there exists a unique complete homomorphism $f^*_\phi : L \to M$ such that $f^*_\phi \circ \phi = f$:

$$\begin{tikzcd} C \arrow[r, "\phi"] \arrow[dr, "f"'] & L \arrow[d, dashed, "f^*_\phi"] \\ & M \end{tikzcd}$$

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

Morris [35] shows that the free completely distributive completion of $(A, \le)$ can be constructed as one of:

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

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

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

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

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

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

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

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

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

further allows angelic and demonic choices to commute, and the status of f^*_ϕ as a complete homomorphism enables the following properties:

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

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

2.2 Game semantics

Game semantics [2, 14, 27] is an approach to compositional semantics which interprets types as two-player games between a program component (the *system*) and the context in which it appears (the *environment*). Terms of a given type are then interpreted as strategies for the corresponding game, specifying for each position in the game the next action to be taken by the system.

For example, in a simple game semantics resembling that of Idealized Algol [3], sequences of actions corresponding to the execution of x := 2 ∗ x could have the form:

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

The moves of the system have been underlined. The environment initiates the execution with the move run. The system move rd_x requests the value of the variable x, communicated in response by the environment move n. The system move

$\mathsf{wr_x}(n)$ requests storing the value n into the variable x, and is acknowledged by the environment move ok. Finally, the system move done expresses termination.

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

Traditionally [4], 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:

$$S_G \subseteq \mathcal{D}(P_G, \sqsubseteq)$$

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

$${\downarrow} s := \{t \in P_G \mid t \sqsubseteq s\}$$

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

Angelic nondeterminism allows us to range over all possible choices of the environment and record the resulting plays. For instance, the strategy for x := 2 ∗ x would be:

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

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

$$\sigma \subseteq \bigcup_{n \in \mathbb{N}} \bigcup_{-1 \le \delta \le 1} {\downarrow}(\mathsf{run} \cdot \underline{\mathsf{rd_x}} \cdot n \cdot \underline{\mathsf{wr_x}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}})$$

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

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

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

We will remedy this by using a richer completion.

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

$$\sigma' := \bigsqcup_{n \in \mathbb{N}} \bigsqcap_{-1 \le \delta \le 1} \phi\big(\mathsf{run} \cdot \underline{\mathsf{rd_x}} \cdot n \cdot \underline{\mathsf{wr_x}[2n+\delta]} \cdot \mathsf{ok} \cdot \underline{\mathsf{done}}\big)$$

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

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

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

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

2.3 Algebraic effects

The framework of *algebraic effects* [40] models computations as terms in an algebra whose operations represent effects: a term $m(x_1, \dots x_n)$ represents a computation which first triggers an effect m, then continues as a computation derived from the subcomputations $x_1, \dots x_n$. For example, the term:

readbit(print["Hello"](done), print["World"](done))

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

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

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

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

which indicates that printing the string s followed by printing the string t is equivalent to printing the string st in one go. In this work, we use effect signatures to represent the possible external interactions of a computation, but we will not use equational theories. We will however make it possible to interpret effects into another signature, modeling a limited form of *effect handlers* [41].

Definition 2.2. An *effect signature* is a set E of operations together with a mapping ar, which assigns to each $e \in E$ a set $\mathsf{ar}(e)$ called the *arity* of e. We will use the notation $E = \{e_1 : \mathsf{ar}(e_1),\ e_2 : \mathsf{ar}(e_2),\ \ldots\}$ to describe effect signatures.

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

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

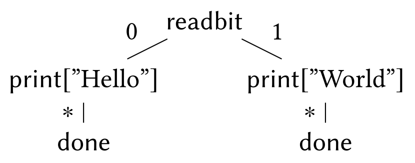
The most direct way to interpret an effect signature is the algebraic point of view, in which it induces a set of terms built out of the signature's operations.

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

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

where s_0 = "Hello" and s_1 = "World". Monads which offer this structure include the free monad on the signature E, which leaves effects entirely uninterpreted; roughly, its computations of type A correspond to terms over the signature $E \uplus \{v : \varnothing \mid v \in A\}$. The interaction specification monad $\mathcal{I}_E$ presented in §3 is a version of the free monad which combines the dual nondeterminism of **FCD** with uninterpreted effects taken in the signature E.

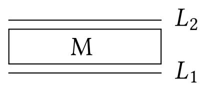
Finally, an effect signature can also be seen as a particularly simple *game*, in which the proponent chooses a question $m \in E$ and the opponent responds with an answer $n \in \mathsf{ar}(m)$. Then the terms induced by the signature are *strategies* for an iterated version of this game. Indeed, the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



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

2.4 Certified abstraction layers

To demonstrate the applicability of our results, we will use them to construct increasingly expressive theories of *certified abstraction layers*. As described in Gu et al. [22], a certified abstraction layer consists of a *layer implementation* together with two *layer interfaces*: the *underlay* provides specifications for the primitives available to the layer implementation; the *overlay* provides specifications for the procedures which the layer implements. A layer M implementing the overlay interface L_2 on top of the underlay interface L_1 can be depicted as follows:



A layer interface L has three components. First, a *signature* enumerates primitive operations together with their types, given as $\mathsf{op} : A \to B$ where A and B are sets. In terms of Def. 2.2, this corresponds to a family $\{\mathsf{op}[a] : B \mid a \in A\}$. Second, the set S contains the *abstract states* of the layer interface. Finally, for each primitive $\mathsf{op} : A \to B$, a *specification* is given as a function:

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

Throughout the paper, we will use $v@k \in V \times S$ to denote a pair containing the value $v \in V$ and the state $k \in S$. In the type of $L.\mathsf{op}$ above, $\mathcal{P}^1$ corresponds to the *maybe* monad:

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

where the empty set $\varnothing \in \mathcal{P}^1(A)$ serves a purpose similar to the one discussed for $\bot$ at the end of §2.1.

As a running example, we will use the certified layer depicted in Fig. 1, which implements a bounded queue with at most N elements using a circular buffer. The underlay interface L_{rb} contains an array $f \in V^N$ with N values of type V and two counters which will take 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.

The overlay L_{bq} features two primitives enq and deq which 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).

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, 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. We will write $M_{\mathsf{bq}}[L_{\mathsf{rb}}]$ to denote the result. Correctness is proved by showing for each operation $\mathsf{op} \in \{\mathsf{enq}(v), \mathsf{deq}(*) \mid v \in V\}$ the simulation:

$$L_{\mathsf{bq}}.\mathsf{op}\ \ [R \to \mathcal{P}^{\leq}(= \times R)]\ \ M_{\mathsf{bq}}[L_{\mathsf{rb}}].\mathsf{op}\,, \qquad (1)$$

$$\boxed{L_{\mathsf{bq}}} \qquad S_{\mathsf{bq}} := V^*$$

$$\begin{aligned}
&\mathsf{enq} : V \to \mathbb{1} &\qquad L_{\mathsf{bq}}.\mathsf{enq}(v)@\vec{q} &:= \{*@\vec{q}v \mid |\vec{q}| < N\} \\
&\mathsf{deq} : \mathbb{1} \to V &\qquad L_{\mathsf{bq}}.\mathsf{deq}(*)@\vec{q} &:= \{v@\vec{p} \mid \vec{q} = v\vec{p}\}
\end{aligned}$$

$$\boxed{M_{\mathsf{bq}}} \qquad R \subseteq S_{\mathsf{bq}} \times S_{\mathsf{rb}}$$

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

$$\boxed{L_{\mathsf{rb}}} \qquad S_{\mathsf{rb}} := V^N \times \mathbb{N} \times \mathbb{N}$$

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

Figure 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.

where the relators $\to, \times, \mathcal{P}^{\leq}$ are defined by:

$$\begin{array}{ccccl}
f & [R_1 \to R_2] & g & \Leftrightarrow & \forall xy \cdot x\ R_1\ y \Rightarrow f(x)\ R_2\ g(y) \\
x & [R_1 \times R_2] & y & \Leftrightarrow & \pi_1(x)\ R_1\ \pi_1(y) \wedge \pi_2(x)\ R_2\ \pi_2(y) \\
s & \mathcal{P}^{\leq}(R) & t & \Leftrightarrow & \forall x \in s \cdot \exists y \in t \cdot x\ R\ y\,.
\end{array}$$

We will write $L_{\mathsf{rb}} \vdash_R M_{\mathsf{bq}} : L_{\mathsf{bq}}$ to express that condition (1) holds for each operation op in the L_{bq} layer interface.

3 The interaction specification monad

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

3.1 Overview

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

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

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

3.2 Plays

We first introduce the partially ordered sets of plays which we use to construct the interaction specification monad. Since we intend to describe *active* computations, we use *odd*-length plays which start with *system* moves, by contrast with the more common approach presented in §2.2.

Definition 3.1. The set $\bar{P}_E(A)$ of *interactions* for an effect signature E and a set of values A is defined inductively:

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

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

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

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

3.3 Interaction specifications

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

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

Definition 3.2. The *interaction specification monad* for an effect signature E maps a set A to the free completely distributive completion of the corresponding poset of plays:

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

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

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

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

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

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

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

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

The most subtle aspect of Def. 3.2 is the case for $\mu^E_A(\underline{m}ns)$, which includes $\underline{m}$ as well as $\underline{m}n\mu^E_A(s)$. This is both to ensure that the effects of the first computation are preserved when the second computation is $\bot$, and to ensure the monotonicity of the underlying function used to define μ^E_A. Consider for example $\underline{m} \sqsubseteq \underline{m}n\underline{\bot} \in \bar{P}_E(\mathcal{I}_E(A))$. Since $\mu^E_A(\underline{\bot}) = \bot$ and the **FCD** extension of the function $s \mapsto \underline{m}ns$ preserves $\bot$, it is not the case that $\underline{m} \sqsubseteq \underline{m}n\mu^E_A(\underline{\bot})$.

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

3.4 Interaction primitives

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

Definition 3.3 (Interaction primitive)**.** For an effect signature E and an operation $m \in E$, the interaction specification $\mathbf{I}^m_E \in \mathcal{I}_E(\mathrm{ar}(m))$ is defined as:

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

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 $\mathbf{I}^m_E$.

To model effect handling for a signature F, we use a family of interaction specifications $(f^m)_{m \in F}$ to provide an interpretation $f^m \in \mathcal{I}_E(\mathrm{ar}(m))$ of each effect $m \in F$ in terms of another effect signature E. This allows us to transform an interaction specification $x \in \mathcal{I}_F(A)$ into an interaction specification $x[f] \in \mathcal{I}_E(A)$, defined as follows. The constructions $\bot$ and $\{P\}$ were discussed in §2.1; they carry similar meanings in the context of the interaction specification monad.

Definition 3.4 (Interaction substitution)**.** Given the effect signatures E, F and the set A, for an interaction specification $x \in \mathcal{I}_F(A)$ and a family $(f^m)_{m \in F}$ with $f^m \in \mathcal{I}_E(\mathrm{ar}(m))$, the *interaction substitution* $x[f] \in \mathcal{I}_E(A)$ is defined by:

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

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

3.5 Categorical structure

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

Definition 3.5 (Morphisms)**.** Consider the effect signatures E, F and G. We will write $f : E \to F$ whenever $(f^m)_{m \in F}$ is a family of interactive computations such that $f^m \in \mathcal{I}_E(\mathrm{ar}(m))$. For $f : E \to F$ and $g : F \to G$, we define $g \circ f : E \to G$ as:

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

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

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

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

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

Lemma 3.6. *Consider the effect signatures E, F, G, H and the morphisms $f : E \to F$, $g : F \to G$ and $h : G \to H$. The following properties hold:*

$$\mathbf{I}_F \circ f = f \circ \mathbf{I}_E = f$$
$$h \circ (g \circ f) = (h \circ g) \circ f$$

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

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

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

Definition 3.7. 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.

3.6 Products

Effect signatures can be combined in the following way.

Definition 3.8. We define the effect signature $1 := \varnothing$. For a family of effect signatures $(E_i)_{i \in I}$, we define:

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

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

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

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

Theorem 3.9. *The category $\mathcal{G}^{ib}_{\sqsubseteq}$ has all products. Objects are given by $\bigotimes_{i \in I} E_i$ and projection arrows are given for each $i \in I$ by the morphism:*

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

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

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

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

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

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

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

□

3.7 Certified abstraction layers

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

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

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

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

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

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

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

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

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

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 $[\![M]\!]@S \circ [\![L]\!] : 1 \to F@S$.

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

Recall the definition of the judgment $L_1 \vdash_R M : L_2$, which means that a layer implementation M correctly implements L_2 on top of L_1 through a simulation relation $R \subseteq S_2 \times S_1$. If we write $L_1' := [\![M]\!]@S_1 \circ [\![L_1]\!]$ for the layer interface obtained by interpreting M on top of L_1, then:

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

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

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

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

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 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 3.10. *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 [36]. □

4 Stateful and reentrant strategies

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

4.1 Overview

As discussed in §3.5, the morphisms of $\mathcal{G}_{\sqsubseteq}^{ib}(E, F)$ correspond to the well-bracketed strategies for the game $!E \multimap F$. As such, they can be promoted to well-bracketed strategies for the more general game $!E \multimap !F$, which allows the environment to ask multiple question of F in a row, and to ask nested questions whenever it is in control.

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

After outlining the construction of a new category $\mathcal{G}_{\sqsubseteq}^{b}$ of games and strategies (§4.2–§4.5), we define an embedding of $\mathcal{G}_{\sqsubseteq}^{ib}$ into $\mathcal{G}_{\sqsubseteq}^{b}$ (§4.6), and show how the states used by a strategy $\sigma : E@S \to F@S$ can be internalized and hidden from its interactions (§4.7).

4.2 Games

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

Definition 4.1. 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 $m \in \Gamma(u, v)$ are called the *questions* of u to v, and the elements $n \in \mathsf{ar}(m)$ are called *answers* to the question m.

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

$$[E, F] = \begin{tikzcd} \mathrm{P} \arrow[r, bend left, "E"] & \mathrm{O} \arrow[l, bend left, "F"] \end{tikzcd}$$

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 interaction sequences used in the definition of strategy composition:

$$[E, F, G] = \begin{tikzcd} \mathrm{O} & \mathrm{P}_1 \arrow[l, "E"'] & \mathrm{P}_2 \arrow[l, "F"'] & \mathrm{O} \arrow[l, "G"'] \end{tikzcd}$$

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

$$[E_1, F_1] \vee [E_2, F_2] = \begin{tikzcd} & \mathrm{P}_1 \arrow[dl, "E_1"'] & \\ \mathrm{O} & & \mathrm{O} \arrow[ul, "F_1"'] \arrow[dl, "F_2"] \\ & \mathrm{P}_2 \arrow[ul, "E_2"] & \end{tikzcd}$$

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

Definition 4.2 (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:

$$\begin{tikzcd}{[E_1, \ldots, E_n, F] :=} & \mathsf{O} & \mathsf{P}_1 \arrow[l, "E_1"'] & \cdots \arrow[l, "E_2"'] & \mathsf{P}_n \arrow[l, "E_n"'] & \mathsf{O} \arrow[l, "F"']\end{tikzcd}$$

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 $\iota_i(m) : \iota_i(u) \to \iota_i(v)$.

4.3 Plays and strategies

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

Definition 4.3. 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 $m_i \in \Gamma(p_{i-1}, p_i)$. We will write this path as $\kappa = m'_1 \cdots m'_n : \mathsf{O} \twoheadrightarrow p \in \Gamma$.

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

Definition 4.4 (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 $m \in \Gamma(u, v)$ and stack $\kappa : \mathsf{O} \twoheadrightarrow u$, there is an edge:

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

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

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

The *plays* over 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 a stack. We 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:

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

4.4 Operations on strategies

Definition 4.5. 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 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 4.6 (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 $m \in \Gamma(u, v)$ and $\kappa : \mathsf{O} \twoheadrightarrow u$, the image of the move $m : (u, \kappa) \to (v, \kappa m)$ is the path:

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

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

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

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

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

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

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

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

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

$$\psi^c_1 := [1, 1, 0] : [E, F, G] \to [E, F]$$
$$\psi^c_2 := [0, 1, 1] : [E, F, G] \to [F, G]$$

defined similarly.

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

$$\psi^c(s_1, s_2) := \bigsqcup_{s \in P[E,F,G]} \{\psi^c_1(s) \sqsubseteq s_1 \wedge \psi^c_2(s) \sqsubseteq s_2\}; \psi^c_X(s) .$$

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:

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

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

$$[E] = \begin{tikzcd} \text{O} \arrow[loop left, "E"] \end{tikzcd}$$

and the transformation:

$$\psi_X^{\mathrm{id}} := [2] : [E] \to [E, E]$$
$$\psi_X^{\mathrm{id}}(\text{O}) := \text{O} \qquad \psi_X^{\mathrm{id}}(m) := mm$$

Then id_E is defined as:

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

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

$$\psi_X^{\otimes}(\text{O}) = \text{O} \qquad \psi_X^{\otimes}(\text{P}_1) = \psi_X^{\otimes}(\text{P}_2) = \text{P}$$
$$\psi_X^{\otimes}(\iota_i(m)) = \iota_i(m)$$

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

$$\psi_i^{\otimes}(p) = \begin{cases} \text{P} & \text{if } p = \text{P}_i \\ \text{O} & \text{otherwise} \end{cases}$$

$$\psi_i^{\otimes}(\iota_j(m)) = \begin{cases} m & \text{if } i = j \\ \epsilon & \text{otherwise} \end{cases}$$

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:

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

4.5 Category

The category $\mathcal{G}_{\sqsubseteq}^{b}$ 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 $\mathcal{G}_{\sqsubseteq}^{ib}$ using the process outlined in the following section. Note however that unlike that of $\mathcal{G}_{\sqsubseteq}^{ib}$, the symmetric monoidal structure of $\mathcal{G}_{\sqsubseteq}^{ib}$ 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.

4.6 Embedding $\mathcal{G}_{\sqsubseteq}^{ib}$

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

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

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

which *prepends* an arbitrary number of copies of f to a play of P_Γ^κ. Starting with $\omega_0^\kappa(t) := t$, we construct a series of approximations:

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

The auxiliary construction:

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

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

$$\bar{\omega}_i^{\kappa q}(\underline{v}, t) = vt$$
$$\bar{\omega}_i^{\kappa q}(\underline{m}, t) = m\, \omega_i^{\kappa q m}(\epsilon)$$
$$\bar{\omega}_i^{\kappa q}(\underline{m} ns, t) = m\, \omega_i^{\kappa q m}(n\, \bar{\omega}_i^{\kappa q}(s, t))$$

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:

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

4.7 Hiding state

The functor $W : \mathcal{G}_{\sqsubseteq}^{ib} \to \mathcal{G}_{\sqsubseteq}^{b}$ can be used to embed the layer theory defined in §3.7 as-is. In addition, the *state* of layer interfaces can be propagated across consecutive calls and eliminated from the representation.

Definition 4.7. The *state-free observation* at $k_0 \in S$ of a partial play $s : (O, \kappa) \twoheadrightarrow (O, \kappa')$ over the signature $[E@S, F@S]$ is written s/k_0 and defined recursively as:

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

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

When the strategy 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.

5 Related work

Game semantics. Research on game semantics for programming languages can be traced back to the model of linear logic proposed by Blass [14]. The fully abstract models of PCF [2, 27] were an important milestone. Subsequent work extended the approach to account for a variety of language features including state [3], control [29] and concurrency [5, 21, 33]. Low-level applications of games have also been proposed, for instance interface theories and interface automata [6, 17–19], and games have been used in the context of modal logic to model properties of open systems [7, 8].

A model of *finite* nondeterminism was first proposed in Harmer and McCusker [25], with a different approximation ordering developed in Murawski [37]. For infinite nondeterminism however, this model exhibits typical problems. Consider a process which nondeterministically chooses $n \in \mathbb{N}$, then performs a given action n times in a row. Although this computation has no possible infinite execution (a^*), it cannot be distinguished through its set of finite prefixes from a computation which allows the action to be performed an infinite number of time ($a^* \cup a^\omega$). This issue is addressed in Tsukada and Ong [44] and Castellan et al. [15] by selectively retaining branching information, but sensitivity to branching makes the resulting refinement algebra less agreeable.

Dual nondeterminism. On the other hand, the problem finds a natural solution in the context of dual nondeterminism, where the choice of n is demonic but divergence is angelic in nature. The algebraic properties of distributive lattices and complete homomorphisms induce our model's insensitivity to branching, which does not usually impact the observable behavior of components [38, 39].

Beyond the work discussed in §1 [11, 12, 20, 35, 36], models featuring dual nondeterminism include binary multirelations [32, 42] and the trace semantics used in Alur et al. [8]. The distinguishing feature of the free completely distributive lattice is that it starts from a poset rather than a set, allowing us to use it in the context of game semantics. The model of the process calculus CSP proposed in Tyrrell et al. [45] comes closest to the models presented here, but to our knowledge its relevance to game semantics has not been fully appreciated.

Algebraic effects and free monads. Algebraic effects were introduced in Plotkin and Power [40]. An important development was the introduction of *effect handlers* [41]. Uses and variants of the free monad on a signature are numerous, but in particular interaction trees [46] share structures and goals with our interaction specification monad (§3).

The implementation of interaction trees in the Coq proof assistant is carefully designed to make them executable and allow their extraction as ML code. However, to make this possible, the definition and theory of interaction trees must handle silent moves, with various notions of simulation and bisimulation taking them into account in different ways. The implementation also has to rely on Coq's support for coinductive types, one of the less commonly well-understood features of Coq which requires special proof techniques.

By contrast, our models are not designed with extraction in mind, but equivalent strategies become equal under predicate extensionality axioms. This enables the use of simple and efficient rewriting techniques in proof assistants, which is important to maintain usability and performance in the practical applications we are envisioning.

Finally, some form of dual nondeterminism and refinement could be modeled in interaction trees by adding choice operators in the effect signature. Refinement would be defined as a new kind of simulation taking these choice operations into account. However, devising a notion of refinement insensitive to branching would be more challenging.

6 Conclusion

Game semantics has tremendous potential for the formal verification of large and complex systems. To deploy game semantics in this context, we have attempted to shift emphasis away from the precise characterization of specific language features, seeking instead to maximize uniformity and expressivity, and to integrate general verification techniques such as stepwise refinement and data abstraction into the game semantics landscape.

This requires a new treatment of nondeterminism in the context of games, and in particular an account of *dual nondeterminism* is crucial to this task. Compared with trace semantics of process calculi, the major distinguishing feature of game semantics is the polarization of moves. This establishes a boundary between the system and the environment, and is a key ingredient in the construction of compositional models [1]. Likewise, recognizing the distinction and the duality between angelic and demonic nondeterminism is crucial to a satisfactory treatment of refinement, especially in the context of games.

It is interesting to note, however, that refinement orderings which bind too closely the polarity of moves with that of nondeterministic choices often end up requiring a fair amount of sophistication, especially when they are constructed after-the-fact, on top of (or by adapting) an existing model. By contrast, our approach embraces dual nondeterminism to a maximal extent and from the ground up, in a way that is only loosely coupled with the structural details of plays. While this carries an initial cost in the complexity of the strategy model, this is more than offset by the simplicity and uniformity of the resulting refinement algebra.

Acknowledgments

We would like to thank anonymous referees for helpful feedback that improved this paper significantly. This research is based on work supported in part by NSF grants 1521523, 1715154, and 1763399.

References

[1] Samson Abramsky. 2010. From CSP to Game Semantics. In *Reflections on the Work of C.A.R. Hoare*, A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood (Eds.). Springer London, London, 33–45. https://doi.org/10.1007/978-1-84882-912-1_2

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full abstraction for PCF. *Information and computation* 163, 2 (2000), 409–470.

[3] 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*. Springer, 297–329.

[4] Samson Abramsky and Guy McCusker. 1999. Game semantics. In *Computational logic: Proceedings of the 1997 Marktoberdorf Summer School*. Springer, 1–56.

[5] S. Abramsky and P. . Mellies. 1999. Concurrent games and full completeness. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. 431–442.

[6] Luca De Alfaro. 2004. Game models for open systems. In *Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday, volume 2772 of LNCS*. Springer, 269–289.

[7] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. 2002. Alternating-time temporal logic. *Journal of the ACM (JACM)* 49, 5 (2002), 672–713.

[8] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. 1998. Alternating refinement relations. In *CONCUR'98 Concurrency Theory*, Davide Sangiorgi and Robert de Simone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–178.

[9] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbr&#252;cken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17. http://dl.acm.org/citation.cfm?id=1987211.1987212

[10] 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. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20160331.

[11] Ralph-Johan Back. 1978. *On the correctness of refinement steps in program development*. Ph.D. Dissertation. Department of Computer Science, University of Helsinki, Helsinki, Finland.

[12] Ralph-Johan Back and Joakim von Wright. 1998. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, New York.

[13] John Baez and Mike Stay. 2010. Physics, topology, logic and computation: a Rosetta Stone. In *New structures for physics*. Springer, 95–172.

[14] Andreas Blass. 1992. A game semantics for linear logic. *Annals of Pure and Applied logic* 56, 1-3 (1992), 183–220.

[15] Simon Castellan, Pierre Clairambault, Jonathan Hayman, and Glynn Winskel. 2018. Non-angelic concurrent game semantics. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, Cham, 3–19.

[16] 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*. ACM, 18–37.

[17] Luca De Alfaro and Thomas A Henzinger. 2001. Interface automata. *ACM SIGSOFT Software Engineering Notes* 26, 5 (2001), 109–120.

[18] Luca De Alfaro and Thomas A Henzinger. 2001. Interface theories for component-based design. In *International Workshop on Embedded Software*. Springer, 148–165.

[19] Luca de Alfaro and Mariëlle Stoelinga. 2004. Interfaces: A game-theoretic framework for reasoning about component-based systems. *Electronic Notes in Theoretical Computer Science* 97 (2004), 3–23.

[20] Edsger W Dijkstra. 1978. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*. Springer, 166–175.

[21] Dan R. Ghica and Andrzej S. Murawski. 2004. Angelic Semantics of Fine-Grained Concurrency. In *Foundations of Software Science and Computation Structures*, Igor Walukiewicz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–225.

[22] 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* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

[23] 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* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

[24] 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 '18)*. ACM, 646–661.

[25] Russell Harmer and Guy McCusker. 1999. A fully abstract game semantics for finite nondeterminism. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. IEEE, 422–430.

[26] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[27] J Martin E Hyland and C-HL Ong. 2000. On full abstraction for PCF: I, II, and III. *Information and computation* 163, 2 (2000), 285–408.

[28] 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*. ACM, 207–220.

[29] James Laird. 1997. Full abstraction for functional languages with control. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 58–67.

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

[31] Xavier Leroy. 2012. Mechanized semantics for compiler verification. In *Asian Symposium on Programming Languages and Systems*. Springer, 386–388.

[32] C.E. Martin, S.A. Curtis, and I. Rewitzky. 2007. Modelling angelic and demonic nondeterminism with multirelations. *Science of Computer Programming* 65, 2 (2007), 140 – 158. https://doi.org/10.1016/j.scico.2006.01.007 Special Issue dedicated to selected papers from the conference of program construction 2004 (MPC 2004).

[33] Paul-André Melliès and Samuel Mimram. 2007. Asynchronous Games: Innocence Without Alternation. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–411.

[34] Carroll Morgan. 1988. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 3 (1988), 403–419.

[35] Joseph M. Morris. 2004. Augmenting Types with Unbounded Demonic and Angelic Nondeterminacy. In *Mathematics of Program Construction*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–288.

[36] Joseph M Morris and Malcolm Tyrrell. 2008. Dually nondeterministic functions. *ACM Transactions on Programming Languages and Systems*

(TOPLAS) 30, 6 (2008), 34.

[37] A. S. Murawski. 2008. Reachability Games and Game Semantics: Comparing Nondeterministic Programs. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 353–363.

[38] Sumit Nain and Moshe Y. Vardi. 2007. Branching vs. Linear Time: Semantical Perspective. In *Automated Technology for Verification and Analysis*, Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–34.

[39] Sumit Nain and Moshe Y Vardi. 2009. Trace semantics is fully abstract. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*. IEEE, 59–68.

[40] Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures*, Furio Honsell and Marino Miculan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–24.

[41] Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming*. Springer, 80–94.

[42] Ingrid Rewitzky. 2003. Binary multirelations. In *Theory and Applications of Relational Structures as Knowledge Instruments*. Springer, 256–271.

[43] Zhong Shao. 2010. Certified Software. *Commun. ACM* 53, 12 (December 2010), 56–66.

[44] Takeshi Tsukada and CH Luke Ong. 2015. Nondeterminism in game semantics via sheaves. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 220–231.

[45] Malcolm Tyrrell, Joseph M. Morris, Andrew Butterfield, and Arthur Hughes. 2006. A Lattice-Theoretic Model for an Algebra of Communicating Sequential Processes. In *Theoretical Aspects of Computing - ICTAC 2006*, Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 123–137.

[46] 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. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.