Chapter 1

Introduction

1.1 Denotational Semantics and Program Equivalence

Given two pieces of computer code, in what circumstances can we say that they are interchangeable? Clearly, the two pieces of code should return the same output for any choice of input values. But – depending on the expressive power of the language – this might not be enough.

For example, the following two Haskell functions appear to do the same thing.

```
f :: Int -> Int
f n = if (n == 0) then 0 else 0

g :: Int -> Int
g n = 0
```

However, if we introduce a non-terminating function

```
diverge :: Int -> Int
diverge x = diverge x
```

then it becomes clear that f and g are not interchangeable: since f always evaluates its argument n, f (diverge 0) will fail to terminate, whereas g (diverge 0) will give us 0, since inputs to functions are evaluated lazily in Haskell.

We can have another go at answering our original question, then, by adding the requirement that the two programs should behave in the same way if they are passed non-terminating inputs. Thus, we add to each datatype an extra distinguished value $\perp$ representing non-termination (so, for example, the type of integers is represented by the set $\mathbb{Z}_\perp = \mathbb{Z} + \{\perp\}$), and then function types are interpreted as functions between these sets – so a term of type **Int** −> **Int** is reprepresented by a function $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$.

We need to be careful, though, since not every such function arises from a program in this way. For example, we cannot write a program corresponding to the function $\chi\colon \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ that sends $\perp$ to 0 and all other values to 1since χ is not constant, such a program would have to evaluate its argument, and would consequently fail to terminate if that argument did not terminate.

If our language admits higher types, then it becomes especially important to exclude such 'impossible' functions from our model. For example, if F and G are two programs of type (**Int** −> **Int**) −> **Int** – i.e., functions that take in a function from integers to integers and return an integer – then we do not want to declare that F and G are different on the basis that $F(\chi) \neq G(\chi)$.

In order to rule out these 'impossible' functions, we define a partial order on the sets $T_\perp$ corresponding to types, defined by setting $x \leq x$ and $\perp \leq x$ for all x. We then require that functions should be monotonic and continuous with respect to this order. For example, a monotonic function $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$ is either constant (correponding to a function that does not evaluate its argument at all) or sends $\perp$ to $\perp$ and is otherwise unconstrained (corresponding to a function that evaluates its argument). It turns out that if we order functions pointwise, then we get the correct constraints at higher types as well.

Even if we get round the problems with divergence, there are other language features that we may need to consider if we want to determine whether two pieces of code are equivalent. If our functions have access to global variables, then we need to check that these variables end up taking the same final value, whatever their initial values were. If we have IO calls in our language, then we need to check that the functions print out the same text, whatever the user input was. If we have a random number generator, then we need to check that our functions return the same *set* of values, whatever the input.

What we have been doing in all these examples is *denotational semantics*: the art of using mathematical objects to study logic and programming languages. In the first case, our denotational semantics was expressed through the mathematics of sets and functions, whereby we captured the behaviour

of a (programming language) function via a (mathematical) function.

Then, following Scott [Sco76], we refind this model to one based on partially-ordered sets – more specifically, *Scott domains* – in which we modelled the behaviour or a program via an associated Scott-continuous functions between domains.

In our other examples, we need to come up with further refinements to our model in order to incorporate the new computational effects. For example, to handle nondeterminism, we might want to switch to using nondeterministic functions, or *relations*, instead of ordinary functions.

The advantages in all of these cases is that the mathematical objects we use are often fairly simple, whereas computer programs, even in simple 'toy' languages, are very complicated to study. A program is, at its heart, a string of symbols governed by a collection of operational rules that govern how such strings should behave. Such an object is very fiddly to reason with directly; indeed, the only way to think about it is as some kind of 'function' from input to outputs. Denotational takes this basic intuition further, and aims to capture features of programming languages through a diverse collection of different mathematical models.

A word of warning: the principal mode of denotational semantics which we shall be studying in this thesis is game semantics, which is much more complicated than the semantics of sets and functions. However, it is still a very valuable tool for determining equivalence of programs.

1.2 Computational Adequacy and Full Abstraction

In order for a denotational semantics to tell us anything, we first need to prove some results that relate it to the language we are studying. For example, if we are hoping to model a programming language with sets and functions, then we need to define a mapping $[\![-]\!]$ (the *denotation*) that takes program types to sets and program functions to functions between those sets, and we also need to prove that this denotation respects the operational rules of the language: for example, we might want to prove that if $f\colon \mathrm{int} \to \mathrm{int}$ is a function and $M\colon \mathrm{int}$ is a term that evaluates to the integer n, then the term fM will evaluate to the integer $[\![f]\!](n)$. This type of result is called *Computational Adequacy*, and relates to programs of a ground or observable type (e.g., a program that returns an integer has ground type, and we can

observe that it either returns an integer or fails to teminate, whereas a program that takes in an integer and returns an integer has a function type: it is not possible to 'observe' its behaviour without substituting in values).

Briefly speaking, a Computational Adequacy result tells us that the behaviour of a program of ground type may be deduced exactly from its denotation. For example, in a domain-theoretic semantics, we might want to say that a program M evaluates to a value v if and only if $[\![M]\!] = v$ and that M fails to terminate if and only if $[\![M]\!] = \bot$.

Such a computational adequacy result extends readily to terms not of ground type. Given two programs M and N of the same type, we say that M and N are *observationally equivalent* if $C[M]$ and $C[N]$ have the same behaviour for any one-holed context $C[-]$ of ground type. If our semantics is *compositional* – so that the denotation of $C[M]$ is obtained by 'applying' the denotation of $C[-]$ to the denotation of M – and computationally adequate, then it follows that the semantics is *equationally sound*: if two terms M and N have the same denotation, then they are observationally equivalent.

If we have an effective way of computing denotations, then this can give us an easy way to prove observational equivalence of terms. However, there is no guarantee that we are able to use such a trick: the terms M and N might be observationally equivalent despite having distinct denotations. The gold standard of denotational semantics – *Full Abstraction* – asserts in addition that the converse of equational soundness holds, so that the denotational semantics completely captures the observational equivalence relation.

An important early success in this direction came with Plotkin's introduction of the stateless sequential programming language PCF [Plo77]. Plotkin was unable to provide a fully abstract denotational semantics for PCF itself, but he showed that if we add a simple parallel construct[1] to PCF, then a denotational semantics based on Scott domains is fully abstract. This astounding result presents us with a world in which we can practically and systematically check observational equivalence (for terms of this extended version of PCF) by computing denotations. This vision is a little rosey-eyed – deciding observational equivalence is stronger than the halting problem and is hence undecidable in general – but if the programs in question are finitely presentable in some sense, then we really can use the denotational

[1] Specifically, 'parallel or', which evaluates its two boolean arguments in parallel, and is thus able to return true if either the left or the right argument returns true, even if the other fails to terminate.

semantics to check observational equivalence.

Unfortunately, this stop working for PCF itself. PCF is a sub-language of the parallelized version, but this also means that the observational equivalence relation is coarser: there may be terms that can be distinguished by a context including the parallel construct that cannot be distinguished inside any purely sequential context. The enterprise was brought down to earth by Ralph Loader's 2001 theorem that observational equivalence for PCF is undecidable, even if we restrict ourselves to a finitary version of the language with no infinite datatypes or recursion beyond a simple non-termination primitive $\bot$. This in particular tells us that no concretely presentable denotational semantics for PCF can possibly be fully abstract, or it would give us an algorithm for deciding observational equivalence in this finite version.

Nevertheless there were, roughly contemporaneous with Loader's result, several fully abstract models of PCF published, in a watershed moment for the subject. The models published by Nickau [Nic94] and O'Hearn and Riecke [OR95] were more or less along domain-theoretic lines, while those of Abramsky, Jagadeesan and Malacaria [AJM00] and Hyland and Ong [HO00] used the relatively new Game Semantics.

These models took a slightly oblique approach to Full Abstraction. First, they defined the notion of *intrinsic equivalence* of terms of the same type T in a denotational model, where two elements σ and τ of the denotation of T are intrinsically equivalent if $\alpha(\sigma) = \alpha(\tau)$ for all functions $\alpha\colon [\![T]\!] \to [\![o]\!]$ from the denotation of T to the denotation of some fixed ground type o. This definition is very closely linked to that of observational equivalence; indeed, if two terms M and N are observationally equivalent, and we are working in a computationally adequate and compositional denotational semantics, then the denotations of M and N will be intrinsically equivalent, since for any ground-type context $C[-]$, we can take α to be the denotation of $C[-]$ in the above definition.

Proving the converse – that intrinsic equivalence of denotations implies observational equivalence – entails going in the opposite direction; i.e., starting with some element α in the model and coming up with a context $C[-]$ in the language whose denotation is α. Thus, proving this direction normally reduces to some kind of *definability* result. Typically, we only need to prove definability for a restricted class of elements α of the denotational model – the *compact* elements.

If we can prove, for some denotational model of a language, that obser-

vational equivalence of terms is equivalent to intrinsic equivalence of their denotations, then we can form a fully abstract model by passing to equivalence classes under the intrinsic equivalence relation. This is the approach taken by the fully abstract semantics that have been given for PCF; there is no contradiction of Loader's theorem, because the intrinsic equivalence relation is itself undecidable.

In this thesis, we shall skip the final step of passing to equivalence classes and declare a denotational semantics to be fully abstract for a language if we can prove that observational equivalence of terms is equivalent to intrinsic equivalence of their denotations. Thus, the Full Abstraction results that we prove will have three main ingredients: compositionality, computational adequacy and definability.

Lastly, we note that the conclusion of Loader's theorem does not necessarily hold for other languages. For example, in Section **??**, we shall demonstrate a denotational characterization of observational equivalence in Idealized Algol, due to Abramsky and McCusker [AM96], which can be adapted to give us an algorithm for deciding observational equivalence for a finitary version of Idealized Algol.

1.3 Categorical Semantics

There is a close link between (typed) programming langauges and categories. Programming languages have things called *types*, and they have *functions* that go from one type to another. Typically, it will be possible to compose two functions together in the language in an associative way, giving us a category. It should come as no surprise, then, that a very important branch of denotational semantics is *categorical semantics*, in which we take some existing category from mathematics, and use its objects and morphisms to represent the types and terms of a programming language.

Typically, each type T of the language will correspond to some object $[\![T]\!]$ of the category, while a term of type T will correspond to a morphism $1 \to [\![T]\!]$, where 1 is some fixed object in the category (usually a terminal object).

Particularly important [Lam68] are the Cartesian closed categories, which have a number of properties making them suitable for denotational semantics:

Product and function spaces Given types S and T, we can define the denotations of the product type $S \times T$ and the function type $S \to T$

to be given by $[\![S]\!] \times [\![T]\!]$ and $[\![T]\!]^{[\![S]\!]}$.

Compositionality Given types S and T, and corresponding objects $[\![S]\!]$ and $[\![T]\!]$ of the category, we can define the denotation of the function type $S \to T$ to be given by the exponentiation $[\![T]\!]^{[\![S]\!]}$ as above. Then we automatically have a recipe for substituting a term of type S into a function of type $S \to T$ via the canonical morphism

$$[\![T]\!]^{[\![S]\!]} \times [\![S]\!] \to [\![T]\!] \ .$$

Abstraction Given a morphism $\sigma \colon A \times B \to C$, we may form a morphism $\Lambda(\sigma) \colon A \to C^B$. This gives us the semantics for λ-abstraction, whereby we pass from a term-in-context

$$\Gamma, x \colon S \vdash M \colon T$$

to the term-in-context

$$\Gamma \vdash \lambda x.M \colon S \to T \, .$$

These rules allow us to build up a model of the simply-typed λ-calculus within any Cartesian closed category, which means we get a large part of the denotation (and the subsequent proof of Computational Adequacy) for free.

This alone would be a good justification for using category theory in denotational semantics, but the benefits go further. The development of programming languages such as Haskell has been strongly influenced by category-theoretic concepts. For example, Moggi's 19xx observation [**?**] that monads on categories provide us with a way of modelling computational effects influenced work that led directly to the introduction of support for monads in Haskell [Jon95], where they have become the primary tool for abstracting out effectful computation.

Monads will be particularly important in this thesis, so it is worth dwelling on them a little further. A *monad* on a category $\mathcal{C}$ is given by a functor $M \colon \mathcal{C} \to \mathcal{C}$, together with natural transformations

$$e \colon \mathrm{id}_{\mathcal{C}} \Rightarrow M \qquad\qquad m \colon M \circ M \Rightarrow M$$

that endow M with an algebraic structure. One example is the non-empty powerset functor on the category of sets, together with the natural transfor-

mations given by

$$\begin{array}{rl} e\colon A \to \mathcal{P}(A) \\ a \mapsto \{a\} \end{array} \qquad \begin{array}{rl} m\colon \mathcal{P}(\mathcal{P}(A)) \to \mathcal{P}(A) \\ \mathcal{A} \mapsto \bigcup_{A \in \mathcal{A}} A\,. \end{array}$$

This powerset monad indicates some kind of nondeterministic choice between elements of A, particularly if we modify the construction to the *nonempty powerset* functor $\mathcal{P}_+$.

Another example in the category of sets is the functor $A \mapsto A + \{\bot\}$, that appends an additional element on to a set. We have natural functions $A \to A + \{\bot\}$ and $A + \{\bot\} + \{\bot\} \to A + \{\bot\}$ that make this into a monad as well. In the study of programming languages, this is often called the *maybe monad*, because $A + \{\bot\}$ indicates an element of A that may or may not be present (with the distinguished value $\bot$ indicating no value).

Given a monad M on a category $\mathcal{C}$, we can form a new category $\mathrm{Kl}_M\,\mathcal{C}$ – the *Kleisli category* of M – whose objects are the objects of $\mathcal{C}$ and where a morphism from A to B is given by a morphism $A \to MB$ in $\mathcal{C}$. The monadic coherence gives us the correct notion of composition: given Kleisli morphisms $\sigma\colon A \to MB$ and $\tau\colon B \to MC$, we may compose them to give a morphism $A \to MC$ via the following formula.

$$A \xrightarrow{\sigma} MB \xrightarrow{M\tau} MMC \xrightarrow{m} MC$$

The Kleisli category of the powerset monad is the category of sets and relations, while the Kleisli category of the maybe monad is the category of sets and partial functions.

There are numerous other monads that can be used to model computational effects, such as the state monad and the exception monad. Work by Plotkin and Power [PP02] makes this more precise, by studying monads that can be built up via algebraic operations and equations. For example, we might want to model nondeterministic choice on a set A via an operation $\sum$ that takes in infinitely many elements of A – so $\sum a_i$ gives us a choice between the a_i. We then impose some axioms on this operation.

Idempotence If $a_i = a$ for all i, then $\sum a_i = a$;

Commutativity $\sum a_i = \sum_{a_{\pi(i)}}$ for any permutation π; and

Associativity $\sum_i (\sum_j a_{ij}) = \sum_{i,j} a_{ij}$.

This is an algebraic theory akin to the theory of groups, and its category of free algebras is isomorphic to the Kleisli category of the powerset monad.

1.4 Game Semantics

Game Semantics gives us a particularly fruitful categorical semantics for programming languages. The underlying idea is that a computer program behaves like a strategy for a two-player game, in that it needs to respond to arbitrary inputs (opponent moves) with its own behaviours (proponent moves). Thus, we represent a programming language type by an idealized game between two players O and P[2], and represent a term of that type by a strategy for that game.

The power of game semantics comes from the fact, first noted in [Bla92] by Blass, that certain natural operations on games correspond to some of the connectives of linear logic. For example, if A and B are two-player games, then we may form their *tensor product* $A \otimes B$, which is played by running the games A and B together in parallel, with the opposing player O allowed to switch between games when it is his turn.

A closely related construction takes games A and B and forms their *linear implication* $A \multimap B$, in which B is played in parallel with the *dual* of A, in which the roles of players P and O are swapped round. This time, player P can choose to switch game when it is her turn.

The remarkable thing about the $\multimap$ construction, first pointed out by Joyal in [Joy77], is that if A, B and C are games, then we may compose a P-strategy for $A \multimap B$ with a P-strategy for $B \multimap C$ to get a P-strategy for $A \multimap C$.

In order to form her strategy, player P sets up a 'scratchpad' consisting of the games $A \multimap B$ and $B \multimap C$ side by side. Now suppose that player O makes a move in $A \multimap C$ that originally came from the game C. Then player P treats this move as a move in $B \multimap C$, and uses her strategy for that game to come up with a reply. If that reply is a move in C, then she plays it as her response. Otherwise, if it is a move in B, she treats that move as an O-move in $A \multimap B$, and therefore has a reply in $A \multimap B$ according to her strategy for that game. Eventually, if she plays a move in A or C, then that will be her reply in the composite strategy. See Figure 1.1 for an illustration.

It is possible instead that player P flips between her two strategies for ever, always playing moves in B and never coming out into A or C. Computationally, this represents 'livelock': a computation that does not terminate

[2] Another convention is to refer to player O as ∀belard and player P as ∃loïse, so I will refer to player O as 'he' and player P as 'she' throughout.

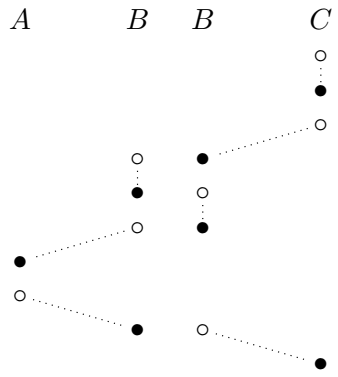

Figure 1.1: Illustration of the composition of strategies in Game Semantics. Working top-to-bottom, the symbol $\circ$ denotes a move by O and the symbol $\bullet$ a move by P. A dotted line indicates that a move is determined by one of player P's strategies for the games $A \multimap B$ and $B \multimap C$. Note that the moves in B are duplicated, so that they may always be considered as an O-move in either $A \multimap B$ or $B \multimap C$. Moves from B are hidden in the composite strategy: in this case, player P's first 'real' move is in C, her second in A and her third in C.

because two subroutines are continually deferring to each other without returning values of their own.

This composition of strategies gives us a category $\mathcal{G}$ in which the objects are games and a morphism from a game A to a game B is a strategy for the game $A \multimap B$. In this category, the identity morphism on a game A is the *copycat strategy* for $A \multimap A$, which responds to an O-move in either copy of A with the identical move in the other copy.

What is more, the connectives $\otimes$ and $\multimap$ make $\mathcal{G}$ into a symmetric monoidal closed category. We typically apply some category-theoretic construction to $\mathcal{G}$ to get a Cartesian closed category, by passing either to the Kleisli category for a linear exponential comonad on $\mathcal{G}$ (as in [AJM00]; see [Sch04]), or to a subcategory of the category of cocommutative comonoids in $\mathcal{G}$ (as in [HO00, AM96]; see [Har99, §3.5.2]). Once we have a Cartesian closed category, we automatically have a way to interpret the simply-typed lambda calculus.

1.5 Game Semantics for Programming Languages

The first triumph of Game Semantics was to solve the Full Abstraction problem for PCF, but a more lasting application of the discipline has been to model more general programming languages with effects such as state. By making changes to the definitions of game and strategy, Game Semantics has proved to be applicable to a wide variety of programming language effects, including exceptions [Lai01], coroutines and continuations [Lai16], nondeterminism [HM99], probability [DH00], and general references [AHM98].

The precise definition of a strategy that we use depends on the language that we are trying to model – a language with less expressive power can realize fewer strategies. For example, the denotations of terms in a stateless langauge such as PCF are *history-free* or *innocent*, in which the proponent's moves can only depend on a particular subsequence of the current sequence of moves – the P-view – rather than on the whole sequence. So for a lot of computational effects, particularly ones that have something to do with state, adding that effect corresponds to a *relaxing* of conditions on strategies.

We model other types of effects by extending the definition of a strategy. For example, if we want to provide a semantics for a language with nondeterminism, then we modify the definition of a strategy so that the propoonent can have multiple replies to each opponent move, as in the work of Harmer and McCusker [HM99]. If we want to model a probabilistic language, then we decorate these different moves with probabilities, as in the work of Danos and Harmer [DH00].

When choosing a definition of a strategy, the aim is to prove a definability result, so that we can prove Full Abstraction. The original proofs of definability of compact innocent strategies in PCF from [AJM00] and [HO00] were intricate and technical. Subsequent work on languages that extend PCF tends to try to prove definability via a *factorization result*, in which we show that every strategy in an extended category of games may be written as the composition of a strategy in an original category of games with some fixed strategy in the new category. Then, if we have a definability result for the original semantics, we can extend it to a definability result in the new category.

For example, the language Idealized Algol is an extended version of PCF that adds some stateful primitives. For example, Abramsky and McCusker's proof of compact definability for Idealized Algol in [AM96] first proves that each compact strategy in their model may be written as the composite of

a compact innocent strategy with the (non-innocent) denotation of one of the new stateful constants. Thus, they can deduce compact definability for Idealized Algol from Hyland and Ong's result that every compact innocent strategy is the denotation of a term of PCF.

Harmer and McCusker develop in [HM99] a model of game semantics in which strategies can be nondeterministic. They show that every such strategy can be written as the composite of a deterministic strategy with some particular fixed nondeterministic strategy. Then, to prove compact definability for nondeterministic Idealized Algol, it suffices for them to exhibit a nondeterministic term whose denotation is that fixed strategy.

We can now summarize the typical process of proving a Full Abstraction language for a language $\mathcal{L}'$ that extends a language $\mathcal{L}$ as follows.

- Starting with an existing computationally adequate model $\mathcal{C}$ of $\mathcal{L}$ that satisfies compact definability, define a categorical model $\mathcal{C}'$ that admits an identity-on-objects functor $J\colon \mathcal{C} \to \mathcal{C}'$. For example, if $\mathcal{C}$ is a category of games and strategies, $\mathcal{C}'$ might be a category whose objects are the same games as $\mathcal{C}$, but where the strategies are less rigidly constrained.
- Prove that the model $\mathcal{C}'$ is a Cartesian closed category and is computationally adequate for $\mathcal{L}'$.
- Prove a factorization result that exhibits every morphism g in $\mathcal{C}'$ as the composition of some morphism Jf in the image of J with one of some fixed collection of morphisms that are known to be definable in $\mathcal{L}'$ (for example, single terms from $\mathcal{L}' \setminus \mathcal{L}$. In addition, if g is compact, then f should be compact in $\mathcal{C}$.

The third bullet point allows us to deduce a compact definability result for $\mathcal{L}'$ in $\mathcal{C}'$ from the compact definability result of $\mathcal{L}$ in $\mathcal{C}$, and then Full Abstraction follows as we have outlined in Section 1.2.

Aside from the proofs of Full Abstraction for stateful and nondeterministic languages that we have mentioned [AM96, HM99], other important Full Abstraction results in Game Semantics that follow this pattern include Danos and Harmer's result for a probabilistic variant of Idealized Algol [DH00], the Full Abstraction result for general references of Abramsky, Honda and McCusker [AHM98], Laird's result for local exceptions [Lai01], Murawski and Tzevelekos's Nominal Game Semantics [MT16] and the result for countable nondeterminism by Laird and the present author [GL18].

It is also worth mentioning several papers that prove Full Abstraction directly, rather than through a factorization result, usually because they depart more radically from traditional game semantics. Such papers include Tsukada and Ong's sheaf-based models for nondeterministic PCF [TO15, TO14], Laird's categorical semantics for coroutines [Lai16] and the results for probabilistic PCF by Castellan, Clairambault, Paquet and Winskel using concurrent games [CCPW18].

1.6 Full Abstraction for Kleisli Categories

One thing which these Full Abstraction results have in common is that they rely on some degree of human intuition to build the original model. This process can often be very difficult. For instance, it seems obvious that if we want to model a nondeterminstic programming language, then we need to relax the determinism constraint on strategies. However, this turns out not to be enough on its own: recall that we model a nonterminating computation in game semantics by a strategy that has no P-reply to a particular O-move. How then do we model a term which chooses between terminating at a value v and not terminating, and how do we distinguish it from a term which always terminates at v? In both cases, player P has the reply v to player O's initial move, but now there is nothing to indicate the possibility of non-termination.

Harmer and McCusker are able to solve this problem, in the finite nondeterminism case, by separately keeping track of divergences in the strategy, but further problems arise when we start to consider countable nondeterminism. Since Game Semantics usually keeps track of finite sequences of moves, nondeterministic strategies are unable to distinguish between a program that nondeterministically chooses a number n and prints "Hello, world" n times, and a program that either does the same thing, or prints out the message infinitely many times. So now we have to add extra information in about infinite sequences of moves (see Levy's work on infinite trace equivalence [Lev08] and the work of Laird and the present author on nondeterministic Idealized Algol [GL18]).

Things get even more difficult when we consider nondeterministic versions of stateless languages such as PCF. Naively relaxing the determinism constraint on the usual definition of an innocent strategy does not give the right notion of a nondeterministic innocent strategy [TO15]. There are definitions of nondeterministic innocence due to Levy [Lev14], and to Tsukada and Ong

[TO15] that use the concept of *morphisms between plays*, but these are already some distance removed conceptually from Hyland and Ong's original paper on deterministic innocence. Tsukada and Ong's paper involves a complete recasting of strategies as sheaves in order to understand what happens when we add nondeterminism to innocence.

Meanwhile, there is plenty of well-known theory that deals with computational effects in a purely systematic way. We have already met two such techniques in section 1.3: monads (and Kleisli categories) and Lawvere theories. Put simply, the purpose of this thesis is to investigate what happens when we try to use these systematic techniques to prove Full Abstraction results for effectful languages.

Let us start by looking at Kleisli categories for monads. Recall that a monad M on a category $\mathcal{C}$ is a functor $M\colon \mathcal{C} \to \mathcal{C}$ that satisfies certain conditions, and that the Kleisli category $\mathrm{Kl}_M\,\mathcal{C}$ of M has the objects of $\mathcal{C}$ as its objects, and that a morphism $a \to b$ in $\mathrm{Kl}_M\,\mathcal{C}$ is given by a morphism $a \to Mb$ in $\mathcal{C}$. There is a natural identity-on-objects functor $J\colon \mathcal{C} \to \mathrm{Kl}_M\,\mathcal{C}$.

In particular, if a is any object of $\mathcal{C}$, then we have a distinguished Kleisli morphism $\phi_A\colon Ma \to a$ in $\mathrm{Kl}_M\,\mathcal{C}$ given by the identity morphism $Ma \to Ma$ in $\mathcal{C}$. Now let $f\colon a \to b$ be an arbitrary morphism in $\mathrm{Kl}_M\,\mathcal{C}$, given by a morphism $\hat{f}\colon a \to Mb$ in $\mathcal{C}$. Then we can write f as the following composite in $\mathrm{Kl}_M\,\mathcal{C}$.

$$a \xrightarrow{J\hat{f}} Mb \xrightarrow{\phi_b} b$$

In other words, the Kleisli category $\mathrm{Kl}_M\,\mathcal{C}$ automatically satisfies a factorization result of the type we have been talking about. So if $\mathcal{C}$ is a model of a language $\mathcal{L}$ that satisfies compact definability, then $\mathrm{Kl}_M\,\mathcal{C}$ will automatically satisfy compact definability for any denotational semantics for a language language $\mathcal{L}'$ that includes (via the functor J) the existing denotational semantics for $\mathcal{L}$ and in which the morphisms ϕ_a are all definable.

Proving computational adequacy is not so automatic. One approach is to treat $\mathrm{Kl}_M\,\mathcal{C}$ like any other model and to apply the usual arguments for proving adequacy. Most computational adequacy arguments rely on order-enriched properties of the underlying categories, particularly for dealing with recursion, and the Kleisli category $\mathrm{Kl}_M\,\mathcal{C}$ automatically inherits order-enriched structure from $\mathcal{C}$, by saying that $f \leq g\colon a \to b$ in $\mathrm{Kl}_M\,\mathcal{C}$ if $f \leq g$ when considered as morphisms $a \to Mb$ in $\mathcal{C}$.

An alternative approach, which we shall use in this thesis, is to try and

deduce computational adequacy for $\mathrm{Kl}_M\,\mathcal{C}$ from a computational adequacy result for the original language $\mathcal{L}$ in $\mathcal{C}$. In this approach, we take a term N of the extended language $\mathcal{L}'$ and note that its denotation $[\![N]\!] : a \to b$ in $\mathrm{Kl}_M\,\mathcal{C}$ is given by some morphism $f\colon a \to Mb$ in $\mathcal{C}$. Typically, it is easy to construct a term P of $\mathcal{L}$ whose denotation in $\mathcal{C}$ is f. We then prove results to peg the operational behaviour of N in $\mathcal{L}'$ to that of P in $\mathcal{L}$, allowing us to deduce a computational adequacy result for our model of $\mathcal{L}'$ from the corresponding result for $\mathcal{L}$.

The last ingredient that we need is to prove that $\mathrm{Kl}_M\,\mathcal{C}$ is a Cartesian closed category. This is not immediate either: $\mathrm{Kl}_M\,\mathcal{C}$ need not be Cartesian, even if $\mathcal{C}$ is. For the purposes of this thesis, we will be considering only monads of a very special form – the *reader monads* R_z on Cartesian closed categories, given by $R_z a = z \to a$. The Kleisli category for a reader monad on a Cartesian closed category is always Cartesian closed [Lam74].

1.7 Difficulties with Countable Nondeterminism

This thesis will focus on nondeterministic effects. Of these, perhaps the trickiest to work with is countable nondeterminism, in which a program can nondeterministically choose between a possibly infinite number of different options, without the possibility of non-termination[3]

There are three main difficulties when dealing with countable nondeterminism in game semantics. The first we have covered in the previous section: we cannot tell everything about a program's behaviour by looking at its possible finite traces, even when the traces are allowed to be arbitrarily long, as in the example where a program nondeterministically chooses a number n and prints out a message n times vs the program which may print out the message infinitely many times.

The fact that finite nondeterminism avoids this behaviour comes down to König's Lemma, which asserts that any finitely branching tree without an infinite branch has bounded height.

This is related to the second problem, which is the failure of continuity of composition, as noticed by Dijkstra in [Dij97, Ch. 9] and later studied by Plotkin and Apt in [AP81]. In order to talk about continuity, we define the

[3]If we have a source of finite nondeterminism, then we can get infinite branching if we accept the possibility that our program might never terminate – for example, count the number of coin tosses it takes before we get the first head.

observational preorder on terms of a language. If $M, N : T$, we write $M \lesssim N$ if for all ground-type contexts $C[-]$ with a hole of type T we have

$$C[M] \text{ always terminates} \Rightarrow C[N] \text{ always terminates.}$$

In particular, M and N are observationally equivalent if and only if $M \lesssim N$ and $N \lesssim M$.

Continuity of composition says that least upper bounds with respect to this order are preserved by function application. Consider, for example, the terms $\leq_m : \mathbb{N} \to \mathbb{N}$ for $m = 0, 1, \ldots$ that return 0 if their argument is less than or equal to m and go into an infinite loop otherwise. It is fairly clear (with a straightforward rigorous proof once we have introduced the denotational semantics) that a least upper bound for the $\leq_m$ is the term $\leq_\infty$ that evaluates its argument and then returns 0 regardless of what value it finds[4].

But now notice what happens if we have a countable nondeterminism primitive ? in our language which when evaluated returns an arbitrarily large natural number. Then the terms $\leq_m ?$ are all observationally equivalent: they either return 0 (when ? returns a number less than or equal to m), or fail to terminate (when ? returns a number greater than m). Therefore, $\leq_m ?$ is a constant sequence and its least upper bound is that constant value, which we can write as

$$0 + \Omega\,.$$

Meanwhile, $\leq_\infty ?$ always returns 0. So application to ? does not preserve least upper bounds.

The reason that this is a problem is that a typical strategy for proving computational adequacy is via an order-enriched category in which the observational preorder at a type matches up with the ordering of morphisms into the denotation of that type. The standard adequacy proof uses continuity of composition in the model in an essential way – see Lemma **??** for an example of such an argument being used to prove computational adequacy for a deterministic language. But if the ordering of morphisms in a denotational model of a language with countable nondeterminism faithfully respects the observational ordering of terms, then it will not have the continuity properties that we need for this proof.

[4] $\leq_\infty$ differs slightly from $\lambda n.0$ – also an upper bound for the $\leq_m$, but not the least upper bound – in that it will fail to terminate if its argument fails to terminate

Our strategy for getting around this problem goes back to Levy's paper *Infinite Trace Equivalence* [Lev08], and essentially involves going via an auxiliary language, in which the nondeterministic oracle ?, when it chooses a value, must print that value to a log. Given a term P of ground type, we write $P \Downarrow_u$ if P terminates whenever it prints the sequence u to the log. Then, if we modify our definition of observational equivalence to say that $M \lesssim N$ if for all suitable contexts $C[-]$ we have

$$C[M] \Downarrow_u \Rightarrow C[N] \Downarrow_u ,$$

then this ordering will be continuous. For example, this extra requirement wrecks our previous example of failure of continuity of composition. Indeed, the sequence $\leq_m ?$ is no longer constant, since, for example, $\leq_5 ?$ does not terminate if it prints 6 to the log, whereas $\leq_{1000} ?$ does. The least upper bound of the $\leq_m ?$ is then the term that terminates whenever it prints a single number to the log; i.e., $\leq_\infty ?$.

Having proved computational adequacy for this new language, we then use operational methods to find a way to identify terms that should be observationally equivalent in the original sense. We form a model for the language with countable nondeterminism by taking our model for the new language and taking the quotient by an appropriate equivalence relation on morphisms.

In order to introduce the last problem that we face with countable nondeterminism, we note that there are two different reasons why an element of a denotational model (e.g., a strategy) might not be definable in a particular reason. One reason is structural: for example, a stateless language such as PCF cannot define a strategy whose behaviour depends on its entire history, since that would indicate stateful behaviour. The other reason is purely computational. Given a non-computable function $f : \mathbb{N} \to \mathbb{N}$ (for example, the function that returns 0 if its argument is source code for a terminating program an 1 otherwise), we have a perfectly well-behaved history free strategy that represents f, but it is nevertheless still not definable in most programming languages. Historically, the study of Full Abstraction has not been too concerned with non-definable elements of the second kind: the reason is that they are do not usually play a role in determining the intrinsic equivalence relation, since such a strategy is necessarily not compact. However, in the presence of countable nondeterminism, there exist terms that can be distinguished in the model that cannot be distinguished by compact elements; indeed, we can use a diagonal argument to define two

functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ that are distinguished only by non-computable functions $\mathbb{N} \to \mathbb{N}$. An example of such a function is found in Proposition **??**; the general idea is due to Kleene and can be found in [Bau06].

What this means is that if we want to model countable nondeterminism then, in addition to the usual structural constraints on strategies relating to the effectful behaviour available in our language, we also need to impose computability constraints that say that our strategies can in some sense be defined using a computable function. Happily, such notions are well studied already, even appearing in the original Hyland-Ong and AJM papers [HO00, AJM00], in which the authors give a complete characterization of those strategies definable in PCF – a *universality* result for the model. Simmilar results can also be found in earlier work on Full Abstraction, such as Plotkin's original PCF paper [Plo77].

The difficulties with countable nondeterminism that we have outlined are essentially domain-theoretic; indeed, the reason that compact strategies suffice to distinguish morphisms in models of finite nondeterminism is that the sets of morphisms in that case form *algebraic domains*, in which every element is the least upper bound of the set of compact elements below it. Algebraicity fails for models of countable nondeterminism.

1.8 Plan for this Thesis

This thesis will develop the theory of Full Abstraction from the point of view of techniques of categorical algebra such as Kleisli categories.

Chapters ?? and ?? give a fairly traditional presentation of a Fully Abstract game semantics model for Idealized Algol. All subsequent results will be for extended versions of Idealized Algol, so this model can serve as the foundation for the rest of our work. Although these chapters form by some distance the longest part of the thesis, they also contain the smallest amount of new material, and owe much to Abramsky and McCusker's Fully Abstract game semantics for Idealized Algol [AM96]. The main difference, which ties in with the general theme of the thesis, is that we prove Computational Adequacy using techniques of categorical algebra rather than via the ad hoc combinatorial arguments found in [AM96]. Specifically, we use Laird's concept of a *sequoidal category* and of the exponential as a final coalgebra, used to prove Full Abstraction for a language with general references [Lai02]. We present the first application of this technique to a lan-

guage with purely local state of ground type. The remainder of the Full Abstraction result is largely as in [AM96].

Chapter ?? presents the general theory of monads and Kleisli categories. It then presents a technique which can be used to prove Full Abstraction for several nondeterministic effects, along the lines we outlined in the previous section.

For the remainder of the thesis, we deal with a generalization of monads – *parametric monads*, in which the action of the monad is parameterized by an object of some monoidal category $\mathcal{X}$ (so that we deal with a functor $\mathcal{X} \times \mathcal{C} \to \mathcal{C}$ rather than a functor $\mathcal{C} \to \mathcal{C}$).

Chapter ?? introduces and defines parametric monads (also known as *lax actions*), and proves a number of technical results which we need. It also introduces the *Melliès category* and the oplax 2-limit $\mathcal{C}/\mathcal{X}$, which give us two related analogues of the Kleisli category in the parametric case. The Melliès category is an enriched category, while $\mathcal{C}/\mathcal{X}$ is an ordinary category obtained from it by change of base. As with monads, we need to specialize to a small class of parametric monads – the *reader actions* – in order to ensure that $\mathcal{C}/\mathcal{X}$ is a Cartesian closed category.

Chapter ?? uncovers a large source of these reader actions, proving that reader actions on the category **Set** of sets are equivalent to lax monoidal functors $\mathbf{Set} \to \mathbf{Set}$.

Chapter ?? takes a small detour away from the main narrative to study the Melliès category from the point of view of profunctors.

Chapter ?? completes our study of parametric monads by proving a Full Abstraction result for a probabilistic language for a category derived from a particular parametric monad, using similar techniques to Chapter **??**.

Lastly, our conclusion in **Chapter ??** provides a glimpse into several further directions that are left unexplored by the rest of the thesis.

1.9 Acknowledgements

TODO

Bibliography

[AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, LICS '98, pages 334–, Washington, DC, USA, 1998. IEEE Computer Society.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409 – 470, 2000.

[AM96] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions: Extended abstract. *Electronic Notes in Theoretical Computer Science*, 3:2 – 14, 1996. Linear Logic 96 Tokyo Meeting.

[AP81] K. R. Apt and G. D. Plotkin. A cook's tour of countable nondeterminism. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, pages 479–494, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[Bau06] Andrej Bauer. König's lemma and the Kleene tree. Published via blog post at `http://math.andrej.com/2006/04/25/konigs-lemma-and-the-kleene-tree/`, April 2006.

[Bla92] Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56(1–3):183 – 220, 1992.

[CCPW18] Simon Castellan, Pierre Clairambault, Hugo Paquet, and Glynn Winskel. The concurrent game semantics of probabilistic pcf. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on*

Logic in Computer Science, LICS '18, pages 215–224, New York, NY, USA, 2018. ACM.

[DH00] V. Danos and R. Harmer. Probabilistic game semantics. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 204–213, June 2000.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[GL18] W. John Gowers and James D. Laird. A Fully Abstract Game Semantics for Countable Nondeterminism. In Dan Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Har99] Russell S. Harmer. Games and full abstraction for nondeterministic languages. Technical report, 1999.

[HM99] R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 422–430, 1999.

[HO00] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285 – 408, 2000.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, Berlin, Heidelberg, 1995. Springer-Verlag.

[Joy77] André Joyal. Remarques sur la théorie des jeux à deux personnes. *Gazette des sciences mathematiques de Quebec*, 1(4), 1977.

[Lai01] J. Laird. A fully abstract game semantics of local exceptions. In *Proceedings of LICS '01*, pages 105–114. IEEE Computer Society Press, 2001.

[Lai02] J. Laird. A categorical semantics of higher-order store. In *Proceedings of CTCS '02*, number 69 in ENTCS. Elsevier, 2002.

[Lai16] James Laird. Higher-order programs as coroutines. to appear, 2016.

[Lam68] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103(2):151–161, 1968.

[Lam74] J. Lambek. Functional completeness of cartesian categories. *Annals of Mathematical Logic*, 6(3):259 – 292, 1974.

[Lev08] Paul Blain Levy. Infinite trace equivalence. *Annals of Pure and Applied Logic*, 151(2):170 – 198, 2008.

[Lev14] Paul Blain Levy. Morphisms between plays. Lecture slides, GaLoP, 2014.

[MT16] Andrzej S. Murawski and Nikos Tzevelekos. Nominal game semantics. *Found. Trends Program. Lang.*, 2(4):191–269, March 2016.

[Nic94] Hanno Nickau. Hereditarily sequential functionals. In *International Symposium on Logical Foundations of Computer Science*, pages 253–264. Springer, 1994.

[OR95] P.W. Ohearn and J.G. Riecke. Kripke logical relations and pcf. *Information and Computation*, 120(1):107 – 116, 1995.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

[PP02] Gordon Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 342–356, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Sch04] Andrea Schalk. What is a categorical model for linear logic?, October 2004.

[Sco76] D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[TO14] Takeshi Tsukada and C.-H. Luke Ong. Innocent strategies are sheaves over plays - deterministic, non-deterministic and probabilistic innocence. *CoRR*, abs/1409.2764, 2014.

[TO15] Takeshi Tsukada and C. H. Luke Ong. Nondeterminism in game semantics via sheaves. In *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, LICS '15, pages 220–231, Washington, DC, USA, 2015. IEEE Computer Society.