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 lan-

guages. 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

[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.

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 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 observational 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.

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 transformations given by

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

This powerset monad indicates some kind of nondeterministic choice between elements of A, particularly if we modify the construction to the *non-empty 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 two-player game and represent a term of that type by a strategy for that game.

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 [**?**] first proves that each compact strategy in their model may be written as the composite of a com-

pact 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.

Bibliography

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

[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.

[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.

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

[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.

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