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.

We can then use concepts from category theory to inform our denotational semantics. The most important of these is that if the category we choose is Cartesian closed, then we can automatically interpret terms of the simply typed lambda calculus as morphisms. This is particularly useful, since many of the languages that we are most interested in studying from a mathematical point of view – in particular, PCF and Idealized Algol – are based upon the simply typed lambda calculus. Thus, if we make sure that our base category is Cartesian closed, then we get a denotational semantics for the lambda calculus-part of the language for free, allowing us to concentrate on the other parts of the language.

Category-theoretic topics have taken on a new importance within functional programminging. In 1991, Eugenio Moggi observed that *monads* on categories provide a way to study computational effects. For example, we have already alluded to the fact that a stateful function $A \to B$ may be identified

with an ordinary stateless function

$$A \times W \to B \times W\,,$$

where W is some fixed set (the *state*). Such a function may alternatively be written as a function

$$A \to (W \to (B \times W))\,;$$

thus, a morphism $A \to B$ in the Kleisli category for the *state monad*

$$B \mapsto W \to (B \times W)\,.$$

Similarly, a nondeterministic function $A \to B$ may be thought of as a function from elements of A to subsets of B, and thus as a morphism in the Kleisli category for the *powerset monad* $\mathcal{P}-$.

The link between monads and computational effects has not been confined to the theoretical domain; indeed, monads quickly became the main way of expressing effectful programming in languages such as Haskell [Jon95].

In 2001, Plotkin and Power [PP02] made this idea more precise by demonstrating that if a computational effect can be given by algebraic operations and equations, then it determines a monad.

More specifically, a *signature* Σ is made up of a set of *operations* σ with (possibly infinite) *arities* $\mathrm{ar}(\sigma)$ and *equations* that encode the rules that hese operations must follow. An *interpretation* of a signature Σ in a category $\mathcal{C}$ with products is given by an object A of $\mathcal{C}$, together with morphisms

$$A^{\mathrm{ar}}(\sigma) \to A$$

for each operation σ in Σ, such that the equations all hold when interpreted as equalities of morphisms in $\mathcal{C}$.

A *homomorphism* between Σ-algebras A and B is a morphism $A \to B$ that respects the Σ-algebra structure on A and B in a natural way. Thus, we get a category of Σ-algebras. Moreover, the natural forgetful functor from this category into $\mathcal{C}$ admits a left adjoint, inducing a monad on $\mathcal{C}$. This monad sends a set A to the underlying set of the *free* Σ*-algebra* on A, given by taking the set of all terms that can be formed from the operations in Σ, with the elements of A as variables, and quotienting by the equivalence generated by the equations in Σ.

For example (see [CBHDP06], for instance), the powerset monad for countable sets is induced from the signature $\Sigma_{\mathcal{P}}$ consisting of a single infinitary operator $\sum$ (i.e., nondeterministic choice between the infinite collection of values), together with equations for infinitary commutativity (reordering the a_i should not change the observed behaviour), associativity ($\sum_i \sum_j a_{ij} = \sum_{i,j} a_{ij}$) and idempotence (repeat entries do not change the behaviour).

If A is a set, then we may identify an element ϕ of the free $\Sigma_{\mathcal{P}}$-algebra on A with the subset of A consisting of all elements $a \in A$ that appear in the term-description of ϕ. For example, if $A = \mathbb{N}$, then we can identify

$$\sum_i 2i$$

with the set of even natural numbers, and

$$\sum_i \sum_j \pi(i)^j$$

with the set of all prime powers. The axioms for commutativity, associativity and idempotence mean that this is a well-defined operation, giving us an equivalence between the free $\Sigma_{\mathcal{P}}$-algebra monad and the powerset monad $\mathcal{P}$.

The state monad with state given by a set W is induced from the signature given by a W-ary operation lookup, corresponding to choosing between W-many options based on the value of the state; and W-many nullary operations update_w, each corresponding to updating the state to a particular value w. There are four axioms relating these two operations, which we adapt from [PP02].

Update with current value

$$\text{lookup}(\text{update}_w(x) : w \in W) = x\,.$$

Looking up the current value and rewriting it is a no-op.

Idempotence of lookup

$$\text{lookup}(\text{lookup}(x_{vw} : v \in W) : w \in W) = \text{lookup}(x_{ww} : w \in W)\,.$$

If we look up the state twice in succession, then we will get the same value both times.

Double update

$$\mathrm{update}_v(\mathrm{update}_w(x)) = \mathrm{update}_w(x)\,.$$

If we update the value twice in succession, then the first update has no effect.

Lookup

$$\mathrm{update}_w(\mathrm{lookup}(x_v \,:\, v \in W)) = \mathrm{update}_w(x_w)\,.$$

If we update the value to w and then look up the state, we get w back.

Plotkin and Power prove in [PP02] that the free algebra monad for this signature is isomorphic to the usual state monad $(W \times -)^W$. We therefore get a strong connection between some very concrete syntactic concepts about what state should look like and the abstract category-theoretic concept of a monad.

Bibliography

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

[CBHDP06] M C. B. Hennessy and G D. Plotkin. *Full Abstraction for a Simple Parallel Programming Language*, volume 74, pages 108–120. 01 2006.

[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.

[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.