Contents

Introduction

Types are types and propositions are propositions; types come from programming languages, and propositions from logic, and they seem to have no relation to each other. We shall see that if we make certain assumptions about both logic and programming, then we can define a system which is *simultaneously* a logic and a programming language, and in which propositions and types are *identical*. This is the system of *constructive type theory*, based primarily on the work of the Swedish logician and philosopher, Per Martin-Löf. In this introduction we examine the background in both logic and computing before going on to look at constructive type theory and its applications. We conclude with an overview of the book proper.

Correct Programming

The problem of correctness is ever-present in computing: a program is written with a particular specification in view and run on the assumption that it meets that specification. As is all too familiar, this assumption is unjustified: in most cases the program does *not* perform as it should. How should the problem be tackled? Testing cannot ensure the absence of errors; only a formal proof of correctness can guarantee that a program meets its specification. If we take a naïve view of this process, we develop the program and then, *post hoc*, give a proof that it meets a specification. If we do this the possibility exists that the program developed doesn't perform as it ought; we should instead try to develop the program in such a way that it *must* behave according to specification.

A useful analogy here is with the types in a programming language. If we use a typed language, we are prevented by the rules of syntax from forming an expression which will lead to a type error when the program is executed. We could *prove* that a similar program in an untyped language shares this property, but we would have to do this for each program developed, whilst in the typed language it is guaranteed in every case.

Our aim, then, is to design a language in which correctness is guaranteed. We look in particular for a *functional programming language* with this property, as semantically the properties of these languages are the most straightforward, with a program simply being a value of a particular explicit type, rather than a state transformer.

How will the new language differ from the languages with which we are familiar?

- The type system will have to be more powerful. This is because we will express a specification by means of a statement of the form

  $$p : P$$

  which is how we write 'the value p has the type P'. The language of types in current programming languages can express the domain and range of a function, say, but cannot express the constraint that for every input value (of numeric type), the result is the positive square root of the value.

- If the language allows general recursion, then every type contains at least one value, defined by the equation $x = x$. This mirrors the observation that a non-terminating program meets every specification if we are only concerned with partial correctness. If we require total correctness we will need to design a language which only permits the definition of *total* functions and fully-defined objects. At the same time we must make sure that the language is expressive enough to be usable practically.

To summarise, from the programming side, we are interested in developing a language in which correctness is guaranteed just as type-correctness is guaranteed in most contemporary languages. In particular, we are looking for a system of types within which we can express logical specifications.

Constructive Logic

Classical logic is accepted as the standard foundation of mathematics. At its basis is a truth-functional semantics which asserts that every proposition is true or false, so making valid assertions like $A \vee \neg A$, $\neg\neg A \Rightarrow A$ and

$$\neg\forall x.\neg P(x) \Rightarrow \exists x.P(x)$$

which can be given the gloss

> If it is contradictory for no object x to have the property $P(x)$, then there is an object x with the property $P(x)$

This is a principle of indirect proof, which has formed a cornerstone of modern mathematics since it was first used by Hilbert in his proof of the Basis Theorem about one hundred years ago. The problem with the principle is that it asserts the existence of an object without giving any indication of what the object is. It is a *non-constructive* method of proof, in other words. We can give a different, constructive, rendering of mathematics, based on the work of Brouwer, Heyting, Bishop and many others, in which every statement has computational content; in the light of the discussion above it is necessary to reject classical logic and to look for modes of reasoning which permit only constructive derivations.

To explain exactly what can be derived constructively, we take a different foundational perspective. Instead of giving a classical, truth-functional, explanation of what is valid, we will explain what it means for a particular object p to be a proof of the proposition P. Our logic is *proof-functional* rather than truth-functional.

The crucial explanation is for the existential quantifier. An assertion that $\exists z.P(z)$ can only be deduced if we can produce an a with the property $P(a)$. A proof of $\exists z.P(z)$ will therefore be a *pair*, (a, p), consisting of an object a and a proof that a does in fact have the property P. A universal statement $\forall z.Q(z)$ can be deduced only if there is a function taking any object a to a proof that $Q(a)$. If we put these two explanations together, a constructive proof of the statement

$$\forall x.\exists y.R(x, y)$$

can be seen to require that there is a function, f say, taking any a to a value so that

$$R(a, f\ a)$$

Here we see that a constructive proof has computational content, in the shape of a function which gives an explicit *witness* value $f\ a$ for each a.

The other proof conditions are as follows. A proof of the conjunction $A \wedge B$ can be seen as a *pair* of proofs, (p, q), with p a proof of A and q of B. A proof of the implication $A \Rightarrow B$ can be seen as a proof transformation: given a proof of A, we can produce a proof of B from it. A proof of the disjunction $A \vee B$ is either a proof of A or a proof of B, together with an indication of which (A or B). The negation $\neg A$ is defined to be the implication $A \Rightarrow \perp$, where $\perp$ is the absurd or false proposition, which has no proof but from which we can infer anything. A proof of $\neg A$ is thus a function taking a proof of A to a proof of absurdity.

Given these explanations, it is easy to see that the law of the excluded middle will *not* be valid, as for a general A we cannot say that either A or $\neg A$ is provable. Similarly, the law of indirect proof will not be valid.

Having given the background from both computing and logic, we turn to examining the link between the two.

The Curry Howard Isomorphism

The central theme of this book is that we can see propositions and types as the same, the *propositions-as-types* notion, also known as the *Curry Howard* isomorphism, after two of the (many) logicians who observed the correspondence.

We have seen that for our constructive logic, validity is explained by describing the circumstances under which 'p is a proof of the proposition P'. To see P as a type, we think of it as the type of its proofs. It is then apparent that familiar constructs in logic and programming correspond to each other. We shall write

$$p : P$$

to mean, interchangeably, 'p is of type P' and 'p is a proof of proposition P'.

The proofs of $A \wedge B$ are pairs (a, b) with a from A and b from B — the conjunction forms the *Cartesian product* of the propositions as types. Proofs of $A \Rightarrow B$ are functions from A to B, which is lucky as we use the same notation for implication and the function space. The type $A \vee B$ is the *disjoint union* or *sum* of the types A and B, the absurd proposition, $\perp$, which has no proofs, is the empty type, and so on.

The correspondence works in the other direction too, though it is slightly more artificial. We can see the type of natural numbers N as expressing the proposition 'there are natural numbers', which has the (constructive!) proofs $0, 1, 2, \ldots$.

One elegant aspect of the system is in the characterisation of inductive types like the natural numbers and lists. Functional programmers will be familiar with the idea that functions defined by recursion have their properties proved by induction; in this system the principles of induction and recursion are *identical*.

The dual view of the system as a logic and a programming language can enrich both aspects. As a logic, we can see that all the facilities of a functional programming language are at our disposal in defining functions which witness certain properties and so forth. As a programming language, we gain various novel features, and in particular the quantified types give us dependent function and sum types. The dependent function space $(\forall x : A) . B(x)$ generalises the standard function space, as the type of the result $B(a)$ depends on the value $a : A$. This kind of dependence is usually not available in type systems. One example of its use is in defining array operations parametrised on the dimension of the array, rather than on the type of the array elements. Dependent sum types $(\exists x : A) . B(x)$ can be used to represent modules and abstract data types amongst other things.

More radically, we have the possibility of combining verification and programming, as types can be used to represent propositions. As an example consider the existential type again. We can think of the elements of $(\exists x : A) . B(x)$ as objects a of type A with the logical property $B(a)$, witnessed by the proof $b : B(a)$. We can give a third interpretation to $p : P$, in the case that P is an existential proposition:

$$(a, b) \; : \; (\exists x : A) . B(x)$$

can be read thus:

a of type A meets the *specification* $B(x)$, as proved by $b : B(a)$

This fulfills the promise made in the introduction to logic that we would give a system of types strong enough to express specifications. In our case the logic is an extension of many-sorted, first-order predicate logic, which is certainly sufficient to express all practical requirements. The system here integrates the process of program development and proof: to show that a program meets a specification we provide the program/proof pair.

As an aside, note that it is misleading to read $p : P$ as saying 'p meets specification P' when P is an arbitrary proposition, an interpretation which seems to be suggested by much of the literature on type theory. This is because such statements include simple *typings* like

$$plus \; : \; N \Rightarrow N \Rightarrow N$$

in which case the right-hand side is a woeful under-specification of addition! The specification statement is an existential proposition, and objects of that type include an explicit *witness* to the object having the required property: in other words we can only state that a program meets its specification when we have a proof of correctness for it.

We mentioned that one motivation for re-interpreting mathematics in a constructive form was to extract algorithms from proofs. A proof of a statement like

$$\forall x . \exists y . R(x, y)$$

contains a description of an *algorithm* taking any x into a y so that $R(x, y)$. The logic we described makes explicit the proof terms. On the other hand it is instead possible to suppress explicit mention of the proof objects, and *extract* algorithms from more succinct derivations of logical theorems, taking us from proofs to programs. This idea has been used with much success in the Nuprl system developed at Cornell University, and indeed in other projects.

Background

Our exposition of type theory and its applications will make continual reference to the fields of functional programming and constructivism. Separate introductions to these topics are provided by the introduction to chapter 2 and by chapter 3 respectively. The interested reader may care to refer to these now.

Section 9.2 contains some concluding remarks.