Constructive logic and type theory *

Erik Palmgren
Department of Mathematics, Uppsala University

March 2004

Constructive logic is based on the leading principle that

(I) *proofs are programs.*

In constructive type theory it has later been joined with the further principle that

(II) *propositions are data types.*

The latter is also called the *propositions-as-types principle*. The meaning of these principles is on first sight far from obvious. The purpose of these lectures is to explain these and their consequences and applications in computer science. A particular aim is to provide background for the study of Martin-Löf type theory, or similar theories of dependent types, and for using proof-checking systems based on such theories. Proofs carried out within constructive logic may be considered as programs in a functional language, closely related to e.g. ML or Haskell. The importance of this is the possibility to extract from an existence proof (that, e.g., there are arbitrarily large prime numbers) a program that finds or constructs the purported object, and further obtain a verification that the program terminates (finds some number) and is correct (finds only sufficiently large prime numbers). The combination of construction and verification of programs has raised considerable interest in computer science. Systems supporting this are known as *integrated program logics*. The combination of a program, and proofs of some of its properties, is called *proof-carrying code* and is of potential importance for improving the trustworthiness of software in network based programming (cf. Java programming).

*Lecture notes for a course in Applied Logic (Tillämpad logik DV1, datavetenskapliga programmet). This is an extension and translation of original notes in Swedish: *Konstruktiv logik*, U.U.D.M. Lecture Notes 2002:LN1. The author is grateful for corrections and suggestions by Peter Dybjer, Thierry Coquand and Fredrik Dahlgren.

Constructive logic has a history well before the advent of electronic computers. Around the turn of the century 1899-1900 there arised doubts about the consistency of the axiomatic and logical foundations for the abstract mathematics that had started to develop with the support of set theory. Bertrand Russell had 1903 with his well-known paradox shown that unrestricted formation of sets from concepts may lead to outright contradictions. (*Is there a set that consists of all sets that are not members of themselves?* Answer: No.) Also other principles in use, such as the Axiom of Choice, had been discovered to have unintuitive and unnatural consequences, even though no paradoxes where known to arise from them. Ernst Zermelo showed 1908 that the set of real numbers may be well-ordered. It was among many mathematicians considered as a serious scientific crisis of the subject, known as the *Grundlagenkrisis*. In that mood of time the outstanding Dutch mathematician, and founder of modern topology, L.E.J. Brouwer started a critical examination and reconstruction of the foundations for mathematics, which went further than previous attempts, and included the very logic and not only the axioms. By the introduction of his *intuitionistic mathematics* he wanted to put mathematics on a secure and intuitive footing. His idea was that every proof must built on a so-called *mental construction*. At that time (1910) there were of course no programming languages, and not even a mathematical notion of algorithm, but it turned out that his notion of mental construction could be interpreted as algorithmic construction in a precise way. This requirement led Brouwer to reject a logical law that had been taken for granted since Aristotle, namely the *Principle of Excluded Middle* or *Tertium Non Datur*. This states that for every proposition A, either A is true or A is false, in logical symbols:

$$A \vee \neg A \qquad (\text{PEM}).$$

For concrete, finitely checkable, propositions there was no reason to doubt the law. The problematic case, according to Brouwer, is when A contains quantification over an infinite set, for instance the set of integers. Brouwer demonstrated that it was possible to develop mathematics also without the principle of excluded middle, and that it in many cases lead to more informative and insightful proofs. The immediate followers of Brouwer were not very numerous, and his use of the theory of choice sequences, which is inconsistent with classical logic, repelled many mathematicians from his philosophy. Later developments of constructive mathematics avoid this theory, and its results are immediately understandable and acceptable to mainstream mathematics (Bishop and Bridges 1985, Bridges and Richman 1987).

1 Non-constructive proofs

We shall illustrate how the principle of excluded middle is used in some non-constructive existence proofs, and how this may lead to the loss of algorithmic content. Here is a famous standard example, chosen for its simplicity rather than mathematical interest.

Proposition 1.1 *There are irrational numbers a and b such that a^b is an rational number.*

Proof. The number $\sqrt{2}$ is irrational (non-rational). Consider the number $\sqrt{2}^{\sqrt{2}}$. By the principle of excluded middle it is either rational or irrational. If it is rational, we may finish the proof by exhibiting $a = b = \sqrt{2}$. If it is irrational, let $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then a^b is rational, indeed we have

$$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^2 = 2. \square$$

Note that in this existence proof it was not decided which of the numbers $a = \sqrt{2}$ or $a = \sqrt{2}^{\sqrt{2}}$, with $b = \sqrt{2}$, that actually gives the intended example. One may actually show, using deeper mathematical theory, that $\sqrt{2}^{\sqrt{2}}$ is irrational, but the standard proof requires several pages.

Many classical existence proofs are indirect, and start "Suppose that there is no object x such that ...". The rest of the proof is devoted to show that a contradiction arises from this assumption. Here is a simple example involving infinite sequences.

Proposition 1.2 *Each infinite sequence of natural numbers*

$$a_1, a_2, a_3, \ldots, a_k, \ldots$$

has a minimal term, i.e. there is some n such that $a_n \leq a_k$ for all k.

Proof. Suppose that there is no minimal term in the sequence. For every index k we may then find $k' > k$ with $a_k > a_{k'}$. Let $k_1 = 1$. By assumption there is then $k_2 > k_1$ with $a_{k_1} > a_{k_2}$. Again, by the assumption, we find $k_3 > k_2$ such that $a_{k_2} > a_{k_3}$. Continuing in this manner we obtain an infinite sequence $k_1 < k_2 < k_3 < \cdots$ such that

$$a_{k_1} > a_{k_2} > a_{k_3} > \cdots.$$

This sequence of natural numbers decrease by at least one unit for every step, so

$$a_{k_n} \leq a_1 - (n-1).$$

Putting $n = a_1 + 2$, we thereby find a term in the sequence less than 0, which is impossible. □

This existence proof gives no information whatsoever where the minimal term is to be found. Not only the proof of this result is non-constructive, but the result is essentially non-constructive. Consider for example the sequence obtained in the following way: given a description of a Turing machine and an input string, let a_k be 0 if the machine has terminated after k steps, and 1 otherwise. If we would be able to find the minimum of this sequence, algorithmically, we could also solve the general halting problem algorithmically. However this is known to be impossible.

This kind of "information-less" existence proofs are not uncommon in mathematical analysis. For instance certain results on the existence of solutions to differential equations build on such proofs (Cauchy-Peano existence proof). There are rather simple examples of ordinary differential equations, whose solutions cannot computed from the parameters, though there are theoretical solutions (cf. Beeson 1985)

Constructive mathematics and logic are founded on the idea that existence is taken more seriously than in classical mathematics: to prove that a certain object exists is the same as giving a method for constructing it. There is also the possibility of further requiring the method to be effective according to some complexity measure (Nelson 1995, Bellantoni and Cook 1992). Such complexity restrictions gives a quite different kind of mathematics, not very well investigated as of yet.

Exercises

1.1. Recall that an algebraic real number is a real number which is a root of a polynomial with integer coefficients. Such numbers are closed under the usual arithmetical operations. Any real number which is not algebraic, is called *transcendental*. The numbers e and π are transcendental. Prove, using only these facts, that $e+\pi$ or $e-\pi$ is transcendental. (It is unknown which one it is, or if both are transcendental).

1.2 (König's lemma). A finite string over the alphabet $\{l, r\}$ is regarded as describing a path in a binary tree, starting from the root. Suppose that P is an infinite set of such paths. Show that there is an infinite string

$$d_1 d_2 d_3 \cdots$$

such that for every n, the string $d_1 d_2 \cdots d_n$ is an initial segment of some path in P.

1.3 (Brouwer's Fan Theorem). Consider a set of paths P such that if w is in P, then so is each of its initial segments, i.e. if $w = uv \in P$, then $u \in P$. An infinite path

$d = d_1 d_2 d_3 \cdots$ is said to be *barred by P*, if there is some n such that $d_1 \cdots d_n \notin P$. Show that if every infinite path d is barred by P, then P must be finite.

1.4. Prove that $b = 2\log_2 3$ is irrational. Use this to give a constructive proof of Proposition 1.1.

2 Typed lambda calculus

We give a short introduction to lambda calculus, with emphasis on those parts which are useful in making the notion of mental construction mathematically precise. For a comprehensive presentation of the theory, see Hindley and Seldin 1986 or Barendregt 1992.

The untyped lambda calculus was introduced in 1932 by the American logician Alonzo Church, originally intended as a foundation for mathematics. Soon after that it came to be used as one of the mathematical models for computability. John McCarthy took this calculus as a starting point when designing the programming language LISP in 1960. Later generations of functional programming languages, such as ML and Haskell, incorporated also strict typing of functions. These languages are closely related to *typed versions* of the lambda calculus.

We introduce and study a typed lambda calculus. First the notion of type is defined inductively.

Definition 2.1 *Types.*

(i) $\mathbb{N}$ is a type (the type of natural numbers).

(ii) If A and B are types, then $(A \times B)$ is a type (the product of A and B).

(iii) If A and B are types, then $(A \to B)$ is a type (the type of functions from A to B).

(iv) If A and B are types, then $(A + B)$ is a type (the disjoint union of A and B).

Examples of types are

$$(\mathbb{N} \to (\mathbb{N} \times \mathbb{N})) \qquad ((\mathbb{N} + (\mathbb{N} \times \mathbb{N})) \to \mathbb{N}).$$

To reduce the number of parentheses we use the convention that $\times$ binds stronger than $+$, which in turn binds stronger than $\to$. The above types are accordingly written

$$\mathbb{N} \to \mathbb{N} \times \mathbb{N} \qquad \mathbb{N} + \mathbb{N} \times \mathbb{N} \to \mathbb{N}.$$

Moreover, we use the convention that $\to$ associates to the right, so that $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ should be read $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ and not $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$.

Remark 2.2 The type constructions (i) – (iii) are familiar from ML, but are there denoted `nat`, `A * B` and `A -> B`, respectively. In Haskell the product of A and B is written `(A,B)`.

Next we define the terms, or "programs", of lambda calculus. The most important constructions are *abstraction* $\lambda x.b$, and *application,* $\mathsf{apply}(f,a)$. Abstraction forms from b a new function $\lambda x.b$ depending on a chosen variable. If b has the type B and x has the type A, then the new function has the type $A \to B$. In $\lambda x.b$ the variable x is bound. The application $\mathsf{apply}(f,a)$ applies a function f to an argument a. The connection between these constructions is given by the β-*rule*

$$\mathsf{apply}(\lambda x.b, a) = b[a/x], \tag{1}$$

where the right hand side means that a has been substituted for x in b. A condition for the validity of the rule is that no variables in a become bound by performing the substitution. As for first-order logic, the name of the bound variables are inessential. This is expressed by the α-rule:

$$\lambda x.b = \lambda y.b[y/x], \tag{2}$$

where y is a variable not occurring in b. By employing this rule we can ensure that the β-rule is applicable. In ML $\lambda x.b$ is written as `(fn x => b)`, and in Haskell this is `\x -> b`.

Another important construction is *pairing:* if a has type A and b has type B, we may form the pair of these $\langle a,b\rangle$, which is of type $A \times B$. There are two projections, or selectors, that extract the first and second component from this pair

$$\#_1(\langle a,b\rangle) = a, \qquad \#_2(\langle a,b\rangle) = b. \tag{3}$$

The third type construction is the disjoint union $A+B$ of two types. It is intuitively the union of A and B, but with a marking of the elements, that indicates from which of the types the element comes, the left or right type. (This may be needed if $A = B$.) The type has two constructors inl and inr. If a has type A, then $\mathsf{inl}(a)$ has type $A+B$. If b has type B, then $\mathsf{inr}(b)$ has type $A+B$. To express that each element has one of these two forms $\mathsf{inl}(a)$ or $\mathsf{inr}(b)$, we introduce a case construction, or a discriminator, when. If f has type $A \to C$ and g has type $B \to C$, then $\mathsf{when}(d,f,g)$ is of type C. It obeys the computation rules

$$\mathsf{when}(\mathsf{inl}(a),f,g) = \mathsf{apply}(f,a), \qquad \mathsf{when}(\mathsf{inr}(b),f,g) = \mathsf{apply}(g,b). \tag{4}$$

In ML there is no primitive type construction which behaves as $+$. However we may define the type construction as

```
datatype ('a,'b) sum = inl of 'a | inr of 'b
```

Whereas in Haskell this would be

```
data Sum a b = Inl a | Inr b
```

The natural numbers are most easily represented using unary notation: 0 has type $\mathbb{N}$; if n has type $\mathbb{N}$, then its successor, $\mathsf{S}(n)$, is of the same type $\mathbb{N}$. The number 3 is for instance represented by the term $\mathsf{S}(\mathsf{S}(\mathsf{S}(0)))$. Such a term is called a *numeral* and is written in $\overline{3}$ in brief form. We introduce an operator rec which allows functions to be defined by a generalised form of primitive recursion, or structural recursion. For f of type A and g of type $\mathbb{N} \to A \to A$ the following equations are valid

$$\mathsf{rec}(0, f, g) = f, \tag{5}$$
$$\mathsf{rec}(\mathsf{S}(n), f, g) = g(n)(\mathsf{rec}(n, f, g)). \tag{6}$$

Next the terms are defined formally. We use the abbreviation $a : A$ for a being a term of type A.

Definition 2.3 *Lambda terms.*

For each type A there are infinitely many variables $x^A, y^A, z^A, \ldots$ of that type.
If $f : A \to B$ and $a : A$, then $\mathsf{apply}(f, a) : B$.
If x^A is a variable and $b : B$, then $\lambda x^A.b : A \to B$, and the variable x^A is bound.
If $a : A$ and $b : B$, then $\langle a, b \rangle : A \times B$.
If $c : A \times B$, then $\#_1(c) : A$ and $\#_2(c) : B$.
If $a : A$ and B is a type, then $\mathsf{inl}(a) : A + B$.
If $b : B$ and A is a type, then $\mathsf{inr}(b) : A + B$.
If $d : A + B$, $f : A \to C$ and $g : B \to C$, then $\mathsf{when}(d, f, g) : C$.
If $a : \mathbb{N}$, $f : A$ and $g : \mathbb{N} \to A \to A$, then $\mathsf{rec}(a, f, g) : A$.

We shall often omit type information from variables, or other terms, when it may be inferred from the context. The expression $\mathsf{apply}(f, a)$ is abbreviated $f(a)$. The abtraction $\lambda x.$ binds most loosely, whence the expression $\lambda x. f(a)$ is read $\lambda x.\mathsf{apply}(f, a)$, while $\mathsf{apply}(\lambda x. f, a)$ is written $(\lambda x. f)(a)$.

Lambda calculus is an *equational theory*, i.e. a theory where only equalities between terms are dealt with. Then logical formulae are thus expressions of the form $s = t$, where s and t are terms of the same type. As axioms we assume all instances of the equalities (1), (2), (3), (4), (5), (6) and $t = t$. The rules of derivation are

$$\frac{t_1 = t_2}{t_2 = t_1} \qquad \frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}$$

$$\frac{t_1 = t_2}{t_1[t/x] = t_2[t/x]} \qquad \frac{t_1 = t_2}{t[t_1/y] = t[t_2/y]} \qquad \frac{t_1 = t_2}{\lambda x^A.t_1 = \lambda x^A.t_2}$$

whose only purpose is to make calculations possible in subexpression. Using these one may calculate as expected.

Example 2.4 *Composition of functions.* Let

$$\mathsf{comp} =_{\text{def}} \lambda f^{B\to C}.\lambda g^{A\to B}.\lambda x^A.f(g(x)).$$

This term has type $(B \to C) \to (A \to B) \to A \to C$. We have $\mathsf{comp}(f)(g)(x) = f(g(x))$.

Example 2.5 The predecessor function $\mathsf{pd} : \mathbb{N} \to \mathbb{N}$ is defined by

$$\mathsf{pd} =_{\text{def}} \lambda x.\mathsf{rec}(x, 0, \lambda n.\lambda y.n).$$

We have

$$\mathsf{pd}(0) = \mathsf{rec}(0, 0, \lambda n.\lambda y.n) = 0$$

and

$$\begin{aligned} \mathsf{pd}(\mathsf{S}(x)) &= \mathsf{rec}(\mathsf{S}(x), 0, \lambda n.\lambda y.n) \\ &= (\lambda n.\lambda y.n)(x)(\mathsf{rec}(x, 0, \lambda n.\lambda y.n)) \\ &= (\lambda y.x)(\mathsf{rec}(x, 0, \lambda n.\lambda y.n)) \\ &= x \end{aligned}$$

Example 2.6 *Multiplication by 2.* Define

$$\mathsf{double} =_{\text{def}} \lambda x.\mathsf{rec}(x, 0, \lambda n.\lambda y.\mathsf{S}(\mathsf{S}(y))).$$

We have $\mathsf{double}(0) = 0$ and $\mathsf{double}(\mathsf{S}(x)) = \mathsf{S}(\mathsf{S}(\mathsf{double}(x)))$.

Example 2.7 *Representation of integers.* Let $\mathbb{Z} = \mathbb{N} + \mathbb{N}$ and let $\mathsf{inl}(m)$ symbolise the negative number $-(m+1)$ while $\mathsf{inr}(n)$ symbolises the non-negative number n. One easily checks that this term changes sign of an integer:

$$\mathsf{neg} =_{\text{def}} \lambda z.\mathsf{when}(z, \lambda u.\mathsf{inr}(\mathsf{S}(u)), \lambda v.\mathsf{rec}(v, \mathsf{inr}(0), \lambda n.\lambda y.\mathsf{inl}(n))).$$

That is we have $\mathsf{neg}(\mathsf{inl}(m)) = \mathsf{inr}(\mathsf{S}(m))$, $\mathsf{neg}(\mathsf{inr}(0)) = \mathsf{inr}(0)$ and $\mathsf{neg}(\mathsf{inr}(\mathsf{S}(m))) = \mathsf{inl}(m)$.

Example 2.8 Define a lambda term I of type $(A \to A) \to \mathbb{N} \to (A \to A)$ which has the property that if applied to a numeral $\overline{n}$, as follows, then

$$I(f)(\overline{n}) = \lambda x.f(f(\cdots f(x)\cdots))).$$

Here the number of f's in the right hand side is n. Observe that the right hand side corresponds to the composition of f with itself n times. To achieve this we use the recursion operator

$$I =_{\text{def}} \lambda f.\lambda n.\mathsf{rec}(n, \lambda x.x, \lambda m.\lambda g.\mathsf{comp}(f,g)).$$

Remark 2.9 It is natural to regard the equalities $(=)$ as computation relations, or reduction relations, directed from left to right, such that for example $\#_1(\langle a,b\rangle) = a$ is read $\#_1(\langle a,b\rangle)$ *computes to* a. In this manner we may regard the typed lambda calculus as a programming language. It has the unusual property that *all programs terminate*. To prove this formally is very complicated, see for instance Hindley and Seldin 1986, for a proof concerning a simplified calculus. The fact that all programs terminate implies that a programming language is not Turing complete, i.e. there is a Turing machine computable function which cannot not be represented as a lambda term. (Cf. Exercise 2.5 below.) However, the calculus contains all computable functions that can be *proved to terminate* in the first order theory of Peano arithmetic.

Remark 2.10 In *pure untyped lambda calculus* the only programming constructions are abstraction and application. It has, as the name indicates, no types so each term may be applied to any term whatsoever, including the term itself. For example, $(\lambda x.x(x))(\lambda x.x(x))$ is a well-formed term. Employing the β-rule the same term is obtained in one computation step! This means that there are non-terminating programs in the calculus. Despite its extreme simplicity the untyped calculus is Turing complete. It is however not easy to demonstrate this, and its was first done by Turing in 1936, building on results of S.C. Kleene.

Exercises

2.1 Verify the equalities in Examples 2.6, 2.7 and 2.8.

2.2 Construct a lambda term $e : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that

$$\begin{aligned} e(0)(0) &= \mathsf{S}(0) \\ e(0)(\mathsf{S}(y)) &= 0 \\ e(\mathsf{S}(x))(0) &= 0 \\ e(\mathsf{S}(x))(\mathsf{S}(y)) &= e(x)(y). \end{aligned}$$

This is thus a function which may be used to decide whether two natural numbers are equal.

2.3 Show that each primitive recursive function may be simulated by a lambda term.

2.4 It is known that the Ackermann function $a : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, defined by

$$\begin{aligned} a(0,n) &= \mathsf{S}(n) \\ a(\mathsf{S}(m),0) &= a(m,\mathsf{S}(0)) \\ a(\mathsf{S}(m),\mathsf{S}(n)) &= a(m,a(\mathsf{S}(m),n)), \end{aligned}$$

grows to quickly for being a primitive recursive function. Prove that it nevertheless may be defined in the typed lambda calculus with the help of the recursion operator rec. [Hint 1: expand the definition of $a(\mathsf{S}(m),n)$ in the third line of the definition. Hint 2: define a function $b : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $a(m,n) = b(m)(n)$. Example 2.8 may also be useful.]

2.5 (a) Let $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and let $g(n) = f(n,n)+1$. Prove that there is no m such that $f(m,n) = g(n)$ for all n. This is the archetypical diagonialisation argument.

(b) Prove that g is computable, if f is computable.

(c) Suppose that the lambda terms contain sufficient information to determined their type in a unique way. This may be done by furnishing the operators $\mathsf{apply}, \mathsf{when}, \mathsf{rec}, \mathsf{inl}$ and inr with type information: $\mathsf{apply}_{A,B}, \mathsf{when}_{A,B,C}, \mathsf{rec}_A, \mathsf{inl}_{A,B}$ and $\mathsf{inr}_{A,B}$ (see Definition 2.3). Argue informally that the following function is totally defined and computable

$$f(m,n) = \begin{cases} k & \text{if } m \text{ in base 2 is a character-code for a lambda term } t \\ & \text{of type } \mathbb{N} \to \mathbb{N}, \text{ and } k \text{ is the value of } t(\overline{n}). \\ 0 & \text{otherwise.} \end{cases}$$

Use the fact that $t(x)$ always terminate with a numeral as value when x is a numeral. (We presuppose that special symbols $\lambda, \ldots$ are character coded in some appropriate way.)

(d) Show using (a) – (c) that there is a computable function which cannot be computed by a term in the typed lambda calculus (presented in this chapter).

3 Constructive interpretation of the logical constants

According to Brouwer's idea every mathematical theorem A must rest on a (mental) construction a. The construction a may be regarded as a *witness* to the truth

of A. We shall here use the lambda terms which were introduced in Chapter 2 as such constructions. This basic constructive interpretation was further clarified by A. Heyting (a student of Brouwer) and by A.N. Kolmogorov (the founder of modern probability theory). Hence it is called the *Brouwer-Heyting-Kolmogorov-interpretation,* or BHK-interpretation for short.

It should be pointed out that the class of constructions is not limited to the lambda terms that we have introduced so far, but may be extended when further construction methods are discovered (cf. Remark 3.4). There are some limits. One may be lead to think that this is a construction

$$f(n) = \begin{cases} 1 & \text{there are } n \text{ consecutive 7's in the decimal expansion of } \pi, \\ 0 & \text{otherwise.} \end{cases}$$

This is, a priori, not a constructive function as no one has (yet) found an algorithm that can decide whether there are n consecutive 7's in the decimal expansion of π or not. Case distinction is allowed only if it can be decided effectively which case is true, for given parameters.

BHK-interpretation. We explain what it means that a is a *witness* to the truth of the proposition A, by induction on the form of A. This will be expressed more briefly as *a is a witness to A*, or that *a testifies A*.

- $\bot$ has no witnesses.
- p testifies $s = t$ iff $p = 0$ and s and t are computed to the same thing. (Here s and t are supposed to be natural numbers, or some similar finitely given mathematical objects.)
- p testifies $A \wedge B$ iff p is a pair $\langle a, b \rangle$ where a testifies A and b testifies B.
- p testifies $A \to B$ iff p is a function which to each witness a to A gives a witness $p(a)$ to B.
- p testifies $A \vee B$ iff p has the form $\mathsf{inl}(a)$, in which case a testifies A, or p has the form $\mathsf{inr}(b)$, in which case b testifies B
- p testifies $(\forall x \in S)A(x)$ iff p is a function which to each element $d \in S$, provides a witness $p(d)$ to $A(d)$.
- p testifies $(\exists x \in S)A(x)$ iff p is a pair $\langle d, q \rangle$ consisting of $d \in S$ and a witness q to $A(d)$.

A proposition A is *valid under the BHK-interpretation,* or is *constructively true,* if there is a construction p such that p testifies A.

Note that this use of the word "witness" extends its usage in classical logic about existence statements. One may say that 2 is a witness to $(\exists x)\, x^2 = 4$ being true. The established standard terminology is rather to say that p is a *proof* for A, and the construction p is called *proof-object.* But to avoid confusion with formal derivations we use here, for pedagogical reasons, the word *witness*.

Here follows some examples of BHK-interpretations. We use constructions from typed lambda calculus.

Examples 3.1 1. $p = \lambda x.x$ is a witness to the truth of $A \to A$. This is clear, since $p(a) = (\lambda x.x)(a) = a$ and if a testifies A, then so does $p(a)$.

2. A witness to $A \wedge B \to B \wedge A$ is given by the construction $f = \lambda x.\langle \#_2(x), \#_1(x)\rangle$.

3. Consider the proposition $\bot \to A$. A witness to this is an arbitrary function f such as $f(x) = 42$: Suppose that a is a witness to $\bot$. But according to the BHK-interpretation $\bot$ has no witness, so we have a contradiction. By the absurdity law, anything follows, in particular that 42 is a witness to A.

Negation is *defined* as $\neg A =_{\text{def}} (A \to \bot)$. To prove $\neg A$ amounts to proving that A leads to a contradiction. As usual we define $A \leftrightarrow B$ to be $(A \to B) \wedge (B \to A)$.

Example 3.2 The contraposition law $(A \to B) \to (\neg B \to \neg A)$ is valid in under the BHK-interpretation. Suppose that f testifies $A \to B$. We wish to find a witness to $(\neg B \to \neg A)$, i.e. $(B \to \bot) \to (A \to \bot)$. Suppose therefore that g testifies $\neg B$ and a testifies A. Thereby $f(a)$ is a witness to B, and hence $g(f(a))$ is a witness to $\bot$. The construction $\lambda a.g(f(a))$ thus testifies $\neg A$. Abstracting on g it is clear that $\lambda g.\lambda a.g(f(a))$ testifies $\neg B \to \neg A$. The construction

$$\lambda f.\lambda g.\lambda a.g(f(a))$$

is finally the witness to the law of contraposition. □

Suppose that we have a witness p to the truth of the proposition

$$(\forall x \in S)\,(\exists y \in T)\, A(x,y). \tag{7}$$

Then we have for each $a \in S$ that $p(a)$ testifies $(\exists y \in T)\, A(a,y)$. But $p(a)$ is a pair $\langle c, q\rangle$ where $c \in T$ and q testifies $A(a,c)$. It follows that $\#_2(p(a))$ testifies $A(a, \#_1(p(a)))$. Hence $f(x) = \#_1(p(x))$ defines a function such that $\lambda x.\#_2(p(x))$ testifies $(\forall x \in S) A(x, f(x))$. This gives a method for computing y from x.

A proposition of the form (7) may for instance be a specification of a program, where S is the type of input data, T is the type of output data, and $A(x,y)$ describes the desired relation between input and output. The witness p now gives a program f which satisfies the specification A.

Remark 3.3 *The Principle of Excluded Middle* (PEM)

$$A \vee \neg A$$

is not obviously valid under the BHK-interpretation, since we would need to find a method, which given the parameters in A, decides whether A is valid or not. If we restrict the possible constructions to computable functions, we may actually show that PEM is not constructively true. It is known that there is a primitive recursive function T such that $T(e,x,t) = 1$ in case t describes a terminating computation (t is, so to say, the complete "trace" of the computation) for the Turing machine e with input x, and having the value $T(e,x,t) = 0$ otherwise. By a suitable coding, the arguments to T may be regarded as natural numbers. The halting problem for e and x may now be expressed by the formula

$$H(e,x) =_{\text{def}} (\exists t \in \mathbb{N})\, T(e,x,t) = 1.$$

According to PEM

$$(\forall e \in \mathbb{N})(\forall x \in \mathbb{N})\, H(e,x) \vee \neg H(e,x).$$

If this proposition were to have a computable witness, then we could decide the halting problem, contrary to Turing's well-known result that this is algorithmically undecidable. The principle of indirect proof, *reductio ad absurdum* (RAA)

$$\neg\neg A \rightarrow A$$

can be shown to be equivalent to PEM within intuitionistic logic, so it is not valid under the BHK-interpretation either. □

We have seen that a witness to the truth of a proposition may be regarded as a program, by letting the constructions be lambda terms. In the following chapter we show how proofs of a proposition A carried out following the rules of *intuitionistic logic* always gives rise to a witness a to A,

The second leading idea, to consider propositions as data types, is realised in the following limited sense. If the witnesses to the truth of proposition A have type S, the witnesses to B have type T, and the witnesses to $C(x)$ have type U, then

- the witnesses to $A \wedge B$ have type $S \times T$,
- the witnesses to $A \rightarrow B$ have type $S \rightarrow T$
- the witnesses to $A \vee B$ have type $S + T$,
- the witnesses to $(\forall x \in \mathbb{N})C(x)$ have type $\mathbb{N} \rightarrow U$,

- the witnesses to $(\exists x \in \mathbb{N})C(x)$ have type $\mathbb{N} \times U$

The witnesses to $s = t$ have type $\mathbb{N}$. Since $\perp$ has no witnesses, we could formally let these have type $\mathbb{N}$ as well. Conceptually there are good reasons to introduce a special *empty type*. Extend Definition 2.1 with

$\emptyset$ is type.

The empty type $\emptyset$ has no elements. The construction $!_A : \emptyset \to A$ is used to express that $\emptyset$ is at least as empty as any other type, namely, if $\emptyset$ has an element c, then every other type A has an element $!_A(c)$.

According to the above it is clear that a type S corresponding to a proposition A may contain terms that are not witnesses to A. For instance, the proposition $A_1 = (\forall n \in \mathbb{N})\, n = n^2$ has corresponding type $\mathbb{N} \to \mathbb{N}$. It contains the term $\lambda x.0$, which is not a witness to A_1.

It would be desirable to be able to identify propositions with data types in such way that a proposition A has a witness if, and only if, it is non-empty regarded as a data type. The logician H.B. Curry realised that for (intuitionistic) propositional logic a formula could be made to correspond exactly with a type by introducing type variables $X, Y, \ldots$ that may stand for arbitrary types, empty or non-empty; see Curry and Feys 1958. W.A. Howard extended, in 1969, Curry's idea to universally quantified propositions. The propositions-as-types principle is sometimes called the *Curry-Howard isomorphism* (in particular, if further relations hold between the reduction rules of the proofs and the programs; see Simmons 2000). A fully fledged version of the propositions-as-types principle came only with the introduction in 1971 of the type theory of Per Martin-Löf. In this and later work it emerged how this principle could be generalised to many other logical construction, even those belonging to higher set theory.

For propositions containing individual variables the associated types becomes much more complicated. In the example A_1 above the proposition $n = n^2$ should be a empty type $S_n = \emptyset$ for $n \geq 2$ and a type containing 0, for instance $S_n = \{0\}$, when $n = 0, 1$. This requires so called *dependent types*. The type S_n depends on $n \in \mathbb{N}$, an element in another type. We return to type theory in Chapter 7.

Remark 3.4 It is also possible to use untyped terms as constructions. An important method is Kleene's recursive realisability interpretation, where the constructions are indices (codes) for partial recursive functions (cf. Troelstra och van Dalen 1988). These indices may be regarded as programs for Turing machines. It may be argued that construction methods beyond these are necessary if one accepts the Church-Turing thesis on computability. While the constructions in typed lambda calculus, or Martin-Löf type theory, may be understood directly and independently of other theories, the properties of the untyped constructions need to justified within some meta-theory.

Exercises

3.1 Prove that the following propositions are valid under the BHK-interpretation

(a) $A \wedge B \to A$ and $A \wedge B \to B$

(b) $A \to (B \to A \wedge B)$

(c) $A \to A \vee B$ and $B \to A \vee B$

(d) $(A \to C) \to ((B \to C) \to (A \vee B \to C))$

(e) $A \to (B \to A)$

(f) $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

(g) $\bot \to A$

(h) $A(t) \to (\exists x \in D)A(x)$

(i) $(\forall x \in D)(A(x) \to B) \to ((\exists x \in D)A(x) \to B))$, where x does not occur free in B

(j) $(\forall x \in D)A(x) \to A(t)$

(k) $(\forall x \in D)(B \to A(x)) \to (B \to (\forall x \in D)A(x))$, where x does not occur free in B.

These are the logical axioms for (first-order) intuitionistic logic. In addition there are the modus ponens rule, and the generalisation rule: from A, derive $(\forall x \in D)A$.

3.2 Show that the following are constructively true:

(a) $\neg\neg\neg A \to \neg A$

(b) $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$.

(c) $\neg(\exists x \in D)A(x) \leftrightarrow (\forall x \in D)\neg A(x)$

3.3 Does any of the following (classical valid) propositions have constructive witnesses? Discuss!

(a) $\neg(\forall x \in D)A(x) \to (\exists x \in D)\neg A(x)$.

(b) $\neg(A \wedge B) \to \neg A \vee \neg B$.

4 Intuitionistic logic

We assume that the reader is familiar with some system of natural deduction for predicate logic; see van Dalen 1997, Huth and Ryan 2000 or Hansen 1997. Here is one such system based on derivation trees.

Derivation rules:

$$\frac{A \qquad B}{A \wedge B}\,(\wedge I) \qquad \frac{A \wedge B}{A}\,(\wedge E1) \qquad \frac{A \wedge B}{B}\,(\wedge E2)$$

$$\frac{\begin{array}{c}\overline{A}^{h}\\ \vdots\\ B\end{array}}{A \to B}(\to I,h) \qquad \frac{A \to B \qquad A}{B}\,(\to E)$$

$$\frac{A}{A \vee B}\,(\vee I1) \qquad \frac{B}{A \vee B}\,(\vee I2) \qquad \frac{A \vee B \quad \begin{array}{c}\overline{A}^{h_1}\\ \vdots\\ C\end{array} \quad \begin{array}{c}\overline{B}^{h_2}\\ \vdots\\ C\end{array}}{C}\,(\vee E,h_1,h_2)$$

$$\frac{\bot}{A}\,(\bot E) \qquad \frac{\begin{array}{c}\overline{\neg A}^{h}\\ \vdots\\ \bot\end{array}}{A}(RAA,h)$$

This is the propositional part of the rules. Note that a discharged assumption A is denoted $\overline{A}^{h}$ in the derivation, where h is a symbol that identifies those assumptions which are discharged at the same time. The symbol is placed also at the rule which does the discharging, and has to be unique.

$$\frac{A}{(\forall x)A}\,(\forall I) \qquad \frac{(\forall x)A}{A[t/x]}\,(\forall E)$$

$$\frac{A[t/x]}{(\exists x)A}\,(\exists I) \qquad \frac{(\exists x)A \quad \begin{array}{c}\overline{A}^{h}\\ \vdots\\ C\end{array}}{C}\,(\exists E,h)$$

The rules for the quantifiers have certain restrictions. For $(\forall I)$ the condition is that x may not be free in undischarged assumptions above A. This blocks a derivation of

$$x = 0 \rightarrow (\forall x)x = 0. \tag{8}$$

For $(\exists E)$ the restriction is that x may not be free in C or in undischarged assumptions other than those marked with h. This blocks derivations of

$$(\exists x)A \rightarrow (\forall x)A. \tag{9}$$

The above rules constitute a system for *classical predicate logic* (with equality). If the rule RAA is omitted, we get a system for *intuitionistic predicate logic.*

Validity under the BHK-interpretation. We show that the rules of the *intuitionistic* subsystem are valid under the BHK-interpretation, that is, if we have witnesses for the propositions above derivation bar then we can construct a witness for the proposition below the derivation bar. The notation $a : A$ is used to express that a is a witness to A. We presuppose that the quantification domain is D. To make an assumption A is interpreted as making the assumption that a variable x is a witness to A, i.e. $x : A$.

$$\frac{a : A \qquad b : B}{\langle a, b\rangle : A \wedge B}\,(\wedge I) \qquad \frac{c : A \wedge B}{\#_1(c) : A}\,(\wedge E1) \qquad \frac{c : A \wedge B}{\#_2(c) : B}\,(\wedge E2)$$

$$\frac{\begin{array}{c}\overline{x : A}^{\,h}\\ \vdots\\ b : B\end{array}}{\lambda x.b : A \rightarrow B}(\rightarrow I, h) \qquad \frac{c : A \rightarrow B \qquad a : A}{\mathsf{apply}(c, a) : B}\,(\rightarrow E)$$

$$\frac{a : A}{\mathsf{inl}(a) : A \vee B}\,(\vee I1) \qquad \frac{b : B}{\mathsf{inr}(b) : A \vee B}\,(\vee I2)$$

$$\frac{\begin{array}{ccc} & \overline{x : A}^{\,h_1} & \overline{y : B}^{\,h_2}\\ & \vdots & \vdots\\ c : A \vee B & d : C & e : C\end{array}}{\mathsf{when}(c, \lambda x.d, \lambda y.e) : C}\,(\vee E, h_1, h_2)$$

$$\frac{c : \bot}{!(c) : A}\ (\bot E)$$

$$\frac{a : A}{\lambda x.a : (\forall x)A}\ (\forall I) \qquad \frac{c : (\forall x)A}{\mathsf{apply}(c,t) : A[t/x]}\ (\forall E)$$

$$\frac{a : A[t/x]}{\langle t, a\rangle : (\exists x)A}\ (\exists I) \qquad \frac{c : (\exists x)A \quad \begin{array}{c}\overline{y : A}^{\,h}\\ \vdots \\ d : C\end{array}}{d[\#_1(c), \#_2(c)/x, y] : C}\ (\exists E, h)$$

We have shown

Theorem 4.1 *The rules for intuitionistic logic are valid under the BHK-interpretation.*

Example 4.2 We derive the contraposition law in intuitionistic logic

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\neg B}^{\,h_2} \quad \dfrac{\overline{A \to B}^{\,h_3} \quad \overline{A}^{\,h_1}}{B}\,(\to E)}{\bot}\,(\to E)}{\neg A}\,(\to I, h_1)}{\neg B \to \neg A}\,(\to I, h_2)}{(A \to B) \to (\neg B \to \neg A)}\,(\to I, h_3)$$

and then do the BHK-interpretation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{g : \neg B}^{\,h_2} \quad \dfrac{\overline{f : A \to B}^{\,h_3} \quad \overline{a : A}^{\,h_1}}{f(a) : B}\,(\to E)}{g(f(a)) : \bot}\,(\to E)}{\lambda a.g(f(a)) : \neg A}\,(\to I, h_1)}{\lambda g.\lambda a.g(f(a)) : \neg B \to \neg A}\,(\to I, h_2)}{\lambda f.\lambda g.\lambda a.g(f(a)) : (A \to B) \to (\neg B \to \neg A)}\,(\to I, h_3)$$

Example 4.3 We derive and then make a BHK-interpretation of $(\exists x)\,(A \wedge B) \to A \wedge (\exists x)\,B$. Here x is not free in A.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z:(\exists x)(A\wedge B)}\,h_2 \quad \dfrac{\dfrac{\dfrac{}{y:A\wedge B}\,h_1}{\#_1(y):A}\,(\wedge E1) \quad \dfrac{\dfrac{\dfrac{}{y:A\wedge B}\,h_1}{\#_2(y):B}\,(\wedge E2)}{\langle x,\#_2(y)\rangle:(\exists x)B}\,(\exists I)}{\langle \#_1(y),\langle x,\#_2(y)\rangle\rangle : A\wedge(\exists x)B}\,(\wedge I)}{\langle \#_1(\#_2(z)),\langle \#_1(z),\#_2(\#_2(z))\rangle\rangle : A\wedge(\exists x)B}\,(\exists E,h_1)}{\lambda z.\langle \#_1(\#_2(z)),\langle \#_1(z),\#_2(\#_2(z))\rangle\rangle : (\exists x)(A\wedge B)\to A\wedge(\exists x)B}\,(\to I,h_2)$$

(What does the witness on the last line do?)

Remark 4.4 * To indicate what open assumptions there are available at a given position in a proof tree one uses *sequent notation.* The expression

$$A_1^{h_1},\ldots,A_n^{h_n}\vdash B$$

states that B is proved under the open assumptions $A_1,\ldots,A_n$. The order of these assumption does not matter, and the same formula may occur several times, but the markers $h_1,\ldots,h_n$ have to be distinct. See Troelstra and Schwichtenberg 1996 for a formulation of natural deduction using this notation. Under the BHK-interpretation the marker becomes superfluous and may be replaced by variables

$$x_1:A_1,\ldots,x_n:A_n\vdash b:B. \tag{10}$$

Model theory of intuitionistic logic.* We have seen that the intuitionistic derivation rules are valid under the BHK-interpretation. However, for this kind of semantics there is no completeness theorem (van Dalen and Troelstra 1988). Intuitionistic logic is complete for Beth-Kripke-Joyal semantics, see van Dalen 1997 for an introduction. This semantics has the same function as Tarski semantics for classical logic and is very useful for demonstrating that a logical formula is unprovable in intuitionistic logic. (For instance one may easily give negative solutions to Exercise 3.3.)

Non-logical axioms: induction. The BHK-semantics has only been applied to pure logic above. It is also possible to verify some non-logical axioms to be valid, for instance the induction scheme for natural numbers. The scheme may also be stated as an elimination rule. We now assume that the domain of quantification is $\mathbb{N}$. The induction rule

$$\dfrac{\begin{array}{c}\\ \\ A(0) \qquad A(\mathsf{S}(x))\end{array} \quad \begin{array}{c}\overline{A(x)}^{h}\\ \vdots \\ A(t)\end{array}}{} \ (\text{induction}, h)$$

may easily be given a BHK-interpretation with the help of the recursion operator:

$$\dfrac{\begin{array}{c}\\ \\ b : A(0) \qquad c : A(\mathsf{S}(x))\end{array} \quad \begin{array}{c}\overline{y : A(x)}^{h}\\ \vdots \\ \mathsf{rec}(t, b, \lambda x.\lambda y.c) : A(t)\end{array}}{} \ (\text{induction}, h).$$

Exercises

4.1 Deduce 3.1 (a)–(k) in intuitionistic logic. (Use the soundness theorem and compare to the constructions you obtained in Exercise 3.1.)

4.2 Deduce 3.2 in intuitionistic logic.

4.3* Prove $\neg\neg\neg A \leftrightarrow \neg A$ in intuitionistic logic.

4.4 Show that the false formulae (8) and (9) are provable, if the restrictions on quantifiers rules are removed.

5 Brouwerian counter examples

In this section we present a common way to show that a proposition is constructively unprovable. There are certain simple logical principles, that are trivially true in classical logic, but which have no constructive proofs. These are the *principles of omniscience* concerning infinite binary sequences

$$1\,0\,0\,1\,1\,0\,0\,0\,1\,0\,\cdots.$$

An *(infinite) binary sequence* is formally a function $\alpha : \mathbb{N} \to \{0,1\}$. We consider here two such principles, the *limited principle of omniscience* and the *lesser limited principle of omniscience*

(LPO) For all binary sequences α either $\exists n\, \alpha_n = 1$ or $\forall n\, \alpha_n = 0$.

(LLPO) Let α be a binary sequence with at most one occurrence of 1. Then $\forall n\, \alpha_{2n} = 0$ or $\forall n\, \alpha_{2n+1} = 0$.

It may easily be shown, constructively, that

$$\text{LPO} \Rightarrow \text{LLPO}.$$

In a recursive realisability interpretation LLPO is false, and hence so is LPO (Troelstra and van Dalen 1988). One may also convince oneself by intuitive considerations that LLPO does not have a constructive proof. Let α be the binary sequence defined by $\alpha_n = 1$ if n is the first position in the decimal expansion of π commencing a run of 100 consecutive 7s, and $\alpha_n = 0$ in other cases. A constructive proof of LLPO would immediately give a method for deciding whether the run is never started at an odd position, or is never started at an even position.

If P is a theorem in classical mathematics whose constructive truth we doubt, we may try to show that P implies of one the principles LPO or LLPO. In case we succeed in this, we should give up looking for a constructive proof of P.

Exercises

5.1 Establish the implication LPO $\Rightarrow$ LLPO in intuitionistic logic, in particular without using PEM or RAA.

5.2 Prove using intuitionistic logic that Theorem 1.2 implies LPO.

6 Classical and intuitionstic proofs

We have seen that proofs in intuitionistic logic have the remarkable property that programs may be extracted from them. Most naturally occurring proofs, in mathematical text books or journals, rely as they stand on some use of classical logic. A natural question is whether there is some mechanical method for translating classical proofs to constructive proofs. It is clear that such methods must have some limitations in view of the counter examples of the previous chapters. Kurt Gödel and Gerhard Gentzen showed that there is a method for purely logical proofs, and certain simple theories, if the proposition proved A may be substituted by a classically equivalent proposition A^*. This substitute may not have the same meaning from a constructive point of view.

In classical logic, the logical constants (connectives and quantifiers) $\exists$ and $\vee$ are actually unnecessary, since we have the following provable equivalences

$$A \vee B \quad \leftrightarrow \quad \neg(\neg A \wedge \neg B) \tag{11}$$

$$(\exists x)C \quad \leftrightarrow \quad \neg(\forall x)\neg C \tag{12}$$

for arbitrary formulae A, B, C. The fact is that if we take the right hand sides as definitions of the corresponding logical constants, and only use RAA, $(\bot E)$ and the introduction and elimination rules for $\wedge, \rightarrow$ and $\forall$, then we get a complete system for classical predicate logic (see e.g. van Dalen 1997). A formula of predicate logic where the only logical constants used are $\bot, \wedge, \rightarrow$ and $\forall$, is called a *non-existential formula.* (We think of $A \vee B$ as a form of existence statement.)

Define the *Gödel-Gentzen negative translation* $(\cdot)^*$ by recursion on non-existential formulae:

- $\bot^* = \bot$,
- $R(t_1, \ldots, t_n)^* = \neg\neg R(t_1, \ldots, t_n)$, if R is a predicate symbol,
- $(A \wedge B)^* = A^* \wedge B^*$,
- $(A \rightarrow B)^* = A^* \rightarrow B^*$,
- $((\forall x)C)^* = (\forall x)C^*$.

It should be clear that the only thing this translation achieves is to insert two negation sign in front of every predicate symbol. Obviously, A is provably equivalent to A^* using RAA.

Example 6.1 Let R be a binary predicate symbol. The formula

$$A = (\forall x)(\exists y)(R(x,y) \vee R(y,x))$$

is classically equivalent to $B = (\forall x)\neg(\forall y)\neg\neg(\neg R(x,y) \wedge \neg R(y,x))$. The Gödel-Gentzen translation B^* is

$$(\forall x)\neg(\forall y)\neg\neg(\neg\neg\neg R(x,y) \wedge \neg\neg\neg R(y,x))$$

Theorem 6.2 *Let A be a non-existential formula. If A is provable in classical predicate logic, then A^* is provable in intuitionistic predicate logic.*

Proof. A formal proof goes by induction on derivations. Since the proof rules are identical for the systems, save for RAA, one needs only to prove

$$\neg\neg A^* \rightarrow A^* \tag{13}$$

in intuitionistic predicate logic, for each non-existential formula A. This is done by induction on the formula A, using the following theorems of intuitionistic logic.

$\vdash \neg\neg\bot \rightarrow \bot$,
$\vdash \neg\neg\neg\neg B \rightarrow \neg\neg B$,
$\neg\neg A \rightarrow A, \neg\neg B \rightarrow B \vdash \neg\neg(A \wedge B) \rightarrow A \wedge B$,
$\neg\neg B \rightarrow B \vdash \neg\neg(A \rightarrow B) \rightarrow (A \rightarrow B)$,
$(\forall x)(\neg\neg A \rightarrow A) \vdash \neg\neg(\forall x)A \rightarrow (\forall x)A$.

We leave their proofs as exercises for the reader. □

A non-existential formula A is called *negative* if every predicate symbol in A is immediately preceded by a negation. For such a formula every predicate symbol in the corresponding translation A^* will be preceded by three negations. Intuitionistically, it holds that $\neg\neg\neg B \leftrightarrow \neg B$. Consequently, every negative formula is equivalent to its own Gödel-Gentzen interpretation. We have

Corollary 6.3 *Let A be a negative, non-existential formula. If A is provable in classical predicate logic, then it is also provable in intuitionistic predicate logic.*

Theorem 6.2 may be extended to some theories T, such that if A is provable in classical predicate logic using axioms from T, then A^* is provable in intuitionistic predicate logic from the axioms of T. ($T = \emptyset$ is thus the theorem proved above.) An important example is $T = PA$, the first-order theory of natural numbers, known

as, *Peano arithmetic.* It has the following axioms:

$$
\begin{aligned}
&(\forall x)\, x = x\\
&(\forall x)(\forall y)[x = y \to y = x]\\
&(\forall x)(\forall y)(\forall z)[x = y \wedge y = z \to x = z]\\
&(\forall x)(\forall y)[x = y \to \mathsf{S}(x) = \mathsf{S}(y)]\\
&(\forall x)(\forall y)(\forall z)(\forall u)[x = z \wedge y = u \to x + y = z + u]\\
&(\forall x)(\forall y)(\forall z)(\forall u)[x = z \wedge y = u \to x \cdot y = z \cdot u]\\
&(\forall x)\, \neg \mathsf{S}(x) = 0\\
&(\forall x)(\forall y)[\mathsf{S}(x) = \mathsf{S}(y) \to x = y]\\
&(\forall x)\, x + 0 = x\\
&(\forall x)(\forall y)\, x + \mathsf{S}(y) = \mathsf{S}(x + y)\\
&(\forall x)\, x \cdot 0 = 0\\
&(\forall x)(\forall y)\, x \cdot \mathsf{S}(y) = x \cdot y + x\\
&A(0) \wedge (\forall x)[A(x) \to A(\mathsf{S}(x))] \to (\forall x) A(x).
\end{aligned}
$$

Here $A(x)$ is an arbitrary formula in the language $\{=, 0, \mathsf{S}, +, \cdot\}$.

For quantifier-free P we have an additional result: if

$$A = (\forall x)(\exists y) P(x, y)$$

is provable from *PA* using classical logic, then A is already provable using intuitionistic logic from the axioms (see Troelstra and van Dalen 1988). Since A has the format of a program specification, it is sometimes possible to use this, and similar results, to extract programs from classical proofs (see Schwichtenberg 1999).

Exercises

6.1. Let $P(x)$ be a predicate symbol and consider $A = (\exists x) P(x) \vee (\forall x) \neg P(x)$. Eliminate all occurrences of $\vee$ and $\exists$ with the help of (11). Then prove the Gödel-Gentzen translated formula in intuitionistic logic.

6.2. Prove the following in intuitionistic logic.

(a) $\vdash \neg\neg\bot \to \bot$,

(b) $\vdash \neg\neg\neg\neg B \to \neg\neg B$,

(c) $\neg\neg A \to A, \neg\neg B \to B \vdash \neg\neg(A \wedge B) \to A \wedge B$,

(d)* $\neg\neg B \to B \vdash \neg\neg(A \to B) \to (A \to B)$,

(e)* $(\forall x)(\neg\neg A \to A) \vdash \neg\neg(\forall x)A \to (\forall x)A$.

6.3.* Suppose that T is a theory such that,

(a) T proves $A \vee \neg A$, for every atomic A, using only intuitionistic logic,

(b) T proves B^* using intuitionistic logic, for every axiom B of T.

Then show that if a non-existential A is provable classically in T, then it is already provable intuitionistically.

6.4.* Prove that PA satisfies the conditions in 6.3. Hint: for (a) use induction on natural numbers.

7 Martin-Löf type theory

The type theory of Martin-Löf 1984 has several notions not present in simple lambda calculus, such as dependent types, Π-types and Σ-types. These were introduced to realise the full propositions-as-types principle.

7.1 Set-theoretic constructions: sums and products.

To explain $\Pi-$ and Σ-types we present some less common set-theoretic constructions. Let I be a set, and let A_i be a set for each $i \in I$. We say that A_i ($i \in I$) is a *family of sets*. The *disjoint union, or sum* of this family is a set of pairs

$$(\Sigma i \in I)A_i = \{\langle i, a\rangle \mid i \in I, a \in A_i\}.$$

(Two alternative notations: $\sum_{i\in I} A_i$ and $\dot{\cup}_{i\in I} A_i$ — note the dot.)

Example 7.1 Let $J = \{1,2,3\}$ and $B_1 = \{0\}$, $B_2 = \{0,1\}$, $B_3 = \{1,2\}$. Then

$$(\Sigma j \in J)B_j = \{\langle 1,0\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle\}.$$

Example 7.2 Let $C_n = \{m \in \mathbb{N} : m \leq n\}$, for $n \in \mathbb{N}$. Then

$$(\Sigma n \in \mathbb{N})C_n = \{\langle n, m\rangle \in \mathbb{N} \times \mathbb{N} \mid m \leq n\}.$$

Consider once more a family A_i ($i \in I$) of sets. The *cartesian product* of this family is a set of functions

$$(\Pi i \in I)A_i = \{ f : I \to \cup_{i \in I} A_i \mid (\forall i \in I)\, f(i) \in A_i \}$$

(Alternative symbolism: $\prod_{i \in I} A_i$.) This is thus the set of functions f defined on I, where for each $i \in I$ the value $f(i)$ belongs to A_i. The range, or codomain, A_i depend on the argument i. Therefore the construction is sometimes called *dependent function space*.

Example 7.3 Let B_j ($j \in J$) be a family of sets as in Example 7.1. Then $(\Pi j \in J)B_j$ consists of four different functions f, g, h, and k, where

$$\begin{array}{llll} f(1) = 0 & g(1) = 0 & h(1) = 0 & k(1) = 0 \\ f(2) = 0 & g(2) = 0 & h(2) = 1 & k(2) = 1 \\ f(3) = 1 & g(3) = 2 & h(3) = 1 & k(3) = 2 \end{array}$$

Example 7.4 Let C_n ($n \in \mathbb{N}$) be as in Example 7.2. Note that $\cup_{n \in \mathbb{N}} C_n = \mathbb{N}$. So $(\Pi \in \mathbb{N})C_n = \{ f : \mathbb{N} \to \mathbb{N} \mid (\forall n \in \mathbb{N})\, f(n) \in C_n \} = \{ f : \mathbb{N} \to \mathbb{N} \mid (\forall n \in \mathbb{N})\, f(n) \leq n \}$.

Remark 7.5 (Binary sums and products.) Let $I = \{0, 1\}$ and $C_0 = A$, $C_1 = B$. Then

$$(\Sigma i \in I)C_i = \{ \langle 0, a \rangle : a \in A \} \cup \{ \langle 1, b \rangle : b \in B \},$$

is a disjoint union of A and B. We denote this set as $A + B$. Furthermore

$$(\Pi i \in I)C_i = \{ f : I \to C_0 \cup C_1 \mid f(0) \in C_0, f(1) \in C_1 \}.$$

Since functions with a two-element domain $I = \{0, 1\}$ may be regarded as pairs, we see that $(\Pi i \in I)C_i$ is a binary cartesian product $A \times B$.

Remark 7.6 Suppose that $A_i = A$ for all $i \in I$. Then we obtain $(\Sigma i \in I)A_i = (I \times A)$ and $(\Pi i \in I)A_i = (I \to A)$. Therefore we may regard Σ and Π as generalisations of the constructions $\times$ and $\to$.

7.2 Propositions as sets

We now investigate how these constructions may be used to give interpretations of propositions as *sets*. Define a set $E_{m,n}$, depending on $m, n \in \mathbb{N}$, by:

$$E_{m,n} = \begin{cases} \{0\} & \text{if } m = n \\ \emptyset & \text{if } m \neq n. \end{cases}$$

Then $E_{m,n}$ is nonempty exactly when $m = n$.

Example 7.7 The proposition $(\exists n \in \mathbb{N})\, m = 2n$ is true if, and only if, m is an even natural number. Consider now the set $S_m = (\Sigma n \in \mathbb{N})\, E_{m,2n}$. The only possibility for it to be non-empty is that $(n, 0) \in S_m$ for some n, which is true when $m = 2n$. We get that

$$S_m \neq \emptyset \Leftrightarrow (\exists n \in \mathbb{N})\, m = 2n.$$

Example 7.8 The proposition $(\forall n \in \mathbb{N})(n+k)^2 = n^2 + 4n + 4$ is true if, and only if, $k = 2$. Form the set $T_k = (\Pi n \in \mathbb{N})\, E_{(n+k)^2, n^2+4n+4}$. This set is non-empty if, and only if, the function $f(n) = 0$ belongs to the set, i.e. if for all $n \in \mathbb{N}$: $E_{(n+k)^2, n^2+4n+4} = \{0\}$, i.e. $(n+k)^2 = n^2 + 4n + 4$. Hence:

$$(\forall n \in \mathbb{N})(n+k)^2 = n^2 + 4n + 4 \Leftrightarrow T_k \neq \emptyset.$$

Example 7.9 The proposition $(\forall n \in \mathbb{N})[n = 0 \vee (\exists m \in \mathbb{N}) n = m + 1]$ says that each natural number is 0 or a successor of another natural number. The corresponding set becomes

$$(\Pi n \in \mathbb{N})\, [E_{n,0} + (\Sigma m \in \mathbb{N}) E_{n,m+1}].$$

Exercise: show that this contains exactly one element.

7.3 The type theory

In the preceding section we gave a *set-theoretic* description of dependent type constructions, and thereby achieved a version of principle (II). However we have no guarantee that an element of a set correspond to an algorithmic construction. That is, we still have to achieve principle (I). To realise both (I) and (II) algorithmic counterpart to dependent types must be defined. This is done in *Martin-Löf type theory,* which is a typed lambda calculus which generalise the calculus from Chapter 2 by introducing dependent types. This system is similar to a BHK-interpreted version of natural deduction system for intuitionistic logic (see Chapter 4). What is derived in type theory are *judgements* of the form $a : A$, which may be read as (i) *a has type A* or (ii) *a is a witness to the proposition A.* It turns out that this idea of identifying propositions with types also leads to conceptual simplifications: $\wedge$ and $\exists$ may be unified in the construction Σ, while $\rightarrow$ and $\forall$ man be unified in the Π-construction. The disjunction $\vee$ is given by the type construction $+$.

A novelty of this system is that types may depend on other types. That a type B depends on $z : A$ means essentially that the variable z occurs free in the type expression B. This dependence may be nested, there may yet another type C depending on $y : B$ and $z : A$, etc. These dependencies are written

B is a type $(z : A)$,

C is a type $(z : A, y : B)$.

We may also say that b is an element of B depending on $z : A$. This is written

$b : B\ (z : A)$,

Thus for $a : A$, we have $b[a/z] : B[a/z]$.

Certain aspects of the formal treatment of dependent types will be omitted here (see however Section 7.4 below).

Π-types. Let B be a type depending on $x : A$. The introduction rule for Π is this:

$$\frac{\begin{array}{c}\overline{x : A}\\ \vdots\\ b : B\end{array}}{\lambda x.b : (\Pi x : A)B}(\Pi I)$$

The elimination rule is

$$\frac{f : (\Pi x : A)B \qquad a : A}{\mathsf{apply}(f, a) : B[a/x]}\ (\Pi E)$$

The associated computation rule is $\mathsf{apply}(\lambda x.b, a) = b[a/x] : B[a/x]$, often called the β-*rule.*

For a type B which is independent of x, we see that this becomes the BHK-interpretation of the rules $(\to I)$ and $(\to E)$ respectively, for intuitionistic logic.

If A is considered as the quantification domain, the rules are similar to those for $\forall$. In single sorted intuitionistic logic the judgements $x : A$ and $a : A$ are hidden, since all variables and terms automatically have the type of the quantification domain.

Σ-types. Let B be a type depending on $x : A$. The introduction for Σ is

$$\frac{a : A \qquad b : B[a/x]}{\langle a, b\rangle : (\Sigma x : A)B}\ (\Sigma I)$$

If A is regarded as the quantification domain and B is regarded as a proposition, we see that (ΣI) may be read as the rule $(\exists I)$.

In case B is independent of x, so that $B[a/x] = B$, we see that the rule (ΣI) has the same shape as $(\wedge I)$. (Set-theoretically it holds in this case that: $(\Sigma x \in A)B = A \times B$, see Remark 7.6.)

The elimination rule for Σ is the following: suppose that C is a type depending on $z : (\Sigma x : A)B$.

$$\frac{\begin{array}{cc} & \overline{x:A} \quad \overline{y:B} \\ & \vdots \quad \vdots \\ c : (\Sigma x : A)B & d : C[\langle x,y\rangle/z] \end{array}}{\mathsf{split}(c, \lambda x.\lambda y.d) : C[c/z]} \qquad (\Sigma E)$$

The associated computation rule is $\mathsf{split}(\langle a,b\rangle, g) = g(a)(b) : C[\langle a,b\rangle/z]$.

If C does not depend on z, and the "hidden" judgements for the quantification domain are taken into account, then the rule (ΣE) is in accordance with $(\exists E)$.

If B is independent of x, $C = A$ and $d = x$ we have that

$$\mathsf{split}(\langle a,b\rangle, \lambda x.\lambda y.x) = (\lambda x.\lambda y.x)(a)(b) = a.$$

We may consider $\mathsf{split}(z, \lambda x.\lambda y.x)$ as the first projection $\#_1(z)$, and hence $(\wedge E1)$ is generalised. (Exercise: how may $\#_2(z)$ be defined in terms of split?)

Remark 7.10 In versions of type theory suitable for computer implementation, for instance Agda or Alf, the terms are provided with complete type information. For instance, the expression $\mathsf{split}(c, \lambda x.\lambda y.d)$ is written as

$$\mathsf{split}(A, (x)B, (z)C, c, \lambda x.\lambda y.d),$$

where $(x)B$ indicates that B is a family of types depending on x.

$+$-types. The introduction rule for the binary sum $+$ is

$$\frac{a : A}{\mathsf{inl}(a) : A+B}\ (+I1) \qquad \frac{b : B}{\mathsf{inr}(b) : A+B}\ (+I2)$$

Let C be a type depending on $z : A+B$. The elimination rule is given by

$$\frac{\begin{array}{ccc} & \overline{x:A} & \overline{y:B} \\ & \vdots & \vdots \\ c : A+B & d : C[\mathsf{inl}(x)/z] & e : C[\mathsf{inr}(x)/z] \end{array}}{\mathsf{when}(c, \lambda x.d, \lambda y.e) : C[c/z]}\ (+E)$$

The computation rule is identical with (4).

We leave it to the reader to show how these rules generalise the rules for $\vee$.

Remark 7.11 *Functional programming and dependent types.* The typing discipline in a programming language has the well-known advantage that many programming errors may be detected already by the compiler. Dependent types make

it possible to sharpen this discipline, and to make further errors detectable at an early stage. An example of a functional language using dependent types is Cayenne (Augustsson 1998).

Suppose that we wish to write a program f for multiplying two matrices A and B. For the matrix product AB to be well-defined the number of columns in A must be the same as the number of rows in B. Denote by $M(r,k)$ the type of $r \times k$-matrices. We will thus have $AB : M(r,k)$ for $A : M(r,n)$ and $B : M(n,k)$. One could imagine designing $f(A,B)$ to take a special error value when the dimensions of the matrices are mismatching. This would however mean that dimension errors would not be discovered at the compilation stage. By using dependent types it is possible to let f have the type

$$(\Pi r : \mathbb{N})(\Pi n : \mathbb{N})(\Pi k : \mathbb{N})[M(r,n) \times M(n,k) \to M(r,k)],$$

in which case it becomes impossible to write a well-typed program which uses f and make a dimension error.

An important application of Σ-types is to form modules. There is no need for a special construction as in ML. A module specification is a type of the form

$$(\Sigma f_1 : A_1) \cdots (\Sigma f_n : A_n) P(f_1, \ldots, f_n),$$

where $f_1, \ldots, f_n$ are functions, or operations, and $P(f_1, \ldots, f_n)$ is regarded as a proposition describing their mutual relations and effects. An element

$$\langle g_1, \ldots, \langle g_n, q \rangle \cdots \rangle$$

of this type is an implementation of the module.

Basic types and recursive types. We give the rules for the types $\emptyset$ and $\mathbb{N}$. There is no introduction rule for $\emptyset$ (as it is supposed to have no elements), on the other hand it has the elimination rule

$$\frac{c : \emptyset}{!_A(c) : A} \quad (\emptyset E)$$

The collection of natural numbers $\mathbb{N}$ considered as a recursive type has the introduction rules

$$0 : \mathbb{N} \qquad \frac{a : \mathbb{N}}{\mathsf{S}(a) : \mathbb{N}} \qquad (\mathbb{N}I).$$

The elimination rule is a fusion of the recursion operator rec and the induction principle. Let C be a type depending on $z : \mathbb{N}$.

$$\frac{\begin{array}{ccc} & \overline{x : \mathbb{N}} \quad \overline{y : C[x/z]} & \\ & \vdots \quad \vdots & \\ t : \mathbb{N} \quad b : C[0/z] & & c : C[\mathsf{S}(x)/z] \end{array}}{\mathsf{rec}(t, b, \lambda x.\lambda y.c) : C[t/z]} \ (\mathbb{N}E).$$

The computation rule is the same as for the recursion operator in Chapter 2.

We introduce a basic dependent type $L(z)$, which depends on $z : \mathbb{N}$.

$$L(0) = \emptyset \qquad L(\mathsf{S}(x)) = \mathbb{N}.$$

Combining this type with the function e in Exercise 2.2 we get a dependent $L(e(m)(n))$ that is empty exactly when $m \neq n$ (cf. $E_{m,n}$ above). It is now easy to write down the sets of Example 7.7 – 7.9 as types in type theory by replacing "$\in$" by ":".

Type theory may easily be extended with rules for enumeration types and recursive types of the kind used in ML or Haskell. The type $\mathbb{B}$ of boolean values is an enumeration type with following introduction and elimination rules:

$$\mathsf{tt} : \mathbb{B} \qquad \mathsf{ff} : \mathbb{B} \qquad (\mathbb{B}I)$$

For a type C that depends on $z : \mathbb{B}$

$$\frac{c : \mathbb{B} \qquad d : C[\mathsf{tt}/z] \qquad e : C[\mathsf{ff}/z]}{\mathsf{if}(c,d,e) : C[c/z]} \qquad (\mathbb{B}E)$$

The computation rules are $\mathsf{if}(\mathsf{tt},c,d) = c$ and $\mathsf{if}(\mathsf{ff},c,d) = d$.

As yet another example of a recursive data type consider the type $\mathrm{List}(A)$ of lists of objects of type A. The introduction rules are

$$\mathsf{nil} : \mathrm{List}(A) \qquad \frac{a : A \qquad \ell : \mathrm{List}(A)}{\mathsf{cons}(a,\ell) : \mathrm{List}(A)}.$$

The formulation of the associated elimination rule is left as an exercise for the reader (see Martin-Löf 1984 for a solution).

Warning. A certain care has to be taken when defining new recursive data types, so that the property that all programs terminate is preserved. A recursive data type D satisfying

$$D = D \to D \tag{14}$$

will allow encoding of untyped lambda terms. Even when worse, the theory may also become inconsistent: if for instance there is a recursive type A given by

$$A = A \to \emptyset. \tag{15}$$

There are syntactical criteria that guarantee termination and relative consistency; see Dybjer 2000. In the system Agda (Chapter 9) there is a partial termination checker, that catches all such non-wellfounded definitions, but also forbids many legitimate ones.

Type universes. A *type universe* is a type U consisting of other types, that is A is a type for each $A : U$. The purpose of a type universe is to admit quantification over types, and thus over families of types. Given a type X and $B : X \to U$, then $\mathsf{apply}(B,x) = B(x)$ is also a type, depending on $x : X$. The type $X \to U$ may thus be viewed as the type of all types in U depending on X.

Example 7.12 It is possible to introduce a very small universe U' consisting only of the types $\emptyset$ and $\mathbb{N}$. From this *define* L by recursion: let $C = U'$

$$L(z) =_{\text{def}} \mathsf{rec}(z, \emptyset, \lambda x.\lambda y.\mathbb{N})$$

The computation rules gives $L(0) = \emptyset$ och $L(\mathsf{S}(x)) = \mathbb{N}$. □

In standard versions of Martin-Löf type theory there is a universe Set containing the basic types $\emptyset$ and $\mathbb{N}$, and which is moreover closed under the formation of Π-, Σ- and $+$-types. The closure condition means that

- $(\Pi x : A)B(x) : \mathsf{Set}$, if $A : \mathsf{Set}$ and $B : A \to \mathsf{Set}$,
- $(\Sigma x : A)B(x) : \mathsf{Set}$, if $A : \mathsf{Set}$ and $B : A \to \mathsf{Set}$,
- $A + B : \mathsf{Set}$, if $A : \mathsf{Set}$ and $B : \mathsf{Set}$.

Example 7.13 With the help of a universe one may define types that lacks counterpart in ML or Haskell. For instance, let for $n : \mathbb{N}$ and $A : \mathsf{Set}$,

$$\begin{aligned} F(0)(A) &= A \\ F(\mathsf{S}(n))(A) &= A \to F(n)(A) \end{aligned}$$

We have $F(n)(A) = A \to A \to \cdots \to A$ (n arrows), so the number of arguments that a function of this type takes *depend* on n. (Compare to the example with matrices above.) Formally we may define F as

$$\lambda n.\mathsf{rec}(n, \lambda A.A, \lambda x.\lambda y.\lambda A.(A \to y(A)),$$

where $F : \mathbb{N} \to \mathsf{Set} \to \mathsf{Set}$. (Exercise: check this using the computation rules.) □

Remark 7.14 Type universes make it possible to type polymorphic functions, i.e. functions which works the same way regardless of type. An example is the appending of lists

$$\mathsf{append} : (\Pi A : U)\,[\mathit{List}(A) \to \mathit{List}(A) \to \mathit{List}(A)].$$

$\mathsf{append}(\mathbb{N})$ is then the append function specialised to lists of natural numbers. □

Sometimes it may be necessary to pile universes on each other. The type $B = (\Sigma A : \mathsf{Set})(\Sigma n : \mathbb{N})F(n)(A)$ does not belong to Set, since it is not the case that $\mathsf{Set} : \mathsf{Set}$. The type $F(n)(B)$ is not well-defined, since F demands that B belongs to Set. To construct an F which can take B as an argument we may introduce a larger universe Set_2 which is closed under Π-, Σ- and $+$-constructions and which is such that

- $\mathsf{Set} : \mathsf{Set}_2$,
- $A : \mathsf{Set}_2$ for all $A : \mathsf{Set}$.

Now, the same problem would occur again if in B the universe Set is replaced by Set_2.

Remark 7.15 The simple and drastic solution to assume $\mathsf{Set} : \mathsf{Set}$ leads to an inconsistent theory; see Martin-Löf 1971. This paradox is called *Girard's paradox*. In practise it seems that one or two levels of universes are enough.

7.4 Type theory as a formal system*

In a formal system for Martin-Löf type theory there is not only rules for judgements of the form $a : A$. The types cannot be defined as grammatically simple as for the lambda calculus in Chapter 2. That an expression $B(a)$ is a type, may depend on that we have previously showed that $a : A$. Therefore one needs a particular judgement form for saying that an expression is a type. For instance rules of the following kind are needed

$$\frac{\begin{array}{c}\overline{x : A}\\ \vdots\\ A\ \text{type} \quad B\ \text{type}\end{array}}{(\Pi x : A)B\ \text{type}} \qquad \frac{z : \mathbb{N}}{L(z)\ \text{type}}$$

Furthermore, the computation rules have a special judgement form $a = b : A$, which states that a and b can be computed to the same element of the type A. Because of constructions as $L(\cdot)$ above, there should be a way of expressing that two types may be computed to the same type, $A = B$. We have for instance $L(e(0)(1)) = L(0) = \emptyset$.

A technique for managing open assumptions is to use assumption lists (see Remark 4.4 above)

$$\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n.$$

Such a list is also called a *context*. For type theory they are rather complicated, since the order between the assumptions is important: A_n may depend on all variables $x_1 : A_1, \ldots, x_{n-1} : A_{n-1}$. This means that, in general, an assumption early in the list may not be possible to discharge, until all later assumptions have been discharged. To ensure correct formation of contexts, there is yet another judgement form. One needs to know that an expression actually is a type before the assumption that a variable has this type can be added to context.

$$\frac{\Gamma \vdash B \text{ type}}{\Gamma, y : B \text{ context}}.$$

The assumption rule is

$$\frac{x_1 : A_1, \ldots, x_n : A_n \text{ context}}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i}.$$

In computer implementations of type theory, contexts and computational equalities are handled by the system. It is thus not necessary worry about those rules, to be able to use the theory.

Exercises

7.1. Let $I_0 = \{0,1,2\}$, $I_1 = \{0,3\}$, $I_2 = \emptyset$, $A_0 = \{0,1\}$, $A_1 = \{0\}$, $A_2 = \{1,2\}$ and $A_3 = \emptyset$. List the elements of the following sets

(a) $(\Sigma i \in I_0)A_i$

(b) $(\Sigma i \in I_1)A_i$

(c) $(\Pi i \in I_0)A_i$

(d) $(\Pi i \in I_1)A_i$

(e) $(\Pi j \in I_0)(\Sigma i \in I_j)A_i$

(f) $(\Sigma j \in I_0)(\Pi \in I_j)A_i$

7.2. Do the exercise in Example 7.9.

7.3. Formulate an elimination rule for the recursive data type $List(A)$.

7.4. Define $\#_2(\cdot)$ in terms of split.

8 Formalising mathematics

In order to mechanically check, or construct, mathematical proofs we need first of all to have a mathematically precise notion of proof. This is achieved by various logical systems, as we have already seen examples of. But we also need some conventions for describing or coding mathematical objects: rational numbers, real numbers, matrices, functions, ordered sets, graphs, trees etc.

The foundation of mathematics in set theory builds on the coding of all objects as sets, in effect built up from the empty set $\emptyset$ and set parentheses $\{\quad\}$. Thus for instance natural numbers are usually coded as

$$0 = \emptyset \quad 1 = \{0, \{0\}\} \quad 2 = \{1, \{1\}\} \quad \cdots \quad .$$

Pairs of elements may be coded as

$$\langle a, b \rangle = \{\{a\}, \{a, b\}\}.$$

The cartesian products $A \times B$ of two sets is the set of all $\langle a, b \rangle$ where $a \in A$ and $b \in B$. Functions $A \to B$ are then subsets of $A \times B$ satisfying the usual condition of graphs of functions. For details of further such codings see e.g. (Krivine 1971).

The next level is to disregard from particular representation of mathematical objects and to describe their properties abstractly. For instance, the cartesian product of two sets A and B may be described as a set $P(A, B)$ together with two "projection" functions

$$\pi_1 : P(A, B) \to A \qquad \pi_2 : P(A, B) \to B,$$

such that for all $a \in A$ and $b \in B$ there exists a unique element $c \in P(A, B)$ with $\pi_1(c) = a$ and $\pi_2(c) = b$. Thus π_k picks out the kth component of the abstract pair. Thereby the reference to the particular coding of ZF set theory may be avoided. In category theory this point of view is taken to its extreme, and one dispenses with the reference to sets altogether. Mathematics may as well be founded on category theory.

At this level of abstraction it makes little difference whether mathematics is founded on set theory or type theory. We shall see below how some important basic notions are formalised in type theory.

8.1 Sets and functions

In mathematics a basic means of abstraction is that of regarding some objects as the "same" from some aspect. This is done is set theory by the introduction of equivalence relations and then equivalence classes. For instance, if for natural

natural numbers $\mathbb{N}$ we want to disregard from everything except from whether they have the same parity, we introduce the equivalence classes of even and odd numbers:

$$\mathbf{0} = \{0,2,4,\ldots\} \qquad \mathbf{1} = \{1,3,5,\ldots\}.$$

As first step an equivalence relation $\equiv$ on $\mathbb{N}$ is defined

$$x \equiv y \Longleftrightarrow_{\text{def}} x - y \text{ is divisible by 2.}$$

The equivalence classes are introduced as subsets of $\mathbb{N}$: $[x] = \{y \in \mathbb{N} : x \equiv y\}$. Then the *quotient set* is

$$(\mathbb{N}/\equiv) = \{[x] : x \in \mathbb{N}\}$$

and note that $[x] = [y]$ iff $x \equiv y$. Hence e.g.

$$\mathbf{0} = [0] = [2] = [4] = \cdots$$

This construction is general and can be made for any set X, and any given equivalence relation $\equiv$ on X.

In constructive mathematics one usually skips the introduction of equivalence classes, following the princple that each set X should come with an explicitly given equivalence relation $=_X$. This has the advantage that the notion of set can be understood in a quite concrete way, and avoiding sets of sets. For instance the quotient set $(\mathbb{N}/\equiv)$ above, would be the pair $(\mathbb{N},\equiv)$. It is actually close to some practise in mathematics to use explicit equivalence relations when there is a possibility of confusion as in

$$7 \equiv 5 \equiv 3 \pmod 2.$$

A *setoid* X will be a type $\underline{X}$ together with an equivalence relation $=_X$ on $\underline{X}$. The latter means that $x =_X y$ is a family of types depending on $x, y : \underline{X}$ and that there are functions ref, sym and tra with

$$\text{ref}(a) : a =_X a \quad (a : \underline{X}),$$

$$\text{sym}(a,b,p) : b =_X a \quad (a : \underline{X}, b : \underline{X}, p : a =_X b),$$

$$\text{tra}(a,b,c,p,q) : a =_X c \quad (a,b,c : \underline{X}, p : a =_X b, q : b =_X c).$$

We shall write $x \in X$ instead of $x : \underline{X}$.

Remark 8.1 In the Bishop tradition of constructive mathematics $\underline{X}$ is called a *preset*, rather than a type, and X is called a *set*.

An *extensional function* f from the setoid X to the setoid Y is a pair $(\underline{f}, \mathrm{ext}_f)$ where $\underline{f} : \underline{X} \to \underline{Y}$ is function so that

$$\mathrm{ext}_f(a,b,p) : \underline{f}(a) =_Y \underline{f}(b) \quad (a,b : \underline{X}, p : a =_X b)$$

Two functions $f, g : X \to Y$ are *extensionally equal* if there is e with

$$e(a) : \underline{f}(a) =_Y \underline{g}(a) \quad (a : \underline{X}).$$

We shall follow the terminology of Bishop, when dealing with setoids, calling an extensional function simply *function*, and calling a function, an *operation.* Also we usuall write f also for $\underline{f}$ when there is no risk of confusion.

We introduce some standard properties for a function $f : X \to Y$ between setoids. It is

- *injective* if $u =_X v$ whenever $f(u) =_Y f(v)$
- *surjective* if for every $y \in Y$ there is some $x \in X$ with $f(x) =_Y y$.
- *split epi* if there is a function $g : Y \to X$ so that $f \circ g = 1_Y$.
- *bijective* if it is injective and surjective.

Clearly, every function $f : X \to Y$ which is split epi is also surjective. The *axiom of choice* in classical ZF set theory can be phrased as: every surjective function is split epi. This is in general too strong for being constructively acceptable. We have however the following results.

An equivalence relation $=_Y$ on $\underline{Y}$ is *finest*, if for any other equivalence relation $\sim$ on $\underline{Y}$,

$$a =_Y b \Longrightarrow a \sim b.$$

A setoid whose equivalence relation is finest is called a *choice setoid.*

Theorem 8.2 *For any setoid X and any choice setoid Y, each surjective function $f : X \to Y$ is split epi.*

Proof. Suppose $f : X \to Y$ is sujective. Then

$$p(y) : (\Sigma x \in X)(f(x) =_Y y) \quad (y \in Y).$$

Let $\underline{g}(y) = \#_1(p(y))$. Then $f(\underline{g}(y)) =_Y y$ is true for all $y \in Y$, but we still do not know that $\underline{g}$ is extensional. Define a new equivalence relation $\sim$ on $\underline{Y}$ by

$$y \sim z \Longleftrightarrow \underline{g}(y) =_X \underline{g}(z).$$

Now, since $=_Y$ is the finest equivalence relation on $\underline{Y}$, we get the desired extensionality

$$y =_Y z \Longrightarrow \underline{g}(y) =_X \underline{g}(z).$$

Thus g is extensional and $f \circ g = 1_Y$. $\Box$

Example 8.3 The setoid of natural numbers $(\mathbb{N}, =_{\mathbb{N}})$ where

$$m =_{\mathbb{N}} n = L(e(m)(n)),$$

is a choice setoid.

A relation R between setoids X and Y is a family of types $R(x,y)$ $(x \in X, y \in Y)$ such that

$$R(x,y), x =_X x', y =_Y y' \Longrightarrow R(x',y').$$

For $X = Y$, we say that R is an relation *on* X

Quotient setoids. Let $X = (\underline{X}, =_X)$ be a setoid and let $\sim$ be a relation on the setoid X, which is an equivalence relation. Then $\sim$ is an equivalence relation on $\underline{X}$, and

$$x =_X y \Longrightarrow x \sim y. \tag{16}$$

Then $X/\sim = (\underline{X}, \sim)$ is a setoid, and $i : X \to X/\sim$ defined by $i(x) = x$ is a function according to (16). We have the following extension property. If $f : X \to Y$ is a function with

$$x \sim y \Longrightarrow f(x) =_Y f(y),$$

then there is a unique function $\overline{f} : X/\sim \to Y$ (up to extensional equality) with

$$\overline{f}(i(x)) =_Y f(x) \qquad (x \in X).$$

This is the same abstract property that quotient sets has in classical ZF set theory.

The identity type construction assigns to each type A a finest equivalence relation $\mathrm{Id}(A, \cdot, \cdot)$, see Nordström *et. al.* 1990. It follows that every setoid is the quotient setoid of a choice setoid, in this version of type theory.

Remark 8.4 For a full development of the basic theory of sets and functions in the constructive setting we refer to Bishop and Bridges 1985 and Mines *et al.* 1988.

9 Implementations of type theory

A modern implementation of Martin-Löf type theory is Agda (C. Coquand 1998). On top of this there is a graphical user interface Alfa (Hallgren 1998), also supporting additional notational features.

We describe the syntax of Agda, which is an "unsugared" version of the Alfa syntax. It is also possible to work with Agda syntax in Alfa by using the command "Edit as Text".

- Type membership is denoted by a double colon $a :: A$ in Agda.
- Function application $f(a)$ is written as a juxtaposition $f\ a$ with a space in between.
- A dependent function type $(\Pi x : A)B$ is written $(x::A)\ \texttt{->}\ B$. For the corresponding lambda abstraction $\lambda x.b$ the argument type is written explicitly, and becomes $\backslash(x::A)\ \texttt{->}\ b$. (When B does not depend on x one may write $A\ \texttt{->}\ B$.)
- The two-place Σ-type is a special case of a more general *record type,* where components are named, and not just numbered, For instance $(\Sigma x : A)B$ corresponds to

  $$\texttt{sig \{ fst::}A\texttt{; snd::}B[\texttt{fst}/x]\texttt{ \}}$$

  Let z be an element of this type. To access the value of the component (or field) with name `fst` one writes z.`fst`. The pair $\langle a, b\rangle : (\Sigma x : A)B$ is written

  $$\texttt{struct \{ fst = }a\texttt{; snd = }b\texttt{ \}}$$

- Enumeration types and recursive data types may be defined similarly as in ML, with the help of the construction `data`. From such a definition constructors and the `case` is automatically generated. We give some simple examples.

  The type $\mathbb{B}$ of booleans is defined by

  ```
  data tt | ff
  ```

  The constructors are called `tt@_` and `ff@_` respectively. Corresponding to $\mathsf{if}(c, d, e)$ we have

  $$\texttt{case }c\texttt{ of \{ tt -> }d\texttt{; ff -> }e\texttt{\}}$$

However, `case` should only be used on variables. If c is non-variable one writes

$$\texttt{let \{y = c\} in case } y \texttt{ of \{ tt -> } d\texttt{; ff -> } e\texttt{\}}$$

The empty type is `data { }` and since it does not have constructors its elimination operator $!(c)$ becomes trivial: `case` c `of { }`

Natural numbers $\mathbb{N}$ are defined recursively by

```
Nat ::  Set = data zero | S (n::Nat).
```

(In this expression `Nat :: Set` means that the new type belongs to the type universe `Set`.) The constructors of this type are `zero@_` and `S@_` respectively. The case-function corresponding to this is

$$\texttt{case } c \texttt{ of \{zero -> } u_1\texttt{; S n -> } u_2\texttt{\}},$$

where n may be free in in u_2.

- In Agda there are type universes `Set` and `Type` (essentially corresponding to Set_1 and Set_2 above). These universes are closed under Π-types, record types and the formation of recursive data types.

Definitions in Agda have the form

$$f\ \texttt{(}v_1\ \texttt{::}\ \ \mathit{intype}_1\texttt{)}\ \cdots\ \texttt{(}v_n\ \texttt{::}\ \ \mathit{intype}_n\texttt{)}\ \texttt{::}\ \ \mathit{outtype}\ \texttt{=}\ \mathit{def}$$

where f is an identifier for the function to be defined, $v_1, \ldots, v_n$ are variables for arguments, *outtype* is the type of the value and *def* is the defining term.

Example 9.1 The +-construction from Chapter 2 is defined in Agda as

```
Plus (A::Set)(B::Set) :: Set
  = data inl (x::A) | inr (y::B)
```

`Plus` is thus a type constructor which takes two types from the universe `Set` and forms the disjoint union of them in the same universe. The when-function may be defined by

```
when (A::Set)(B::Set)(C::Set)
     (c:Plus A B)(d::A->C)(e::B->C) :: C
  = case c of {(inl x) -> d x;
               (inr y) -> e y}
```

`when` thus has six arguments, of which the first three are types (these were omitted in Chapter 2; see however Exercise 2.3). □

Example 9.2 The recursion operator `rec` (Chapter 2) is defined as

```
rec (A::Set)(a::Nat)(f::A)(g::Nat->A->A) :: A
  = case a of { zero     -> f;
                (S x)    -> g x (rec A x f g)}
```

□

In these examples there no essential use of dependent types. The type universes make it easy to handle such types. The type `A -> Set` consists of all families of types in `Set` that depends on `A`. We may introduce the abbreviation `Fam A` for this type, in Agda:

```
Fam (A::Set) :: Type
  = A -> Set
```

Example 9.3 The general recursion operator (Chapter 7) also gives the induction principle, and is expressed as follows

```
rec (A::Fam Nat)
    (a::Nat)
    (f::A zero@_)
    (g::(x::Nat)->(y::A x)->A (S@_ x)) :: A a
  = case a of { zero    -> f;
                (S x)   -> g x (rec A x f g)}
```

□

Example 9.4 The dependent type L is defined by

```
L (z::Nat) :: Set
  = case z of {zero -> Empty;
               (S x) -> Nat}
```

where `Empty::Set = data`.

Example 9.5 A module of functions that find sufficiently large prime numbers may be written as the record type

```
Primefinder :: Set
   = sig {f:: Nat -> Nat;
          correct1 :: (n::Nat) -> Prime (f n);
          correct2 :: (n::Nat) -> LessEq n (f n)}
```

where `Prime` is a predicate which which is true precisely for prime number, and `LessEq` is the relation $\leq$ on $\mathbb{N}$. For `z::Primefinder` the field `z.f` is a function that given n gives a prime number $p \geq n$. (Exercise: define `Prime` and `LessEq`.)

Bibliography

A good introduction to lambda calculus is Hindley and Seldin 1986, which is mainly devoted to untyped and simply type calculus. For a deeper study of untyped calculus Barendregt 1977 is the standard reference. Troelstra and Schwichtenberg 1996 is a thorough introduction to both typed lambda calculus and proof theory for intuitionistic predicate logic. Barendregt 1992 treats lambda calculus with dependent types. Martin-Löf 1984 gives an elegant presentation of his own type theory with "meaning explanation" and philosophical justifications. The original paper Martin-Löf 1972 is also strongly recommended. For an introduction to type theory and its applications in computer science, see Coquand *et. al.* 1994, Nordström *et. al.* 1990 and Thompson 1991. A short introduction to intuitionistic logic and its semantics is in van Dalen 1997. The standard tomes on constructive logical systems are van Dalen och Troelstra 1988, vol. I and II.

Augustsson, Lennart (1998), Cayenne - a language with dependent types. *Proc. of the International Conference on Functional Programming (ICFP'98).* ACM Press, September 1998. [See also URL: `www.cs.chalmers.se/~augustss/`.]

Barendregt, H.P. (1977), *The Lambda Calculus.* North-Holland.

Barendregt, H.P. (1992), Lambda calculi with types, pp. 118 – 279 i (S. Abramsky, D.M. Gabbay and T.S.E. Maibaum eds.) *Handbook of Logic in Computer Science,* vol 2. Oxford University Press.

Beeson, Michael (1985), *Foundations of Constructive Mathematics.* Springer-Verlag.

Bellantoni, Stephen, Cook, Stephen (1992). A New Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity* vol. 2, 97-110.

Bishop, Errett and Bridges, Douglas S. (1985), *Constructive Analysis.* Springer-Verlag.

Bridges, Douglas S. och Richman, Fred (1987), *Varieties of Constructive Mathematics.* London Mathematical Society Lecture Notes, Vol. 97. Cambridge University Press.

Coquand, Catarina. *Agda.* URL: `www.cs.chalmers.se/~catarina/agda/`

Coquand, Thierry, Nordström, Bengt, Smith, Jan och von Sydow, Björn (1994), Type theory and Programming. *The EATCS bulletin,* February 1994. [Also available at URL: `www.cs.chalmers.se/~smith/`.]

Curry, H.B. and Feys, R. (1958), *Combinatory Logic.* North-Holland.

Dybjer, P. (2000), A general formulation of simultaneous inductive-recursive definitions in type theory, *Journal of Symbolic Logic* 65(2000).

Hallgren, Thomas. *The Proof Editor Alfa.*
URL: `www.cs.chalmers.se/~hallgren/Alfa/`

Hansen, Kaj B. (1997), *Grundläggande Logik.* Studentlitteratur.

Hansen, Michael R. och Rischel, Hans (1999), *Introduction to Programming Using SML.* Addison-Wesley.

Heyting, Arend (1971), *Intuitionism.* North-Holland.

Hindley, J.R. och Seldin, J.P. (1986), *Introduction to Combinators and Lambda Calculus.* Cambridge University Press.

Howard, W.A. (1980), The Formulae-as-Types Notion of Construction. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* J.P. Seldin and J.R. Hindley (red.), Academic Press, pp. 479 – 490.

Martin-Löf, Per (1972), *An intuitionistic theory of types.* Technical report, Department of Mathematics, University of Stockholm. Published in (Sambin and Smith 1998).

Martin-Löf, Per (1984), *Intuitionistic Type Theory.* Bibliopolis.

Martin-Löf, Per (1985), Constructive Mathematics and Computer Programming. *Mathematical Logic and Computer Languages.* C.A.R. Hoare and J.C. Shepherdson (red.). Prentice-Hall.

Mines, Ray, Richman, Fred, Ruitenburg, Wim (1988), *A Course in Constructive Algebra.* Springer.

Mints, Grigori (2000), *A Short Introduction to Intuitionistic Logic.* Kluwer Academic/Plenum Publishers.

Nelson, Edward (1995). *Ramified recursion and intuitionism.* Preprint.
`www.math.princeton.edu/~nelson/papers.html`

Nordström, Bengt, Peterson, Kent and Smith, Jan (1990), *Programming in Martin-Löf's Type Theory.* Oxford University Press. [The book is out of print, but is accessible in electronic form at URL:
`www.cs.chalmers.se/Cs/Research/Logic/book/` .]

Nordström, Bengt, Peterson, Kent och Smith, Jan (2000), Martin-Löf's type theory. *Handbook of Logic in Computer Science*, Vol. 5. Oxford University Press.

Salling, Lennart (1999), *Formella språk, automater och beräkningar.* Eget förlag, Uppsala.

Sambin, Giovanni and Smith, Jan (1998) (eds.) *Twenty-Five Years of Type Theory.* Oxford University Press.

Schwichtenberg, Helmut (1999), *Classical Proofs and Programs.* Marktoberdorf Summer School ’99. Available at URL:
`www.mathematik.uni-muenchen.de/~schwicht/`

Simmons, Harold (2000), *Derivation and Computation.* Cambridge University Press.

Stirling, Colin (1992), Modal and temporal logics, pp. 477 – 563 i (S. Abramsky, D.M. Gabbay and T.S.E. Maibaum eds.) Handbook of Logic in Computer Science, vol. 2, Oxford University Press.

Thompson, Simon (1991), *Type Theory and Functional Programming.* Addison-Wesley.

Thompson, Simon (1999), *The Craft of Functional Programming.* Addison-Wesley.

Troelstra, Anne S. and Schwichtenberg, Helmut (1996), *Basic Proof Theory*, Cambridge University Press.

Troelstra, Anne S. och van Dalen, Dirk (1988), *Constructivism in Mathematics, Vol. I & II.* North-Holland.

van Dalen, Dirk (1997) *Logic and Structure,* Third edition. Springer.