CHAPTER 18

Proof-Assistants Using Dependent Type Systems

Henk Barendregt

Herman Geuvers

Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

1. Proof checking

Proof checking consists of the automated verification of mathematical theories by first fully formalizing the underlying primitive notions, the definitions, the axioms and the proofs. Then the definitions are checked for their well-formedness and the proofs for their correctness, all this within a given logic. In this way mathematics is represented on a computer and also a high degree of reliability is obtained.

After a certain logic is chosen (e.g. classical logic or intuitionistic logic; first-, second- or higher-order logic) there are still several ways in which a theory can be developed. The Cantor-Hilbert-Bourbaki style is to use set-theory, say Zermelo-Fraenkel set-theory with the axiom of choice formalized in first-order classical logic (ZFC)[1]. Indeed, the great attraction of set-theory is the fact that *in principle* it can be used to formalize most mathematical notions. But set-theory has as essential problem that it cannot capture computations very well. Computations are needed for applications of theories and—as we will see later—also for providing proofs. In both cases we want, say for a function $f : \mathbb{N} \to \mathbb{N}$, that for numbers $n, m \in \mathbb{N}$ such that $f(n) = m$, we can find a formal proof of $\underline{f}(\underline{n}) = \underline{m}$, where the underlinings stand for representations in the theory. Although this is theoretically possible for set-theory, *in practice* this may not be feasible. This is because a computation has to be coded in set-theory as a sequence of sets being a formal description of a computation path (consecutive states) according to some computational model.

Type theory presents a powerful formal system that captures both the notion of *computation* (via the inclusion of functional programs written in typed λ-calculus) and of *proof* (via the so called 'propositions-as-types embedding', where types are viewed as propositions and terms as proofs). As a matter of fact there are various type theories, capturing various notions of computation (e.g. primitive recursion, recursion over higher types) and various logical systems (e.g. first order, higher order). In this article we will not attempt to describe all the different possible choices of type theories. Instead we want to discuss the main underlying ideas, with a special focus on the use of type theory as the formalism for the description of theories including proofs.

Once a theory is formalized, its correctness can be verified by a small program, the *proof checker*. But in order to make the formalization process feasible, an interactive *proof-development system* is needed. This is a proof environment that stands next to the proof-checker and helps the human to develop the proofs. The combination of a proof-development system and a proof checker is called a *proof-assistant*. Such a combination is different form a 'theorem prover'. This is a computer system that allows the user to check the validity of mathematical theorems by generating them automatically. Of course, for proof-assistants the end goal is also to prove theorems. But this is not done by implementing a number of smart algorithms (like resolution or binary decision diagrams), but by letting the user generate a *proof*, interactively with the system. So, the user of proof-assistants is very much in control: by means

[1] Or perhaps some stronger versions with large cardinals, e.g. for the formalization of category theory

of 'tactics' (that are input to the system) a so-called 'proof-term' is created that closely corresponds to a standard mathematical proof (in natural deduction style).

For machine assisted theorem proving (via automated theorem proving or via interactive proof generation or a combination of the two) the main goal is to increase the reliability of mathematical results[2]. Roughly there are two reasons why mathematical results may be difficult to verify. The first is *complexity*: the problem is very big, the number of cases to be distinguished being very large, etcetera. This is a situation that one often encounters in computer science, where, e.g. in a protocol one has to go through all possible states of a system. The second problem is *depth*: the problem is very deep, very complicated. This is a situation that is more often encountered in pure mathematics, e.g. Fermat's last theorem is an example. In case of *complexity*, we may expect help from an automated reasoning tool, e.g. to go through a huge number of cases that each by themselves is easily verified. In case of *depth*, an automated reasoning tool will be of little use, but we may expect some help from a proof assistant that does the bookkeeping and prevents us from overseeing details. In the latter case, we might also want to use the proof assistant as a tool for exploring new fields. At this moment however, there is not yet a user-friendly system that provides machine assistance for doing mathematical research. But the potential is there.

Proof assistants based on type theory present a general specification language to define mathematical notions and formulas. Moreover, it allows to construct algorithms and proofs as first class citizens. The advantages are that a user can define his or her own structures in a very flexible way, including the (executable) functions that are part of these structures. Furthermore—and this is what distinguishes the type theoretic approach to theorem proving from most of the other ones—presented in this style, theorem proving consists of the (interactive) construction of a proof-term, *which can be easily checked independently*. These issues will be discussed in more detail below. Again we want to point out that type theory presently does not provide a fast tool for automated theorem proving: there is (in general) not much automation and the fact that explicit proof-terms are constructed slows down the implementation. Also as a research tool proof-assistants are not yet mature. However, they provide a very high reliability, both because of the explicit proof-terms and their well-understood meta-theory. Another good point is their expressive flexibility. For further reading on these issues, beyond the scope of this Chapter, we advise [Luo 1994] or [Nordström, Petersson and Smith 1990].

Another possible (future) application of machine assisted theorem proving is the field of *computer mathematics*. Right now, computers are used in various parts of mathematics, notably for computer algebra and numerical methods. Each of such applications requires the formalization of a specific part of mathematics, covering the domain of the application. To have various systems interact with each other and with the user would require a formalization of substantial parts of mathematics.

[2]There are systems, like JAPE [1997], Mathpert [1997] and Hyperproof, see [Barwise and Etchemendy 1995], that have mainly an educational goal and are not geared towards proving large mathematical theorems. However these systems are comparable since they want to prevent their users from erroneous reasoning.

For example the language OpenMath [1998] is aiming at providing an intermediate level between such mathematical computer applications. Reaching even further is the idea, laid down the QED-manifesto (see [Bundy 1994]), of creating an electronic library of completely formalized and checked mathematical results, that one can refer to, browse through, use and extend. For this it is necessary that the proof-assistants become much more user-friendly. This would first of all require a very general and flexible mathematical vernacular by means of which ordinary mathematicians can do the work of formalizing and interact with the library. We believe that type theory can provide such a language. As it stands, only the Mizar project (see [Mizar 1989]) has created and maintains a large collection of mathematical results. There is, however, no obvious way of transferring a result from the Mizar theorem prover to another proof-assistant and also it is hard to find results in the Mizar library.

2. Type-theoretic notions for proof checking

The type systems that are used as a foundational theory are influenced by several people. We mention them here and name their important contribution. Brouwer and Heyting for intuitionistic logic; Russell for the notion of type and for the use of higher order quantification to define logical operations; Gentzen and Prawitz for natural deduction; Church and Curry for typed lambda terms; Howard for the propositions-as-types interpretation; de Bruijn for introducing dependent types and for type conversion for δ- and β-reduction; Scott for inductive types; Martin-Löf for the use of inductive types to define the logical operations, thereby completing the propositions-as-types interpretation, and for type conversion for iota-reduction; Girard for higher order type systems and their normalization; Coquand and Huet for building a type system that incorporates all the previous notions.

Besides this we mention the following people. McCarthy [1962] for his idea of proof checking, including symbolic computing. He did not, however, consider representing proofs in natural deduction form, nor did he have the use of higher types for making appropriate abstractions. De Bruijn for his vigorous plea for proof checking and revitalizing type systems for this purpose. Martin-Löf for his emphasis on reliability (by requiring a clear phenomenological semantics) and consequent proposal to restrict to predicative type systems.

2.1. Proof checking mathematical statements

Mathematics is usually presented in an informal but precise way. One speaks about 'informal rigor'. A typical result in mathematics is presented in the following form.

In situation Γ we have A. Proof. p. ■

Informal mathematics

Here Γ is an informally described set of assumptions and A is an informally given statement. Also the proof p is presented informally. In logic the statements Γ, A become formal objects and so does the notion of provability. Proofs still are presented in an informal way, but theoretically they can be formalized as a derivation-tree (following some precisely given set of rules).

$$\boxed{\begin{array}{l}\Gamma \vdash_L A \\ \text{Proof. } p. \blacksquare\end{array}}$$

Mathematics formalized in logic

It turns out that there are several natural ways to translate propositions as *types* (for the moment one may think of these as 'sets') and proofs as *terms inhabiting* ('elements of') these types. The intuitive difference between sets and types is that an object can be in several different sets, but only in one type. Moreover, a type is a rather 'simple' kind of set: whether a term is of a certain type is usually decidable, due to the fact that 'being of a type' is a syntactic criterion. In the context of type theory, membership of a term a to the type A is denoted by $a{:}A$ rather than $a{\in}A$. Writing the translation of proposition A as $[A]$ and of a proof p as $[p]$ one has

$$\vdash A \text{ using proof } p \;\Leftrightarrow \vdash [p] : [A],$$

and hence

$$A \text{ is provable} \quad \Leftrightarrow \quad [A] \text{ is inhabited.}$$

Therefore the formalization of mathematics in type theory becomes the following (we do not write the [ ] but identify a proposition or proof with its translation).

$$\boxed{\Gamma \vdash_T p : A}$$

Mathematics formalized in type theory

Now all of Γ, A and p are formalized linguistic objects. The statement $\Gamma \vdash_T p : A$ is equivalent to

$$\boxed{\text{Type}_\Gamma(p) = A}$$

Proof checking

Here, $\text{Type}_{\_}(-)$ is a function that finds for p a type in the given context Γ. The decidability of type-checking follows from:

- $\text{Type}_\Gamma(p)$ generates a type of p in context Γ or returns 'false' (if p has no such type).
- The equality $=$ is decidable.

The story is a little bit more complicated. First there are several possible logics (e.g. first or second order logic; intuitionistic or classical logic). This will give rise to several type theories. Secondly the equality $=$ in the last statement depends on the type theory: it is a conversion relation generated from a specific set of elementary reductions.

In the practice of an interactive proof assistant based on type theory, the proof-terms are generated interactively between the user and the proof development system. The user types in so called *tactics*, guiding the proof development system to construct a proof-term. At the end, this term is type checked and the type is compared with the original goal. In connection to proof checking, decidability problems that we can distinguish.

$$
\begin{array}{llll}
\Gamma & \vdash_T & M : A? & \text{TCP,} \quad \text{Type Checking Problem;} \\
\Gamma & \vdash_T & M : ? & \text{TSP,} \quad \text{Type Synthesis Problem;} \\
\Gamma & \vdash_T & ? : A & \text{TIP,} \quad \text{Type Inhabitation Problem.}
\end{array}
$$

If we think of A as a formula and M as its proof, then the TCP asks to verify *whether an alleged proof M indeed proves A*. TSP asks to verify *whether the alleged proof M is a proof at all.* TIP asks to verify *whether A is provable.* It will be clear that TIP is undecidable for any type theory that is of interest for formalizing mathematics (i.e. for any T in which enough first order predicate logic can be done). Whether TCP and TSP are decidable depends in general on the rules of the type theory and especially on how much type-information is added in the term M. In all of the systems that we discuss, both TCP and TSP are decidable. Decidability of TCP and TSP conforms with the intuition that, even though we may not be able to *find* a proof of a given formula ourselves, we can *recognize* a proof if presented to us.

Software (like our proof development system) is *a priori* not reliable, so why would one believe a system that says it has verified a proof? This is a good question. The pioneer of computer verified proofs, N.G. de Bruijn, has given a satisfactory answer. We should take care that the verifying program (the type checker) *is a very small program*; then this program can be verified by hand, giving the highest possible reliability to the proof checker. This is the so called *de Bruijn criterion.*

> A proof assistant satisfies the de Bruijn criterion if it generates 'proof-objects' (of some form) that can be checked by an 'easy' algorithm.

In the late sixties de Bruijn made an impressive start with the technology of proof checking. He designed formal systems for the efficient representation of proofs allowing a verifying algorithm that can be coded in 200 lines of imperative code. These systems were given the collective name Automath, see [Nederpelt, Geuvers and de Vrijer 1994] for an up to date survey. As to the point of reliability, de Bruijn has remarked that one cannot obtain absolute certainty. There always can be some kind of electronic failure that makes a proof-assistant accept a wrong proof (actually this is very unlikely; there is a bigger chance that a correct proof is not accepted). But formalized proofs provide results with the *highest possible reliability.* The reliability of machine checked proofs can be summarized as follows.

> Proof-objects may be large, possibly several Mb; but they are self-evident. This means that a small program can verify them; the program just follows whether locally the correct steps are being made.

We can summarize the type theoretic approach to interactive theorem proving as follows.

$$\begin{array}{rcl} \text{provability of formula } A & = & \text{`inhabitation' of the type } A \\ \text{proof checking} & = & \text{type checking} \\ \text{interactive theorem proving} & = & \text{interactive construction of a term of} \\ & & \text{a given type.} \end{array}$$

So the decidability of type checking is at the core of the type-theoretic approach to theorem proving.

2.2. Propositions as types

It is possible to represent proofs in a different and more efficient way as formal terms. The intuition behind this is inspired by intuitionistic (constructive) logic. In this philosophy a proof of an implication $A \supset B$ is a method that transforms a proof of A into a proof of B. A proof of $A \mathbin{\&} B$ is a pair $\langle p, q\rangle$ such that p is a proof of A and q one of B. A proof of $A \vee B$ is a pair $\langle b, p\rangle$, where b is either 0 or 1 and if $b = 0$, then p is a proof of A; if $b = 1$ then p is a proof of B. There is no proof of $\perp$, the false proposition. A proof of $\forall x{\in}X.Ax$ is a method p that transforms every element $a{\in}A$ into a proof of Aa. Finally a proof of $\exists x{\in}X.Ax$ is a pair $\langle a, p\rangle$ such that $a{\in}A$ and p is a proof of Aa. Here, $\supset, \&, \vee, \perp, \forall$ and $\exists$ are the usual logical connectives and quantifiers. Negation is defined as $\neg A = A \supset \perp$.

The propositions as types interpretation intuitively can be defined as follows. A sentence A is interpreted as $[A]$, defined as the collection of proofs of A. Then, according to the intuitionistic interpretation of the logical connectives one has

$$\begin{array}{rcl} [A \supset B] & = & [A] \to [B] \\ [A \mathbin{\&} B] & = & [A] \times [B] \\ [A \vee B] & = & [A] \cup [B] \\ [\perp] & = & \emptyset \\ [\forall x{\in}X.Ax] & = & \Pi x{:}X.[Ax] \\ [\exists x{\in}X.Ax] & = & \Sigma x{:}X.[Ax] \end{array}$$

The operations $\to, \times$ and $\cup$ are respectively the formation of functions spaces, Cartesian products and disjoint unions. Intuitively this means the following.

$$\begin{array}{rcl} P \to Q & = & \{f \mid \forall p{:}P.f(p) : Q\}; \\ P \times Q & = & \{\langle p, q\rangle \mid p{:}P \text{ and } q{:}Q\}; \\ P \cup Q & = & \{\langle 0, p\rangle \mid p{:}P\} \cup \{\langle 1, q\rangle \mid q{:}Q\}. \end{array}$$

Furthermore, $\emptyset$ is the empty type. Finally, the (Cartesian) product and sum of a family $\{Px\}_{x:A}$ of types are intuitively defined as

$$\begin{aligned}\Pi x{:}A.Px &= \{f{:}(A \to \cup_{x:A} Px) \mid \forall x{:}A\ (fx : Px)\}\\ \Sigma x{:}A.Px &= \{(x,p) \mid x{:}A \text{ and } p{:}(Px)\}.\end{aligned}$$

Now, a statement A is provable if $[A]$ is inhabited, i.e. if there is a p such that $p : A$ holds in type theory.

2.3. *Examples of proofs as terms*

To get an idea of what proof-objects really look like and how type checking works, we look at an example: we construct a proof-object and type-check it. This example should be understandable without any further knowledge of the typing rules: some basic 'programmers' intuition of types should suffice.

The first non-trivial example in predicate logic is the proposition that a binary antisymmetric relation is irreflexive.

Let X be a set and let R be a binary relation on X. Suppose

$$\forall x, y{\in}X.Rxy \supset \neg Ryx.$$

Then $\forall x{\in}X.\neg Rxx$.

We want to formalize this. In the type theory we have two *universes*, Set and Prop. The idea is that a term X of type Set, notation $X{:}\mathsf{Set}$, is a type that represents a *domain* of the logic. (In logic one also speaks of *sorts* or just *sets*.) A term $A{:}\mathsf{Prop}$, is a type that represents a *proposition* of the logic, the idea being that A is identified with the type of its proofs. So A is provable if we can find a term $p : A$.

Based on this idea, a predicate on $X(:\mathsf{Set})$ is represented by a term $P : X{\to}\mathsf{Prop}$. This can be understood as follows.

$t(:X)$ satisfies the predicate P iff the type Pt is inhabited,

i.e. there is a proof-term of type Pt. So the collection of predicates over X is represented as $X{\to}\mathsf{Prop}$ and similarly, the collection of binary relations over X is represented as $X{\to}(X{\to}\mathsf{Prop})$.

One of the basic operations of mathematics (even though it is not formally treated in ordinary logic!) is *defining*. This is formally captured in type theory via a kind of 'let' construction. Let us give some definitions.

$$\begin{aligned}\mathsf{Rel} &:= \lambda X{:}\mathsf{Set}.X{\to}(X{\to}\mathsf{Prop}),\\ \mathsf{AntiSym} &:= \lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\,X).\forall x,y{:}X.(Rxy) \supset ((Ryx) \supset \bot),\\ \mathsf{Irrefl} &:= \lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\,X).\forall x{:}X.(Rxx) \supset \bot.\end{aligned}$$

These definitions are *formal constructions* in type theory with a computational behavior, so-called δ-reduction, by which definitions are unfolded. Rel takes a domain

X and returns the domain of binary relations on X:

$$\begin{array}{lll}(\mathsf{Rel}\,X) & \to_\delta & (\lambda X{:}\mathsf{Set}.X{\to}(X{\to}\mathsf{Prop}))X \\ & \to_\beta & X{\to}(X{\to}\mathsf{Prop}).\end{array}$$

So by one definition unfolding and one β-step we find that $(\mathsf{Rel}\,X) = X{\to}(X{\to}\mathsf{Prop})$. Similarly, for $X : \mathsf{Set}$ and $Q : X{\to}(X{\to}\mathsf{Prop})$,

$$\begin{array}{lll}(\mathsf{AntiSym}\,XQ) & = & \forall x,y{:}X.(Qxy) \supset ((Qyx) \supset \bot), \\ (\mathsf{Irrefl}\,XQ) & = & \forall x{:}X.(Qxx) \supset \bot.\end{array}$$

The type of $\mathsf{AntiSym}$ is $\Pi X{:}\mathsf{Set}.(X{\to}(X{\to}\mathsf{Prop})){\to}\mathsf{Prop}$, the type of operators that, given a set X and a binary relation over this X, return a proposition. Here we encounter a *dependent type*, i.e. a type of functions f where the range-set depends on the input value. See the previous Section for a set-theoretic understanding.
The formula $\forall x,y{:}X.(Qxy) \supset ((Qyx) \supset \bot)$ is translated as the dependent function type

$$\Pi x,y{:}X.(Qxy){\to}((Qyx){\to}\bot).$$

(For now, we take $\bot$ to be some fixed closed term of type Prop.) Given the (informal) explanation of the Π-type given before, we observe the following two rules for term-construction related to the dependent functions type.

- If $F : \Pi x{:}A.B$ and $N : A$, then $FN : B[N/x]$, (B with N substituted for x).
- If $M : B$ under the assumption $x : A$ (where x may possibly occur in M or B), then $\lambda x{:}A.M : \Pi x{:}A.B$

Let's now try to prove that anti-symmetry implies irreflexivity for binary relations R. So, we try to find a proof-term of type

$$\Pi X{:}\mathsf{Set}.\Pi R{:}(\mathsf{Rel}\,X)(\mathsf{AntiSym}\,X\,R){\to}(\mathsf{Irrefl}\,X\,R).$$

We claim that the term

$$\lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\,X).\lambda h{:}(\mathsf{AntiSym}\,X\,R).\lambda x{:}X.\lambda q{:}(Rxx).hxxqq$$

is a term of this type. We have encountered a TCP; the verification of our claim is performed by the type-checking algorithm. Most type-checking algorithms work as follows:

1. First solve the TSP
   (compute a type C of the term
   $\lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\,X).\lambda h{:}(\mathsf{AntiSym}\,X\,R).\lambda x{:}X.\lambda q{:}(Rxx).hxxqq$),
2. Then compare the computed type with the given type
   (check if $C =_{\beta\delta} \Pi X{:}\mathsf{Set}.\Pi R{:}(\mathsf{Rel}\,X)(\mathsf{AntiSym}\,X\,R){\to}(\mathsf{Irrefl}\,X\,R)$).

So a TCP is solved by solving a TSP and checking an equality. Note that this method is only complete if types are *unique up to equality*: if M has type A and type B, then $A =_{\beta\delta} B$. For the algorithm to terminate we must assure that TSP and equality checking are decidable.

For our example we solve the TSP step by step; there are two main steps

1. For a λ-abstraction $\lambda x{:}X.M$, we first compute the type of M, under the extra condition that x has type X. Say we find B as type for M. Then $\lambda x{:}X.M$ receives the type $\Pi x{:}X.B$.
2. For an application FN, we first compute the type of N. Say we find A as type for N. Then we compute the type of F, say C. Now we check whether C reduces to a term of the form $\Pi x{:}D.B$. If so, we check if $D =_{\beta\delta} C$. If this is the case, FN receives the type $B[N/x]$.

If a check fails, we return 'false', meaning that the term has no type.
For our example term $\lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\,X).\lambda h{:}(\mathsf{AntiSym}\,X R).\lambda x{:}X.\lambda q{:}(Rxx).hxxqq$, we compute the type

$$\Pi X{:}\mathsf{Set}.\Pi R{:}(\mathsf{Rel}\,X).\Pi h{:}(\mathsf{AntiSym}\,X R).\Pi x{:}X.\Pi q{:}(Rxx).C,$$

with C the type of $hxxqq$ under the conditions $X{:}\mathsf{Set}$, $R{:}(\mathsf{Rel}\,X)$, $h{:}(\mathsf{AntiSym}\,X R)$, $x{:}X$ and $q{:}(Rxx)$. Now, $h : (\mathsf{AntiSym}\,X R)$, which should be applied to x, of type X. We reduce $(\mathsf{AntiSym}\,X R)$ until we obtain $\Pi x, y{:}X.(Rxy)\rightarrow((Ryx)\rightarrow\bot)$. So, hx receives the type $\Pi y{:}X.(Rxy)\rightarrow((Ryx)\rightarrow\bot)$. The term hx has a Π-type with the right domain (X), so it can be applied to x, obtaining

$$hxx : (Rxx)\rightarrow((Rxx)\rightarrow\bot).$$

This again can be applied to q (twice), obtaining $hxxqq : \bot$, so TSP finds as type

$$\Pi X{:}\mathsf{Set}.\Pi R{:}(\mathsf{Rel}\,X).\Pi h{:}(\mathsf{AntiSym}\,X R).\Pi x{:}X.\Pi q{:}(Rxx).\bot.$$

We easily verify that this type is $\beta\delta$-convertible with the desired type and conclude that indeed

$$\begin{aligned}\lambda X{:}\mathsf{Set}.\lambda R{:}(\mathsf{Rel}\,X).\lambda h{:}(\mathsf{AntiSym}\,X R).\lambda x{:}X.\lambda q{:}(Rxx).hxxqq \quad &: \\ \Pi X{:}\mathsf{Set}.\Pi R{:}(\mathsf{Rel}\,X).(\mathsf{AntiSym}\,X R)\rightarrow(\mathsf{Irrefl}\,X R).&\end{aligned}$$

By convention, $\forall$ and Π will often be used as synonymous, and similarly $\supset$ and $\rightarrow$.

¿From this example, one can get a rough idea of how type synthesis works: the structure of the term dictates the form of the type that is synthesized. For the type synthesis algorithm to terminate we need the convertibility $=_{\beta\delta}$ to be decidable. This is usually established by proving that $\beta\delta$-reduction is Normalizing (every term M $\beta\delta$-reduces to a normal form) and Confluent (if M $\beta\delta$-reduces to both P_1 and P_2, then there is a Q such that both P_1 and P_2 $\beta\delta$-reduce to Q). Then the question "$M{=_{\beta\delta}}N$?" can be decided by reducing both M and N to normal form and comparing these terms lexically. It should be pointed out here that comparing normal forms is often a very inefficient procedure for checking convertibility. (See [Coquand 1991] for a different approach to checking conversion in a dependent type theory.) Therefore, the convertibility checking algorithm will reduce only if necessary. (There is always a 'worst case' where we really have to go all the way to the normal forms.) In particular, this means that definitions are unfolded as little as possible: although the real complexity of $=_{\beta\delta}$ is in the β-reductions, the definitions 'hide' most of the β-redexes. This can be seen from the fact that proof-terms are almost always in β-normal form (but certainly not in δ-normal form). See [COQ 1999] and [van Benthem Jutting, McKinna and Pollack 1994] for more information on type-checking and checking convertibility in dependent type theories. In Section 3.2 we discuss in detail a type-checking algorithm for one specific type system.

2.4. Intermezzo: Logical frameworks

What has been described in the previous two Sections is sometimes called the *direct* encoding of logic in type theory. The logical constructions (connectives) each have a counterpart in the type theory, *implication*, for example, is mirrored by the *arrow type* in type theory. Moreover, the elimination and introduction rules for a connective also have their counterpart in type theory (λ-abstraction mirrors implication introduction and application mirrors implication elimination). In the rest of this paper we restrict ourselves to this direct encoding. There is, however, a second way of interpreting logic in type theory, which is called the *logical frameworks* encoding or also the *shallow* encoding. As the name already indicates, the type theory is then used as a *logical framework*, a meta system for encoding a specific logic one wants to work with. The encoding of a logic L is done by choosing an appropriate context Γ_L, in which the language of L (including the connectives) and the proof rules are declared. This context is usually called a *signature*. In the direct encoding, a context is used for declaring variables (e.g. declaring that the variable x is of domain A) or for making assumptions (by declaring $z : \varphi$, for φ a proposition, we *assume* φ). In logical frameworks, the context is used also to 'declare' the logic itself. One of the reasons that (even rather simple) type systems provide a very powerful logical framework is that type theory is very accurate in dealing with variables (binding, substitution, α-conversion). Hence, when encoding a logic, all issues dealing with variables can be left to the type theory: the logical framework is used as the underlying calculus for substitution and binding. How this works precisely is illustrated by three small examples. For further details on logical frameworks we refer to [Pfenning 2001] (Chapter 17 of this Handbook) or to [Harper, Honsell and Plotkin 1993, Pfenning 1991, de Bruijn 1980]. It should also be remarked here that, even though we do not treat the technical details of logical frameworks based on type theory and the encoding of logics in them, much of our discussions also apply to these type systems, notably the issue of type checking. We now recapitulate the main differences between the two encodings.

Direct encoding	Shallow encoding
One type system $\sim$ One logic	One type system $\sim$ Many logics
Logical rules $\sim$ type theoretic rules	Logical rules $\sim$ Context declarations

The encoding of logics in a logical framework based on type theory will be shown by giving three examples

1. The $\{\supset\}$-fragment of minimal propositional logic,
2. The $\{\supset,\forall\}$-fragment of minimal predicate logic,
3. The untyped λ-calculus.

Minimal propositional logic
The formulas are built up from atomic ones using implication ($\supset$) as only logical operator. In order to translate propositions as types, one postulates the 'signature':

$$\mathsf{prop} : \mathtt{type} \tag{2.1}$$

$$\mathsf{imp} : \mathsf{prop}{\to}\mathsf{prop}{\to}\mathsf{prop} \tag{2.2}$$

Now define the encoding of propositions $[-]$ as follows.

$$[A \supset B] = \mathsf{imp}[A][B].$$

Then one has for example $[A \supset A] = \mathsf{imp}[A][A]$ and $[A \supset A \supset B] = \mathsf{imp}[A](\mathsf{imp}[A][B])$. The type prop can be seen as the type of 'names' of propositions: a term of type prop is not a proposition itself, because it can not be inhabited (i.e. proved), as it is not a type. In order to state that e.g. $[A \supset A]$ is valid, one intoduces the following map:

$$\mathsf{T} \quad : \quad \mathsf{prop}{\to}\mathtt{type}. \tag{2.3}$$

The intended meaning of $\mathsf{T}p$ is 'the collection (type) of proofs of p', so T maps a 'name' of a proposition to the type of its proofs. Therefore it is natural to interpret 'p is valid' by '$\mathsf{T}p$ is inhabited'. In order to show now that tautologies like $A \supset A$ are valid in this sense (after translation), one postulates

$$\mathsf{imp\_intr} \quad : \quad \Pi p, q : \mathsf{prop}.(\mathsf{T}p{\to}\mathsf{T}q){\to}\mathsf{T}(\mathsf{imp}\ p\ q), \tag{2.4}$$

$$\mathsf{imp\_el} \quad : \quad \Pi p, q : \mathsf{prop}.\mathsf{T}(\mathsf{imp}\ p\ q){\to}\mathsf{T}p{\to}\mathsf{T}q. \tag{2.5}$$

Then indeed the translation of e.g. $A \supset A$, which is $\mathsf{imp}[A][A]$, becomes valid:

$$\mathsf{imp\_intr}[A][A](\lambda x{:}\mathsf{T}[A].x) : \mathsf{T}(\mathsf{imp}[A][A]),$$

since clearly $(\lambda x{:}\mathsf{T}[A].x) : (\mathsf{T}[A]{\to}\mathsf{T}[A])$. Similarly one can construct proofs for other tautologies (e.g. $(A \supset A \supset B) \supset A \supset B$. In fact one can show by an easy induction on derivations in the logic L that

$$\vdash_{\mathrm{PROP}} A \quad \Rightarrow \quad \Sigma_{\mathrm{PROP}}, a_1{:}\mathsf{prop}, \ldots, a_n{:}\mathsf{prop} \vdash p : \mathsf{T}[A], \text{ for some } p.$$

Here $\{a_, \ldots, a_n\}$ is the set of basic proposition symbols in A and $\Sigma_{\mathtt{PROP}}$ is the *signature* of our minimal propositional logic PROP, i.e. the set of declarations (1-5). Property (6) is called *adequacy* or soundness of the encoding. The converse of it, *faithfulness* (or completeness), is also valid, but more involved to prove.

Minimal predicate logic

We consider the $\{\supset, \forall\}$-fragment of (one-sorted) predicate logic. Suppose we have a logical signature with one constant, one unary function and one binary relation. This amounts to the following (first part of the) type theoretic signature.

$$\begin{aligned}
\mathsf{prop} &: \mathtt{type}, && (2.6)\\
\mathsf{A} &: \mathtt{type}, && (2.7)\\
\mathsf{c} &: \mathsf{A}, && (2.8)\\
\mathsf{f} &: \mathsf{A}\to\mathsf{A}, && (2.9)\\
\mathsf{R} &: \mathsf{A}\to\mathsf{A}\to\mathsf{prop}, && (2.10)\\
\mathsf{imp} &: \mathsf{prop}\to\mathsf{prop}\to\mathsf{prop}, && (2.11)\\
\mathsf{imp_intr} &: \Pi p, q : \mathsf{prop}.(\mathsf{T}p\to\mathsf{T}q)\to\mathsf{T}(\mathsf{imp}\ p\ q), && (2.12)\\
\mathsf{imp_el} &: \Pi p, q : \mathsf{prop}.\mathsf{T}(\mathsf{imp}\ p\ q)\to\mathsf{T}p\to\mathsf{T}q. && (2.13)
\end{aligned}$$

This covers the language and the implicational part (copied from the logic PROP). Now one has to encode $\forall$, which is done by observing that $\forall$ takes a function from A to prop, $\forall : (\mathsf{A}\to\mathsf{prop})\to\mathsf{prop}$. The introduction and elimination rules for $\forall$ are then remarkably straightforward.

$$\begin{aligned}
\mathsf{forall} &: (\mathsf{A}\to\mathsf{prop})\to\mathsf{prop}, && (2.14)\\
\mathsf{forall_intr} &: \Pi P{:}\mathsf{A}\to\mathsf{prop}.(\Pi x{:}\mathsf{A}.\mathsf{T}(Px))\to\mathsf{T}(\mathsf{forall}P), && (2.15)\\
\mathsf{forall_elim} &: \Pi P{:}\mathsf{A}\to\mathsf{prop}.\mathsf{T}(\mathsf{forall}P)\to\Pi x{:}\mathsf{A}.\mathsf{T}(Px). && (2.16)
\end{aligned}$$

Now we translate universal quantification as follows.

$$[\forall x{:}A.Px] = \mathsf{forall}(\lambda x{:}A.[Px]).$$

The proof of an implication like

$$\forall z{:}A(\forall x, y{:}A.Rxy) \supset Rzz$$

is now mirrored by the proof-term

$$\mathsf{forall\_intr}[\_](\lambda z{:}\mathsf{A}.\mathsf{imp\_intr}[\_][\_](\lambda h{:}\mathsf{T}([\forall x, y{:}A.Rxy]).\mathsf{forall\_elim}[\_](\mathsf{forall\_elim}[\_]hz))),$$

where we have replaced – for readability – the instantiations of the Π-type by $[\_]$. This term is of type

$$\mathsf{forall}(\lambda z{:}\mathsf{A}.\mathsf{imp}(\mathsf{forall}(\lambda x{:}\mathsf{A}.(\mathsf{forall}(\lambda y{:}\mathsf{A}.\mathsf{R}xy))))(\mathsf{R}zz)).$$

Again one can prove adequacy

$$\vdash_{\text{PRED}} \varphi \quad\Rightarrow\quad \Sigma_{\text{PRED}}, x_1{:}\mathsf{A}, \ldots, x_n{:}\mathsf{A} \vdash p : T[\varphi], \text{ for some } p,$$

where $\{x_1, \ldots, x_n\}$ is the set of free variables in φ and Σ_{PRED} is the signature consisting of the declarations (6–16). Faithfulness can be proved as well.

Untyped λ-calculus
Perhaps more unexpected is that untyped λ-calculus can be modeled in a rather simple type theory (the same as for PRED and PROP). The needed signature Σ_{lambda} now is

$$\mathsf{D} : \mathtt{type}; \tag{2.17}$$

$$\mathsf{app} : \mathsf{D}\to(\mathsf{D}\to\mathsf{D}); \tag{2.18}$$

$$\mathsf{abs} : (\mathsf{D}\to\mathsf{D})\to\mathsf{D}. \tag{2.19}$$

Now every variable x in the λ-calculus is represented by the variable $x : \mathsf{D}$ in the type system. The translation of untyped λ-terms is defined as follows.

$$\begin{aligned} [x] &= x; \\ [PQ] &= \mathsf{app}\,[P]\,[Q]; \\ [\lambda x.P] &= \mathsf{abs}\,(\lambda x{:}\mathsf{D}.[P]). \end{aligned}$$

We now have to express that e.g. $(\lambda x.x)y = y$, and then we have to prove that this equality is valid. As to the statement of equalities, one declares a term

$$\mathsf{eq}{:}\mathsf{D}\to\mathsf{D}\to\mathtt{type}. \tag{2.20}$$

The λ-calculus equation $P = Q$ is now translated as the type $\mathsf{eq}\ [P]\ [Q]$. The validity of this equation is by definition equivalent to the inhabitation of this type. In order to ensure this we need the following axioms.

$$\mathsf{refl} : \Pi x{:}\mathsf{D}.\mathsf{eq}\ x\ x, \tag{2.21}$$

$$\mathsf{sym} : \Pi x,y{:}\mathsf{D}.\mathsf{eq}\ x\ y\to\mathsf{eq}\ y\ x, \tag{2.22}$$

$$\mathsf{trans} : \Pi x,y,z{:}\mathsf{D}.\mathsf{eq}\ x\ y\to\mathsf{eq}\ y\ z\to\mathsf{eq}\ x\ z, \tag{2.23}$$

$$\mathsf{mon} : \Pi x,x',z,z'{:}\mathsf{D}.\mathsf{eq}xx'\to\mathsf{eq}zz'\to\mathsf{eq}(\mathsf{app}\ z\ x)(\mathsf{app}\ z'\ x'), \tag{2.24}$$

$$\mathsf{xi} : \Pi F,G{:}\mathsf{D}\to\mathsf{D}.(\Pi x{:}\mathsf{D}.\mathsf{eq}(Fx)(Gx))\to\mathsf{eq}(\mathsf{abs}\ F)(\mathsf{abs}\ G), \tag{2.25}$$

$$\mathsf{beta} : \Pi F{:}\mathsf{D}\to\mathsf{D}.\Pi x{:}\mathsf{D}.\mathsf{eq}(\mathsf{app}(\mathsf{abs}\ F)x)(Fx). \tag{2.26}$$

Now one can proof the adequacy

$$P =_\beta Q \;\Rightarrow\; \Sigma_{\mathsf{lambda}}, x_1{:}\mathsf{D},\ldots,x_n{:}\mathsf{D} \vdash p : \mathsf{eq}\ [P]\ [Q],\ \text{for some } p.$$

Here, $x_1,\ldots,x_n$ is the list of free variables in PQ and Σ_{lambda} is the signature for untyped λ-calculus, consisting of declarations (17–26). Again the opposite implication, faithfulness, also holds.

The three examples show that using type theories as logical framework is flexible, but somewhat tiresome. Everything has to be spelled out. Of course, in a concrete implementation this can be overcome by having some of the arguments inferred automatically. Note that for each formalization the faithfulness has to be proved separately.

2.5. Functions: algorithms versus graphs

In type theory there is a type of *functions* $A{\to}B$, for A and B types. Which functions there are depends on the derivation rules that tell us how to construct functions. Usually (certainly for the systems in this paper) we see three ways of constructing functions.

- Axiomatically declare $f : A{\to}B$ for a new symbol f.
- Given that $M : B$ in a context containing $x : A$ (and no other dependencies on x in the context), we construct, using the λ-rule,

$$\lambda x{:}A.M : A{\to}B.$$

- Via primitive recursion: given $b : B$ and $f : \mathsf{nat}{\to}B{\to}B$ we can construct

$$\mathsf{Rec}\ b\, f : \mathsf{nat}{\to}B.$$

These functions also *compute*: there are reduction rules associated to them, the β and ι rules:

$$\begin{array}{rcl}(\lambda x{:}A.M)N & \to_\beta & M[N/x],\\ \mathsf{Rec}\ b\, f\, 0 & \to_\iota & b,\\ \mathsf{Rec}\ b\, f\, (\mathsf{S}^+\, x) & \to_\iota & f\, x\, (\mathsf{Rec}\ b\, f\, x).\end{array}$$

So, terms of type $A{\to}B$ denote *algorithms*, whose operational semantics is given by the reduction rules. In this view we can see a declaration $f : A{\to}B$ as an 'unknown' algorithm.

At the same time the set-theoretic concept of a *function as a graph* is also present in type theory. If $R : A{\to}B{\to}\mathsf{Prop}$ (R is a binary relation over A and B) and we have a proof-term of type $\forall x{:}A.\exists! y{:}B.Rxy$, then we can of course view this R as a function (graph) in the set-theoretic way. Note, however, that we have no way of really talking about the 'R-image' of a given $a : A$, because we can't give it a name (like $f(a)$). In terms of formal logic, the only way to use it is under an $\exists$-elimination, where we have given the y a name – locally – and we know it to be unique. So the set-theoretic concept of 'function' doesn't give us an algorithm that computes. To remedy this situation one can add a constant – Church [1940] uses the ι for this – that extracts a 'witness' from a predicate. In Church's higher order logic, if P is a predicate over A (i.e. $P : A{\to}\mathsf{Prop}$ in type-theoretical terms), then $\iota P : A$ and there is an axiom saying $\forall P{:}A{\to}\mathsf{Prop}(\exists! x{:}A.Px) \to P(\iota P)$. So, if there is a unique element for which P holds, then ιP denotes this element, otherwise ιP is an arbitrary unspecified element. Obviously, the latter aspect of the ι is not so nice, especially in a system with inductive types like nat, where we now will encounter closed terms of type nat (e.g. $\iota(\geq 0)$) that are not in constructor form, i.e. equal to $S^n 0$ for some $n{\in}\mathbb{N}$.

In constructive systems, there is a different way to obtain a 'witness' from a proof of an existential statement: if $\forall x{:}A\exists y{:}B.Rxy$ holds constructively, then there is a function (algorithm) that computes the y from the x. (This could almost be taken as a definition of what it means for a logic to be constructive.)

> If p is a closed proof-term of type $\forall x{:}A\exists y{:}B.Rxy$, then p contains a term $f{:}A{\rightarrow}B$ and a proof-term of type $\forall x{:}A.Rx(fx)$.

Note that this is a *meta-theoretic* property of constructive systems: there is not (necessarily) a function *inside* the system that extracts the $f{:}A{\rightarrow}B$ from the proof-term p. In some systems, most notably the constructive type theories of Martin-Löf ([Martin-Löf 1984], [Nordström et al. 1990]), this property has been internalised by interpreting an existential fomula $\exists y{:}B.\varphi$ as a Σ-type $\Sigma y{:}B.\varphi$, consisting of pairs $\langle b, q\rangle$ with $b : B$ and $q : \varphi[b/x]$. So, the only way to construct a term of the Σ-type $\Sigma y{:}B.\varphi$ is by giving a $b : B$ for which $\varphi[b/x]$ holds. From a term $t : \Sigma y{:}B.\varphi$, one can extract the two components by projections: $\pi_1 t : B$ and $\pi_2 t : \varphi[\pi_1 t/x]$. These are the Σ-introduction and the Σ-elimination rules, respectively. This implies that from a proof-term $p : \forall x{:}A\exists y{:}B.Rxy$, we can immediately extract the *function* $f : A{\rightarrow}B$ defined by $\lambda x{:}A.\pi_1(px)$ and we can prove for this f that $\forall x{:}A.Rx(fx)$ holds. (The proof-term is $\lambda x{:}A.\pi_2(px)$.) The extracted function also has a proper computational behavior: a closed proof-term $p : \forall x{:}A\exists y{:}B.Rxy$ has the form $\lambda x{:}A.\langle t, q\rangle$; the function extracted from this p is (indeed) $\lambda x{:}A.t$.

The internalisation of the (constructive) existence property via a Σ-type may seem a neat way to solve the problem of 'functional-relations-not-being-functions'. However, every advantage has its disadvantage, in this case that we loose the immediate connection between type theory and logic. The reason is that with the Σ-type we can construct *objects that depend on proofs*, a feature alien to ordinary logic. The simplest example is where we have a proof p of $\Sigma x{:}\mathsf{nat}.A$, from which we get the object $\pi_1 p : \mathsf{nat}$. Ordinary logic is built up in *stages*, where

- in the first stage one defines what the domains and the terms of the domains are;
- in the second stage one defines the formulas (or one singles out the formulas from the collection of terms);
- in the third stage one defines what a proof is.

This built-up makes it impossible for objects to depend on proofs, for the simple reason that the objects were already there before we even thought about proofs. Note that Church' approach, using the ι operator, conforms with the conception of ordinary logic that we have just sketched: the object ιP does *not* depend on the proof of $\exists ! y{:}A.Px$, but only on the object P. Choosing a type theory in which objects do not depend on proofs has some clear advantages if we want to explain and understand the system in terms of ordinary logic. We come back to this later in 2.9. Here we just remark that if a type theory is to be used as a basis for a theorem prover, a clear connection to some well-known standard logic is desirable.

We conclude that, if we look at functions in type theory, there is a clear distinction between algorithms ($f : A{\rightarrow}B$) and graphs ($R : A{\rightarrow}B{\rightarrow}\mathsf{Prop}$ such that $\forall x{:}A.\exists ! y(Rxy)$ holds). Even if we allow to extract from a proof of $\forall x{:}A.\exists ! y(Rxy)$ an $f : A{\rightarrow}B$, there is still a clear distinction: the proof is not the same as the function.

2.6. Subject Reduction

The property of Subject Reduction (SR) can be seen as the 'sine qua non' of type theory. It states that the set of typed terms of a given type A is closed under reduction. More formally: if $M : A$ and $M \twoheadrightarrow N$, then $N : A$. For A representing a data type, we can understand this as saying that A is closed under evaluation. The rules for evaluation are β, δ and ι that we have already encountered. We illustrate the use of reduction by an example.

Suppose we have as definition $\mathsf{plus} := \lambda x, y{:}\mathsf{nat}.\mathsf{Rec}\, x(\lambda z{:}\mathsf{nat}.\mathsf{S}^+)y$. Then the 'value' of the expression $\mathsf{plus}\,1\,0$ is computed by first unfolding the plus (one δ-reduction step), then performing two β-steps and then one ι-step, to obtain 1. The Subject Reduction property says that all expressions in this computation are of type nat.

In a proof-term, reduction captures the well-known proof-theoretical notion of *cut-elimination*. A *cut* in a proof is a situation where an introduction rule (I) for a connective is immediately followed by an elimination rule (E) for that connective. It is then possible to make a 'shortcut', eliminating the consecutive application of the (I) rule and the (E) rule. (Note that this may not always make the proof literally shorter.) Suppose we have the proof-term $\lambda h{:}A{\to}A{\to}B.\lambda z{:}A.hzz : (A{\to}A{\to}B){\to}(A{\to}B)$, corresponding to the standard natural deduction proof of this fact, ending with an introduction rule. Now, if we also have a proof $q : A{\to}A{\to}B$ we can eliminate the implication obtaining $(\lambda h{:}A{\to}A{\to}B.\lambda z{:}A.hzz)q : A{\to}B$. If we do one β-step we eliminate the cut obtaining the proof-term $\lambda z{:}A.qzz : A{\to}B$. So, for proof-terms,

> the Subject Reduction property states that *cut-elimination is correct* in the sense that if p is a proof of A and we obtain p' by eliminating some cuts from p, then also p' is a proof of A.

In practice, we seldom wish to perform β-reduction on proof-terms: once we have proved a result (i.e. we have constructed a term $p : A$), we are mainly interested in its statement (the type A) and the fact that there is *some* proof (inhabitant) of it. The proof is only inspected if we want to study its structure (e.g. to try to reuse it for proving similar statements). The actual situation is that once we have proved a lemma, say we have constructed $\lambda h{:}A{\to}A{\to}B.\lambda z{:}A.hzz : (A{\to}A{\to}B){\to}(A{\to}B)$ as above, we will *save* this lemma under a name, say lemma_1, and we will only refer to this 'name' lemma_1. In type theory, what happens is that we introduce a definition $\mathsf{lemma}_1 := \lambda h{:}A{\to}A{\to}B.\lambda z{:}A.hzz$ and we use lemma_1 as a constant of type $(A{\to}A{\to}B){\to}(A{\to}B)$. It is a *defined* constant, but in implementations it will be *opaque*, meaning that it will never be unfolded by δ.

2.7. Conversion and Computation

We have already encountered three notions of computation: β-, ι- and δ-reduction. For most type theories these reduction relations together are confluent and normalizing, yielding a decidable *conversion* relation $=_{\beta\iota\delta}$ on the set of well-typed terms. This decidability also makes the type checking algorithm work, see Section 2.1. We will look more closely at the use of conversion.

Suppose again we have the definition of plus as given above and we want to prove $2 > 0$ from $p : \forall x, y, z{:}\mathsf{nat}.(x > (\mathsf{plus}\, y z)) \to (x > z)$ and $q : 2 > 1$. Now $p\,2\,1\,0 : (2 > (\mathsf{plus}\,1\,0)) \to (2 > 0)$ and we want to apply this proof to q to obtain the proof-term $p\,2\,1\,0\,q : (2 > 0)$. The application can only work if we first reduce the type $(2 > (\mathsf{plus}\,1\,0)) \to (2 > 0)$ to $(2 > 1) \to (2 > 0)$, which is done by one δ-reduction (unfolding the definition of plus), two β-steps and a ι-step. We can depict this in a deduction as follows.

$$\dfrac{\dfrac{\dfrac{p : \forall x, y, z{:}\mathsf{nat}.(x > (\mathsf{plus}\, y z)) \to (x > z)}{p\,2\,1\,0 : (2 > (\mathsf{plus}\,1\,0)) \to (2 > 0)}}{p\,2\,1\,0 : (2 > 1) \to (2 > 0)}\,(\mathrm{conv}) \qquad q : (2 > 1)}{p\,2\,1\,0\,q : (2 > 0)}$$

Here we see an application of the conversion rule:

$$(\mathrm{conv})\,\frac{M : \varphi \quad \psi : \mathsf{Prop}}{M : \psi}\ \text{if } \varphi =_{\beta\iota\delta} \psi$$

In the example above, M is $p\,2\,1\,0$, φ is $(2 > (\mathsf{plus}\,1\,0)) \to (2 > 0)$ and ψ is $(2 > 1) \to (2 > 0)$. The proof-term M is left unchanged under the transition from φ to ψ. This poses no problem for the type checking algorithm, because the conversion $=_{\beta\iota\delta}$ is decidable. (So, if we are given a term M and we want to check whether M is of type ψ we only have to check whether M has a type and if so, verify whether it's convertible with ψ.) In case the equality in the side-condition to the conversion rule is not decidable (which is the situation in the type theory of Nuprl, [Constable et al. 1986]), the conversion from type φ to ψ would have to leave a 'trace' in the term M in order to make type checking decidable. (The trace could be the reduction sequence from φ to ψ.) One could also leave a trace of the conversion in order to help the type checking algorithm, but this is usually not done: it makes proof-terms unnecessarily complicated. Moreover we want to follow the so-called *Poincaré principle*, which can be stated intuitively as follows.

> There is a distinction between *computations* and *proofs* and computations do not require a proof.

This implies, for example, that the equality of $\mathsf{plus}\,1\,0$ and 1 does not require a proof: $\mathsf{plus}\,1\,0$ and 1 are computationally equal, so $\mathsf{plus}\,1\,0 = 1$ follows trivially (from the reflexivity of $=$). The power of the Poincaré principle depends on the expressivity of the type theory in terms of algorithms that can be written. Imagine the situation where we have a class of formulas that can be encoded syntactically in our type system. That is, we have a (inductive) type 'Class-of-Form' together with a 'decoding function' $\mathsf{Dec} : \mathsf{Class\text{-}of\text{-}Form} \to \mathsf{Prop}$ such that every formula $T : \mathsf{Prop}$

in our class has a syntactic representation t : $\mathsf{Class\text{-}of\text{-}Form}$ with $\mathsf{Dec}\,t =_{\beta\iota\delta} T$. Suppose that we can write a decision algorithm in our type system, i.e. we have a term $\mathsf{Check} : \mathsf{Class\text{-}of\text{-}Form}{\to}\mathsf{Prop}$ such that if $\mathsf{Check}\,t =_{\beta\iota\delta} \top$, then t encodes a provable formula from our class. ($\top$ is the proposition with one unique proof, true.) In more precise type-theoretic terms: suppose we have a proof-term ok with

$$\mathsf{ok} : \forall t{:}\mathsf{Class\text{-}of\text{-}Form}.(\mathsf{Check}\,t) \to (\mathsf{Dec}\,t).$$

Then, to prove that a formula $T : \mathsf{Prop}$ from our class is provable, we only have to find its encoding $t : \mathsf{Class\text{-}of\text{-}Form}$ and then

$$\mathsf{ok}\,t\,\mathsf{true} : T.$$

if T is indeed provable (inhabited), which can be verified by the type checker. In this example, the main task of the type checker is to execute the algorithm Check. This use of the Poincaré principle shows how automated theorem proving can be done (safely) *inside* type theory. This technique is usually called *reflection* (reflecting (part of) the language in itself). The origins date back to Howe [1988]. It has been used succesfully in the Coq system to write a tactic for deciding equality in ring-structures. See also [Barthe, Ruys and Barendregt 1996] – where it is called the 'two-level approach' – and [Oostdijk and Geuvers 2001]. To get really fast automated theorem proving, it is advisable to use a special purpose automated theorem prover, which has the extra adavantage that one doesn't have to program (and prove correct!) the decision procedures oneself. If one uses reflection (and the Poincaré principle) one obtains a medium fast descision procedure but very reliable proof-terms, which can be checked independently.

2.8. Equality

Note that we have not included η-reduction in the conversion rule, but just β, δ and ι. This may seem remarkable, because for the untyped λ-calculus, many nice results of β-reduction (like confluence) extend to $\beta\eta$. This is however not the case for typed λ-calculus. The snag lies in the fact that our typed terms have a type attached to the bound variable in the λ-abstraction ($\lambda x{:}\underline{A}.M$). This information is crucial for the type checking algorithm (without it, type checking in dependent type theory is undecidable [Dowek 1993]), but it complicates the combination of β and η. For example consider $\lambda x{:}A.(\lambda y{:}B.y)x$,

$$\begin{array}{lll} \lambda x{:}A.(\lambda y{:}B.y)x & \to_\beta & \lambda x{:}A.x \\ \lambda x{:}A.(\lambda y{:}B.y)x & \to_\eta & \lambda y{:}B.y \end{array}$$

The terms on the right hand side have a common reduct only if A and B do. This complication of η was already known to the Automath community [Nederpelt 1973]; Confluence and Normalization for types systems from the Automath family was proved by Daalen [1980]. For a study and proof of the general situation see

[Geuvers 1992], [Geuvers 1993]. For a study of type theory with λ-terms *without* types attached to the bound variables, see [Barthe and Sørensen 2000], where it is shown that the type checking (notably its undecidability) is not completely hopeless. In [Magnusson 1994], an implementation of a proof assistant based on such a type theory (without types attached to the bound variables) is described.

There are several other ways of extending the equality in the conversion rule. A prominent example is the *extensional equality* on functions. In mathematics, if $f, g : A \to B$, the f and g would be considered to be equal if they have the same graph, i.e. $f = g$ iff $\forall x{:}A(f\ x = g\ x)$. If we want to view the functions not so much as algorithms, but more abstractly as graphs, the inclusion of extensional equality in the convertibility (as side condition in the conversion rule) would be very natural. If we want to do this, it is required that we introduce an *equality judgment* of the form

$$\Gamma \vdash M = N : A.$$

Before discussing extensionality further, we first focus on the different notions of equality.

Equality as a judgment or as a type

As rules for deriving an equality judgment we would have β, δ and ι plus the normal rules for making it an equivalence relation (reflexivity, symmetry, transitivity) plus rules for making the equality compatible with the term-constructions. For example, we would have

$$(\beta)\quad \frac{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}C.B \quad \Gamma \vdash N : C}{\Gamma \vdash (\lambda x{:}A.M)N = M[N/x] : B[N/x]}$$

$$(\delta)\quad \frac{\Gamma_1, c := M : A, \Gamma_2 \vdash c : B}{\Gamma \vdash c = M : B}$$

$$(\text{refl})\quad \frac{\Gamma \vdash M : B}{\Gamma \vdash M = M : B}$$

$$(\text{trans})\quad \frac{\Gamma \vdash M = N : B \quad \Gamma \vdash N = P : B}{\Gamma \vdash M = P : B}$$

$$(\text{app-comp})\quad \frac{\Gamma \vdash M = N : \Pi x{:}A.B \quad \Gamma \vdash P = Q : A}{\Gamma \vdash MP = NQ : B[P/x]}$$

$$(\text{abs-comp})\quad \frac{\Gamma, x{:}A \vdash M = N : B \quad \Gamma \vdash A = C : D}{\Gamma \vdash \lambda x{:}A.M = \lambda x{:}D.N : \Pi x{:}A.B}$$

The conversion rule then takes the following form

$$(\text{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A = B}{\Gamma \vdash M : B}$$

The addition of extensionality would amount to the rule

$$\text{(ext)}\ \frac{\Gamma \vdash M, N : A \to B \quad \Gamma \vdash p : \Pi x{:}A.(Mx = Nx)}{\Gamma \vdash M = N : A \to B}$$

In the (ext) rule, the equality in the premise ($=_B$)is an equality that can be proved; we could call it a *logical equality*, but in type-theory it is usually called *book equality*, as it is thought of as the user-defined equality in 'the book'. (In Automath systems, the notion 'book' has a very precise formal meaning; it corresponds roughly to the user-defined context that represents some specific theory.) The equality in the conclusion of the (ext) rule is the 'internal equality' of the type system, usually called the *definitional equality*. This definitional equality can be represented by a judgment itself (as above), but often it is represented as a 'convertibility side condition', like in 2.7. In the latter case, the convertibility $A =_{\beta\delta\iota} B$ is understood as an equality on a set of 'pseudo terms', including the well-typed ones. Let us summarize the different equalities.

1. *Definitional equality.* The 'underlying equality' of the type system. Captures β, δ and, if present, also ι. Can be *judgemental* (i.e. built into the formal system) or a *convertibility side condition*.
2. *Book equality.* The 'equality provable' inside the type system. If $M =_A N$ is a book equality, then it is a type ($M =_A N : \mathsf{Prop}$ for $M, N : A$) and we can try to find a proof-term inhabiting it ($p : M =_A N$). Such an equality can be defined by the user.

Book equality comes in various flavours, depending not only on the user's choice, but also on the type theory, because most type theories (and certainly their implementations) have a 'built-in' or 'preferred' equality. We give a short overview of some options. First of all, we want the following from a book-equality.

- The equality should be an equivalence relation on the carrier type: for $A : \mathsf{Set}$, $=_A : A \to (A \to \mathsf{Prop})$ should be an equivalence relation.
- Substitution property. We want to replace 'equal terms in a proposition'. In type theoretical terminology, we want the following rule to be derivable (for some term construction $S(\_,\_)$).

$$\frac{\Gamma \vdash N : A(t) \quad \Gamma \vdash e : t =_A q}{\Gamma \vdash S(N, e) : A(q)}$$

To achieve this we distinguish the following three treatments of equality.

1. *Leibniz equality*, defined in higher order logic. We want to say (following Leibniz) that $t =_A q$ if for all predicates P over A, P holds for t iff P holds for q. In type theory:
$$t =_A q \quad := \quad \Pi P{:}A \to \mathsf{Prop}(P\ t) \to (P\ q).$$
Note that this equality looks asymmetric; however, it can be shown that $=_A$ is symmetric.

2. *Inductively defined equality.* Equality $=_A$: $A \to (A \to \mathsf{Prop})$ is the 'smallest' reflexive relation on A, i.e. the 'smallest' relation R on A for which $\forall x{:}(R\ x\ x)$ holds. In type theoretic syntax this would look like

$$\begin{aligned} \mathsf{Inductive}\quad & \mathsf{Eq}_A : A \to A \to \mathsf{Prop} := \\ & \mathsf{Refl} : \Pi x{:}A.(\mathsf{Eq}\ x\ x). \end{aligned}$$

   This specific form of definition, to be treated in more detail in Section 3.8, says that Refl is the only *constructor* for the inductively defined relation Eq. This is made precise by an *induction* principle that comes along with this definition.
3. *Special type with special rules*, roughly reflecting the inductivity of $=_A$, as in 2. In Martin-Löf's type theory (see [Martin-Löf 1984], [Nordström et al. 1990]), equality is taken as a basic type constructor:

$$\frac{\Gamma \vdash A : \mathsf{Set}}{\Gamma, x, y{:}A \vdash (\mathsf{Id}_A\ x\ y) : \mathsf{Set}} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash (\mathsf{Refl}_A M) : (\mathsf{Id}_A\ MM)}$$

   We don't give the full elimination (induction) principle, but only one of its instances:

$$\frac{\Gamma \vdash P : A \to \mathsf{Set} \quad \Gamma \vdash q : (Pa) \quad \Gamma \vdash e : (\mathrm{Id}_A\, ab)}{\Gamma \vdash (\mathsf{idrec}\, qe) : (Pb)}$$

Note that in the third approach, the identity type $(\mathsf{Id}_A\, ab)$ is of type Set, and not of type Prop. This is not a peculiar aspect of Martin-Löf's approach to equality, but a consequence of his approach to logic in general: there is no distinction between sets and propositions; both 'live' in the universe Set (and hence there is no universe Prop).

There are some clear differences, e.g. Leibniz equality requires impredicativity to be definable, while the inductively defined equality requires inductive types. However, each of these approaches to equality yields an equivalence relation for which the substitution property holds. Let us discuss one example where the different equalities diverge. Suppose we have defined (inductively) a map $\mathsf{Fin} : \mathsf{nat} \to \mathsf{Set}$ such that $(\mathsf{Fin}\, n)$ represents the n-element type. Then one would like $(\mathsf{Fin}\, n)$ and $(\mathsf{Fin}\, m)$ to be isomorphic if n and m are equal. So we want (at least) the following to be derivable (for some some term construction $E(\_,\_)$).

$$\frac{\Gamma \vdash t : (\mathsf{Fin}\, n) \quad \Gamma \vdash e : n =_{\mathsf{nat}} m}{\Gamma \vdash E(t,e) : (\mathsf{Fin}\, m)}$$

For Leibniz equality ((1) above), we can not construct such a term $E(t,e)$, because it allows elimination 'over Prop' only. For the inductive equality, it depends on the elimination rules that are allowed in the type system (e.g. the type system of COQ [1999] does not allow it). For Martin-Löf's type theory, the above rule is obviously derivable, because Prop and Set are the same universe, and one can eliminate over it.

Extensionality versus intensionality
The definitional equality can be *intensional* or *extensional*. In the first case, we do not have a derivation rule (ext), and hence equality of functions is equality of algorithms. In the second case, we have a derivation rule (ext), and hence equality of functions is equality of graphs.

It follows from our discussion of TCP and TIP in 2.2 that the addition of the rule (ext) renders TCP undecidable. Viz. suppose $H : (A{\to}B){\to}\mathsf{Prop}$ and we know $x : (H\ f)$; then $x : (H\ g)$ iff there is a term of type $\Pi x{:}A.f\ x = g\ x$. So for this TCP to be solvable, we need to solve a TIP.

The first type systems by Martin-Löf (see [Martin-Löf 1984]) were extensional. Later he rejected extensionality, because of the implied undecidability of type checking. The interactive theorem prover Nuprl of Constable et al. [1986] is based on extensional type theory. It is clear that from a more classical view on mathematics (identifying functions with graphs in set-theoretic way), extensionality is very desirable. Recently, work has been done (mainly by Hofmann [1994]) showing how to encode (or explain) extensional equality in an intensional type theory. The idea is to translate an extensional type to a pair consisting of an intensional type and an equivalence relation on it. Here, the equivalence realtion is a user-defined (book) equality, built up according to the type constructions from basic equalities, which are the inductively defined one for inductive types and an axiomatically declared one for basic variable types.

Setoids
A pair $[A, =]$ with $A : \mathsf{Set}$, $= : A{\to}(A{\to}\mathsf{Prop})$ such that $=$ is an equivalence relation on A is called a *setoid*. In the translation of extensional types to setoids (in intensional type theory) one has to also translate compound types, like $A{\to}B$ and $\Pi x{:}A.B$, this amounts to defining the function space and the dependent function space between setoids. To give the idea we treat the function space here. Given two setoids $[A, =_A]$ and $[B, =_B]$, we define the *function space setoid* $[A\overset{s}{\to}B, =_{A\overset{s}{\to}B}]$ by

$$\begin{array}{rcl} A\overset{s}{\to}B & := & \Sigma f{:}A{\to}B.(\Pi x, y{:}A.(x =_A y){\to}((f\ x) =_B\ (f\ y))), \\ f =_{A\overset{s}{\to}B} g & := & \Pi x{:}A.(\pi_1\ f\ x) =_B (\pi_1\ g\ x). \end{array}$$

Note that, $f =_{A\overset{s}{\to}B} g$ is equivalent to $\Pi x, y{:}A.(x =_A y){\to}(\pi_1\ f\ x) =_B (\pi_1\ g\ x)$, because we require f and g to preserve $=_A$. Given A with equality $=_A$ and B with equality $=_B$, this is the 'canonical equality' on $A{\to}B$. Note that the carrier set $A\overset{s}{\to}B$ is not just $A{\to}B$, but the 'subset' of those $f : A{\to}B$ that respect the equalities R_A and $=_B$. Such an f is also called a *setoid function* from $[A, =_A]$ to $[B, =_B]$. In type theory, such a subset (of setoid functions) is represented by a Σ-type, consisting of pairs $\langle f, p, \rangle$ with (in this case) $f : A{\to}B$, $p : \Pi x, y{:}A.(x =_A y){\to}((f\ x) =_B (f\ y))$. The equivalence relation $=_{A\overset{s}{\to}B}$ ignores the proof-terms, so $\langle f, p\rangle =_{A\overset{s}{\to}B} \langle f, q\rangle$ holds for all elements of the carrier set $A\overset{s}{\to}B$.

The canonical equality on $A{\to}B$ is the extensional equality of functions. Therefore, the interpretation of extensional type theory in intensional type theory is sound. (Of course, the other type constructions still have to be verified; see [Hofmann 1994] for details.) It has been observed that setoids present a general and practical way of dealing with extensional equality and with mathematical constructions in general. If, in mathematics one speaks informally of a 'set', we encode this in type theory by a 'setoid'. To show the flexibility we show how a *quotient* and a *subset* can be represented using setoids.

Given a setoid $[A, =_A]$ and an equivalence relation Q over this setoid, we define the *quotient-setoid* $[A, =_A]/Q$. Note that the fact that Q is an equivalence relation over the setoid $[A, =_A]$ means that

- $Q : A \to (A \to \mathsf{Prop})$ is an equivalence relation,
- $=_A \subset Q$, i.e. $\forall x, y{:}A.(x =_A y) \to (Q\ x\ y)$.

We define the quotient setoid $[A, =_A]/Q$ simply as $[A, Q]$. It is an easy exercise to show how a setoid function f from $[A, =_A]$ to $[B, =_B]$ that respects Q (i.e. $\forall x, y{:}A.(Q\ x\ y) \to ((f\ x) =_B (f\ y)))$ induces a setoid function from $[A, =_A]/Q$ to $[B, =_B]$.

Given a setoid $[A, =_A]$ and a predicate P on A, we define the *sub-setoid* $[A, =_A]|P$ as the pair $[\Sigma x{:}A.(P\ x), =_A|P]$, where $=_A|P$ is $=_A$ restricted to P, i.e. for q, r : $\Sigma x{:}A.(P\ x)$,

$$q\ (=_A|P)\ r \quad := \quad (\pi_1\ q) =_A (\pi_1\ r).$$

In defining a subsetoid, we do not require the predicate P to respect the equality $=_A$. (That is, we do *not* require $\forall x, y{:}A(x = y \wedge Px) \to Py$ to hold.) So, in taking a subsetoid we may remove elements from the =-equivalence classes. This is natural, because we are not interested in the elements of A, but in the =-equivalence classes. Consider the following example where this appears rather naturally. Let $A := \mathsf{int} \times \mathsf{nat}$ be the type of pairs of an integer and a natural number. To represent the rationals we define, for $\langle x, p\rangle, \langle y, q\rangle{:}A$,

$$\langle x, p\rangle =_A \langle y, q\rangle \quad := \quad x(q+1) = y(p+1).$$

Now consider the predicate P on A defined by

$$P\ \langle x, p\rangle \quad := \quad \gcd(x, p+1) = 1.$$

The subsetoid $[A, =_A]|P$ is isomorphic to $[A, =_A]$ itself, but all equivalence classes have been reduced to a one element set.

Subtypes and coercions

When using setoids to formalize the notion of set, one encounters a typing problem. Suppose we have the setoid $[A, =_A]$. Now, $A : \mathsf{Set}$, but the setoid $[A, =_A]$ is not of type Set, but of type $\Sigma A{:}\mathsf{Set}.A \to (A \to \mathsf{Prop})$ Hence we can not declare a variable $x : [A, =_A]$ (because we can only declare a variable $x : B$ if $B : \mathsf{Set}$ or $B : \mathsf{Prop}$). Similarly, if $a : A$, then a is not of type $[A, =_A]$.

As a matter of fact, a setoid consists of a *triple*

$$[A, =_A, \mathsf{eq\_rel\_proof}] : \Sigma A{:}\mathsf{Set}.\Sigma R{:}A \to (A \to \mathsf{Prop}).(\mathsf{Is\_eq\_rel}\ \mathsf{A}\ \mathsf{R}),$$

where $\mathsf{eq\_rel\_proof}$ is a proof of $(\mathsf{Is\_eq\_rel\,A\,R})$, stating that $=_A$ is an equivalence relation over A. If we formalize the type of equivalence relations over a fixed A as

$$\mathsf{Eq\_Rel_A} := \Sigma R{:}A{\to}(A{\to}\mathsf{Prop}).(\mathsf{Is\_eq\_rel\,A\,R}),$$

then, if $R : \mathsf{Eq\_Rel_A}$ and $a, a' : A$, one would like to write Raa', but this is not a proposition. (The R is really a *pair* consisting of a binary relation and a proof.)

If we look at the formalization of subsets as subsetoids, we encounter a similar problem. If $[A, =_A]|P$ is a subsetoid of $[A, =_A]$, then an 'element' of this subsetoid is given by a *pair* $\langle a, p\rangle$, where $a : A$ and $p : Pa$, but this is not an 'element' of $[A, =_A]$. Indeed, if $F : A{\to}B$ and $x : [A, =_A]|P$, we can not write Fx, as x itself is not of type A.

The problem lies in the fact that our terms are very explicit, whereas we would like to be more implicit. This situation is also encountered in mathematics, where one defines, for example a 'group' as a tuple $\mathcal{A} = \langle A, \circ, \mathsf{inv}, e\rangle$, where A is a set, $\circ$ a binary operation, inv a unary operation and e an element of A, satisfying the group axioms. Then one speaks of 'elements of the group $\mathcal{A}$', where one really means 'elements of the (carrier) set A'. So, one (deliberately) mixes up the group $\mathcal{A}$ and its carrier set A. This is not sloppiness, but convenience: some of the details are deliberately omitted, knowing that one can fill them in if necessary. This is sometimes called 'informal rigor'.

As was first noted by Aczel, one would like to have a similar mechanism in type theory, for being able to use informal rigor. A way to do this is by creating a level on top of the type theory, where one can use more informal language, which is then translated to the formal level. This requires that the informal expressions are expanded in such a way that they become well-formed in the underlying formal type theory. It turns out that in this expansion, the type synthesis algorithm is very useful, as it generates the missing information. This can be made formally precise by introducing the notion of *coercion*.

Some of the problematic examples that we gave above can be seen as instances of the *sub-typing problem*. In type theory as we have discussed until now, there is no notion of subtype: we can not say that $A \subseteq B$, with as intended meaning that if $a : A$ then also $a : B$. It turns out that if one adds such a sub-typing relation, the decidability of type checking becomes rather problematic. Moreover, there are various ways in which the sub-typing relation can be lifted along the type constructions (like Π and $\to$). On the other hand, some of the problems discussed above can be solved using sub-typing:

$$\begin{array}{rll} \text{If } \Sigma A{:}\mathsf{Set}.A{\to}(A{\to}\mathsf{Prop}) \subseteq \mathsf{Set}, & \text{then} & x : [A, =_A] \text{ can be declared,} \\ \text{If } \mathsf{Eq\_Rel_A} \subseteq A{\to}(A{\to}\mathsf{Prop}), & \text{then} & R : \mathsf{Eq\_Rel_A}, a, a' : A \vdash Raa' : \mathsf{Prop}, \\ \text{If } [A, =_A]|P \subseteq [A, =_A], & \text{then} & F : A{\to}B, a : [A, =_A]|P \vdash Fa : B. \end{array}$$

Note, however that this does not solve all problems: if $a : A$, we can not write $a : [A, =_A]$ (the $\subseteq$ needs to be reversed). Furthermore, the meaning of $[A, =_A]|P \subseteq [A, =_A]$ is not so clear, as both are not themselves types.

A related but different solution can be found by making the inclusions $A \subseteq B$ explicit by a *coercion map*. Then we have e.g.

$$\begin{aligned} \pi_1 &: \Sigma A{:}\mathsf{Set}.A{\to}(A{\to}\mathsf{Prop}) \subseteq \mathsf{Set}, \\ \pi_1 &: \mathsf{Eq\_Rel_A} \subseteq A{\to}(A{\to}\mathsf{Prop}). \end{aligned}$$

We have no map from $[A, =_A]|P$ to $[A, =_A]$, as these are not types. The maps here are just definable terms and we can replace the $\subseteq$ by an $\to$. But then we are back to the original formulations where we have to give all terms explicitly everywhere. The idea is to declare the coercions as special maps, to be used by the type checker to type expressions. So the user does not have to insert these maps, but the type checker will do so to compute a type. Essentially, there are three ways in which a type checking algorithm can use a coercion map.

$$\left.\begin{array}{l} c : A \subseteq \mathsf{Set} \\ (\text{or } c : A \subseteq \mathsf{Prop}) \end{array}\right\} \quad \text{the declaration } x : A \text{ is expanded to } x : (cA).$$

$$\left.\begin{array}{l} G : D \\ c : D \subseteq A{\to}B \\ a : A \end{array}\right\} \quad G\,a \text{ is expanded to } c\,G\,a \text{ of type } B.$$

$$\left.\begin{array}{l} F : A{\to}B \\ c : D \subseteq A \\ a : D \end{array}\right\} \quad F\,a \text{ is expanded to } F\,(c\,a) \text{ of type } B.$$

It should also be possible to use multiple coercion maps: if there are coercions $c_1 : A \subseteq B$ and $c_2 : B \subseteq C$, then there is a coercion $\lambda x{:}A.c_2(c_1 x) : A \subseteq C$. So the coercions are really just definable λ-terms that can be composed. Of course, there should be only one coercion between two types A and B and there should be no coercion from a type A to itself. This has to be checked by the system at the moment a coercion is declared: it should go through the 'coercion graph' to verify that it is still a tree. For more on coercions see [Barthe 1996] or [Luo 1999]. Another approach to subtypes is to treat them as real subsets: if $M : A$ and A is a subtype of B, then $M : B$ (without coercion). We will not discuss this possibility here; for a possible set of typing rules for subtypes we refer to [Zwanenburg 1999].

2.9. Connection between logic and type theory

When doing formal proofs with the help of some computer system, one may wonder what one is really proving. Or, to put it differently,

> what is the *semantics* of the formal objects that the system (and the user) is dealing with?

The systems that we are concerned with here are based on type theory, which moves the semantics-question from the level of the computer system to the level of the formal system:

> what do the expressions of the type theory mean?

Note that this only gives a satisfactory answer in case the computer system is a faithful implementation of the type theory. The actual situation is as follows: the interactive proof development system (where the proof-terms are *created*) is not fully explained in terms of the type theory; however, the *proof checker* (which is executed after the proof-term has been completed) is completely faithful to the type theory.

So, we will confine ourselves to the question what the expressions of the type theory mean. This question can be dealt with in different ways. First we can look at some (preferred) model, $\mathcal{M}$, of a piece of mathematics and ask what the type theoretical expressions mean in $\mathcal{M}$. Second, we can look at some logic $\mathcal{L}$ and ask what the meaning of the type theoretical expressions in $\mathcal{L}$ is. This results in the following questions.

- What is the interpretation of the expressions in the model $\mathcal{M}$ and is there a soundness and/or completeness result? For $A : \mathsf{Prop}$,

$$\mathcal{M} \models A \text{ iff } \exists p(\vdash p : A)?$$

- What is the interpretation of the expressions in the logic $\mathcal{L}$ and, for $A : \mathsf{Prop}$, is A provable in $\mathcal{L}$ iff A is inhabited?

$$\vdash_{\mathcal{L}} A \text{ iff } \exists p(\vdash p : A)?$$

As type theory is generic, we are mainly interested in the second question. The connection with logic is even more relevant as type theory seeks to represent *proofs* as terms; these proof-terms then better have some relation to a proof in logic. Following the Curry-Howard-de Bruijn *propositions-as-types-embedding*, formulas of logic are interpreted as types, and at the same time, (natural deduction) derivations are interpreted as proof-terms. So, the answer to the question whether proof-terms in type theory represent proofs is affirmative: proof-terms represents natural deduction proofs. Of course, the situation is more complicated: there are a lot of logics and a lot of type theories. But if we choose, given our logic in natural deduction style $\mathcal{L}$, an appropriate type theory $S(\mathcal{L})$, we have the following

Soundness of the propositions-as-types embedding:

$$\vdash^{\Sigma}_{\mathcal{L}} \varphi \quad \Rightarrow \quad \Gamma \vdash_{S(\mathcal{L})} [\![\Sigma]\!] : \varphi,$$

where Σ denotes the deduction of φ in $\mathcal{L}$ and $[\![\Sigma]\!]$ its encoding as a term in $S(\mathcal{L})$. Γ is a context in which the relevant variables are declared. In Section 3.2, we describe the propositions-as-types embedding in more detail for higher order predicate logic and its corresponding type system.

The other way around, we may wonder whether, if φ is inhabited in $S(\mathcal{L})$, then φ is derivable in $\mathcal{L}$ (where $\varphi : \mathsf{Prop}$).

Completeness of the propositions-as-types embedding:

$$\Gamma \vdash_{S(\mathcal{L})} M : \varphi \quad \stackrel{?}{\Rightarrow} \quad \vdash_{\mathcal{L}} \varphi,$$

where Γ is again a context in which the relevant variables are declared. If we take into account that a term $M : \varphi$ is intended to represent a natural deduction proof, we may strengthen our completeness by requiring an embedding $[\_]$ from proof-terms to deductions.

Strong Completeness of the propositions-as-types embedding:

$$\Gamma \vdash_{S(\mathcal{L})} M : \varphi \quad \stackrel{?}{\Rightarrow} \quad \vdash_{\mathcal{L}}^{[M]} \varphi.$$

Completeness is not in all cases so easy. Consider for example the Martin-Löf's type theories, where there is no distinction between 'sets' and 'propositions' – both are of type Set. We have already discussed this situation in Section 2.5, where we pointed out that in ordinary logic there is a sharp distinction between Prop and Set from the very start. It is just the way logic is defined, in *stages*, where one first defines the terms (including the domains), then the formulas and then the derivations. That means that for Martin-Löf's type theories, it is not so easy to define a mapping back to the logic (in this case first order intuitionistic logic). For example, look at the context

$$A{:}\mathsf{Set}, a{:}A, P{:}A{\to}\mathsf{Set}, h{:}(Pa), Q{:}(Pa){\to}\mathsf{Set}, f{:}(P\ a){\to}A.$$

If we try to interpret this in first order intuitionistic logic, we can view A as a domain, a as an element of A, P as a predicate on A and h as the assumption that $(P\ a)$ holds (h is an *assumed proof* of $(P\ a)$). But then Q can only be understood as a predicate on the set of proofs of $(P\ a)$[3], and f as a map from the proofs of $(P\ a)$ to the domain A. It will be clear that there are many types $X{:}\mathsf{Set}$ in the type theory that have no interpretation, neither as a 'domain' nor as a 'proposition', in first order intuitionistic logic. As a consequence, Strong Completeness fails for Martin-Löf's type theory. It has been shown – but the proof is really intricate, see [Swaen 1989] – that completeness (the weaker variant) holds. However, if we extend these type theories to higher order, we obtain either an inconsistent system (if we interpret the higher order $\exists$ as a Σ-type, see [Coquand 1986]), or (if we interpret the higher order $\exists$ impredicatively) a system for which completeness fails with respect to constructive higher order predicate logic; see [Berardi 1990], [Geuvers 1993].

Summarizing, we observe the following possible points of view: (1) first order predicate logic is incomplete, as it does not allow objects to depend on proofs, whereas both are just 'constructions'; (2) the idea of unifying the Prop and the Set universe into one (Set) is wrong, as it creates objects depending on proofs, a feature alien to ordinary logic. We tend to have the second view, although the situation is

[3] A – proof-theoretically – interesting predicate on proofs may be 'to be cut-free'. However, a predicate can not distinguish between β-equal terms, so this predicate can not be expressed.

not so easy, as can be seen from the two examples below, where we apply the idea of letting objects depend on proofs.

With respect to the interpretation of the constructive existential quantifier, there are also two possible positions: (I) interpret $\exists$ by the Σ-type, which does not work well for higher order logic, (but higher-order logic is often considered as non-constructive – because impredicative – anyway); (II) interpret it in a different way (e.g. using a higher order encoding or an inductive encoding) that avoids the projections of proofs to objects. Obviously, position (I) on the existential quantifier interpretation goes well with position (1) on the Prop-Set-issue above. Similarly (II) goes well with (2) above.

Concluding this discussion on the precise choice of the type theoretical rules to interpret the logic, we note the following. The build up of logic in *stages*, as described before, is very much related to a Platonist view of the world, where

> the objects are just *there* and logic is a means of deriving true properties about these objects.

So an object is not affected by our reasoning about it. In the constructive view,

> both objects and proofs are *constructions* and the only objects that exist are the ones that can be constructed.

Then a formula is identified with the set of its proofs and there is *a priori* no problem with constructing an element of one set (say the set nat) out of another set (say a formula A). So, if we take the constructive view as a starting point, the dependency of objects on proofs is no problem. Note that this still leaves a choice of really identifying the universe of sets and propositions (then $A : \mathsf{Set}$ for sets A and $A : \mathsf{Set}$ for formulas A) or keeping the distinction (then $A : \mathsf{Set}$ for sets A and $A : \mathsf{Prop}$ for formulas A). In this article we start from type systems where objects do not depend on proofs.

If one chooses a type theory that remains quite closely to the original constructive logic (in natural deduction style), it is not so difficult (although laborious) to prove Strong Completeness of the propositions-as-types embedding. See [Geuvers 1993] for some detailed proofs and examples.

Examples of objects depending on proofs

In the discussion above, we promoted the idea of not letting objects depend on proofs. Although this solves some of the completeness questions in a relatively easy way, this position is not so simple to be maintained. If one *really* starts formalizing mathematics in type systems, objects depending on proofs occur quite naturally.

Consider a $A : \mathsf{Set}$ that we want to show to be a field. That means that we have to define all kinds of objects (0, 1) and functions ($\mathsf{mult}, \ldots$) on A and to prove that together they satisfy the field-axioms. Now what should the type of the reciprocal be, given that the reciprocal of 0 is not defined? An option is to let $\mathsf{recip} : A{\to}A$ with the property that $\forall x{:}A.x \neq 0 \to \mathsf{mult}\, x(\mathsf{recip}\, x) = 1$. However, this is not very nice: $\mathsf{recip}0$ should be undefined (whereas now it is an 'unspecified' element of A). In type theory there is a different solution to this: construct

$$\mathsf{recip} : (\Sigma x{:}A.x \neq 0) \to A.$$

Then recip is only defined on the subset of elements that are non-zero: it receives a pair $\langle a, p\rangle$ with $a : A$ and $p : a \neq 0$ and returns $\mathsf{recip}\langle a, p\rangle : A$. But how should one understand the dependency of this object (of type A) on the proof p in terms of ordinary mathematics?

A possible solution is provided by the setoids approach (see also the previous Section). We take as the carrier of a field a setoid $[A, =_A]$, so $A : \mathsf{Set}$ and $=_A$ is an equivalence relation on A. The operations on the field are now taken to be setoid functions, so e.g. mult has to preserve the equality: if $a =_A a'$ and $b =_A b'$, then $(\mathsf{mult} ab) =_A (\mathsf{mult} a'b')$. Similarly, all the properties of fields are now denoted using the setoid equality $=_A$ instead of the general equality $=$. For the reciprocal, this amounts to

$$\mathsf{recip} : [A, =_A] | (\lambda x{:}A.x \neq_A 0) \to [A, =_A],$$

a setoid function from the subsetoid of non-zeros to $[A, =_A]$ itself. In this case, recip still takes a pair of an object and a proof $\langle a, p\rangle$, with $a : A$ and $p : a \neq_A 0$, and returns $\mathsf{recip}\langle a, p\rangle : A$. The difference however is that recip now is a *setoid function*, which implies the following.

$$\text{If } a, a' : A \text{ with } a =_A a', p : a \neq_A 0, q : a' \neq_A 0, \text{ then } \mathsf{recip}\langle a, p\rangle =_A \mathsf{recip}\langle a', q\rangle.$$

So, the value of $\mathsf{recip}\langle a, p\rangle$ does not depend on the actual p; the only thing to ascertain is that such a term exists (i.e. that $a \neq_A 0$ is true).

We conjecture that if the objects that depend on proofs only occur in the context of setoids, as above, we can make sense of these objects in terms of standard mathematics. The general principle that for an object $t(p) : A$, where $p : \varphi$ denotes a sub-term of t,

$$t(p) = t(q) \text{ for all } p, q : \varphi$$

is called the *principle of Proof Irrelevance*. It states that the actual proof p of φ is irrelevant for the value of $t(p)$. The setoid equality discussed before obeys this principle, due to the way the setoid equality is promoted to subsetoids.

Another example of objects depending on proofs occurs for example in the definition of the absolute value in an ordered field. Suppose

$$p : \Pi x{:}F.(x \geq 0 \vee x \leq 0).$$

Then define the absolute value function abs as follows.

$$\mathsf{abs} \quad := \quad \lambda x{:}F.\,\mathsf{case}\ (p\,x)\ \mathsf{of}\ \begin{array}{lll} (\mathsf{inl}\,\_) & \Rightarrow & x \\ (\mathsf{inr}\,\_) & \Rightarrow & -x \end{array}$$

This function distinguishes cases according to the value of px. If it is of the form $\mathsf{inl}\,r$ (with $r : x \geq 0$), we take x; if it is of the form $\mathsf{inr}\,r$ (with $r : x \leq 0$), we take

$-x$. Now, for $a : F$, the term $(\mathsf{abs}\ a)$ contains a proof-term p. We want to prove that the values of abs do not depend on the actual value of p. In the context of setoids, this means that if we have two definitions of the absolute value function, abs_p and abs_q, one defined using the proof $p : \Pi x{:}F.x \geq 0 \vee x \leq 0$ and one using the proof q of the same type, we have to prove

$$\Pi x, x'{:}F.(x =_F x') \to (\mathsf{abs}_p x) =_F (\mathsf{abs}_q x').$$

Note that it may be the case that for some x, the value of $p\,x$ is $\mathsf{inl}\_$, while the value of $q\,x$ is $\mathsf{inr}\_$. Then $\mathsf{abs}_p x$ has value x and $\mathsf{abs}_q x$ has value $-x$. One then has to prove that in this overlapping case $x =_F -x$, which holds, as it only occurs if $x =_F 0$.

3. Type systems for proof checking

As we see it, there is not one 'right' type system. The widely used theorem provers that are based on type theory all have inductive types. But then still there are other important parameters: the choice of allowed quantification and the choice of reduction relations to be used in the type conversion rule. We have already mentioned the possibility of allowing impredicative quantification or not. Also, we mentioned the β, δ, ι and η rules as possible reduction rules. A very powerful extension of the reduction relation is obtained by adding a fixed-point-operator $\mathsf{Y}{:}\Pi A{:}\mathsf{Set}.(A{\to}A){\to}A$ satisfying

$$\mathsf{Y} f \to_{\mathsf{Y}} f(\mathsf{Y} f).$$

With this addition the reduction of the type system does not satisfy strong normalization and proof-objects are *potential* ones. It has been shown in [Geuvers, Poll and Zwanenburg 1999] that under mild conditions the Y-rules are conservative over the ones without a Y. A similar extension of type theory with fixed points is discussed in [Audebaud 1991], where the fixed points are used to define recursive data types.

It is outside the scope of this article to discuss the technical details of various different type systems. However, we do want to give some of the underlying theory, to show the sound theoretical base of type theoretical theorem provers and to make concrete some of the issues that were discussed in the previous Sections. Therefore we start off by considering one specific type system in detail. We define a type theory for higher order predicate logic, $\lambda\mathsf{HOL}$ and show how mathematical notions can be interpreted in it. To make the latter precise, we first look into higher order predicate logic itself. Then we study the formal interpretation from higher order predicate logic into $\lambda\mathsf{HOL}$, both as a motivation for the definition of $\lambda\mathsf{HOL}$ and as an illustration of how precisely mathematics is dealt with in type theory. Then we define a more general class of type systems. We discuss the essential properties and how type systems are used to create an interactive theorem prover. For $\lambda\mathsf{HOL}$ itself we give—in detail—the type checking algorithm, which is at the core of every type theoretical theorem prover.

By examples,we give some possible extensions of λHOL with other type constructions, like inductive types. The type systems that we discuss here all adhere to the principle that objects do not depend on proofs and that there is a distinction between sets and formulas. This is mainly done to keep the 'logical' explanation clear; see also the discussion in Section 2.9. We also give no formal treatment of definitions here (the δ-rule for unfolding definitions etc., see Section 2.8). Definitions are very prominent in a theorem prover and we believe that (hence) definitions are an important formal notion, but we want to restrict to the main issues here. See [Severi and Poll 1994] for the extension of type systems with a formal notion of definition.

3.1. Higher order predicate logic

If we want to do proof checking, we first have to make a choice for a logic. There are various possibilities: first order, second order, higher order. It is also possible to choose between either classical or intuitionistic logic, or between natural deduction and sequent calculus.

For checking formal proofs in a system based on type theory, it turns out that a calculus of intuitionistic natural deduction is the most adequate. Although it is not difficult to add classical reasoning, type theory is more tailored towards constructive reasoning. Furthermore, typed λ-terms are a faithful term representation of natural deductions. (In sequent calculus there is much more 'bureaucracy'.) The choice between first order, second order or higher order can be made by adapting the rules of the type system; we will come to that later. So, to set our logical system we choose constructive higher order predicate logic in natural deduction style.

3.1. Definition. The language of **HOL** is defined as follows.

1. The set of *domains*, D is defined by

$$D ::= B \mid \Omega \mid D {\to} D,$$

   where B represents a *basic domain* (we assume that there are countably many basic domains) and Ω represents the *domain of propositions.*
2. For every $\sigma \in D$, the set of *terms of type* σ, Term_σ is inductively defined as follows. (As usual we write $t : \sigma$ to denote that t is a term of type σ.)
   (a) the constants $c_1^\sigma, c_2^\sigma, \ldots$ are in Term_σ,
   (b) the variables $x_1^\sigma, x_2^\sigma, \ldots$ are in Term_σ,
   (c) if $\varphi : \Omega$ and x^σ is a variable, then $(\forall x^\sigma.\varphi) : \Omega$,
   (d) if $\varphi : \Omega$ and $\psi : \Omega$, then $(\varphi \Rightarrow \psi) : \Omega$,
   (e) if $M : \sigma{\to}\tau$ and $N : \sigma$, then $(MN) : \tau$,
   (f) if $M : \tau$ and x^σ is a variable, then $(\lambda x^\sigma.M) : \sigma{\to}\tau$.
3. The set of terms of **HOL**, Term, is defined by $\mathrm{Term} := \cup_{\sigma\in D}\mathrm{Term}_\sigma$.
4. The set of formulas of **HOL**, form, is defined by $\mathsf{form} := \mathrm{Term}_\Omega$.

We adapt the well-known notions of *free* and *bound* variable, substitution, β-reduction and β-conversion to the terms of this system.

There are no 'product' domains $(D \times D)$ in our logic. We present functions of higher arity by *Currying*: a binary function on D is represented as a term in the domain $D \to (D \to D)$. A predicate is represented as a function to Ω, following the idea (probably due to Church; it appears in [Church 1940]) that a predicate can be seen as a function that takes a value as input and returns a formula. So, a binary relation over D is represented as a term in the domain $D \to (D \to \Omega)$. (If $R : D \to (D \to \Omega)$ and $t, q : D$, then $((Rt)q) : \Omega$.) The logical connectives are just implication and universal quantification. Due to the fact that we have *higher order* universal quantification, we can express all other quantifiers using just $\Rightarrow$ and $\forall$. See 3.6 for more details.

3.2. NOTE. We fix the following notational conventions.

- Outside brackets are omitted.
- In the domains we omit the brackets by letting them associate to the right, so $D \to D \to \Omega$ denotes $D \to (D \to \Omega)$.
- In terms we omit brackets by associating them to the left, so Rtq denotes $(Rt)q$. Note that in ordinary mathematics, this is usually written as $R(t, q)$.
- If we write Rab, we *always* mean $((R\ a)\ b)$, so R applied to a, and then this applied to b. If we want to introduce a name (as an abbreviation), we will use the sans serif font, e.g. in writing trans as an abbreviation of the transitivity property.

3.3. EXAMPLE. Before giving the logical rules of **HOL**, we treat some examples of terms and formulas that can be written down in this language. Let the following be given: domains $\mathbb{N}$ and A, the relation-constant $>: \mathbb{N} \to \mathbb{N} \to \Omega$, the relation-variables $R, Q : A \to A \to \Omega$ and the function-constants $0 : \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$.

1. The predicate 'being larger than 0' is expressed by the term $\lambda x^{\mathbb{N}}.x > 0 : \mathbb{N} \to \Omega$.
2. Induction over $\mathbb{N}$ can be expressed by the (second order) formula ind defined as
$$\forall P^{\mathbb{N} \to \Omega}.(P0) \Rightarrow (\forall x^{\mathbb{N}}.(Px \Rightarrow P(Sx))) \Rightarrow \forall x^{\mathbb{N}}.Px.$$
3. The formula $\mathsf{trans}(R)$, defined as $\forall x^A y^A z^A (Rxy \Rightarrow Ryz \Rightarrow Rxz)$ denotes the fact that R is transitive. So, $\mathsf{trans} : (A \to A \to \Omega) \to \Omega$. Note that we write $\forall x^A y^A z^A$ as a shorthand for $\forall x^A.\forall y^A.\forall z^A$.
4. The term $\subseteq : (A \to A \to \Omega) \to (A \to A \to \Omega) \to \Omega$ is defined by
$$R \subseteq Q := \forall x^A y^A.(Rxy \Rightarrow Qxy).$$
(We informally use the infix notation $R \subseteq Q$ to denote $\subseteq RQ$.)
5. The term $\lambda x^A y^A.(\forall Q^{A \to A \to \Omega}.(\mathsf{trans}(Q) \Rightarrow (R \subseteq Q) \Rightarrow Qxy))$ is of type $A \to A \to \Omega$. It denotes the transitive closure of R. We use $\lambda x^A y^A$ as a shorthand for $\lambda x^A.\lambda y^A$.

The derivation rules of **HOL** are given in a natural deduction style.

(axiom)	$\overline{\Gamma \vdash \varphi}$	if $\varphi \in \Gamma$
($\Rightarrow$ -introduction)	$\dfrac{\Gamma \cup \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi}$	
($\Rightarrow$ -elimination)	$\dfrac{\Gamma \vdash \varphi \quad \Gamma \vdash \varphi \Rightarrow \psi}{\Gamma \vdash \psi}$	
($\forall$-introduction)	$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x^\sigma . \varphi}$	if $x^\sigma \notin \mathrm{FV}(\Gamma)$
($\forall$-elimination)	$\dfrac{\Gamma \vdash \forall x^\sigma . \varphi}{\Gamma \vdash \varphi[t/x^\sigma]}$	if $t : \sigma$
(conversion)	$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \psi}$	if $\varphi =_\beta \psi$

Figure 1: Deduction rules of **HOL**

3.4. Definition. The notion of *provability*, $\Gamma \vdash \varphi$, for Γ a finite set of formulas (terms of type form) and φ a formula, is defined inductively by the rules in Fig. 1

3.5. Remark. The rule (conversion) is an operationalization of the Poincaré principle discussed in Section 2.8. The rule says that we don't want to distinguish between β-equal propositions.

3.6. Example. A well-known fact about this logic is that the connectives $\&, \vee, \perp, \neg$ and $\exists$ are definable in terms of $\Rightarrow$ and $\forall$. (This is due to [Russell 1903].) For $\varphi, \psi : \Omega$, define

$$
\begin{aligned}
\varphi \& \psi &:= \forall x^\Omega . (\varphi \Rightarrow \psi \Rightarrow x) \Rightarrow x, \\
\varphi \vee \psi &:= \forall x^\Omega . (\varphi \Rightarrow x) \Rightarrow (\psi \Rightarrow x) \Rightarrow x, \\
\perp &:= \forall x^\Omega . x, \\
\neg \varphi &:= \varphi \Rightarrow \perp, \\
\exists x^\sigma . \varphi &:= \forall z^\Omega . (\forall x^\sigma . (\varphi \Rightarrow z)) \Rightarrow z.
\end{aligned}
$$

It's not difficult to check that the intuitionistic elimination and introduction rules for these connectives are sound.

Equality between terms of a fixed type σ is definable by saying that two terms are equal if they share the same properties. This equality is usually called *Leibniz equality* and is defined by

$$t =_A t' := \forall P^{A\to\Omega}.(Pt \Rightarrow Pt'), \text{ for } t, t' : A.$$

It is not difficult to see that this equality is reflexive and transitive. It is also symmetric: Let Q be a predicate variable over A (so $Q : A\to\Omega$). Take $\lambda y^A.Qy \Rightarrow Qt$ for P. The deduction is as follows. (At the left we apply the ($\forall$-elim) rule followed by the (conv) rule.)

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \forall P^{A\to\Omega}(Pt \Rightarrow Pt')}{\Gamma \vdash (Qt \Rightarrow Qt) \Rightarrow (Qt' \Rightarrow Qt)} \qquad \dfrac{\Gamma, Qt \vdash Qt}{\Gamma \vdash Qt \Rightarrow Qt}}{\Gamma \vdash Qt' \Rightarrow Qt}}{\Gamma \vdash \forall Q^{A\to\Omega}.(Qt' \Rightarrow Qt)}$$

Impredicativity In the definition of the connectives (Example 3.6) and in the definition of equality, one makes use of *impredicativity*, that is

> the possibility of constructing a term of a certain domain by abstracting over that same domain or over a domain of the same 'order'.

E.g. in Example 3.6 one constructs the *proposition* $\varphi\&\psi$ by abstracting (using the universal quantifier) over the collection of *all propositions*. Similarly in the definition of Leibniz equality one defines a binary *relation on* A by abstracting over the collection of all *predicates on* A. Both are domains of second order. (The basic domains are of first order.) The fact that this logic is higher order allows us to make these impredicative constructions.

The notion of order was first introduced by Russell (see [Whitehead and Russell 1910, 1927]) in his ramified type theory, to prevent the paradoxes arising from a naive conception of the notion of set. Later it was noted by Ramsey [1925] that the simple types suffice to avoid the syntactic paradoxes. The semantic paradoxes can be avoided by making a clear distinction between syntax (formal system) and semantics (models). In [Whitehead and Russell 1910, 1927] this distinction was not made and the ramification was used to prevent the semantical paradoxes.

Impredicativity is often seen as 'non-constructive': an impredicative definition can not really be understood as a *construction*, but only as a *description* of an object whose existence we assume on other grounds. For example, the definition of Leibniz equality *describes* a binary relation by quantifying over the collection of all predicates. This is not a *construction*, as that would require that the collection of all predicates had already been constructed, before we construct this binary relation. Therefore, impredicativity is seen as alien to constructive logic. We will still call our logic constructive, as it lacks the double negation law (and hence it is not classical). Moreover, the logic enjoys the *disjunction property* (if $\vdash \varphi\vee\psi$, then $\vdash \varphi$ or $\vdash \psi$) and the *existence property* (if $\vdash \exists x{:}A.\varphi$, then $\vdash \varphi[a/x]$ for some $a : A$) that we know from constructive logics. If we charachterize a logic as constructive if it satisfies the disjunction and the existence property, then our higher order predicate logic is constructive.

3.2. *Higher order typed λ-calculus*

In type theory, one interprets formulas and proofs via the well-known 'propositions-as-types' and 'proofs-as-terms' embedding, originally due to Curry, Howard and de Bruijn. (See [Howard 1980, de Bruijn 1970].) Under this interpretation, a formula is viewed as the type of its proofs. It turns out that one can define a typed λ-calculus $\lambda\mathsf{HOL}$ that represents **HOL** in a very precise way. What *very precise* means will not be defined here, but see e.g. [Barendregt 1992] or [Geuvers 1993]. Here, we just define the system $\lambda\mathsf{HOL}$, using the intuitions of **HOL**. In order to get a better understanding we note a few things.

1. The language of **HOL** as presented in 3.1 is a typed language already. This language will be a part of $\lambda\mathsf{HOL}$
2. In $\lambda\mathsf{HOL}$, formulas like $\varphi \Rightarrow \psi$ and $\forall x^A.\varphi$ will become types. However, these 'propositional' types are not the same as the 'set' types like e.g. $\mathbb{N}$. Hence there will be two 'universes': Prop, containing the 'propositional' types, and Type, containing the 'set' types. Prop itself is a 'set' type.
3. The deductions are represented as typed λ-terms. The discharging of hypotheses is done by *λ-abstraction*. The modus ponens rule is interpreted via *application*.

The *derivable judgments* of $\lambda\mathsf{HOL}$ are of the form

$$\Gamma \vdash M : A,$$

where Γ is a *context* and M and A are terms. A context is of the form $x_1{:}A_1, \ldots, x_n{:}A_n$, where $x_1, \ldots, x_n$ are variables and $A_1, \ldots, A_n$ are terms. The variables that occur in M and A are given a type in a context. If, in the judgment $\Gamma \vdash M : A$, the term A is a 'propositional type' (i.e. $\Gamma \vdash A : \mathsf{Prop}$), we view M as a *proof* of A. If the term A is a 'set type' (i.e. $\Gamma \vdash A : \mathsf{Type}$), we view M as an *element* of the set A.

3.7. DEFINITION. The typed λ-calculus $\lambda\mathsf{HOL}$, representing higher order predicate logic, is defined as follows. The set of *pseudo terms* $\mathcal{T}$ is defined by

$$\mathcal{T} ::= \mathsf{Prop} \mid \mathsf{Type} \mid \mathsf{Type}' \mid \mathcal{V} \mid (\Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}) \mid (\lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T}) \mid \mathcal{T}\mathcal{T}.$$

Here, $\mathcal{V}$ is a set of variables. The set of *sorts*, $\mathcal{S}$ is $\{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$.

The typing rules, that select the *well-typed* terms from the pseudo terms, are given in Figure 2. Here, s ranges over the set of sorts $\mathcal{S}$.

In the rules (var) and (weak) it is always assumed that the newly declared variable is fresh, that is, it has not yet been declared in Γ. The equality in the conversion rule (conv) is the β-equality on the set of pseudo terms $\mathcal{T}$.

(axiom)	$\vdash \mathsf{Prop} : \mathsf{Type}$	$\vdash \mathsf{Type} : \mathsf{Type}'$
(var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$	
(weak)	$\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$	
(Π)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$	if $(s_1, s_2) \in \{ (\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop}) \}$
(λ)	$\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$	
(app)	$\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$	
(conv)	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	if $A =_\beta B$

Figure 2: Typing rules for $\lambda\mathsf{HOL}$

We see that there is no distinction between types and terms in the sense that the types are formed first and then the terms are formed using the types. A pseudo term A is *well-typed* if there is a context Γ and a pseudo term B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. The set of well-typed terms of $\lambda\mathsf{HOL}$ is denoted by Term($\lambda\mathsf{HOL}$). A context Γ is *well-formed* if it appears in some derivable statement, i.e. if there are some M and A such that $\Gamma \vdash M : A$ is derivable.

The only type-forming operator in this language is the Π, which comes in three flavors, depending on the type of the *domain* (the A in $\Pi x{:}A.B$) and the type of the *range* (the B in $\Pi x{:}A.B$). Intuitively, a Π-type should be read as a set of functions. If we depict the occurrences of x in B explicitly by writing $B(x)$, the intuition is:

$$\Pi x{:}A.B(x) \approx \prod_{a \in A} B(a) = \{ f \mid \forall a{\in}A[f\ a{\in}B(a)]\}.$$

So, $\Pi x{:}A.B$ is the *dependent function type* of functions taking a term of type A as input and delivering a term of type B in which x is replaced by the input. We therefore immediately recover the ordinary function type as a special instance.

3.8. REMARK. In case $x \notin \mathrm{FV}(B)$, we write $A \to B$ for $\Pi x{:}A.B$. We call this a *non-dependent* function type.

As examples we list all instances of the Π-type that can be encountered in λHOL.

3.9. EXAMPLE.

1. Using the combination (Type,Type), we can form the function type $A \to B$ for $A, B{:}\mathsf{Type}$. This also comprises the types of unary predicates and binary relations: $A \to \mathsf{Prop}$ and $A \to A \to \mathsf{Prop}$. Furthermore, it also extends to higher order predicate types like $(A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$.
   If $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x{:}A \vdash B : \mathsf{Type}$, then $x \notin \mathrm{FV}(B)$ in λHOL, so all types formed by (Type,Type) are non-dependent function types.
2. Using (Prop,Prop), we can form the propositional type $\varphi \to \psi$ for $\varphi, \psi{:}\mathsf{Prop}$. This is to be read as an *implicational formula*.
   If $\Gamma \vdash \varphi : \mathsf{Prop}$ and $\Gamma, x{:}\varphi \vdash \psi : \mathsf{Prop}$, then $x \notin \mathrm{FV}(\psi)$ in λHOL, so all types formed by (Prop,Prop) are non-dependent types.
3. Using (Type,Prop), we can form the dependent propositional type $\Pi x{:}A.\varphi$ for $A{:}\mathsf{Type}$, $\varphi{:}\mathsf{Prop}$. This is to be read as a *universally quantified formula*. This quantification can also range over higher order domains, like in $\Pi P{:}A \to A \to \mathsf{Prop}.\varphi$.
   If $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x{:}A \vdash \varphi : \mathsf{Prop}$, then it *can* happen that $x \in \mathrm{FV}(\varphi)$ in λHOL.

We do not define formal interpretations from **HOL** to λHOL and back. See e.g. [Barendregt 1992] for details. Instead, we motivate the interpretation by some (suggestive) examples. Then we discuss the main assets of the interpretation and motivate its completeness.

For a good reading of the examples below, we recall the notational conventions introduced in 3.2: Rab denotes $((R\ a)\ b)$, so R applied to a and that together applied to b. Moreover, application binds strong, so $Rab \to Rbc$ denotes $(Rab) \to (Rbc)$ and $\lambda x{:}Rab.M$ denotes $\lambda x{:}(Rab).M$. As usual, arrow associate to the right, so $A \to A \to \mathsf{Prop}$ denotes $A \to (A \to \mathsf{Prop})$.

3.10. EXAMPLE.

1. $\mathbb{N}{:}\mathsf{Type}, 0{:}\mathbb{N}, >{:}\mathbb{N} \to \mathbb{N} \to \mathsf{Prop} \vdash \lambda x{:}\mathbb{N}.x>0 : \mathbb{N} \to \mathsf{Prop}$. Here we see the use of λ-abstraction to define predicates.
2. $$\mathbb{N}{:}\mathsf{Type}, 0{:}\mathbb{N}, S{:}\mathbb{N} \to \mathbb{N} \vdash \ \Pi P{:}\mathbb{N} \to \mathsf{Prop}.(P0) \to (\Pi x{:}\mathbb{N}.(Px \to P(Sx))) \to \Pi x{:}\mathbb{N}.Px : \mathsf{Prop}.$$
   This is the formula for induction written down in λHOL as a term of type Prop.
3. $A{:}\mathsf{Type}, R{:}A \to A \to \mathsf{Prop} \vdash \Pi x, y, z{:}A.Rxy \to Ryz \to Rxz : \mathsf{Prop}$. (Transitivity of R)
4. $A{:}\mathsf{Type} \vdash \lambda R, Q{:}A \to A \to \mathsf{Prop}.\Pi x, y{:}A.Rxy \to Qxy :$
   $(A \to A \to \mathsf{Prop}) \to (A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$. (Inclusion of relations)

5. $A{:}\mathsf{Type} \vdash \lambda x, y{:}A.\Pi P{:}A{\to}\mathsf{Prop}.(Px{\to}Py) : A{\to}A{\to}\mathsf{Prop}$.
   This is 'Leibniz equality' and is usually denoted by $=_A$, mentioning the domain type explicitly.
6. $A{:}\mathsf{Type}, x, y{:}A \vdash \lambda r{:}(x =_A y).\lambda P{:}A{\to}\mathsf{Prop}.r(\lambda z{:}A.Pz{\to}Px)(\lambda q{:}Px.q) :$ $(x =_A y){\to}(y =_A x)$. The proof of the fact that Leibniz equality is symmetric.

Just as in **HOL**, it is possible to define the ordinary connectives &, $\vee$, $\bot$, $\neg$ and $\exists$ in λHOL. For $\varphi, \psi{:}\mathsf{Prop}$, define

$$\begin{array}{rcl} \varphi\&\psi & := & \Pi\alpha{:}\mathsf{Prop}.(\varphi{\to}\psi{\to}\alpha){\to}\alpha, \\ \varphi \vee \psi & := & \Pi\alpha{:}\mathsf{Prop}.(\varphi{\to}\alpha){\to}(\psi{\to}\alpha){\to}\alpha, \\ \bot & := & \Pi\alpha{:}\mathsf{Prop}.\alpha, \\ \neg\varphi & := & \varphi{\to}\bot, \\ \exists x{:}A.\varphi & := & \Pi\alpha{:}\mathsf{Prop}.(\Pi x{:}A.(\varphi{\to}\alpha)){\to}\alpha. \end{array}$$

To form these propositions (terms of type Prop), the rules (Prop,Prop) (for all the arrows) and (Type,Prop) (for all the Π-types) are used.

The logical rules for these connectives can be derived. For example, for $\varphi\&\psi$, we have terms $\pi_1 : (\varphi\&\psi){\to}\varphi$ and $\pi_2 : (\varphi\&\psi){\to}\psi$ (the projections) and a term $\langle -, - \rangle : \varphi{\to}\psi{\to}(\varphi\&\psi)$ (the pairing constructor). One can easily verify that if we take

$$\begin{array}{rcl} \pi_1 & := & \lambda p{:}(\varphi\&\psi).p\varphi(\lambda x{:}\varphi.\lambda y{:}\psi.x), \\ \pi_2 & := & \lambda p{:}(\varphi\&\psi).p\psi(\lambda x{:}\varphi.\lambda y{:}\psi.y), \\ \langle -, - \rangle & := & \lambda x{:}\varphi.\lambda y{:}\psi.\lambda\alpha{:}\mathsf{Prop}.\lambda h{:}(\varphi{\to}\psi{\to}\alpha).hxy, \end{array}$$

then these terms are of the right type. Hence the introduction and elimination rules for the connective & are definable. They also have the correct reduction behavior, corresponding to cut-elimination in the logic:

$$\begin{array}{rcl} \pi_1\langle t_1, t_2\rangle & \twoheadrightarrow_\beta & t_1, \\ \pi_2\langle t_1, t_2\rangle & \twoheadrightarrow_\beta & t_2. \end{array}$$

Similarly for the other connectives, the introduction and elimination rules can be defined.

Note that on the Type level, it is not possible to define data types, like the product type. A product type is equivalent to the conjunction (&), but the construction above for & can only be done at the Prop level.

Propositions-as-types for higher order predicate logic

The propositions-as-types interpretation from higher order predicate logic **HOL** into λHOL maps a formula to a type and a proof (a derivation in natural deduction) of a formula φ to a term (i.e. a typed λ-term) of the type associated with φ:

$$\begin{array}{c}\boxed{\;\Sigma\;}\\ \psi\end{array} \quad \mapsto \quad [\![\Sigma]\!] : (\!|\psi|\!)$$

where $(\!|-|\!)$ denotes the interpretation of formulas as types and $[\![-]\!]$ denotes the interpretation of derivations as λ-terms. In a derivation, we use expressions from the logical language (e.g. to instantiate the $\forall$), which may contain free variables, constants and domains (e.g. in $f(\lambda x{:}A.c)$). In type theory, in order to make sure that all terms are well-typed, the basic items (like variables and domains) have to be declared explicitly in the context. Also, a derivation will in general contain non-discharged assumptions $(\varphi_1, \ldots, \varphi_n)$ that will appear as variable declarations $(z_1 : \varphi_1, \ldots, z_n : \varphi_n)$ in the type theoretic context. So the general picture is this.

$$\begin{array}{c}\varphi_1 \ldots \varphi_n\\ \boxed{\;\Sigma\;}\\ \psi\end{array} \quad \mapsto \quad \Gamma_\Sigma, z_1 : \varphi_1, \ldots, z_n : \varphi_n \vdash [\![\Sigma]\!] : (\!|\psi|\!),$$

where Γ_Σ is the context that declares all domains, constants and free variables that occur in Σ.

As an example we treat the derivation of irreflexivity from anti-symmetry for a relation R. The derivation is as follows. (Γ denotes $\forall x^A y^A Rxy \Rightarrow Ryx \Rightarrow \bot$, Γ' denotes Γ, Rxx.)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash \forall x^A y^A.Rxy \;\Rightarrow\; Ryx \;\Rightarrow\; \bot}{\Gamma' \vdash \forall y^A.Rxy \;\Rightarrow\; Ryx \;\Rightarrow\; \bot}}{\Gamma' \vdash Rxx \;\Rightarrow\; Rxx \;\Rightarrow\; \bot} \qquad \Gamma' \vdash Rxx}{\Gamma' \vdash Rxx \;\Rightarrow\; \bot} \qquad \Gamma' \vdash Rxx}{\dfrac{\Gamma' \vdash \bot}{\Gamma \vdash Rxx \;\Rightarrow\; \bot}}}{\Gamma \vdash \forall x^A.Rxx \;\Rightarrow\; \bot}$$

This derivation is mapped to the typed λ-term $\lambda x{:}A.\lambda q{:}(Rxx).zxxqq$. This term is well-typed in the context $A : \mathsf{Type}, R : A{\to}A{\to}\mathsf{Prop}, z : \Pi x, y{:}A.(Rxy{\to}Ryx{\to}\bot)$, yielding the following judgment, derivable in $\lambda\mathsf{HOL}$ if we take for Γ the context $\Gamma = \{A{:}\mathsf{Type}, R{:}A{\to}A{\to}\mathsf{Prop}, z{:}\Pi x, y{:}A.(Rxy{\to}Ryx{\to}\bot)\}$.

$$\Gamma \vdash \lambda x{:}A\lambda q{:}(Rxx).zxxqq : (\Pi x{:}A.Rxx{\to}\bot).$$

The context Γ_Σ here consists of $A : \mathsf{Type}, R : (A{\to}A{\to}\mathsf{Prop})$.

Now one may wonder if the type system $\lambda\mathsf{HOL}$ is really faithful to higher order predicate logic **HOL**. Put differently, one can ask the question of *completeness*: given a proposition of **HOL** such that $\Gamma_\varphi \vdash M : (\!|\varphi|\!)$ in $\lambda\mathsf{HOL}$, is φ derivable in **HOL**? It turns out that this is the case. Even though the number of rules of $\lambda\mathsf{HOL}$ is limited (where one rule serves several different purposes, e.g. the (λ)-rule allows to form both functions, proofs of an implication and proofs of a universal quantification) and there seems to be hardly any distinction in treatment between the propositions (terms of type Prop) and the sets (terms of type Type), we can completely *disambiguate* the syntax. This is stated by the following Lemma.

3.11. Lemma (Disambiguation Lemma). *Given a judgment $\Gamma \vdash M : A$ in $\lambda\mathsf{HOL}$, there is a $\lambda\mathsf{HOL}$-context $\Gamma_D, \Gamma_L, \Gamma_P$ such that*

1. $\Gamma_D, \Gamma_L, \Gamma_P$ *is a* permutation *of* Γ,
2. $\Gamma_D, \Gamma_L, \Gamma_P \vdash M : A$
3. Γ_D *consists only of declarations* $A : \mathsf{Type}$,
4. Γ_L *consists only of declarations* $x : A$ *with* $\Gamma_D \vdash A : \mathsf{Type}$,
5. Γ_P *consists only of declarations* $z : \varphi$ *with* $\Gamma_D, \Gamma_L \vdash \varphi : \mathsf{Prop}$.

Moreover the following are the case.

- *If* $\Gamma \vdash A : \mathsf{Type}$, *then* $\Gamma_D \vdash A : \mathsf{Type}$ *and* $A \equiv B_1 \to \cdots \to B_n$ $(n \geq 1)$ *and* $\Gamma_D \vdash B_i : \mathsf{Type}$ *for all* i.
- *If* $\Gamma \vdash M : A$ *where* $\Gamma \vdash A : \mathsf{Type}$, *then* $\Gamma_D, \Gamma_L \vdash M : A$.
- *If* $\Gamma \vdash \Pi x{:}A.B : \mathsf{Prop}$ *where* $\Gamma \vdash A : \mathsf{Prop}$, *then* $x \notin \mathrm{FV}(B)$ *(and so* $\Pi x{:}A.B \equiv A \to B$, *representing a real implication).*

The Disambiguation Lemma really states that $\lambda\mathsf{HOL}$ represents **HOL** very closely. Note that it says—among other things—that proof-terms (terms M with $M : \varphi$ for some $\varphi : \mathsf{Prop}$) do not occur in object-terms (terms $t : A$ with for some $A : \mathsf{Type}$). Using the Lemma, one can define a mapping back from $\lambda\mathsf{HOL}$ to **HOL** that constructs a derivation out of a proof-term. Let a ψ with $\Gamma \vdash \psi : \mathsf{Prop}$ be given.

$$\Gamma \vdash M : \psi \quad \mapsto \quad \dfrac{\begin{array}{c}\varphi_1 \ldots \varphi_n \\ {[M]}\end{array}}{\psi}$$

Here the $\varphi_1 \ldots \varphi_n$ are computed from Γ, using Lemma 3.11, in such a way that $\Gamma_P = z_1 : \varphi_1, \ldots, z_n : \varphi_n$.

The mapping back from $\lambda\mathsf{HOL}$ to **HOL** proofs the completeness of the propositions-as-types interpretation: if $\Gamma \vdash M : \varphi$, then φ is derivable in **HOL** from the assumptions listed in Γ_P.

Type Checking

An important property of a type system is *computability of types*, i.e. given Γ and M compute an A for which $\Gamma \vdash M : A$ holds, and if there is no such A, return 'false'. This is usually called the *type synthesis problem, TSP* or the *type inference problem*. In this Section, a type synthesis algorithm for the system λHOL is given, which is quite reminiscent for type synthesis algorithms in general. Before discussing the details we briefly recapitulate some generalities on type synthesis and type checking. See also Sections 2.2, 2.3.

A problem related to type synthesis is *decidability of typing*, i.e. given Γ, M and A, decide whether $\Gamma \vdash M : A$ holds. This is usually called the *type checking problem, TCP*. Both problems are very much related, because in the process of type checking, one has to solve type synthesis problems as well: for example when *checking* whether $MN : C$, one has to *infer* a type for N, say A, and a type for M, say D, and then to check whether for some B, $D =_\beta \Pi x{:}A.B$ with $B[N/x] =_\beta C$. It should be clear from this case that type synthesis and type checking are closely entwined. (See Section 2.3 for an extended example.) The crucial algorithm to construct is an algorithm $\mathrm{Type}_{-}(-)$, that takes a context Γ and a term M such that

$$\mathrm{Type}_\Gamma(M) =_\beta A \quad \Leftrightarrow \quad \Gamma \vdash M : A.$$

Hence, one will need an algorithm for *β-equality checking* to decide typing.

There are two important properties that solve the decidability of β-equality checking: Confluence for β-reduction and Strong Normalization for β-reduction. (This is a well-known fact from rewriting: if a rewriting relation is confluent and strongly normalizing, then the induced equality relation is decidable: to determine $M =_\beta N$, one reduces M and N to normal form and compares these normal forms.)

3.12. Proposition (Confluence). *On the set of pseudo terms $\mathcal{T}$, β-reduction is* confluent *i.e. for all $M, N_1, N_2 \in \mathcal{T}$, if $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then there exists a $P \in \mathcal{T}$ such that $N_1 \twoheadrightarrow_\beta P$ and $N_2 \twoheadrightarrow_\beta P$.*

Confluence for β can be proved by following the well-known proofs for confluence for the untyped λ-calculus. Another important property of λHOL is Subject Reduction.

3.13. Proposition (Subject Reduction). *The set of well-typed terms of a given type is closed under reduction. That is, for Γ a context and M, N, A in $\mathcal{T}$, if $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash N : A$.*

See Section 2.6 for a discussion on Subject Reduction and Section 3.3 for a list of properties for λHOL (among which Subject Reduction). The following is a consequence of confluence on $\mathcal{T}$ and Subject Reduction.

3.14. Corollary (Confluence on well-typed terms). *On the set of well-typed terms of λHOL, β-reduction is confluent. That is, for M well-typed, if $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then there exists a well-typed term P such that $N_1 \twoheadrightarrow_\beta P$ and $N_2 \twoheadrightarrow_\beta P$. Moreover, N_1 and N_2 are well-typed.*

3.15. PROPOSITION (Strong Normalization). *For any term M well-typed in $\lambda\mathsf{HOL}$, there are no infinite β-reduction paths starting from M. (Put differently: all reductions starting from a well-typed term terminate.)*

The proof of this Proposition is rather involved. See [Barendregt 1992] for references to proofs.

The *type synthesis algorithm* $\mathrm{Type}_{\_}(-)$ attempts to apply the typing rules in the reverse direction. For example, computing $\mathrm{Type}_\Gamma(\lambda x{:}A.M)$ is done by computing $\mathrm{Type}_{\Gamma,x:A}(M)$, and if this yields B, computing $\mathrm{Type}_\Gamma(\Pi x{:}A.B)$. If this returns a $s{\in}\{\mathsf{Prop},\mathsf{Type}\}$, then we return $\Pi x{:}A.B$ as result of $\mathrm{Type}_\Gamma(\lambda x{:}A.M)$. So, we read the ($\lambda$)-rule in the reverse direction.

There is a potential problem in this way of constructing the $\mathrm{Type}_{\_}(-)$ algorithm by reversing the rules: a conclusion $\Gamma \vdash \lambda x{:}A.M : C$ need not have been obtained from the (λ)-rule. (It could also be a conclusion of the (weak)-rule or the (conv)-rule. This situation is usually referred to as the 'non-syntax-directedness' of the derivation rules. A set of derivation rules is called *syntax-directed* if, given a context Γ and a term M, *at most* one rule can have as conclusion $\Gamma \vdash M : C$ (for some C). See [Pollack 1995] and [van Benthem Jutting et al. 1994] for more on syntax-directed sets of rules for type systems and their advantages. We will treat the (potential) problem of non-syntax-directedness later when we discuss the soundness and completeness of the $\mathrm{Type}_{\_}(-)$ algorithm.

Another part of the algorithm that needs some special attention is the variable case. The result of $\mathrm{Type}_\Gamma(x)$ should be A if $x{:}A$ occurs in Γ and 'false' otherwise. But, if Γ is not a well-formed context, we want to return 'false' as well! So we have to check the well-formedness of Γ. A type synthesis algorithm consists of two mutually dependent recursive functions: $\mathrm{Type}_{\_}(-)$, the real type synthesis algorithm, and the *context checking algorithm* $\mathrm{Ok}(-)$. The latter takes as input a context and returns 'true' if and only if the context is well-formed (and 'false' otherwise).

3.16. DEFINITION. We define the algorithms $\mathrm{Ok}(-)$, taking a context and returning 'true' or 'false', and $\mathrm{Type}_{\_}(-)$, taking a context and a term and returning a term or 'false', as follows. Here x denotes a variable.

$$
\begin{array}{rcl}
\mathrm{Ok}(<>) & = & \text{`true' (the empty context)},\\
\mathrm{Ok}(\Gamma, x{:}A) & = & \mathrm{Type}_\Gamma(A){\in}\{\mathsf{Prop},\mathsf{Type},\mathsf{Type}'\},\\
\mathrm{Type}_\Gamma(x) & = & \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{and}\ x{:}A{\in}\Gamma\,\mathsf{then}\ A\ \mathsf{else}\ \text{`false'},\\
\mathrm{Type}_\Gamma(\mathsf{Prop}) & = & \mathsf{if}\ \mathrm{Ok}(\Gamma)\mathsf{then}\ \mathsf{Type}\ \mathsf{else}\ \text{`false'},\\
\mathrm{Type}_\Gamma(\mathsf{Type}) & = & \mathsf{if}\ \mathrm{Ok}(\Gamma)\mathsf{then}\ \mathsf{Type}'\ \mathsf{else}\ \text{`false'},\\
\mathrm{Type}_\Gamma(\mathsf{Type}') & = & \text{`false'},\\
\mathrm{Type}_\Gamma(MN) & = & \mathsf{if}\ \mathrm{Type}_\Gamma(M) = C\ \mathsf{and}\ \mathrm{Type}_\Gamma(N) = D\\
 & & \quad \mathsf{then} \quad \mathsf{if}\ C \twoheadrightarrow_\beta \Pi x{:}A.B\ \mathsf{and}\ A =_\beta D\\
 & & \qquad\qquad \mathsf{then}\ B[N/x]\ \mathsf{else}\ \text{`false'}\\
 & & \quad \mathsf{else} \quad \text{`false'},
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{Type}_\Gamma(\lambda x{:}A.M) & = & \mathsf{if}\ \mathrm{Type}_{\Gamma,x:A}(M) = B \\
& & \quad \mathsf{then} \quad \mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\} \\
& & \qquad\qquad \mathsf{then}\ \Pi x{:}A.B\ \mathsf{else}\ \text{‘false’} \\
& & \quad \mathsf{else}\ \text{‘false’}, \\
\mathrm{Type}_\Gamma(\Pi x{:}A.B) & = & \mathsf{if}\ \mathrm{Type}_\Gamma(A) = s_1\ \mathsf{and}\ \mathrm{Type}_{\Gamma,x:A}(B) = s_2 \\
& & \qquad\qquad \mathsf{and}\ s_1, s_2 \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\} \\
& & \quad \mathsf{then} \quad \mathsf{if}\ (s_1, s_2) \in \{\ (\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), \\
& & \qquad\qquad\qquad (\mathsf{Type}, \mathsf{Prop})\ \} \\
& & \qquad\qquad \mathsf{then}\ s_2\ \mathsf{else}\ \text{‘false’} \\
& & \quad \mathsf{else}\ \text{‘false’},
\end{array}
$$

The intuition behind the type synthesis algorithm being clear, we want to prove that it is sound and complete. This means proving the following.

3.17. DEFINITION. The type synthesis algorithm $\mathrm{Type}_{\_}(-)$ is *sound* if for all Γ and M,

$$\mathrm{Type}_\Gamma(M) = A \ \Rightarrow\ \Gamma \vdash M : A.$$

The type synthesis algorithm $\mathrm{Type}_{\_}(-)$ is *complete* if for all Γ, M and A,

$$\Gamma \vdash M : A \ \Rightarrow\ \mathrm{Type}_\Gamma(M) =_\beta A.$$

Note that completeness of $\mathrm{Type}_{\_}(-)$ implies that if $\mathrm{Type}_\Gamma(M) =$ ‘false’, then M is not typable in Γ. The definition of completeness only makes sense if we have *uniqueness of types*:

$$\text{If } \Gamma \vdash M : A \text{ and } \Gamma \vdash M : B, \text{ then } A =_\beta B.$$

This property holds for $\lambda\mathsf{HOL}$. Without uniqueness of types, we would have to let $\mathrm{Type}_{\_}(-)$ generate a *set of possible types*, for otherwise it could happen that a valid type A for M in Γ is not computed (up to $=_\beta$) by $\mathrm{Type}_\Gamma(M)$.

Besides soundness and completeness, we want to know that $\mathrm{Type}_{\_}(-)$ terminates on all inputs, i.e. it should be a *total* function. (A sound and complete algorithm may still not terminate on some non-typable term.) We will deal with soundness, termination and completeness now.

3.18. PROPOSITION (Soundness of $\mathrm{Type}_{\_}(-)$). *The* type synthesis algorithm *and the* context checking algorithm*,* $\mathrm{Type}_{\_}(-)$ *and* $\mathrm{Ok}(-)$*, are sound, i.e. if* $\mathrm{Type}_\Gamma(M) = A$*, then* $\Gamma \vdash M : A$ *and if* $\mathrm{Ok}(\Gamma) =$ ‘true’*, then* Γ *is well-formed.*

The proof of soundness of $\mathrm{Type}_{\_}(-)$ and $\mathrm{Ok}(-)$ is simultaneously, by induction on the number of evaluation-steps required for the algorithm to terminate. (Soundness states a property only for those inputs for which the algorithm terminates.) The only interesting case is $\text{Type}_\Gamma(MN)$, where one has to use the Subject Reduction property and Confluence.

The termination of $\text{Type}_\_(-)$ and $\text{Ok}(-)$ should also be proved simultaneously, by devising a measure that decreases with every recursive call. We define the measure m for a context Γ or a pair of a context Γ and a term M as follows.

$$\begin{aligned} \mathrm{m}(\Gamma) &:= \#\{\text{symbols in } \Gamma\}, \\ \mathrm{m}(\Gamma, M) &:= \#\{\text{symbols in } \Gamma, M\}. \end{aligned}$$

Now, m decreases for every recursive call of $\text{Type}_\_(-)$ or $\text{Ok}(-)$, except for the case of $\text{Type}_\Gamma(\lambda x{:}A.M)$, where $\mathrm{m}(\Gamma, \Pi x{:}A.B)$ may be larger than $\mathrm{m}(\Gamma, \lambda x{:}A.M)$ (if B is longer then M). So, the only problem with termination is in the side-condition of the (λ)-rule, where we have to verify whether $\Pi x{:}A.B$ is a well-typed type. This is a situation encountered very generally in type synthesis algorithms for dependent type theory. See [Pollack 1995] and [Severi 1998] for some general solutions to this problem and a discussion. In the case of $\lambda\mathsf{HOL}$, there is a rather easy way out: we can replace the side-condition $\Gamma \vdash \Pi x{:}A.B : s$ in the (λ)-rule by an equivalent but simpler one.

3.19. LEMMA. *Let $\Gamma, x{:}A$ be a context and B be a term. Suppose $\Gamma, x{:}A \vdash M : B$ for some M. Then the following holds.*

$$\begin{array}{lll} \Gamma \vdash \Pi x{:}A.B : s & \Leftrightarrow & \textit{if}\ B \equiv C_0 \to \cdots \to C_n \ \textit{for some}\ n \in \mathbb{N}\ \textit{with} \\ & & (C_n \equiv \mathsf{Prop} \vee (C_n \equiv z \ \textit{for some}\ z \textit{with}\ (z{:}\mathsf{Type}) \in \Gamma)) \\ & & \quad \textit{then} \quad \Gamma \vdash A : \mathsf{Type} \\ & & \quad \textit{else} \quad \textit{if}\ B \not\equiv \mathsf{Type}, \mathsf{Type}'\ \textit{then}\ \Gamma \vdash A : \mathsf{Prop} \end{array}$$

When applying the type synthesis algorithm to a λ-abstraction, we will replace the part '$\mathsf{if}\ \text{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$' by the equivalent condition given in the Lemma.

3.20. DEFINITION. The new *type synthesis algorithm* $\text{Type}_\_(-)$ and the *context checking algorithm* $\text{Ok}(-)$ are defined by replacing in the case $\text{Type}_\Gamma(\lambda x{:}A.B)$ the part
$\mathsf{if}\ \text{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$ by

$$\begin{array}{l} \mathsf{if} \quad B \equiv C_0 \to \cdots \to C_n \ \text{for some}\ n \in \mathbb{N}\ \mathsf{with} \\ \quad (C_n \equiv \mathsf{Prop} \vee (C_n \equiv z \ \text{for some}\ z\ \mathsf{with}\ z{:}\mathsf{Type} \in \Gamma)) \\ \qquad \mathsf{then} \quad \text{Type}_\Gamma(A) = \mathsf{Type} \\ \qquad \mathsf{else} \quad \mathsf{if}\ B \not\equiv \mathsf{Type}, \mathsf{Type}' \mathsf{then}\ \text{Type}_\Gamma(A) = \mathsf{Prop} \end{array}$$

Note that the algorithm only verifies this condition when the premise in the Lemma is satisfied. The new condition may look rather complicated, but it is decidable and now all the recursive calls are done to inputs with a smallest measure. We remark that, this slight variation of the type synthesis algorithm is still sound.

To establish termination, we have to verify that all side conditions are decidable. Here the only work is in the application case: in computing $\mathrm{Type}_\Gamma(MN)$, we have to check a β-equality and we have to check whether a term reduces to a Π-type. In general, checking β-equality on pseudo terms is *not decidable* because we have the full expressive power of the untyped λ-calculus. However, due to the soundness of the algorithm (Proposition 3.18), we know that the intermediate results in the computation of $\mathrm{Type}_\Gamma(MN)$, C and D, are typable terms. Now, β-equality is decidable for typable terms, due to Strong Normalization and Confluence. Hence all side conditions are decidable. To make the algorithm fully deterministic we search the $\Pi x{:}A.B$ (in $C \twoheadrightarrow_\beta \Pi x{:}A.B$) by computing the *weak-head-normal-form* (which exists, due to Strong Normalization).

3.21. PROPOSITION. *The algorithms* $\mathrm{Type}_{\_}(-)$ *and* $\mathrm{Ok}(-)$ *terminate on all inputs.*

Now we come to the completeness of the algorithms. Usually this is proved by defining a different set of derivation rules (1) that is equivalent to the original one (i.e. they have the same set of derivable statements $\Gamma \vdash M : A$), (2) for which the completeness of the algorithm are easy to prove. In order to achieve (2), we define a derivation system that is close to the type synthesis algorithm.

3.22. DEFINITION. The *modified derivation rules* of $\lambda\mathsf{HOL}$ are to derive two forms of judgment: $\Gamma \vdash^{\mathrm{tc}} M : A$ and $\Gamma \vdash^{\mathrm{tc}} \mathsf{ok}$. They are given by the original rules of $\lambda\mathsf{HOL}$, except that

- The rules (ax), (weak), (var) and (conv) are removed,
- The following rules are added.

(empty)	$\langle\rangle \vdash^{\mathrm{tc}} \mathsf{ok}$	
(proj)	$\dfrac{\Gamma \vdash^{\mathrm{tc}} \mathsf{ok}}{\Gamma \vdash^{\mathrm{tc}} x : A}$	if $(x{:}A) \in \Gamma$
(sort)	$\dfrac{\Gamma \vdash^{\mathrm{tc}} \mathsf{ok}}{\Gamma \vdash^{\mathrm{tc}} \mathsf{Prop} : \mathsf{Type}}$	$\dfrac{\Gamma \vdash^{\mathrm{tc}} \mathsf{ok}}{\Gamma \vdash^{\mathrm{tc}} \mathsf{Type} : \mathsf{Type}'}$
(context)	$\dfrac{\Gamma \vdash^{\mathrm{tc}} A : s}{\Gamma, x{:}A \vdash^{\mathrm{tc}} \mathsf{ok}}$	

- The (app) rule is replaced by

$$\text{(app)} \quad \frac{\Gamma \vdash^{\mathrm{tc}} M : C \quad \Gamma \vdash^{\mathrm{tc}} N : D}{\Gamma \vdash^{\mathrm{tc}} MN : B[N/x]} \quad \text{if } C \twoheadrightarrow_\beta \Pi x{:}A.B \text{ and } D =_\beta A$$

We state the following properties for the modified derivation rules.

3.23. PROPOSITION. *1. Soundness of the modified rules*

$$\Gamma \vdash^{\mathrm{tc}} M : A \Rightarrow \Gamma \vdash M : A$$

2. Completeness of the modified rules

$$\Gamma \vdash M : A \Rightarrow \exists A_0[A_0 =_\beta A \;\&\; \Gamma \vdash M : A_0]$$

3. Completeness of the modified rules w.r.t $\mathrm{Type}_{-}(-)$ *and* $\mathrm{Ok}(-)$

$$\begin{array}{rcl} \Gamma \vdash^{\mathrm{tc}} M : A & \Rightarrow & \mathrm{Type}_\Gamma(M) =_\beta A, \\ \Gamma \vdash^{\mathrm{tc}} \mathsf{ok} & \Rightarrow & \mathrm{Ok}(\Gamma) = \text{‘true’}. \end{array}$$

All cases in the proof of this Proposition are by an easy induction.

3.3. Pure Type Systems

The system $\lambda\mathsf{HOL}$ is just an instance of a general class of typed λ calculi, the so-called ‘Pure Type Systems’ or PTSs. These were first introduced by Berardi [1988] and Terlouw [1989], under different names and with slightly different definitions, as a generalization of the so called *λ-cube*, see [Barendregt 1992]. The reason for defining the class of PTSs is that many known systems are (or better: can be seen as) PTSs. This makes it fruitful to study the general properties of PTSs in order to obtain many specific results for specific systems as immediate instances. In what follows we will mention a number of these properties. Another advantage is that the PTSs can be used as a framework for comparing type systems and for defining translations between them.

Pure Type Systems are an immediate generalization of $\lambda\mathsf{HOL}$ if we just note the following parameters in the definition of $\lambda\mathsf{HOL}$.

- The set of ‘sorts’ $\mathcal{S}$ can be varied. (In $\lambda\mathsf{HOL}$: $\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$.)
- The relation between the sorts can be varied. (In $\lambda\mathsf{HOL}$: $\{\ \mathsf{Type} : \mathsf{Type}', \mathsf{Prop} : \mathsf{Type}\}$.)
- The combinations of sorts for which we allow the construction of Π-types can be varied. (In $\lambda\mathsf{HOL}$: $(\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})$.)

3.24. DEFINITION. For $\mathcal{S}$ a set (the set of sorts), $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ (the set of axioms)and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ (the set of rules), the *Pure Type System* $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is the typed λ-calculus with the deduction rules given in Figure 3. If $s_2 \equiv s_3$ in a triple $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$. In the derivation rules, the expressions are taken from the set of *pseudo terms* $\mathcal{T}$ defined by

$$\mathcal{T} ::= \mathcal{S} \,|\, \mathcal{V} \,|\, (\Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}) \,|\, (\lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T}) \,|\, \mathcal{T}\mathcal{T}.$$

The pseudo term A is *well-typed* if there is a context Γ and a pseudo term B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. The set of well-typed terms of $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is denoted by $\mathrm{Term}(\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R}))$.

(sort)	$\vdash s_1 : s_2$	if $(s_1, s_2) \in \mathcal{A}$
(var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$	if $x \notin \Gamma$
(weak)	$\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$	if $x \notin \Gamma$
(Π)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3}$	if $(s_1, s_2, s_3) \in \mathcal{R}$
(λ)	$\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$	
(app)	$\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$	
(conv)	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A =_\beta B$

Figure 3: Typing rules for PTS

It is instructive to define some PTSs to see how flexible the notion is. In the following, we describe a PTS by just listing the sort, the axioms and the rules in a box. For $\lambda\mathsf{HOL}$ this amounts to the following.

$\lambda\mathsf{HOL}$	
$\mathcal{S}$	$\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$
$\mathcal{A}$	$\mathsf{Prop} : \mathsf{Type}, \mathsf{Type} : \mathsf{Type}'$
$\mathcal{R}$	$(\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Prop})$

To define first order predicate logic as a PTS, we have to make a syntactical distinction between 'first order domains' (over which one can quantify) and 'higher order domains' (over which quantification is not allowed). Therefore, a sort Set is introduced, the sort of first order domains, and associated with that a sort Type^s, the type of Set. The Pure Type System λPRED, representing first order predicate logic, is defined as follows.

λPRED	
$\mathcal{S}$	$\mathsf{Set}, \mathsf{Type}^s, \mathsf{Prop}, \mathsf{Type}$
$\mathcal{A}$	$\mathsf{Set} : \mathsf{Type}^s, \mathsf{Prop} : \mathsf{Type}$
$\mathcal{R}$	$(\mathsf{Set}, \mathsf{Set}), (\mathsf{Set}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Set}, \mathsf{Prop})$

We briefly explain the rules. The rule $(\mathsf{Prop}, \mathsf{Prop})$ is the usual for forming the implication. With $(\mathsf{Set}, \mathsf{Type})$ one can form $A{\to}\mathsf{Prop} : \mathsf{Type}$ and $A{\to}A{\to}\mathsf{Prop} : \mathsf{Type}$, the domains of unary predicates and binary relations. The rule $(\mathsf{Set}, \mathsf{Prop})$ allows the quantification over Set-types: one can form $\Pi x{:}A.\varphi$ ($A : \mathsf{Set}$ and $\varphi : \mathsf{Prop}$, which is to be read as a universal quantification). Using $(\mathsf{Set}, \mathsf{Set})$ one can define function types like the type of binary functions: $A{\to}A{\to}A$, but also $(A{\to}A){\to}A$, which is usually referred to as a 'higher order function type'. So note that λPRED is first order only in the *logical* sense, i.e. quantification over predicate domains (like $A{\to}A{\to}\mathsf{Prop}$) is not allowed.

The system λPRED, as described above, captures quite a lot of first order predicate logic. As a matter of fact it precisely captures *minimal* first order predicate logic with *higher order functions*. The minimality means that there are only two connectives: implication and first order universal quantification. As we are in a first order framework, the other connectives can not be defined. This makes the expressibility rather low, as one can not write down negative formulas. On the other hand, we do have higher order function types. It is possible to define a PTS that captures minimal first order predicate logic exactly (i.e. λPRED without higher order functions). See [Barendregt 1992] for details.

To regain all connectives, λPRED can be extended to the second order or higher order predicate logic (where all connectives are definable). We only treat the extension to higher order predicate logic (λPREDω) here and compare it with $\lambda\mathsf{HOL}$.

λPREDω	
$\mathcal{S}$	$\mathsf{Set}, \mathsf{Type}^s, \mathsf{Prop}, \mathsf{Type}$
$\mathcal{A}$	$\mathsf{Set} : \mathsf{Type}^s, \mathsf{Prop} : \mathsf{Type}$
$\mathcal{R}$	$(\mathsf{Set}, \mathsf{Set}), (\mathsf{Set}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}),$
	$(\mathsf{Set}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})$

The rule $(\mathsf{Type}, \mathsf{Prop})$ allows quantification over domains of type Type, which are $A{\to}\mathsf{Prop}$, $A{\to}A{\to}\mathsf{Prop}$ etcetera. The addition of $(\mathsf{Type}, \mathsf{Type})$ implies that now also $(A{\to}\mathsf{Prop}){\to}\mathsf{Prop} : \mathsf{Type}$ and $((A{\to}\mathsf{Prop}){\to}\mathsf{Prop}){\to}\mathsf{Prop} : \mathsf{Type}$. Quantification is over Type, which covers all higher order domains.

Other well-known typed λ-calculi that can be described as a PTS are simple typed λ-calculus, polymorphic typed λ-calculus (also known as system F, [Girard 1972], [Girard, Lafont and Taylor 1989]), higher order typed λ-calculus (also known as Fω, [Girard 1972]). All these systems can be seen as subsystems of the Calculus of Constructions, [Coquand 1985], [Coquand and Huet 1988]. We define the Calculus of Constructions (CC) as the following PTS.

CC	
$\mathcal{S}$	$*, \Box$
$\mathcal{A}$	$* : \Box$
$\mathcal{R}$	$(*,*), (*,\Box), (\Box,*), (\Box,\Box)$

The aforementioned subsystems can be obtained from this specification by restricting the set of rules $\mathcal{R}$. This decomposition of the Calculus of Constructions is also known as the *cube of typed λ-calculi*, see [Barendregt 1992] for further details. In view of higher order predicate logic, one can understand CC as the system obtained by smashing the sorts Prop and Set into one, $*$. Hence, higher order predicate logic can be done inside the Calculus of Constructions. We describe the map from λPREDω to CC later in this Section in detail.

3.4. Properties of Pure Type Systems

As has already been mentioned, an important motivation for the definition of the general framework of Pure Type Systems is the fact that many important properties can be proved for all PTSs at once. Here, we list the most important properties and discuss them briefly. Proofs can be found in [Geuvers and Nederhof 1991] and [Barendregt 1992]. In the following, unless explicitly stated otherwise, $\vdash$ refers to derivability in an arbitrary PTS. As in λHOL, we define a context Γ to be *well-formed* if $\Gamma \vdash M : A$ for some M and A.

Two basic properties are Thinning, saying that typing judgments remain valid in an extended context, and Substitution, saying that typing judgments remain valid if we substitute well-typed terms.

3.25. Proposition (Thinning). *For Γ a context, Γ' a* well-formed *context and M and A in $\mathcal{T}$, if $\Gamma \vdash M : A$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash M : A$. Here, $\Gamma \subseteq \Gamma'$ denotes that all declarations that occur in Γ, also occur in Γ'.*

3.26. Proposition (Substitution). *For $\Gamma_1, x{:}B, \Gamma_2$ a context, and M, N and A in $\mathcal{T}$, if $\Gamma_1, x{:}B, \Gamma_2 \vdash M : A$ and $\Gamma_1 \vdash N : B$, then $\Gamma_1, \Gamma_2[N/x] \vdash M[N/x] : A[N/x]$. Here, $M[N/x]$ denotes the substitution of N for x in M, which is straightforwardly extended to contexts by substituting in all types in the declarations.*

Two other properties we want to mention here are Strengthening, saying that variables that do not appear in the terms can be omitted from the context, and Subject Reduction, saying that typing is closed under reduction.

3.27. Proposition (Strengthening). *For $\Gamma_1, x{:}B, \Gamma_2$ a context, and M, A in $\mathcal{T}$,*

$$\Gamma_1, x{:}B, \Gamma_2 \vdash M : A \ \ \& \ \ x \notin \mathrm{FV}(\Gamma_2, M, A) \ \Rightarrow\ \Gamma_1, \Gamma_2 \vdash M : A.$$

This property, though intuitively very plausible, is difficult to prove and requires a deep analysis of the typing judgment (see [van Benthem Jutting 1993]). (Note that Strengthening is not an immediate consequence of Substitution, because types may not be inhabited, i.e. there may not be an N such that $\Gamma_1 \vdash N : B$.)

3.28. PROPOSITION (Subject Reduction). *For Γ a context and M, N and A in $\mathcal{T}$, if $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash N : A$.*

There are also many (interesting) properties that hold for specific PTSs or specific classes of PTSs. We mention some of these properties, but first we introduce a new notion.

3.29. DEFINITION. A PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *functional*, also called *singly sorted*, if the relations $\mathcal{A}$ and $\mathcal{R}$ are functions, i.e. if the following two properties hold

$$\begin{aligned} \forall s_1, s_2, s_2' {\in} \mathcal{S} (s_1, s_2), (s_1, s_2') {\in} \mathcal{A} \quad &\Rightarrow \quad s_2 = s_2', \\ \forall s_1, s_2, s_3, s_3' {\in} \mathcal{S} (s_1, s_2, s_3), (s_1, s_2, s_3') {\in} \mathcal{R} \quad &\Rightarrow \quad s_3 = s_3' \end{aligned}$$

All the PTSs that we have encountered so far are functional. In general it is hard to find a 'natural' PTS that is not functional. Functional PTSs share the following nice property.

3.30. PROPOSITION (Uniqueness of Types). *This property holds for* functional *PTSs only. For Γ a context, M, A and B in $\mathcal{T}$, if $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.*

One can sometimes relate results of two different systems by defining an embedding between them. There is one very simple class of embeddings between PTSs.

3.31. DEFINITION. For $T = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ and $T' = \lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ PTSs, a *PTS-morphism from T to T'* is a mapping $f : \mathcal{S} \to \mathcal{S}'$ that preserves the axioms and rules. That is, for all $s_1, s_2 {\in} \mathcal{S}$, if $(s_1, s_2) {\in} \mathcal{A}$ then $(f(s_1), f(s_2)) {\in} \mathcal{A}'$ and if $(s_1, s_2, s_3) {\in} \mathcal{R}$ then $(f(s_1), f(s_2), f(s_3)) {\in} \mathcal{R}'$.

A PTS-morphism f from $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ to $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ extends immediately to a mapping f on pseudo terms and contexts. Moreover, this mapping preserves reduction in a faithful way: $M \to_\beta N$ iff $f(M) \to_\beta f(N)$. We have the following property.

3.32. PROPOSITION. *For T and T' PTSs and f a PTS-morphism from T to T', if $\Gamma \vdash M : A$ in T, then $f(\Gamma) \vdash f(M) : f(A)$ in T'.*

Not all PTSs are Strongly Normalizing. We have the following well-known theorem.

3.33. THEOREM. *The Calculus of Constructions, CC, is Strongly Normalizing.*

The proof is rather involved and can be found in [Geuvers and Nederhof 1991, Coquand and Gallier 1990, Berardi 1990]. More general approaches to proving strong normalization for type systems with dependent types can be found in [Mellies and Werner 1998, Geuvers 1995].

As a consequence we find that many other PTSs are Strongly Normalizing as well. This comprises all the sub-systems of CC and also all systems T for which there is a PTS-morphism from T to CC. (Note that a PTS-morphism preserves infinite reduction paths.)

3.34. COROLLARY. *The following PTSs are all Strongly Normalizing. All subsystems of CC;* λPRED*;* λPREDω*.*

A well-known example of a PTS that is not Strongly Normalizing is $\lambda *$. This generalizes the Calculus of Constructions to the extent where $*$ and $\Box$ are unified, or put differently, the sort of types, $*$, is itself a type.

$\lambda *$	
$\mathcal{S}$	$*$
$\mathcal{A}$	$* : *$
$\mathcal{R}$	$(*, *)$

This PTS is also *inconsistent* in the sense that all types are inhabited (which means, if we view—following the propositions-as-types embedding—the type system as a logic, that all propositions are provable). The original proof of inconsistency of $\lambda *$ is in [Girard 1972]; a very clear exposition can be found in [Coquand 1986], while [Hurkens 1995] has improved and shortened the inconsistency proof considerably. From the inconsistency it easily follows that the system is not normalizing. The PTS $\lambda *$ is also the terminal object in the category of PTSs with PTS-morphisms as arrows.

As a matter of fact, we now have two formalizations of higher order predicate logic as a PTS: $\lambda\mathsf{HOL}$ and λPREDω. We employ the notion of PTS-morphism to see that they are equivalent. ¿From λPREDω to $\lambda\mathsf{HOL}$, consider the PTS-morphism f given by

$$\begin{aligned} f(\mathsf{Prop}) &= \mathsf{Prop}, \\ f(\mathsf{Set}) &= \mathsf{Type}, \\ f(\mathsf{Type}) &= \mathsf{Type}, \\ f(\mathsf{Type}^s) &= \mathsf{Type}'. \end{aligned}$$

One verifies immediately that f preserves $\mathcal{A}$ and $\mathcal{R}$, hence we have

$$\Gamma \vdash_{\lambda\text{PRED}\omega} M : A \Rightarrow f(\Gamma) \vdash_{\lambda\mathsf{HOL}} f(M) : f(A).$$

The inverse of f can almost be described as a PTS-morphism, but not quite. Define the PTS-morphism g from λPREDω to $\lambda\mathsf{HOL}$ as follows.

$$g(\mathsf{Prop}) = \mathsf{Prop},$$

$$\begin{array}{rcl} g(\mathsf{Type}) & = & \mathsf{Set}, \\ g(\mathsf{Type}') & = & \mathsf{Type}^s \end{array}$$

(In $\lambda\mathsf{HOL}$ the sort Type' can not appear in a context nor in a term on the left side of the ':'.) We extend g to derivable judgments of $\lambda\mathsf{HOL}$ in the following way.

$$\begin{array}{rcl} g(\Gamma \vdash M : A) & = & g(\Gamma) \vdash g(M) : g(A), \text{ if } A \neq \mathsf{Type}, \\ g(\Gamma \vdash M : \mathsf{Type}) & = & g(\Gamma) \vdash g(M) : \mathsf{Set}, \text{ if } M \equiv \cdots \to \alpha, (\alpha \text{ a variable}), \\ g(\Gamma \vdash M : \mathsf{Type}) & = & g(\Gamma) \vdash g(M) : \mathsf{Type}, \text{ if } M \equiv \cdots \to \mathsf{Prop}. \end{array}$$

By easy induction one proves that g preserves derivations. Furthermore, $f(g(\Gamma \vdash M : A)) = \Gamma \vdash M : A$ and $g(f(\Gamma \vdash M : A)) = \Gamma \vdash M : A$. Hence, λPREDω and $\lambda\mathsf{HOL}$ are equivalent systems. This equivalence implies that the system $\lambda\mathsf{HOL}$ is Strongly Normalizing as well.

3.5. Extensions of Pure Type Systems

Several features are not present in PTSs. For example, it is possible to define data types (in a polymorphic sort, e.g. Prop in $\lambda\mathsf{HOL}$ or $*$ in CC), but one does not get induction over these data types for free. (It is possible to define functions by recursion, but induction has to be assumed as an axiom.) Therefore, 'inductive types' an extra feature. The way we present them below, they were first defined in [Coquand and Paulin-Mohring 1990]. (See also [Paulin-Mohring 1994].) Inductive types are present in all widely used type-theoretic theorem provers, like [COQ 1999, LEGO 1998, Agda 2000].

Another feature that we will discuss is the notion of product and (strong) Σ-type. A Σ-type is a 'dependent product type' and therefore a generalisation of product type in the same way that a Π-type is a generalisation of arrow type: $\Sigma x{:}A.B$ represents the type of pairs (a, b) with $a : A$ and $b : B[a/x]$. (If $x \notin \mathrm{FV}(B)$, we just end up with $A \times B$.) Besides a pairing construction to create elements of a Σ-type, we have projections to take a pair apart: if $t : \Sigma x{:}A.B$, then $\pi_1 t : A$ and $\pi_2 t : B[\pi_1 t/x]$. Σ-types are very natural for doing abstraction over theories, as was first explained in [Luo 1989]. Products can be defined inside the system if one has polymorphism, but Σ-types cannot.

3.6. Products and Sums

We have already seen how to define conjunction and disjunction in $\lambda\mathsf{HOL}$. These are very close to product-types and sum-types. In Figure 4 the desired rules for a product-type are given. In presence of polymorphism, these constructions are all definable. For example in $\lambda\mathsf{HOL}$ we have products in the sort Prop. Let $A_1, A_2 : \mathsf{Prop}$ and define

$$A_1 \times A_2 \quad := \quad \Pi\alpha{:}\mathsf{Prop}.(A_1 \to A_2 \to \alpha) \to \alpha,$$

$$
\begin{array}{|ll|}
\hline
\text{(products)} & \dfrac{\Gamma \vdash A_1 : s \;\; \Gamma \vdash A_2 : s}{\Gamma \vdash A_1 \times A_2 : s} \\[2ex]
\text{(projection)} & \dfrac{\Gamma \vdash p : A_1 \times A_2}{\Gamma \vdash \pi_i p : A_i} \\[2ex]
\text{(pairing)} & \dfrac{\Gamma \vdash t_1 : A_1 \;\; \Gamma \vdash t_2 : A_2}{\Gamma \vdash \langle t_1, t_2 \rangle : A_1 \times A_2} \\[2ex]
\text{computation rule:} & \pi_i \langle t_1, t_2 \rangle \to t_i \\
\hline
\end{array}
$$

Figure 4: Rules for product types

$$
\begin{array}{rcl}
\pi_1 & := & \lambda p{:}(A_1 \times A_2).pA_1(\lambda x{:}A_1.\lambda y{:}A_2.x), \\
\pi_2 & := & \lambda p{:}(A_1 \times A_2).pA_2(\lambda x{:}A_1.\lambda y{:}A_2.y), \\
\langle -, - \rangle & := & \lambda x{:}A_1.\lambda y{:}A_2.\lambda \alpha{:}\mathsf{Prop}.\lambda h{:}(A_1 \to A_2 \to \alpha).hxy,
\end{array}
$$

For sum-types one would like to have the rules of Figure 5. This can also be

$$
\begin{array}{|ll|}
\hline
\text{(sums)} & \dfrac{\Gamma \vdash A_1 : s \;\; \Gamma \vdash A_2 : s}{\Gamma \vdash A_1 + A_2 : s} \\[2ex]
\text{(injection)} & \dfrac{\Gamma \vdash p : A_i}{\Gamma \vdash \mathsf{in}_i\ p : A_1 + A_2} \\[2ex]
\text{(case)} & \dfrac{\Gamma \vdash f_1 : A_1 \to C \;\; \Gamma \vdash f_2 : A_2 \to C}{\Gamma \vdash \mathsf{case}(f_1, f_2) : (A_1 + A_2) \to C} \\[2ex]
\text{computation rule:} & \mathsf{case}(f_1, f_2)(\mathsf{in}_i\ p) \to f_i p \\
\hline
\end{array}
$$

Figure 5: Rules for sum types

defined in a polymorphic sort (inspired by the $\vee$-construction). Let in $\lambda\mathsf{HOL}$, A_1, A_2 and C be of type Prop, $f_1 : A_1 \to C$ and $f_2 : A_2 \to C$.

$$
\begin{array}{rcl}
A_1 + A_2 & := & \Pi \alpha{:}\mathsf{Prop}.(A_1 \to \alpha) \to (A_2 \to \alpha) \to \alpha, \\
\mathsf{in}_1 & := & \lambda p{:}A_1.\lambda \alpha{:}\mathsf{Prop}.\lambda h_1{:}(A_1 \to \alpha).\lambda h_2{:}(A_2 \to \alpha).h_1 p, \\
\mathsf{in}_2 & := & \lambda p{:}A_2.\lambda \alpha{:}\mathsf{Prop}.\lambda h_1{:}(A_1 \to \alpha).\lambda h_2{:}(A_2 \to \alpha).h_2 p,
\end{array}
$$

$$\mathsf{case}(f_1, f_2) \quad := \quad \lambda x{:}(A_1 + A_2).xCf_1f_2.$$

3.7. Σ-types

In mathematics one wants to be able to reason about abstract notions, like the *theory of groups*. Therefore, in the formalization of mathematics in type theory, we have to be able to form something like the 'type of groups'. As an example, let us see what a group looks like in $\lambda\mathsf{HOL}$. Given $A : \mathsf{Type}$, a *group over* A is a tuple consisting of the terms

$$\begin{array}{rcl} \circ & : & A{\to}A{\to}A \\ e & : & A \\ \mathsf{inv} & : & A{\to}A \end{array}$$

(the group-structure) such that the following types are inhabited (we use infix-notation for readability).

$$\begin{array}{rcl} \Pi x, y, z{:}A.(x \circ y) \circ z & = & x \circ (y \circ z), \\ \Pi x{:}A.e \circ x & = & x, \\ \Pi x{:}A.(\mathsf{inv}\ x) \circ x & = & e. \end{array}$$

For the type of the group-structure we can use the product: the *type of group-structures over* A, $\mathsf{Group\text{-}Str}(A)$, is $(A{\to}A{\to}A) \times (A \times (A{\to}A))$. If $t : \mathsf{Group\text{-}Str}(A)$, then $\pi_1 t : A{\to}A{\to}A$, $\pi_1(\pi_2 t) : A$, etcetera. However, this does not yet capture the axioms of group-theory. For this we can use the Σ-type: the *type of groups over* A, $\mathsf{Group}(A)$, is defined by

$$\begin{array}{rl} \mathsf{Group}(A) := \Sigma \circ{:}A{\to}A{\to}A.\Sigma e{:}A.\Sigma \mathsf{inv}{:}A{\to}A. & (\Pi x, y, z{:}A.(x \circ y) \circ z = x \circ (y \circ z)) \wedge \\ & (\Pi x{:}A.e \circ x = x) \wedge \\ & (\Pi x{:}A.(\mathsf{inv}\ x) \circ x = e). \end{array}$$

Now, if $t : \mathsf{Group}(A)$, we can extract the elements of the group structure by projections as before: $\pi_1 t : A{\to}A{\to}A$, $\pi_1(\pi_2 t) : A$, etcetera. One can also extract proof-terms for the group-axioms by projection: $\pi_1(\pi_2(\pi_2(\pi_2 t))) : \Pi x, y, z{:}A.\pi_1 t(\pi_1 txy)z = \pi_1 tx(\pi_1 tyz)$, representing the associativity of the operation $\pi_1 t$.

Similarly, if $f : A{\to}A{\to}A$, $a : A$ and $h : A{\to}A$ with p_1, p_2, p_3 and p_4 proof-terms of the associated group-axioms, then

$$\langle f, \langle a, \langle h, \langle p_1, \langle p_2, \langle p_3, p_4\rangle\rangle\rangle\rangle\rangle\rangle : \mathsf{Group}(A).$$

The precise rules of the Σ-types in $\lambda\mathsf{HOL}$ are as in Figure 6.

These rules allow the formation of the 'dependent tuples' we need for formalizing notions like Group and Ring. An even more general approach towards the theory

(Σ)	$\dfrac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma, x{:}A \vdash \varphi : \mathsf{Prop}}{\Gamma \vdash \Sigma x{:}A.\varphi : \mathsf{Type}}$
$(\langle -, - \rangle)$	$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash p : \varphi[a/x] \quad \Gamma \vdash \Sigma x{:}A.\varphi : \mathsf{Type}}{\Gamma \vdash \langle a, p \rangle : \Sigma x{:}A.\varphi}$
(π_1)	$\dfrac{\Gamma \vdash t : \Sigma x{:}A.\varphi}{\Gamma \vdash \pi_1 t : A}$
(π_2)	$\dfrac{\Gamma \vdash t : \Sigma x{:}A.\varphi}{\Gamma \vdash \pi_2 t : \varphi[\pi_1 t/x]}$
computation rules:	$\pi_1 \langle a, p \rangle \to a$ $\pi_2 \langle a, p \rangle \to p$

Figure 6: Rules for Σ-types

of groups would be to also abstract over the carrier type, obtaining

$$\begin{aligned}\mathsf{Group} := \quad & \Sigma A{:}\mathsf{Type}.\Sigma \circ{:}A{\to}A{\to}A.\Sigma e{:}A.\Sigma \mathsf{inv}{:}A{\to}A.\\ & (\Pi x, y, z{:}A.(x \circ y) \circ z = x \circ (y \circ z)) \wedge\\ & (\Pi x{:}A.e \circ x = x) \wedge\\ & (\Pi x{:}A.(\mathsf{inv}\ x) \circ x = e).\end{aligned}$$

This can be done by an easy extension of the rules, allowing to form $\Sigma x{:}A.B$ also for $A : \mathsf{Type}'$:

$$(\Sigma') \quad \frac{\Gamma \vdash A : \mathsf{Type}' \quad \Gamma, x{:}A \vdash B : \mathsf{Type}}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Type}}$$

However, if we want the system to remain consistent, it is not possible to allow $\Sigma x{:}\mathsf{Type}.B : \mathsf{Type}$. We must put $\Sigma x{:}\mathsf{Type}.B : \mathsf{Type}'$. This implies that $\mathsf{Group} : \mathsf{Type}'$, which may not be desirable.

We may observe that the Σ-type behaves very much like an *existential quantifier*. Apart from the fact that $\Sigma x{:}A.\varphi$ is not a proposition, but a type, we see that a (proof)term of type $\Sigma x{:}A.\varphi$ is constructed from a term a of type A for which $\varphi[a/x]$ holds. The other way around, from a (proof)term t of type $\Sigma x{:}A.\varphi$ one can construct the *witness* $\pi_1 t$ and the proof that for this witness φ holds. This very closely reflects the constructive interpretation of the existential quantifier ('if $\exists x{:}A.\varphi$ is derivable, then there exists a term a for which $\varphi[a/x]$ is derivable'). The use of Σ-types for the existential quantifier requires that $\Sigma x{:}A.\varphi : \mathsf{Prop}$ (not of type Type) in $\lambda\mathsf{HOL}$.

In order to achieve this we could modify the Σ-rule as follows.

$$(\Sigma) \quad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma, x{:}A \vdash \varphi : \mathsf{Prop}}{\Gamma \vdash \Sigma x{:}A.\varphi : \mathsf{Prop}}$$

However, the addition of this rule to $\lambda\mathsf{HOL}$ makes the system inconsistent. In the case of $\lambda\mathrm{PRED}\omega$, it is possible to add a Σ-type that represents the existential quantifier, while remaining consistent, but only for $A : \mathsf{Set}$. On the other hand, one may wonder whether a Σ-type is the correct formalization of the constructive existential quantifier, because it creates set-terms that depend on proof-terms. For example, if we put $z : \Sigma x{:}A.\varphi$ in the context where $\Sigma x{:}A.\varphi$ is a proposition ($\Sigma x{:}A.\varphi : \mathsf{Prop}$), then $\pi_1 z : A$ ($A : \mathsf{Set}$). So we have an element-expression ($\pi_1 z$) that depends on a proof (z), a feature alien to ordinary predicate logic, where the expression-language is built up independently of the proofs.

3.8. Inductive Types

A basic notion in logic and set theory is induction: when a set is defined inductively, we understand it as being 'built up from the bottom' by a set of basic constructors. Elements of such a set can be decomposed in 'smaller elements' in a well-founded manner. This gives us the principles of 'proof by induction' and 'function definition by recursion'.

If we want to add inductive types to our type theory, we have to add a definition mechanism that allows us to introduce a new inductive type, by giving the name and the constructors of the inductive type. The theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion. It turns out that this can be done very generally in type theory, including very many instances of induction. Here we shall use a variant of the inductive types that are present in the system COQ [1999] and that were first defined in Coquand and Paulin-Mohring [1990]. Another approach to inductive types is to encode them as 'well-ordering types', also called W-types. The W-type can be used to encode arbitrary inductive types, but only if we are in extensional type theory. As we are in an intensional framework, we do not pursue that thread; see e.g. [Goguen and Luo 1993] for details.

We illustrate the rules for inductive types in $\lambda\mathsf{HOL}$ by first treating the (very basic) example of natural numbers nat. We would like the user to be able to write something like

$$\begin{array}{ll} \mathsf{Inductive} & \mathsf{nat} : \mathsf{Type} := \\ & 0 : \mathsf{nat} \\ & \mid S : \mathsf{nat} \to \mathsf{nat}. \end{array}$$

to obtain the following rules.

$$(\text{elim}_1)\quad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : \mathsf{nat}{\rightarrow}A{\rightarrow}A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{nat}} f_1 f_2 : \mathsf{nat}{\rightarrow}A}$$

$$(\text{elim}_2)\quad \frac{\Gamma \vdash P : \mathsf{nat}{\rightarrow}\mathsf{Prop} \quad \Gamma \vdash f_1 : P0 \quad \Gamma \vdash f_2 : \Pi x{:}\mathsf{nat}.Px{\rightarrow}P(Sx)}{\Gamma \vdash \mathsf{Rec}_{\mathsf{nat}} f_1 f_2 : \Pi x{:}\mathsf{nat}.Px}$$

The rule (elim$_1$) allows the definition of functions by primitive recursion. The rule (elim$_2$) allows proofs by induction. To make sure that the functions defined by (elim$_1$) compute $\mathsf{Rec}_{\mathsf{nat}}$ has the following reduction rule.

$$\begin{array}{rcl} \mathsf{Rec}_{\mathsf{nat}} f_1 f_2 0 & \rightarrow_\iota & f_1 \\ \mathsf{Rec}_{\mathsf{nat}} f_1 f_2 (St) & \rightarrow_\iota & f_2 t(\mathsf{Rec}_{\mathsf{nat}} f_1 f_2 t) \end{array}$$

It is understood that the additional ι-reduction is also included in the *conversion*-rule (conv), where we now have '$A =_{\beta\iota} B$' as a side-condition. The subscript in $\mathsf{Rec}_{\mathsf{nat}}$ will be omitted, when clear from the context.

An example of the use of (elim$_1$) is in the definition of the 'double' function d, which is defined by

$$d := \mathsf{Rec}_{\mathsf{nat}} 0(\lambda x{:}\mathsf{nat}.\lambda y{:}\mathsf{nat}.S(S(y))).$$

Now, $d0 \twoheadrightarrow_{\beta\iota} 0$ and $d(Sx) \twoheadrightarrow_{\beta\iota} S(S(dx))$. The predicate of 'being even', $\mathsf{even}(-)$, can also be defined by using (elim$_1$):

$$\mathsf{even}(-) := \mathsf{Rec}_{\mathsf{nat}}(\top)(\lambda x{:}\mathsf{nat}.\lambda\alpha{:}\mathsf{Prop}.\neg\alpha).$$

We obtain indeed that

$$\begin{array}{rcl} \mathsf{even}(0) & \twoheadrightarrow_{\beta\iota} & \top, \\ \mathsf{even}(Sx) & \twoheadrightarrow_{\beta\iota} & \neg\mathsf{even}(x) \end{array}$$

An example of the use of (elim$_2$) is the proof of $\Pi x{:}\mathsf{nat}.\mathsf{even}(dx)$. Say that true is some canonical inhabitant of type $\top$. Using $\mathsf{even}(d(Sx)) =_{\beta\iota} \neg\neg\mathsf{even}(dx)$ we find that $\lambda x{:}\mathsf{nat}.\lambda h{:}\mathsf{even}(dx).\lambda z{:}\neg\mathsf{even}(dx).zh$ is of type $\Pi x{:}\mathsf{nat}.\mathsf{even}(dx){\rightarrow}\mathsf{even}(d(Sx))$. So we conclude that

$$\vdash \mathsf{Rec}_{\mathsf{nat}}\mathsf{true}(\lambda x{:}\mathsf{nat}.\lambda h{:}\mathsf{even}(dx).\lambda z{:}\neg\mathsf{even}(dx).zh) : \Pi x{:}\mathsf{nat}.\mathsf{even}(dx).$$

Another well-known example is the type of lists over a domain D. It is defined as follows.

$$\begin{array}{ll} \mathsf{Inductive} & \mathsf{List} : \mathsf{Type} := \\ & \mathsf{Nil} : \mathsf{List} \\ & \mid \mathsf{Cons} : D{\rightarrow}\mathsf{List}{\rightarrow}\mathsf{List} \end{array}$$

with the following rules.

$$(\text{elim}_1)\quad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : D \to \mathsf{List} \to A \to A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : \mathsf{List} \to A}$$

$$(\text{elim}_2)\quad \frac{\Gamma \vdash P : \mathsf{List} \to \mathsf{Prop} \;\; \Gamma \vdash f_1 : P\mathsf{Nil} \;\; \Gamma \vdash f_2 : \Pi d{:}D.\Pi x{:}\mathsf{List}.Px \to P(\mathsf{Cons}\; dx)}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : \Pi x{:}\mathsf{List}.Px}$$

The rule (elim_1) allows the definition of functions by primitive recursion, while the rule (elim_2) allows proofs by induction. To make sure that the functions compute in the correct way, $\mathsf{Rec}_{\mathsf{List}}$ has the following reduction rule.

$$\begin{array}{rcl} \mathsf{Rec}_{\mathsf{List}} f_1 f_2 \mathsf{Nil} & \to_\iota & f_1 \\ \mathsf{Rec}_{\mathsf{List}} f_1 f_2 (\mathsf{Cons}\; dt) & \to_\iota & f_2 dt (\mathsf{Rec}_{\mathsf{List}} f_1 f_2 t) \end{array}$$

An example of the use of $\mathsf{Rec}_{\mathsf{List}}$ is in the definition of the 'map' function that takes a function $f : D \to D$ and returns the function (of type $\mathsf{List} \to \mathsf{List}$) that applies f to all elements of the list. Define

$$\begin{array}{rcl} \mathsf{map} & := & \lambda f{:}D \to D.\lambda l{:}\mathsf{List}.\mathsf{Rec}_{\mathsf{List}} \mathsf{Nil}(\lambda d{:}D.\lambda k{:}\mathsf{List}.\lambda h{:}\mathsf{List}.\mathsf{Cons}\;(fd)h) \\ & : & (D \to D) \to \mathsf{List} \to \mathsf{List}. \end{array}$$

Then $\mathsf{map}\; f(\mathsf{Cons}\; dt) =_{\beta\iota} \mathsf{Cons}\;(fd)\mathsf{map}\; ft$.

Of course, there is a more general pattern behind these two examples. The extension of λHOL with inductive types is defined by adding the following scheme.

$$\begin{array}{ll} \mathsf{Inductive} & \mu : \mathsf{Type} := \\ & \mathsf{constr}_1 : \sigma_1^1(\mu) \to \cdots \sigma_{m_1}^1(\mu) \to \mu \\ & \vdots \\ & \mid \mathsf{constr}_n : \sigma_1^n(\mu) \to \cdots \sigma_{m_n}^n(\mu) \to \mu \end{array}$$

where the $\sigma_j^i(\mu)$ are all 'type schemes with a strictly positive occurrence of μ', i.e. each $\sigma_j^i(\mu)$ is of the form $A_1 \to \cdots A_n \to X$ with no occurrence of μ in the A_k and either $X \equiv \mu$ or μ not in X. This declaration of μ introduces μ as a defined type and it generates the constructors $\mathsf{constr}_1, \ldots, \mathsf{constr}_n$ plus the associated elimination rules and the reduction rules. For a general picture on inductive types we refer to [Paulin-Mohring 1994].

To illustrate the generality of inductive types, we give an example of an inductive type that is more complicated than nat and List. We want to define the type Tree of countably branching trees with labels in D. (So a term of type Tree represents a tree where the nodes and leaves are labelled with a term of type D and where at every node there are countably many subtrees.) The definition of Tree is as follows.

$$\begin{array}{ll} \mathsf{Inductive} & \mathsf{Tree} : \mathsf{Type} := \\ & \mathsf{Leaf} : D \to \mathsf{Tree} \\ & \mid \mathsf{Join} : D \to (\mathsf{nat} \to \mathsf{Tree}) \to \mathsf{Tree} \end{array}$$

Here, Leaf creates a tree consisting of just a leaf, labelled by a term of type D. The constructor Join takes a label (of type D) and an infinite (countable) list of trees to create a new tree. The (elim_1) rule is as follows.

$$(\mathrm{elim}_1) \quad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash f_1 : D{\to}A \quad \Gamma \vdash f_2 : D{\to}(\mathsf{nat}{\to}\mathsf{Tree}){\to}(\mathsf{nat}{\to}A){\to}A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{Tree}} f_1 f_2 : \mathsf{Tree}{\to}A}$$

$\mathsf{Rec}_{\mathsf{Tree}}$ has the following reduction rule.

$$\begin{array}{rcl} \mathsf{Rec}_{\mathsf{Tree}} f_1 f_2(\mathsf{Leaf}\, d) & \to_\iota & f_1 d \\ \mathsf{Rec}_{\mathsf{Tree}} f_1 f_2(\mathsf{Join}\, d\, t) & \to_\iota & f_2 dt(\lambda x{:}\mathsf{nat}.\mathsf{Rec}_{\mathsf{Tree}} f_1 f_2(tx)) \end{array}$$

It is an interesting exercise to define all kinds of standard functions on Tree, like the function that takes the nth subtree (if it exists and take $\mathsf{Leaf}a$ otherwise) or the function that decides whether a tree is infinite (or just a single leaf).

For Tree, we have the following (elim_2) rule.

$$(\mathrm{elim}_2) \quad \frac{\begin{array}{ll} \Gamma \vdash P : \mathsf{Tree}{\to}\mathsf{Prop} & \Gamma \vdash f_1 : \Pi d{:}D.P(\mathsf{Leaf}\, d) \\ \multicolumn{2}{l}{\Gamma \vdash f_2 : \Pi d{:}D.\Pi t{:}\mathsf{nat}{\to}\mathsf{Tree}.(\Pi n{:}\mathsf{nat}.P(tn)){\to}P(\mathsf{Join}\, d\, t)} \end{array}}{\Gamma \vdash \mathsf{Rec}_{\mathsf{Tree}} f_1 f_2 : \Pi x{:}\mathsf{Tree}.Px}$$

Another interesting example of inductive types are inductively defined *propositions*. An example is the conjunction, which has one constructor (the pairing). Given φ and ψ of type Prop, it can be defined as follows.

$$\begin{array}{ll} \mathsf{Inductive} & \varphi \wedge \psi : \mathsf{Prop} := \\ & \mathsf{Pair} : \varphi{\to}\psi{\to}(\varphi \wedge \psi) \end{array}$$

As we do not have the $(\mathsf{Prop}, \mathsf{Type})$ rule in $\lambda\mathsf{HOL}$, we can only consider the second elimination rule, which will only appear in the case where P is a constant of type Prop. (So $P : \mathsf{Prop}$ instead of $P : \varphi \wedge \psi{\to}\mathsf{Prop}$.) The elimination rule (elim_2) rule is then as follows.

$$(\mathrm{elim}_2) \quad \frac{\Gamma \vdash P : \mathsf{Prop} \quad \Gamma \vdash f_1 : \varphi{\to}\psi{\to}P}{\Gamma \vdash \mathsf{Rec}_{\wedge} f_1 : (\varphi \wedge \psi){\to}P}$$

By taking φ (respectively ψ) for P and $\lambda x{:}\varphi.\lambda y{:}\psi.x$ (respectively $\lambda x{:}\varphi.\lambda y{:}\psi.y$) for f_1, one easily recovers the well-known projection from $\varphi \wedge \psi$ to φ (respectively ψ). The logical operators $\vee$ and $\exists$ can similarly be defined inductively.

More general inductive definitions

Above we have restricted ourselves to a specific class of inductive types. This class is very general, covering all the so called 'algebraic types', but it still can be extended. There are three main extensions that we discuss briefly by some motivating examples. They are

1. Parametric Inductive Types

2. Inductive Types with Dependent Constructors
3. Inductive Predicates

Many of these extensions occur together in more interesting examples.

Probably the most well-known situation of a ‘parametric type’ is the type of ‘lists over a type D’. Here the type D is just a parameter: primitive recursive operations on lists do not depend on the specific choice for D. A possible way for defining the type of *parametric lists* would be the following.

$$\begin{array}{ll} \mathsf{Inductive} & \mathsf{List} : \mathsf{Type} {\rightarrow} \mathsf{Type} := \\ & \mathsf{Nil} : \Pi D{:}\mathsf{Type}.(\mathsf{List} D) \\ & \mid \mathsf{Cons} : \Pi D{:}\mathsf{Type}.D {\rightarrow} (\mathsf{List} D) {\rightarrow} (\mathsf{List} D). \end{array}$$

Which would generate the following elimination rules and reduction rule.

$$(\mathrm{elim}_1)\quad \frac{\Gamma \vdash D : \mathsf{Type} \quad \Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : D {\rightarrow} (\mathsf{List} D) {\rightarrow} A {\rightarrow} A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : (\mathsf{List} D) {\rightarrow} A}$$

$$(\mathrm{elim}_2)\quad \frac{\begin{array}{ll} \Gamma \vdash D : \mathsf{Type} & \Gamma \vdash f_1 : P(\mathsf{Nil} D) \\ \Gamma \vdash P : (\mathsf{List} D) {\rightarrow} \mathsf{Prop} & \Gamma \vdash f_2 : \Pi d{:}D.\Pi x{:}(\mathsf{List} D).Px {\rightarrow} P(\mathsf{Cons}\ Ddx) \end{array}}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : \Pi x{:}(\mathsf{List} D).Px}$$

$$\begin{array}{rcl} \mathsf{Rec}_{\mathsf{List}} f_1 f_2 (\mathsf{Nil} D) & \rightarrow_\iota & f_1 \\ \mathsf{Rec}_{\mathsf{List}} f_1 f_2 (\mathsf{Cons}\ Ddt) & \rightarrow_\iota & f_2 dt (\mathsf{Rec}_{\mathsf{List}} f_1 f_2 t) \end{array}$$

To be able to write down the type of the constructors Nil and Cons, we need the rule $(\mathsf{Type}', \mathsf{Type})$ in $\lambda\mathsf{HOL}$, which makes the system inconsistent. Therefore, this extension works much better in a system like $\lambda\mathrm{PRED}\omega$, where we can consistently allow quantification over Set. We will not be concerned with these precise details here however.

In the example of parametric lists we have already seen constructors that have a dependent type. It turns out that this situation occurs more often. With respect to the general scheme, the extension to include dependent typed constructors is a straightforward one: all definitions carry through immediately. We treat an interesting example of an inductive type (the Σ-type), which is defined using a constructor that has a dependent type. Let B : Type and $Q : A {\rightarrow} \mathsf{Prop}$ and suppose we have added the rule $(\mathsf{Prop}, \mathsf{Type})$ to our system.

$$\begin{array}{ll} \mathsf{Inductive} & \mu : \mathsf{Type} := \\ & \mathsf{In} : \Pi z{:}B.(Qz) {\rightarrow} \mu. \end{array}$$

$$(\mathrm{elim}_1)\quad \frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash f_1 : \Pi z{:}B.(Qz)\to A}{\Gamma \vdash \mathsf{Rec}_{\mu} f_1 : \mu \to A}$$

$$(\mathrm{elim}_2)\quad \frac{\Gamma \vdash P : \mu \to \mathsf{Prop} \quad \Gamma \vdash f_1 : \Pi z{:}B.\Pi y{:}(Qz).P(\mathsf{In}zy)}{\Gamma \vdash \mathsf{Rec}_{\mu} f_1 : \Pi x{:}\mu.(Px)}$$

The ι-reduction rule is

$$\mathsf{Rec}_{\mu} f_1(\mathsf{In}bq) \quad \to_\iota \quad f_1 bq$$

Now, taking in (elim_1) B for A and $\lambda z{:}B.\lambda y{:}(Qz).z$ for f_1, we find that

$$\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).z)(\mathsf{In}bq) \twoheadrightarrow b.$$

Hence, we define $\pi_1 := \mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).z)$. Now, taking $\lambda x{:}\mu.Q(\pi_1 x)$ for P in (elim_2) and $\lambda z{:}B.\lambda y{:}(Qz).y$ for f_1, we find that $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y) : \Pi z{:}\mu.Q(\pi_1 z)$. Furthermore, $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y)(\mathsf{In}bq) \twoheadrightarrow q$. Hence, we define $\pi_2 := \mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y)$ and we remark that μ together with In (as pairing constructor) and π_1 and π_2 (as projections) represents the Σ-type.

An example of an inductively defined predicate is the equality, which can be defined as follows.

$$\begin{array}{ll}\mathsf{Inductive} & \mathsf{Eq} : D\to D\to \mathsf{Prop} := \\ & \mathsf{Refl} : \Pi x{:}D.(\mathsf{Eq}xx).\end{array}$$

Just like in the example for the conjunction, we only have the second elimination rule for the non-dependent case (i.e. P only depends on $x, y{:}D$ but not on a proof of $\mathsf{Eq}xy$). So we have

$$(\mathrm{elim}_2)\quad \frac{\Gamma \vdash P : D\to D\to \mathsf{Prop} \quad \Gamma \vdash f_1 : \Pi x{:}D.(Pxx)}{\Gamma \vdash \mathsf{Rec}_{\mathsf{Eq}} f_1 : \Pi x, y{:}D.(\mathsf{Eq}xy)\to(Pxy)}$$

The ι-reduction rule is

$$\mathsf{Rec}_{\mathsf{Eq}} x x f_1(\mathsf{Refl}x) \quad \to_\iota \quad f_1 x$$

4. Proof-development in type systems

In this section we will show how a concrete proof-assistant works. First we show in what way the human has to interact with the system. Then a small proof-development is partially shown (most proof-objects are omitted). Finally it is shown how computations can be captured in formalized theories.

4.1. Tactics

In Section 2.1 and Section 4.3 examples will be given of an easy, and a more involved theorem with full proofs. Even before these examples are given, the reader will probably realize that constructing fully formalized proofs (the proof-objects) is relatively involved. Therefore tools have been developed—so-called *proof-assistants*—that make this task more easy. A proof assistant consists of a proof checker and an interactive proof-development system. We have depicted the situation graphically in Figure 7. In the proof-development system one chooses a context and formu-

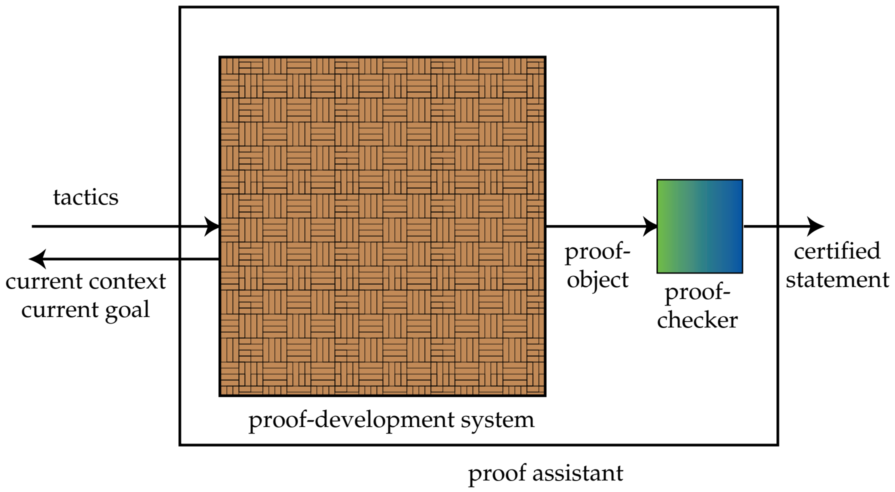

Figure 7: A proof-assistant and its components

lates a statement to be proved relative to that context. This statement is called the *goal*. Rather than constructing the required proof-object directly, one uses so-called *tactics* that give a hint to the machine as to what the proof-object looks like. For example, if one wants to prove

$$\forall x{:}A.(Px \;\Rightarrow\; Qx)$$

in context $A : \mathsf{Set}, P, Q : A \rightarrow \mathsf{Prop}$, then there is a tactic ('Intros') that changes the context by picking a fresh ('arbitrary') $x{:}A$ and assumes Px, the goal now becoming Qx. To be more precise, we give some extracts of Coq sessions. In Coq, the Π-abstraction and the λ-abstraction are represented by brackets: `(x:A)B` denotes $\Pi x{:}A.B$ and `[x:A]M` denotes $\lambda x{:}A.M$. Furthermore, `->` and abstraction bind stronger than application, so we have to put brackets around applications, writing `(x:A)(P x)->(Q x)` for $\Pi x{:}A.Px \rightarrow Qx$. In the following, `Unnamed_thm <` and `Coq <` are the Coq prompts at which the user is expected to type some command: at `Coq <`, the system is in 'declaration mode', where the user can extend the context with new declarations or definitions; at `Unnamed_thm <`, the system is in 'proof mode', where the user can type in tactics to solve the goal(s).

```
Coq < Variable A:Set; Variable P,Q:A->Prop.
A is assumed
P is assumed
Q is assumed

Coq <  Goal (x:A)(P x) -> (Q x).
1 subgoal

  ============================
   (x:A)(P x)->(Q x)

Unnamed_thm < Intros.
1 subgoal

  x : A
  H : (P x)
  ============================
   (Q x)
```

The `H: (P x)` means that we assume that `H` is a proof of `(P x)` (in order to construct a proof `q` of `(Q x)`, thereby providing a proof of `(P x) -> (Q x)`, namely `[H:(P x)]q`, and hence of `(x:A)(P x) -> (Q x)`, namely `[x:A][H:(P x)]q`.

Another tactic is 'Apply'. If the current context contains `a:A` and `p: (x:A)(P x) -> (Q x)` and the current goal is `(Q a)`, then the command `Apply p` will change the current goal into `(P a)`. This is done by *matching* the type of `p` with the current goal where the universal variables (here just `x`) are the ones to be instantiated. So, the system matches `(Q x)` with `(Q a)`, finding the instantiation of `a` for `x`. The proof-term that the system constructs is in this case `p a ?`, with `?` the yet to be constructed proof of `(P a)`.

```
Coq < Variable a:A; Variable p : (x:A) (P x) -> (Q x).
a is assumed
p is assumed

Coq < Goal (Q a).
1 subgoal

  ============================
   (Q a)

Unnamed_thm < Apply p.
1 subgoal

  ============================
   (P a)
```

Another essential tactic is concerned with inductive types. For example the type of natural numbers is defined by

```
Inductive nat := O :nat | S: nat -> nat.
```

This type comes together with an induction principle

```
nat_ind
    : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

The way this can be used is as follows. If the (current) goal is `(Q n)` in context containing `n : nat`, then the tactic `Elim n` will produce the new goals `(Q O)` and `(n : nat)(Q n)-> (Q (n+1))`. Indeed, if `p` is a proof of `(Q O)` and `q` of `(n:nat)(Q n)->(Q(n+1))`, then `(nat_ind Q p q n)` will be a proof of `(Q n)`.

Also this type `nat` comes with a recursor `nat_rec` satisfying

```
    (nat_rec a b O)  =  a;
(nat_rec a b (S n))  =  (b n (nat_rec a b n)).
```

Indeed, going from left to right, these are ι-reductions that fall under the Poincaré principle.

As logical operators are defined inductively, we basically have all tools to develop mathematical proofs. The interactive session continues until all goals are solved. Then the system is satisfied and the proved result can be stored under a name that is chosen by the user.

```
Subtree proved!

Unnamed_thm < Save fst_lemma.
<tactics>

fst_lemma is defined
```

In the place of `<tactics>`, the system repeats the series of tactics that was typed in by the user to solve the goal. The system adds a definition `fst_lemma := ...` to the context, where `...` is the proof term (a typed λ-term) that was interactively constructed. Then later the user can use the lemma by referring to `fst_lemma`, for example in the Apply tactic: `Apply fst_lemma`.

The set of tactics and its implementation together with the user interface will yield a large proof-development system. For example, several techniques of automated deduction may be incorporated as tactics. But even if the resulting proof-development system as subunit in general will be large, the reliability of the proof-assistant as such is still high, provided that the proof checker is small, i.e. satisfies the de Bruijn criterion.

4.2. Examples of Proof Development

Given a mathematical statement within a certain context, a *proof development* consists of a formalization of the context Γ and statement A and a construction of a proof-object for it, i.e. a term p such that

$$\Gamma \vdash p : A.$$

A substantial part of a proof development consists of a *theory development*, a name coined by Peter Aczel. This consists of a list of primitive and defined notions and axioms and provable theorems culminating in the goal A to be proved. In this section we will present such a theory development in the system Coq for the statement that every natural number greater than one has a prime divisor.[4]

Two aspects of the development are of interest. Whereas the logical operators $\to$ and $\forall$ are primitive notions of type theory (when translated as Π), the operators conjunction $\wedge$, disjunction $\vee$, false `FF`, negation `~` and existence $\exists$ are also definable using inductive types, see [Martin-Löf 1984]. For example

```
Inductive or [A:Prop; B:Prop] : Prop :=
        or_introl : A->(or A B)
      | or_intror : B->(or A B)
```

Here, the abstraction `[A:Prop; B:Prop]` says that `A` and `B` are parameters of the definition. Some pretty printing, a syntactic definition can be added, allowing to write `A \/ B` for `(or A B)`. The inductive definition implies that `A \/ B` comes together with maps

```
or_introl  : (A,B:Prop)A->A\/B
or_intror  : (A,B:Prop)B->A\/B
```

We also need a map corresponding to the elimination principle for disjunction (for example to prove that `A\/B -> B\/A`):

```
or_ind     : (A,B,P:Prop)(A->P)->(B->P)->A\/B->P
```

It is also possible to define the operations $\wedge$, $\vee$, `FF`, `~` and $\exists$ without inductive types, using higher order quantification, as in [Russell 1903]. For example disjunction becomes

$$A \vee B \equiv \Pi C{:}\texttt{Prop}.(A{\to}C){\to}(B{\to}C){\to}A \vee B{\to}C.$$

In this way the elimination principle is the term

$$\lambda f{:}(A{\to}C)\lambda g{:}(B{\to}C)\lambda h{:}(A \vee B).hCfg.$$

The logical definitions defined this way turn out to be equivalent with the inductively defined ones. Following Martin-Löf we use the inductive definitions, because this way one can avoid impredicative notions like higher order quantification.

[4]From this statement Euclid's theorem that there are infinitely many primes is not far removed: consider a prime factor of $n! + 1$ and show that it is necessarily $> n$. Thus one obtains $\forall n \exists p > n.\texttt{prime}\ p$. A slightly different formalization is possible in type theory, where one can prove the statement $\forall n{:}\mathbb{N}\forall p_1, \ldots, p_n{:}\mathbb{N}[\texttt{prime}\ p_1 \wedge \ldots \wedge \texttt{prime}\ p_n \Rightarrow \exists x{:}\mathbb{N}[\texttt{prime}\ x \wedge x \neq p_1 \wedge \ldots \wedge x \neq p_n]]$. Note that it is impossible to even state this as a theorem in Peano Arithmetic, because of the use of n as a parameter denoting the length of the sequence $\vec{p}$ and the number of disjunctions $x \neq p_i$. In type theory it can be stated because of the rules for inductive types. In arithmetic one would have to go to second order logic to state (and prove) this theorem

Another point of interest is that inductive types are freely generated by their constructors. This has for example as consequence that for the type of natural numbers one can prove

```
  (n : nat) ~( (S n) = O )
(n,m : nat) (S n) = (S m) -> n = m
```

Thus we see that within type theory with inductive types, Heyting arithmetic can be formalized, without assuming additional axioms or rules. To quote Randy Pollack: "Type theory with inductive types is intuitionistic: mathematical principles are wired in."

Now we will present a theory development in Coq (version 6.3), for the statement that every natural number has a prime divisor. The mathematics behind this is very elementary. Logic is introduced.[5] After the introduction of the natural numbers, plus and times are defined recursively. Then division and primality are defined. In order to prove our result the usual ordering $<$ is defined (first $\leq$) and course of value induction[6] is used. Text written between `(* ... *)` serves as a comment. In the following, the proofs are omitted but the definitions are given explicitly.

```
(**************** A simple proof-development ****************)

(****   Propositional connectives defined inductively.   ****)

Inductive and [A:Prop; B:Prop] :  Prop
                               := conj : A->B->(and A B).

Inductive TT :  Prop
             := trivial : TT.

Inductive FF : Prop
             :=.

Definition not : Prop->Prop
               := [A:Prop]A->FF.

Definition iff := [A,B:Prop](and (A->B)(B->A)).

(* For pretty printing syntactic definitions (not shown) are
   introduced that allow to use the following notations

     ~A  for (not A)
   A/\B  for (and A B)
   A\/B  for (or A B)
   A<->B for (iff A B)                                     *)

(* Introduction and elimination rules. *)
```

[5] In fact classical logic. An intuitionistic proof is much better, as it provides an algorithm to find the prime divisor. But this requires more work.

[6] If for every $n \in \mathbb{N}$ one has $(\forall m < n. Pm) \rightarrow Pn$, then $\forall n \in \mathbb{N}. Pn$.

```
Lemma  and_in : (a,b:Prop)a->b->(and a b).
Lemma and_ell : (a,b:Prop)(and a b)->a.
Lemma and_elr : (a,b:Prop)(and a b)->b.

Lemma false_el : (a:Prop) FF->a.

Lemma or_inl : (a,b:Prop)a->(or a b).
Lemma or_inr : (a,b:Prop)b->(or a b).
Lemma  or_el : (a,b,c:Prop)(a->c)->(b->c)->(or a b)->c.

(* Lemmas combining connectives. *)

Lemma non_or : (a,b:Prop)~(or a b)->~a/\~b.
(* We show the proof-object (generated by the tactics):
   non_or =
   [a,b:Prop; p:(not (or a b))]
    (and_in (not a) (not b) [q:a](p (or_inl a b q))
      [q:b](p (or_inr a b q)))
        : (a,b:Prop)(not (or a b))->(and (not a) (not b))   *)

(* Some lemmas omitted *)

(*******************   Predicate logic.   *******************)

Inductive ex [A : Set; P : A->Prop] :  Prop
                                  := ex_intro : (x:A)(P x)->(ex A P).

(*  A syntactic definition (not shown) is given that allows one
to write the usual

(EX x:A|(P x)) for ex A [x:A](P x)                           *)

Section Pred.

Variables A : Set; P : A->Prop; Q : A ->Prop.

Lemma all_el : (x:A)((y:A)(P y))->(P x).
Lemma  ex_in : (x:A)(P x)->(EX y:A|(P y)).

Lemma  non_ex : (~(EX x:A|(P x)))->(x:A)~(P x).
Lemma all_not : ((x:A)~(P x))->~(EX x:A|(P x)).
Lemma all_and : ((x:A)(P x)/\(Q x))->((x:A)(P x))/\((x:A)(Q x)).
Lemma   ex_or : (EX x:A|(P x)\/(Q x))
->(EX x:A|(P x))\/(EX x:A|(Q x)).

End Pred.

(* Classical logic. *)
```

```
Axiom DN : (a:Prop)(~~a->a).

Lemma dn_c : (a:Prop)~~a<->a.

Lemma (* Excluded middle: tertium non datur. *)
      tnd : (a:Prop)(a\/~a).

(* Some lemmas omitted *)

Section Pred_clas.

Variable A:Set; P:A->Prop.

Lemma non_all : (~(x:A)(P x))->(EX x:A|~(P x)).
Lemma    ex_c : (EX x:A|(P x))<->~(x:A)~(P x).

(* This lemma has the following proof-object. [Note the presence of DN]
   ex_c =
   (conj (EX x:A | (P x))->~((x:A)~(P x))
         ~((x:A)~(P x))->(EX x:A | (P x))
     [H:(EX x:A | (P x)); H0:((x:A)(P x)->FF)]
      (ex_ind A [x:A](P x) FF [x:A; H1:(P x)](H0 x H1) H)
     [H:(~((x:A)~(P x)))]
      (DN (EX x:A | (P x))
        [H0:((EX x:A | (P x))->FF)]
         (H [x:A; H1:(P x)](H0 (ex_intro A [x0:A](P x0) x H1)))))
        : (EX x:A | (P x))<->~((x:A)~(P x))                          *)

End Pred_clas.

(*******************     Arithmetic   ************************)

Inductive eq [A:Set;x:A] :  A->Prop
                         := refl_equal : (eq A x x).

(* A syntactic definition (not shown) is introduced in order to use
   the abbreviation
                x = y  for  (eq A x y).
   In this syntactic definition, the type A can be used as an
   `implicit argument'. It is reconstructed by the type checking
   algorithm from the type of x                                      *)

Lemma   sym_eq : (A:Set)(x,y:A)(x = y)->(y = x).
Lemma     leib : (A:Set)(P:A->Prop)(x,y:A)(x = y)->(P x)->(P y).
Lemma eq_ind_r : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)(y=x)->(P y).
Lemma  f_equal : (A,B:Set; f:(A->B); x,y:A)(x=y) -> ((f x)=(f y)).

Inductive nat : Set := O : nat | S : nat->nat.
```

```
Definition   one : nat := (S O).
Definition   two : nat := (S one).

Definition Is_suc := [n:nat]
Cases n of
                     O    => FF
                  | (S p) => TT
               end.

Lemma no_conf : (n:nat)~(O= (S n)).

Inductive leseq [n:nat] :  nat->Prop :=
     leseq_n   : (leseq n n)
                  |  leseq_suc : (m:nat)(leseq n m)->(leseq n (S m)).

Definition lthan := [n,m:nat](leseq (S n) m).

Lemma leseq_trans : (x,y,z:nat)(leseq x y)->(leseq y z)->(leseq x z).
Lemma lthan_leseq : (n,m:nat)((lthan n m)->(leseq n m)).
Lemma     non_lt0 : (n:nat)~(lthan n O).
Lemma   suc_leseq : (n,m:nat)(leseq (S n)(S m))->(leseq n m).
Lemma        lt01 : (x:nat)(x=O\/x=one\/(lthan one x)).
Lemma      nOn1lt : (n:nat)(~(n=O)->~(n=one)->(lthan one n)).

Definition before [n:nat; P:nat->Prop] := ((k:nat)(lthan k n)->(P k)).

Lemma (* Course of value induction *)
cv_ind : (P:nat->Prop)((n:nat)((before n P) -> (P n))-> (n:nat)(P n)).

Fixpoint plus [n:nat] :  nat -> nat := [m:nat]
Cases n of
                      O    => m
                   | (S p) => (S (plus p m))
                end.

Lemma  plus_altsuc : (n,m:nat)(plus n (S m))=(S(plus n m)).
Lemma plus_altzero : (n:nat) (plus n O)=n.
Lemma     plus_ass : (n,m,k: nat)(n,m,k: nat)
                     (plus n (plus m k))=(plus(plus n m)k).
Lemma     plus_com : (n,m:nat)(plus n m)=(plus m n).

Fixpoint times [n:nat] :  nat -> nat := [m:nat]
Cases n of
                   O    => O
               | (S p) => (plus (times p m) m)
                end.
```

```
Lemma        distr : (n,m,k: nat)(n,m,k: nat)
                     (times (plus n m) k)=(plus(times n k)(times m k)).
Lemma timesaltzero : (n:nat)(times n O)=O.
Lemma  timesaltsuc : (n,m:nat)(times n (S m))=(plus(times n m) n).
Lemma    times_ass : (n,m,k:nat)(times n(times m k))=(times(times n m)k).

Definition      div :  (nat->nat->Prop)
                    := [d,n:nat](EX x:nat|(times x d)=n).
Definition  propdiv :  (nat->nat->Prop)
                    := [d,n:nat]((lthan one d)/\(lthan d n)/\(div d n)).
Definition    prime :  nat -> Prop
                    := [n:nat]((lthan one n)/\~(EX d:nat|(propdiv d n))).
Definition primediv :  nat->nat->Prop
                    := [p,n:nat](prime p)/\(div p n).

(* Some lemmas omitted *)

(* has prime divisor *)
Definition HPD  :  nat->Prop := [n:nat](EX p:nat|(primediv p n)).

Theorem numbers_gt1_have_primediv : (n:nat)(lthan one n)->(HPD n).

(*****************************************************************)
```

As stated before, from here one can prove Euclid's theorem that there are infinitely many primes. In order to do this one needs to know that if d divides both a and $a+b$, then it divides b (introduce cut-off subtraction for this and prove some lemmas about it).

4.3. Autarkic Computations

We have so far described how to formalize definitions, statements and proofs. Another important aspect of mathematics is *computing*. (In order to decide whether statements are true or simply because a numerical value is of interest). The following examples are taken from [Barendregt 1997]. These are examples of statements for which computations are needed.

(1) $[\sqrt{45}] = 6,$ where $[r]$ is the integer part of a real

(2) $\mathtt{Prime}(61)$

(3) $(x+1)(x+1) = x^2+2x+1$

In principle computations can be done within an axiomatic framework, in particular within predicate logic with equality. But then proofs of these statements become rather long. E.g.

$$(x+1)^2 = (x+1)\cdot(x+1)$$

$$\begin{aligned} &= (x+1)\cdot x + (x+1)\cdot 1 \\ &= x\cdot x + 1\cdot x + x\cdot 1 + 1\cdot 1 \\ &= x^2 + x + x + 1 \\ &= x^2 + 2\cdot x + 1. \end{aligned}$$

This is not even the whole story. Each use of '=' has to be justified by applying an axiom, substitutions and the fact that + preserves equality[7].

A way to handle (1) is to use the Poincaré principle extended with the reduction relation $\twoheadrightarrow_\iota$ for primitive recursion on the natural numbers. Operations like $f(n) = [\sqrt{n}\,]$ are primitive recursive and hence are λ-definable (using $\twoheadrightarrow_{\beta\iota}$) by $\mathsf{Rec}_{\mathtt{nat}}$ introduced in Section 3.8. Then, writing $\ulcorner 0 \urcorner = \mathtt{0}, \ulcorner 1 \urcorner = \mathtt{S\ 0}, \ldots$, it follows from the Poincaré principle that the same is true for

$$F\ulcorner 45 \urcorner = \ulcorner 6 \urcorner,$$

since $\ulcorner 6 \urcorner = \ulcorner 6 \urcorner$ is formally derivable and we have $F\ulcorner 45 \urcorner \twoheadrightarrow_{\beta\iota} \ulcorner 6 \urcorner$. Usually, a proof obligation arises that F is adequately constructed. For example, in this case it could be

$$\forall n\ (F\,n)^2 \ \leq\ n\ < ((F\,n)+1)^2.$$

Such a proof obligation needs to be formally proved, but only once; after that reductions like

$$F\ulcorner n \urcorner \twoheadrightarrow_{\beta\iota} \ulcorner f(n) \urcorner$$

can be used freely many times.

In a similar way, a statement like (2) can be formulated and proved by constructing a λ-defining term $K_{\mathtt{Prime}}$ for the characteristic function of the predicate $\mathtt{Prime}$. This term should satisfy the following statement

$$\begin{aligned} \forall n \quad &[(\mathtt{Prime}\,n \ \leftrightarrow\ K_{\mathtt{Prime}}\,n = \ulcorner 1 \urcorner)\ \& \\ &(K_{\mathtt{Prime}}\,n = \ulcorner 0 \urcorner\ \vee\ K_{\mathtt{Prime}}\,n = \ulcorner 1 \urcorner)]. \end{aligned}$$

which is the proof obligation.

Statement (3) corresponds to a symbolic computation. This computation takes place on the syntactic level of formal terms. There is a function g acting on syntactic expressions satisfying

$$g((x+1)(x+1)\,) = x^2 + 2x + 1,$$

that we want to λ-define. While $x+1 : \mathsf{Nat}$ (in context $x{:}\mathsf{Nat}$), one has '$x + 1$' : $\mathtt{term}(\mathsf{Nat})$. Here $\mathtt{term}(\mathsf{Nat})$ is an inductively defined type consisting of the terms over the structure $\langle \mathsf{Nat}, +, \times, 0, 1\rangle$. Using a reduction relation for primitive recursion over this data type, one can represent g, say by G, so that

$$G\,`(x+1)(x+1)\,\text{'} \twoheadrightarrow_{\beta\iota} `x^2 + 2x + 1\text{'}.$$

[7]This is why some mathematicians may be turned off by logic. But these steps have to be done. Usually they are done within a fraction of a second and unconsciously by a mathematician.

Now in order to finish the proof of (3), one needs to construct a self-interpreter E, such that for all expressions $p : \mathsf{Nat}$ one has

$$\mathsf{E}\,`p' \twoheadrightarrow_{\beta\iota} p$$

and prove the proof obligation for G which is

$$\forall t{:}\mathtt{term}(\mathsf{Nat})\ \mathsf{E}(G\,t) \ = \ \mathsf{E}\,t.$$

It follows that

$$\mathsf{E}(G\,`(x+1)(x+1)\,') \ = \ \mathsf{E}\,`(x+1)(x+1)\,'.$$

Now, since

$$\begin{array}{rcl}\mathsf{E}(G\,`(x+1)(x+1)\,') & \twoheadrightarrow_{\beta\iota} & \mathsf{E}\,`x^2+2x+1' \\ & \twoheadrightarrow_{\beta\iota} & x^2+2x+1 \\ \mathsf{E}\,`(x+1)(x+1)\,' & \twoheadrightarrow_{\beta\iota} & (x+1)(x+1),\end{array}$$

we have by the Poincaré principle

$$(x+1)(x+1) \ = \ x^2+2x+1.$$

Bureaucratic details how to treat free variables under E are omitted.

The use of inductive types like Nat and $\mathtt{term}(\mathsf{Nat})$ and the corresponding reduction relations for primitive reduction was suggested by Scott [1970] and the extension of the Poincaré principle for the corresponding reduction relations of primitive recursion by Martin-Löf [1984]. Since such reductions are not too hard to program, the resulting proof checking still satisfies the de Bruijn criterion.

The general approach is as follows. In computer algebra systems algorithms are implemented by special purpose term rewriting. For example for polynomial expressions p one has for (formal) differentiation and simplification the following.

$$\boxed{\begin{array}{llllll} p & \rightarrow_{\mathtt{diff}} & \cdots & \rightarrow_{\mathtt{diff}} & p^{\mathtt{diff\text{-}nf}} & = p_1; \\ p & \rightarrow_{\mathtt{simpl}} & \cdots & \rightarrow_{\mathtt{simpl}} & p^{\mathtt{simpl\text{-}nf}} & = p_2. \end{array}}$$

In this way the functions $f_{\mathtt{diff}}(p) = p_1$ and $f_{\mathtt{simpl}}(p) = p_2$ are computed. In type theory with inductive types and ι-reduction these computations can be captured as follows.

$$\boxed{\begin{array}{rcl} F_{\mathtt{diff}}\,p & \twoheadrightarrow_{\beta\delta\iota} & p_1; \\ F_{\mathtt{simpl}}\,p & \twoheadrightarrow_{\beta\delta\iota} & p_2. \end{array}}$$

This is like replacing special purpose computers by the universal Turing-von Neumann computer with software.

In [Oostdijk and Geuvers 2001] a program is presented that, for every primitive recursive predicate P, constructs the lambda term K_P defining its characteristic function and the proof of the adequacy of K_P. That is, one proves $\forall n{:}\mathsf{Nat}.P(n) \leftrightarrow K_P(n) = 1$ (generically for all primitive recursive predicates P). In this way, proving $P(n)$ can be replaced by computing $K_P(n)$. The resulting computations for $P =$ `Prime` are not efficient, because a straightforward (non-optimized) translation of primitive recursion is given and the numerals (represented numbers) used are in a unary (rather than n-ary) representation; but the method is promising. In [Caprotti and Oostdijk 2001], a more efficient ad hoc definition of the characteristic function of `Prime` is given, using Pocklington's criterion, based on Fermat's small theorem about primality. Also the required proof obligation is given. In this way it can be proved, formally in Coq, that a number like 1223334444555554444333221 is prime (but also bigger numbers, some of 44 digits!) So the statements in the beginning of this subsection can be obtained by computations.

Another use of reflection is to show that a function like

$$f(x) = e^{3x^2} + \sqrt{1 + \mathtt{sin}^2 x} + \cdots$$

is continuous. Rather than proving this by hand one can introduce a formal language L, such that a description of f is among them, and show that every expression $e : L$ denotes a continuous function.

5. Proof assistants

Proof assistants are interactive programs running on a computer that help the user to obtain verified statements (within a given mathematical context). This verification can be generated in two ways: automatically by a theorem prover, or provided by the user with a proof that is checked by the machine.

It is clear that proof checking is not automated deduction. The problem of deciding whether a putative proof is indeed a proof is decidable; on the other hand the problem whether a putative theorem is indeed a theorem is undecidable. Having said this, it is nevertheless good to remark that there is a spectrum ranging from on the one hand pure proof-checkers to on the other hand pure automated theorem provers. A pure proof-checker, to which one has to present an entire fully formalized proof, is impractical, because it is difficult to provide these proof-objects. On the other hand a pure automated theorem prover (that finds a proof if a statement A is provable and tells us that there is none otherwise) is impossible for theorems in theories as simple as predicate logic. Automated deduction is in general only possible as a partial algorithm (providing a proof if there is one, running forever otherwise).

For some special theories, like elementary geometry (which is decidable), a total algorithm may be possible (in the case of geometry there is the excellent theorem prover of Wu [1994]). In most cases an automated theorem prover requires that the user gives hints. Although this chapter is not about automated theorem provers, we would like to mention Otter [1998] for classical predicate logic, the system of Bibel and Schmitt [1998] for classical predicate logic with equality, Boyer and Moore's

[1997] theorem prover Nqthm, based upon primitive recursive arithmetic, and Wu's [1994] geometry theorem prover that was already mentioned.

At the other end of the spectrum a user-friendly proof-checker usually has some form of automated deduction in order to make it more easy for the user to provide proof-objects. Proof-assistants consists of a proof-development system together with a proof-checker.

5.1. Comparing proof-assistants

We will discuss several proof-assistants. All systems except Agda work with *proof scripts* that are a list of *tactics* needed to make the proof-assistant to verify the validity of the statement. The proof-assistants fall into two classes: those with *proof-objects* and those without proof-objects.

In the case of a proof-assistant with proof-objects the script generates and stores a term that is (isomorphic to) a proof that can be checked by a simple proof checker. This makes these systems highly reliable. In principle someone, who is doubtful whether a certain statement is valid, can download a proof-object via the internet and locally verify it using his or her own trusted proof checker of relatively small size.

Proof-assistants that have no proof-objects come in two classes. The first one consists of systems that in principle can translate the proof-script into a proof-object that can be verified by a small checker. In this case the proof-script can be considered as a *non-standard proof-object*. In order to make this translation these systems just need some system specific preprocessor after which a trustworthy check can be performed. The second class consists of proof-assistants for which there is not (yet) a way to provide a proof-object with high reliability. So for the correctness of theorems accepted by assistants in this class one has to trust these systems. The advantage of these kind of systems usually is their larger automated deduction facilities and (therefore) their larger user-friendliness.
We will discuss the following proof-assistants.

system	proof-objects
Coq, Lego, Agda	yes
Nuprl, HOL, Isabelle	non-standard
Mizar, PVS, ACL2	no

Coq, Lego and Agda

Of these three systems Coq is the most developed one. The systems Coq and Lego are based on versions of the calculus of constructions extended with inductive types. For the logical power of this formal system, see [Aczel 1999] and the references contained therein. An important difference between the proof-assistants is in their computational power. Both systems admit the Poincaré principle for $\beta\delta\iota$-conversion. This means that there are deduction steps like the following ones.

$$\frac{\text{Reflexive}(R)}{\forall x.Rxx}\,\delta \qquad \frac{A(\mathsf{I}x)}{A(x)}\,\beta\delta \quad \text{and} \quad \frac{A(\texttt{fac}(4))}{A(24)}\,\iota\delta,$$

[Here one assumes to have defined Reflexive$(R) \equiv \forall x.Rxx$, $\mathsf{I} \equiv \lambda x.x$ and `fac` as the function representing the factorial.] One of the differences between Coq and Lego is that in Lego one can introduce other notions of reduction for which the Poincaré principle is assumed to hold (including non-terminating ones).

Both Coq and Lego create proof-objects from the proof-scripts and store them. These proof-objects are isomorphic to natural deduction proofs. The two systems allow impredicative arguments as used in actual mathematics, but argued to be potentially unreliable by Poincaré and Martin-Löf. The system Agda is similar to Coq and Lego, except that it is based on Martin-Löf type-theory in which impredicative quantifications are not allowed. The Poincaré principle can be assumed by the user for any notion of reduction that is proved to be strongly normalizing. Agda is not so much 'tactics based' as Coq and Lego. In Agda one edits a proof term by 'filling in the holes' in an open term. The system acts as a structure editor, providing support for term construction.

Nuprl, HOL and Isabelle

Constable et al.'s [1986] system Nuprl does have proof-objects, but a judgment

$$\vdash p : A,$$

indicating that p is a proof of A, is not decidable. The reason for this is that the Poincaré principle is assumed not only for $\beta\delta\iota$-conversion, (the intensional equality) but also for *extensional equality*. See Section 2.8. So there is a rule

$$\frac{p : A(t) \qquad q : (t = s)}{p : A(s)}$$

So, Nuprl is based on an extensional type system. This implies that type checking $p : A$? (TCP, see Section 2.1) is no longer decidable and therefore proofs cannot be checked. However, there are 'expanded' proof-objects d that can establish that $p : A$. In fact, the d takes into account the terms q for which $q : t = s$. So these d serve as the 'real' proof-objects.

The proof-assistant HOL [1998] is based on Church's [1940] simple type theory. This is a classical system of higher order logic. That HOL uses non-standard proof-objects has a different reason. HOL does not satisfy the Poincaré principle for any conversion relation. As a consequence computations involving recursion become quite lengthy when converted to a proof-object (for example establishing by a proof that $\vdash \texttt{fac}\ \mathbf{c}_n = \mathbf{c}_{n!}$). Therefore the design decision was made that proof-objects are not stored, only the proof-scripts. Even if a proof of $\texttt{fac}\, \mathbf{c}_n = \mathbf{c}_{n!}$ may be long, it is possible to give it a short description in the proof-script. Induction is done by defining an inductive predicate in a higher order way as the smallest set satisfying a closure property.

Also Isabelle is based on intuitionistic simple type theory. But this proof-assistant is fine-tuned towards using this logic as a meta-logic in which various logics (for example first-order predicate logic, the systems of the lambda cube or higher order logic) are described internally, in the Logical Framework style. This makes it having non-standard proof-objects. Again the system does not satisfy the Poincaré principle, but avoids the problem by not considering proof-objects. Both assistants HOL and Isabelle have pretty good rewrite engines, needed to run the non-standard proof-objects.

It should be emphasized that HOL and Isabelle did not fail to adept the Poincaré principle because it was forgotten, but because the problem of equational reasoning was solved in a different way, by the non-standard proof-objects in the form of the tactics. It makes formalizing more easy, but one cannot use proof-objects for example to see details of the proof or for program extraction. However, it is in principle not difficult to modify either HOL or Isabelle to create and store proof objects.

Mizar, ACL2, PVS

Mizar [1989] is based on a form of set theory (Tarski-Grothendieck, that is ZFC extended with an axiom expressing the existence arbitrary large cardinals). It does not work with proof-objects nor does it have the Poincaré principle. The system has some automated deduction and a user-friendly set of tactics. In fact a nice feature of the system is that the proof-script is close to an ordinary proof in mathematics (which are internally represented as proofs in set theory). An impressive collection of results is in the Mizar library. It seems that in principle it is possible that the Mizar scripts are translated into a proof-object.

ACL2 [2000] is an extension of the theorem prover of Boyer-Moore. It is based on classical primitive recursive arithmetic and it is used in industry. It is not possible for the user to construct inductive types, but there is a powerful built-in induction: a user can define his own well-founded recursive functions (up to ϵ_0 recursion) and let the system compute with them. (The functions are actually Lisp functions.)

PVS [1999] again is based on classical simple type theory. It is without proof-objects and exploits this by allowing all kind of rewriting, for numeric and symbolic equalities. The system is very user-friendly because of automated deduction that is built in. The system allows subtypes of the form

$$A = \{x : B \mid P(x)\}.$$

If the system has to check $a : A$ it will generate a proof-obligation for the reader: "prove $P(a)$". Up to our knowledge no effort has been made to provide PVS with proof-objects.

Comparison

The proof-assistants considered follow the following pattern:

Agda-Coq-Lego-Nuprl-HOL-Isabelle-Mizar-ACL2-PVS.

Agda, Coq and Lego are to the left, indicating reliability (Agda given the first place because it has only predicative logic; Coq coming second, since only strongly normalizing rewrite rules may be added). After that follow Nuprl, HOL and Isabelle, with their non-standard proof-objects (Nuprl coming first for the same reasons as Agda; Isabelle coming last, because the extra layer making things a bit harder to manage). Finally come Mizar, ACL2 and PVS, because they do not work with proof-objects. We put PVS last, because every now and then bugs are found in this system).

On the other hand, the order for internal automation is the opposite: ACL2 and PVS win and Agda loses. Of course eventually proof-assistants should be developed that are both reliable and user-friendly. The following judgments are based on some intuition and should not be taken too seriously.

Ass.	p.o.	reliab.	PP	logic	dep.t.	ind.t	autom.	#users
Agda	yes	+++	$\beta\delta\iota R_1$	int. pred.	yes	yes	none[8]	-
Coq	yes	++	$\beta\delta\iota R_2$	int.	yes	yes	+	++
Lego	yes	++	$\beta\delta\iota R_1$	int.	yes	yes	+	+
Nuprl	n.s.	++	$\beta\delta\iota R_3$	int.	yes	yes	+	++
HOL	n.s.	++	none	cl.	no	yes	++	++
Isabelle	n.s.	++	none	t.b.s.	no	no	++	++
Mizar	none	+	none	cl.	yes	no	+	++
ACL2	none	+	R_4	pra	no	yes[9]	+++	+++
PVS	none	$-$	none	cl.	no	no	+++	+++

[8]There is a little use of higher order unification

[9]Basically, there's only one inductive type in which the user 'codes' his induction

Legenda

Ass.	name of the Proof Assistant;
p.o.	proof-objects;
n.s.	non-standard;
reliab.	reliability;
PP	Poincaré principle;
dep.t.	dependent types;
ind.t.	inductive types;
autom.	degree of automation;
int.	intuitionistic logic preferred;
pred.	only predicative quantification;
cl.	classical logic;
pra	primitive recursive arithmetic (so no quantifiers);
t.b.s.	to be specified by the user;
R_1	arbitrary notion of reduction;
R_2	structurally well-founded recursion;
R_3	arbitrary provable equality;
R_4	ϵ_0-recursion.

There are very many other proof-assistants. See [Digimath 2000] for an impressive list.

5.2. Applications of proof-assistants

At present there are two approaches to the mechanical verification of complicated statements. The first one, that we may call the *pragmatic approach*, uses proof assistants with many complex tools to verify the correctness of statements. These tools include theorem provers and computer algebra systems, the correctness of which has not been verified (as a matter of fact, computer algebra systems are often not formally correct at all). Even if these systems may contain bugs the correctness of hardware systems and (relatively small but critical) software systems (like protocols) is dramatically increased, see [Rushby and Henke 1993] and [Ruess, Shankar and Srivas 1996]. Proof-assistants that are used include PVS, Nuprl, Isabelle and HOL.

The other approach, that we may call the *fundamental* one, aims at the highest degree of reliability. In this approach one only uses proof-assistants with a proof-checker that satisfies the de Bruijn criterion, i.e. have a small verifying program.

In this chapter we have focused our attention on the second approach. It should be remarked that even in this approach there is some spectrum of reliability. If the Poincaré principle is adopted for $\beta\delta\iota$-conversion, the verifying program is more complex than the one for just $\beta\delta$-conversion. This is natural and fair, since adopting the Poincaré principle for ι-conversion has as consequence that primitive recursive computations within a proof come without proof obligations. In fact the pragmatic proof-assistants can be viewed as a strong use of the computational power as provided by a form of the Poincaré principle.

Another parameter in a fundamental proof-assistant is the choice of strength of the underlying type system and hence the related logical system. For example, one may use first-order, second-order or higher-order logic. This parameter determines the *logical* strength of the proof system.

Rather than making a choice for the computational and logical strength one may think of a universal[10] system in which these two can be set according to the taste and application area of the user. It is hoped (and expected) that it is possible to construct a universal proof-assistant that is sufficiently efficient. Also there is a considerable foundational interest in the enterprise of constructing user-friendly proof-assistants. One has to realize which steps are obvious to the mathematician and provide suitable tools.

It is a (possibly long term) goal of the second approach to make the formalization of an informally known mathematical proof as easy as writing a mathematical paper say in LaTeX. At the same time the efficiency should be comparable to efficient systems for computer algebra.

Several notions in classical mathematics are not directly available in the constructive approach of type theory. Next to the failure of the excluded middle these include quotient sets, subsets defined by a property and partial functions. It is for good reasons that these constructions are not available. In the constructive type theoretic approach the notion $a : A$ should be decidable, a property that is lost in the presence of types representing undecidable sets.

In order to increase the ease of formalizing proofs several tools are being constructed that enhance the power of the fundamental approach. In this way eventually the power of the fundamental approach may be equal to that of the present day pragmatic one.

When the goal of easy formalization has been reached not only spin-off in system design, but also in the development of mathematics is expected. First of all there may emerge a different system of refereeing. People will only submit papers that are correct. The referee can focus on the judgment whether the paper is of interest and point out relations with other work. Then there will be an impact on teaching mathematics. The notion of proof can be taught by patient computers.

It is also to be expected that eventually proof-assistants will help the working mathematician. Arbitrary mathematical notions can be represented on a computer; not just the computable ones, as is presently the case in systems of computer algebra. The interaction between humans and computers may lead to fruitful new mathematics, where humans provide the intuition and machines take over part of

[10]Of course there cannot be a universal proof-assistant, due to Gödel's theorem. The word universal is used in the same way as ZFC is seen as a universal foundation: it captures large parts of mathematics

the craftsmanship.

Next to these theoretical aspects, there is a potential practical spin-off in the form of program extraction. In case a statement of the form

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

has been proved constructively, an algorithm finding the y in terms of the x can be extracted automatically. See [Mohring 1986, Paulin-Mohring and Werner 1993, Parent 1995].

For a discussion of issues related to (the future of) proof-assistants, see also the QED-manifesto in [Bundy 1994] (pp. 238–251).

Many (often smaller) proof-assistants we have not mentioned. For a (probably incomplete) but extended survey see [Digimath 2000].

Acknowledgments

We thank all people from the EC Working Group 'Types' and its predecessor 'Logical Frameworks' for the pleasant cooperation and the lively discussions over the years. In particular we want to thank Ana Bove, Thierry Coquand, Wolfgang Naraschewski, Randy Pollack, Dan Synek, Freek Wiedijk, Jan Zwanenburg and the readers for their very useful suggestions and comments.

Bibliography

ABRAMSKY, S., GABBAY, D. M. AND MAIBAUM, T., EDS [1992], *Handbook of Logic in Computer Science, Volume 2: Background: Computational Structures*, Oxford University Press.

ACL2 [2000], 'Applicative Common Lisp'. Architects: M. Kaufmann and J. Strother Moore.
URL: *http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html*

ACZEL P. [1999], On relating type theories and set theories, *in* 'Altenkirch, Naraschewski and Reus [1999]'.

AGDA [2000], 'A system for incrementally developing proofs and programs'. Architect: C. Coquand.
URL: *http://www.cs.chalmers.se/ catarina/agda/*

ALTENKIRCH, T., NARASCHEWSKI, W. AND REUS, B., EDS [1999], *International Workshop TYPES '98, Kloster Irsee, Germany, 1998: selected papers*, Vol. 1657, Springer-Verlag, Berlin.

AUDEBAUD P. [1991], Partial objects in the Calculus of Constructions, *in* 'Proceedings of the Symposium on Logic in Computing Science', IEEE, Amsterdam, NL, pp. 86–95.

BARENDREGT H. [1992], Lambda calculi with types, *in* 'Abramsky, Gabbay and Maibaum [1992]', Oxford University Press, pp. 117–309.

BARENDREGT H. [1997], 'The impact of the lambda calculus', *Bulletin of Symbolic Logic* **3**(2), 181–215.

BARENDREGT, H. AND NIPKOW, T., EDS [1994], *Types for proofs and programs: international workshop TYPES '93, Nijmegen, The Netherlands, 1993: selected papers*, Vol. 806 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

BARTHE G. [1996], Implicit coercions in type systems, *in* 'Berardi and Coppo [1996]', pp. 1–15.

BARTHE G., RUYS M. AND BARENDREGT H. [1996], A two-level approach towards lean proof-checking, *in* 'Berardi and Coppo [1996]', pp. 16–35.

Barthe G. and Sørensen M. [2000], 'Domain-free Pure Type Systems', *Journal of Functional Programming* **10**, 417–452. Preliminary version in S. Adian and A. Nerode, editors, Proceedings of LFCS'97, LNCS 1234, pp 9-20.

Barwise J. and Etchemendy J. [1995], *Hyperproof*, Cambridge University Press.

Berardi S. [1988], Towards a mathematical analysis of the Coquand-Huet Calculus of Constructions and the other systems in Barendregt's cube, Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino.

Berardi S. [1990], Type Dependence and Constructive Mathematics, PhD thesis, Dipartimento Matematica, Università di Torino.

Berardi, S. and Coppo, M., eds [1996], *Types for proofs and programs: international workshop TYPES '95, Torino, Italy, 1995: selected papers*, Vol. 1158, Springer-Verlag, Berlin.

Bezem, M. and Groote, J., eds [1993], *Typed Lambda Calculi and Applications, TLCA'93*, Vol. 664 of *Lecture Notes in Computer Science*, Springer, Berlin.

Bibel, W. and Schmitt, P., eds [1998], *Automated Deduction—A Basis for Applications*, Vol. I,II,III, Kluwer, Dordrecht.

Boyer R. and Moore J. [1997], *A Computational Logic Handbook*, second edn, Academic Press, London.

Bundy, A., ed. [1994], *Automated deduction, CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June 26–July 1, 1994: proceedings*, Vol. 814 of *Lecture Notes in Artificial Intelligence and Lecture Notes in Computer Science*, Springer-Verlag Inc.

Caprotti O. and Oostdijk M. [2001], 'Formal and efficient primality proofs by use of computer algebra oracles', *Journal of Symbolic Computation* **to appear**. Special Issue on Computer Algebra and Mechanized Reasoning.

Church A. [1940], 'A formulation of the simple theory of types', *Journal of Symbolic Logic* **5**, 56–68.

Constable et al. R. [1986], *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, New Jersey.

COQ [1999], 'The Coq proof assistant version 6.9'. Architects: Chr. Paulin-Mohring et al.
URL: *http://pauillac.inria.fr/coq/assis-eng.html*

Coquand T. [1985], Une théorie des Constructions, PhD thesis, Université Paris VII, Thèse de troisième cycle.

Coquand T. [1986], An analysis of Girard's paradox, *in* 'Proceedings of the Symposium on Logic in Computing Science', IEEE, Cambridge, Massachusetts.

Coquand T. [1991], An algorithm for testing conversion in type theory, *in* 'Huet and Plotkin [1991]', Cambridge University Press.

Coquand T. and Gallier J. [1990], A proof of strong normalization for the theory of Constructions using a Kripke-like interpretation, *in* G. Huet and G. Plotkin, eds, 'Preliminary Proceedings 1st Annual Workshop on Logical Frameworks, Antibes, France, 7–11 May 1990', pp. 479–497.
URL: *ftp://ftp.inria.fr/INRIA/Projects/coq/types/Proceedings/book90.dvi*

Coquand T. and Huet G. [1988], 'The Calculus of Constructions', *Information and Computation* **76**, 95–120.

Coquand T. and Paulin-Mohring C. [1990], Inductively defined types, *in* 'Martin-Löf and Mints [1990]', Vol. 417 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Daalen D. v. [1980], The Language Theory of Automath, PhD thesis, Eindhoven University of Technology, The Netherlands.

de Bruijn N. [1980], A survey of the project Automath, *in* 'Seldin and Hindley [1980]', Academic Press, pp. 579–606. Also in Nederpelt et al. [1994], pp 141–161.

de Bruijn N. G. [1970], The mathematical language AUTOMATH, its usage and some of its extensions, *in* M. Laudet, D. Lacombe and M. Schuetzenberger, eds, 'Symposium on Automatic Demonstration', Springer Verlag, Berlin, 1970, IRIA, Versailles, pp. 29–61. Lecture Notes in Mathematics **125**; also in Nederpelt et al. [1994].

DEZANI-CIANCAGLINI, M. AND PLOTKIN, G., EDS [1995], *Second International Conference on Typed Lambda Calculi and Applications, TLCA'95*, Vol. 902 of *Lecture Notes in Computer Science*, Springer, Berlin.

DIGIMATH [2000], 'A list of computer math systems'. F. Wiedijk.
URL: *http://www.cs.kun.nl/ freek/digimath*

DOWEK G. [1993], The undecidability of typability in the $\lambda\pi$-calculus, *in* 'Bezem and Groote [1993]', pp. 139–145.

DYBJER, P., NORDSTRÖM, B. AND SMITH, J., EDS [1995], *Types for proofs and programs: international workshop TYPES '94, Båstad, Sweden, 1994: selected papers*, Vol. 996 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

GEUVERS H. [1992], The Church-Rosser property for $\beta\eta$-reduction in typed lambda calculi, *in* 'Proceedings of the seventh annual symposium on Logic in Computer Science, Santa Cruz, Cal.', IEEE, pp. 453–460.

GEUVERS H. [1993], Logics and Type Systems, PhD thesis, Catholic University of Nijmegen, The Netherlands.

GEUVERS H. [1995], A short and flexible proof of strong normalization for the Calculus of Constructions, *in* 'Dybjer, Nordström and Smith [1995]', pp. 14–38.

GEUVERS H. AND NEDERHOF M. [1991], 'A modular proof of strong normalization for the Calculus of Constructions', *Journal of Functional Programming* **1**(2), 155–189.

GEUVERS H., POLL E. AND ZWANENBURG J. [1999], Safe proof checking in type theory with Y, *in* F. Flum and M. Rodriguez-Artalejo, eds, 'Computer Science Logic (CSL'99)', Vol. 1683 of *LNCS*, Spinger-Verlag, pp. 439–452.

GIMÉNEZ, E. AND PAULIN-MOHRING, C., EDS [1998], *International Workshop TYPES '96, Aussois, France, 1996: selected papers*, Vol. 1512 of *LNCS*, Springer-Verlag, Berlin.

GIRARD J.-Y. [1972], Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur, PhD thesis, Thèse d'Etat, Université Paris VII.

GIRARD, J.-Y., ED. [1999], *Typed Lambda Calculus and Applications, TLCA '99*, Vol. 1581 of *Lecture Notes in Computer Science*, Springer, Berlin.

GIRARD J.-Y., LAFONT Y. AND TAYLOR P. [1989], *Proofs and Types*, Vol. 7 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.

GOGUEN H. AND LUO Z. [1993], Inductive data types: Well-orderings revisited, *in* 'Huet and Plotkin [1993]', Cambridge University Press, pp. 198–218.

HARPER R., HONSELL F. AND PLOTKIN G. [1993], 'A framework for defining logics', *Journal of the ACM* **40**(1), 143–184.

HOFMANN M. [1994], Elimination of extensionality and quotient types in Martin-Löf type theory, *in* 'Barendregt and Nipkow [1994]', pp. 166–190.

HOL [1998], 'Higher order logic theorem prover'. Architects: K. Slind et al.
URL: *http://www.cl.cam.ac.uk/Research/HVG/HOL/*

HOWARD W. [1980], The formulas-as-types notion of construction, *in* 'Seldin and Hindley [1980]', Academic Press, pp. 479–490.

HOWE D. [1988], Computational metatheory in Nuprl, *in* E. Lusk and R. Overbeek, eds, 'Proceedings of the Ninth International Conference of Automated Deduction', number 310 *in* 'LNCS', Springer, Berlin, pp. 238–257.

HUET, G. AND PLOTKIN, G., EDS [1991], *Logical Frameworks*, Cambridge University Press.

HUET, G. AND PLOTKIN, G., EDS [1993], *Logical Environments*, Cambridge University Press.

HURKENS A. [1995], A simplification of Girard's paradox, *in* 'Dezani-Ciancaglini and Plotkin [1995]', pp. 266–278.

JAPE [1997], 'A framework for building interactive proof editors'. Architects: B. Sufrin and R. Bornat.
URL: *http://users.comlab.ox.ac.uk/bernard.sufrin/jape.html*

LEGO [1998], 'The Lego proof assistant'. Architect: R. Pollack.
URL: *http://www.dcs.ed.ac.uk/home/lego/*

LUO Z. [1989], ECC, the Extended Calculus of Constructions, *in* 'Logic in Computer Science', IEEE Computer Society Press.

LUO Z. [1994], *Computation and Reasoning: A Type Theory for Computer Science*, Vol. 11 of *Intl. Series of Monographs in Computer Science*, Clarendon Press.

LUO Z. [1999], 'Coercive subtyping', *Journal of Logic and Computation* **9**(1).

MAGNUSSON L. [1994], The implementation of ALF: a proof-editor based on Martin-Löf's monomorphic type theory with explicit substitution, PhD thesis, Dept. of Comp. Science, Chalmers University, Sweden.

MARTIN-LÖF P. [1984], *Intuitionistic Type Theory*, Studies in Proof Theory, Bibliopolis, Napoli.

MARTIN-LÖF, P. AND MINTS, G., EDS [1990], *COLOG-88: International conference on computer logic*, Vol. 417 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

MATHPERT [1997], 'Mathpert'. Architect: M. Beeson.
URL: *http://www.mathpert.com*

MCCARTHY J. [1962], Computer programs for checking mathematical proofs, *in* 'Proceedings of the Symposium in Pure Mathematics 5', American Mathematical Society.

MELLIES P. AND WERNER B. [1998], A generic proof of strong normalisation for Pure Type Systems, *in* 'Giménez and Paulin-Mohring [1998]'.

MIZAR [1989]. Architects: Andrzej Trybulec, Czeslaw Bylinski.
URL: *http://www.mizar.org*

MOHRING C. [1986], Algorithm development in the Calculus of Constructions, *in* 'Proceedings of the First Symposium on Logic in Computer Science, Cambridge, Mass.', IEEE, Washington DC, pp. 84–91.

NEDERPELT R. [1973], Strong normalisation in a lambda calculus with lambda structured types, PhD thesis, Eindhoven University of Technology, The Netherlands.

NEDERPELT, R., GEUVERS, H. AND DE VRIJER, R., EDS [1994], *Selected Papers on Automath*, Studies in Logic and the Foundations of Mathematics **133**, North-Holland, Amsterdam.

NORDSTRÖM B., PETERSSON K. AND SMITH J. [1990], *Programming in Martin-Löf's Type Theory*, Oxford University Press.

OOSTDIJK M. AND GEUVERS H. [2001], 'Proof by computation in the Coq system', *Theoretical Computer Sci.* **to appear**.

OPENMATH [1998].
URL: *http://www.nag.co.uk/projects/openmath/omsoc*

OTTER [1998]. Architect: William McCune.
URL: *http://www.mcs.anl.gov/AR/otter*

PARENT C. [1995], Synthesizing proofs from programs in the Calculus of Inductive Constructions, *in* B. Möller, ed., 'Proceedings 3rd Intl. Conf. on Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany, 1995', Vol. 947 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 351–379.

PAULIN-MOHRING C. [1994], Inductive definitions in the system Coq; rules and properties, *in* 'Bezem and Groote [1993]', pp. 328–345.

PAULIN-MOHRING C. AND WERNER B. [1993], 'Synthesis of ML programs in the system Coq', *Journal of Symbolic Computation* **15**, 607–640.

PFENNING F. [1991], Logic programming in the LF logical framework, *in* 'Huet and Plotkin [1991]', Cambridge University Press, pp. 149–181.

PFENNING F. [2001], Logical frameworks, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 17, pp. 1063–1147.

POLLACK R. [1995], A verified type checker, *in* 'Dezani-Ciancaglini and Plotkin [1995]', pp. 365–380.

PVS [1999], 'Specification and verification system'. Architects: J. Rushby et al.
URL: *http://pvs.csl.sri.com/*

RAMSEY F. [1925], 'The foundations of mathematics', *Proceedings of the London Mathematical Society* pp. 338–384.

RUESS H., SHANKAR N. AND SRIVAS M. [1996], Modular verification of SRT division, *in* 'Proceedings of the 8th International Conference on Computer Aided Verification, New Brunswick, NJ, USA, eds. R. Alur and T.A. Henzinger', Vol. 1102 of *Lecture Notes in Computer Science*, Springer, pp. 123–134.

RUSHBY J. AND HENKE F. V. [1993], 'Formal verification of algorithms for critical systems', *IEEE Transactions on Software Engineering* **19**(1), 13–23.

RUSSELL B. [1903], *The Principles of Mathematics*, Allen & Unwin, London.

SCOTT D. [1970], Constructive validity, *in* D. L. M. Laudet and M. Schuetzenberger, eds, 'Symposium on Automated Demonstration', Vol. 125 of *Lecture Notes in Mathematics*, Springer, Berlin, pp. 237–275.

SELDIN, J. AND HINDLEY, J., EDS [1980], *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press.

SEVERI P. [1998], 'Type inference for Pure Type Systems', *Information and Computation* **143-1**, 1–23.

SEVERI P. AND POLL E. [1994], Pure Type Systems with definitions, *in* A. Nerode and Y. Matiyasevich, eds, 'Proceedings of LFCS'94, St. Petersburg, Russia', number 813 *in* 'LNCS', Springer Verlag, Berlin, pp. 316–328.

SWAEN M. [1989], Weak and strong sum-elimination in intuitionistic type theory, PhD thesis, University of Amsterdam.

TERLOUW J. [1989], Een nadere bewijstheoretische analyse van GSTT's (Dutch), Technical report, Department of Computer Science, Catholic University of Nijmegen.

VAN BENTHEM JUTTING L. [1993], 'Typing in Pure Type Systems', *Information and Computation* **105**(1), 30–41.

VAN BENTHEM JUTTING L., MCKINNA J. AND POLLACK R. [1994], Checking algorithms for Pure Type Systems, *in* 'Barendregt and Nipkow [1994]', pp. 19–61.

WHITEHEAD A. AND RUSSELL B. [1910, 1927], *Principia Mathematica Vol 1 (1910) and 2 (1927)*, Cambridge University Press.

WU W. [1994], *Mechanical Theorem Proving in Geometries*, Texts and Monographs in Symbolic Computation, Springer.

ZWANENBURG J. [1999], Pure Type Systems with subtyping, *in* 'Girard [1999]', pp. 381–396.

Index

Name index