# Proof-assistants using Dependent Type Systems

Henk Barendregt  Herman Geuvers

December 10, 1999

## Contents

**1 Proof-checking** **2**

**2 Typetheoretic notions for proof-checking** **4**
2.1 Propositions as types . . . . . 4
2.2 Proofs as terms . . . . . 7
2.3 Functions: algorithms versus graphs . . . . . 10
2.4 Subject Reduction . . . . . 12
2.5 Conversion and Computation . . . . . 12
2.6 Equality . . . . . 14
2.7 Connection between logic and type theory . . . . . 18

**3 Type systems for proof-checking** **22**
3.1 Higher order predicate logic . . . . . 23
3.2 Higher order typed $\lambda$-calculus . . . . . 26
3.3 Pure Type Systems . . . . . 36
3.4 Properties of Pure Type Systems . . . . . 39
3.5 Extensions to Pure Type Systems . . . . . 41
3.6 Products and Sums . . . . . 42
3.7 $\Sigma$-types . . . . . 43
3.8 Inductive Types . . . . . 44

**4 Proof-development in type systems** **49**
4.1 Tactics . . . . . 49
4.2 Examples of Proof Development . . . . . 51
4.3 Autarkic Computations . . . . . 57

**5 Proof assistants** **60**
5.1 Comparing proof-assistants . . . . . 60
5.2 Applications of proof-assistants . . . . . 63

**Abstract**

Science and technology depend on true statements. Most of these are obtained via reasoning from experimental data using mathematics (models). Therefore the notion of *proof* that warrants correctness plays a highly important role. Even if a statement is proved, it may not be true; simply because a proof can be incorrect. Therefore a reliable technique of proof-checking has been proposed. For this method one needs fully formalised proofs. Since provability is undecidable, proofs in general cannot be generated automatically by machines. They have to be developed in an interactive fashion between a human and a machine. First a short description of the methodology of the development and verification of proofs is given. Then a class of formal systems is introduced that is particularly useful for this goal: various systems of dependent type theory. One of the reasons for this is that in type theory there is a seemless collaboration between computations and statements. Other reasons are that type theory forms a very natural way to formalise parts of mathematics and allows a compact way of representing proofs.

## 1. Proof-checking

Proof-checking consists of the automated verification of mathematical theories by first fully formalising 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 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 formalised 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 formalise 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 practise* this is not 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 $\lambda$-calculus) and of *proof* (via the so called 'formulas-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 formalised, its correctness can be verified by a small program, the *proof-checker*. But in order to make the formalisation 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

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

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 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.

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. For example the language OpenMath [1998] is aiming at providing an intermediate level between such mathematical computer applications. Reaching

[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.

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 formalising 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 way of transferring a result from the Mizar theorem prover to another application and also it is hard to find results in the Mizar library.

*Further reading*

The following is relevant literature. Books on type theory: Martin-Löf [1984], Nordström et al. [1990], Nederpelt et al. [1994], Luo [1994]. Proceedings on typed lambda calculus and applications: Bezem and Groote [1993], Dezani-Ciancaglini and Plotkin [1995], de Groote and Hindley [1997] and Girard [1999]. Proceedings of the TYPES workinggroup: Barendregt and Nipkow [1994], ... .

We now first will discuss some specific features of type theory useful for formalisation.

## 2. Typetheoretic 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 delta- and beta-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 normalisation; 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, want to represent 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 revitalising type systems for this purpose. Martin-Löf for his emphasis on reliability (by requiring a clear phenomenological semantics) and proposal to restrict to predicative type systems.

### 2.1. Propositions as types

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

| In situation $\Gamma$ we have $A$.<br>Proof. $p$. ■ |
|---|

*Informal mathematics*

Here $\Gamma$ 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 $\Gamma, A$ become formal objects and so does the notion of provability. Proofs still are presented in an informal way, but theoretically they can be formalised as a derivation-tree (following some precisely given set of rules).

| $\Gamma \vdash_L A$ |
| --- |
| Proof. $p$. ■ |

*Mathematics formalised in logic*

It turns out that it is possible to represent proofs in an 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 $A \,\&\, B$ 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 $\bot$, 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, $\supset, \&, \vee, \bot, \forall$ and $\exists$ are the usual logical connectives and quantifiers. Negation is defined as $\neg A = A \supset \bot$.

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, acccording to the intuitionistic interpretation of the logical connectives one has

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

On the right hand side we have *types* rather than sets. The intuitive difference 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 $a{\in}A$.

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

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

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. Therefore the formalization of mathematics in type theory becomes the following (we do not write the [ ] but identify $A$ with $[A]$).

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

*Mathematics formalised in type theory*

Type theory not only consists of a formal system to generate types, it also generates terms that belong to certain of these types. For example $\mathtt{id}_A \equiv \lambda x : A.x$ is the identity on $A$ and hence of type $A \to A$.

The statement $\Gamma \vdash_T p : A$ is equivalent to

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

*Proof-checking*

Here, $\mathrm{Type}_{_}(-)$ is the type checking function. The decidability of type-checking follows from:

- $\mathrm{Type}_\Gamma(p)$ generates a type of $p$ in context $\Gamma$ 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, there are three main decidability problems that we can distinguish.

| | | | | |
|---|---|---|---|---|
| $\Gamma$ | $\vdash_T$ | $M : A?$ | TCP, | Type Checking Problem; |
| $\Gamma$ | $\vdash_T$ | $M : ?$ | TSP. | Type Synthesis Problem; |
| $\Gamma$ | $\vdash_T$ | $? : A$ | TIP, | Type Inhabitation Problem. |

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. 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 *recognise* 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 et al. [1994] for an up to date survey. At 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 formalised 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.

| | | |
|---|---|---|
| provability of formula $A$ | $=$ | 'inhabitation' of the type $A$ |
| proof-checking | $=$ | type-checking |
| interactive theorem proving | $=$ | interactive construction of a term of a given type. |

So the decidability of type checking is at the core of the type-theoretic approach to theorem proving. To get an idea of what these proof-objects really look like and how type checking really 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.

### 2.2. Proofs as terms

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 $R$ is anti-symmetric*
>
> $$\forall x, y{\in}X.R(x,y) \supset \neg R(y,x).$$
>
> *Then $R$ is irreflexive* $\forall x{\in}X.\neg R(x,x)$.

We want to formalise this. In the type theory we have two *universes*, $\mathsf{Set}$ and $\mathsf{Prop}$. The idea is that a term $X$ of type $\mathsf{Set}$, 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 $P\ t$ is inhabited*,

i.e. there is a proof-term of type $P\ t$. 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 logic!) is *defining*. This is formally captured in type theory via a kind of 'let' construction. Let us give some definitions. (We start using notation more akin to typetheory.)

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

These definitions are *formal constructions* in type theory with a computational behaviour, so-called $\delta$-reduction, by which definitions are unfolded. $\mathsf{Rel}$ takes a domain $X$ and returns the domain of binary relations on $X$:

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

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

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

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

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

(For now, we take $\bot$ to be some fixed closed term of type $\mathsf{Prop}$.) Given the (informal) explanation of the $\Pi$-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

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

We claim that the term

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

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

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 $\lambda$-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 $\lambda X{:}\mathsf{Set}\lambda R{:}(\mathsf{Rel}\ X)\lambda h{:}(\mathsf{AntiSym}\ X R)\lambda x{:}X\lambda q{:}(R\ x\ x).h\ x\ x\ q\ q$, we compute the type

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

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

$$h\ x\ x : (R\ x\ x){\to}((R\ x\ x){\to}\bot).$$

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

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

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

$$\lambda X{:}\mathsf{Set}\lambda R{:}(\mathsf{Rel}\ X)\lambda h{:}(\mathsf{AntiSym}\ X R)\lambda x{:}X.\lambda q{:}(R\ x\ x).h\ x\ x\ q\ q \quad : \\ \Pi X{:}\mathsf{Set}\Pi R{:}(\mathsf{Rel}\ X)(\mathsf{AntiSym}\ X R){\to}(\mathsf{Irrefl}\ X R).$$

We understand that $\forall$ and $\Pi$ often will be used as synonymous, and similarly $\supset$ and $\to$.

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. 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 $\beta$-reductions, the definitions 'hide' most of the $\beta$-redexes. This can be seen from the fact that proof-terms are almost always in $\beta$-normal form (but certainly not in $\delta$-normal form). See Coquand [1996] and Pollack [1995] 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.

### 2.3. 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 contaning $x : A$ (and no other dependencies on $x$ in the context), we construct, using the $\lambda$-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 $\beta$ and $\iota$ 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 $\forall x{:}A\, \exists!y{:}B.R\ x\ y$, 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 $\iota$ 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 $\forall P{:}A{\to}\mathsf{Prop}(\exists!x{:}A.P\ x) \to P(\iota P)$. So, if there is a unique element for which $P$ holds, then $\iota P$ denotes this element, otherwise $\iota P$ is an arbitrary unspecified element. Obviously, the latter aspect of the $\iota$ 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 $\forall x{:}A \exists y{:}B.R\ x\ y$ 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 $\forall x{:}A \exists y{:}B.R\ x\ y$, then $p$ contains a term $f{:}A \to B$ and a proof-term of type $\forall x{:}A.R\ x\ (f\ x)$.

Note that this is a *meta-theoretic* property of constructive systems: there is not (necessarily) a function *inside* the system that extracts the $f{:}A \to B$ 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 $\Sigma$-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 $\Sigma$-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 $\Sigma$-introduction and the $\Sigma$-elimination rules, respectively. This implies that from a proof-term $p : \forall x{:}A \exists y{:}B.R\ x\ y$, we can immediately extract the *function* $f : A \to B$ defined by $\lambda x{:}A.\pi_1\ (p\ x)$ and we can prove for this $f$ that $\forall x{:}A.R\ x\ (f\ x)$ holds. (The proof-term is $\lambda x{:}A.\pi_2\ (p\ x)$.) The extracted function also has a proper computational behaviour: a closed proof-term $p : \forall x{:}A \exists y{:}B.R\ x\ y$ 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 $\Sigma$-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 $\Sigma$-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 $\iota$ operator, conforms with the conception of ordinary logic that we have just sketched: the object $\iota P$ 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 2.7. 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 ($f : A \to B$) and graphs (that is, a relation $R : A \to B \to \mathsf{Prop}$ 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 $f : A \to B$, there is still a clear distinction: the proof is not the same as the function.

### 2.4. 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 $\beta$, $\delta$ and $\iota$ that we have already encountered. We illustrate the use of reduction by an example. Suppose we have as definition $\mathsf{plus} := \lambda x, y{:}\mathsf{nat}.\mathsf{Rec}\ x(\lambda z{:}\mathsf{nat}.\mathsf{succ})y$. Then the 'value' of the expression $\mathsf{plus}\,1\,0$ is computed by first unfolding the plus (one $\delta$-reduction step), then performing two $\beta$-steps and then one $\iota$-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 $\lambda h{:}(A{\to}A{\to}B)\lambda z{:}A.h\ z\ z : (A{\to}A{\to}B){\to}(A{\to}B)$, corresponding to the standard natural deduction proof of this fact, ending with an introduction rule. Now, if we also have a proof $q : A{\to}A{\to}B$, qq then we can eliminate the implication obtaining $(\lambda h{:}(A{\to}A{\to}B)\lambda z{:}A.h\ z\ z)q : A{\to}B$. If we do one $\beta$-step we eliminate the cut obtaining the proof-term $\lambda z{:}A.q\ z\ z : A{\to}B$. 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 $\beta$-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 $\lambda h{:}(A{\to}A{\to}B)\lambda z{:}A.h\ z\ z : (A{\to}A{\to}B){\to}(A{\to}B)$ as above, we will *save* this lemma under a name, say $\mathsf{lemma}_1$, and we will only refer to this 'name' $\mathsf{lemma}_1$. In type theory, what happens is that we introduce a definition $\mathsf{lemma}_1 := \lambda h{:}(A{\to}A{\to}B)\lambda z{:}A.h\ z\ z$ and we use $\mathsf{lemma}_1$ 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 $\delta$.

### 2.5. Conversion and Computation

We have already encountered three notions of computation: $\beta$-, $\iota$- and $\delta$-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 Sections 2.1 and 3.2. 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 $p : \forall x, y, z{:}\mathsf{nat}.x > (\mathsf{plus}yz){\to}x > z$ 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 $\delta$-reduction (unfolding the definition of $\mathsf{plus}$), two $\beta$-steps and a $\iota$-step. Here we see an application of the conversion rule:

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

In the example above, $M$ is $p\ 2\ 1\ 0$, $\varphi$ is $(2>(\mathsf{plus}\ 1\ 0))\to(2>0)$ and $\psi$ is $(2>1)\to(2>0)$. The proof-term $M$ is left unchanged under the transition from $\varphi$ to $\psi$. 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 $\psi$ we only have to check whether $M$ has a type and if so, verify whether it's convertible with $\psi$.) 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 $\varphi$ to $\psi$ 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 $\varphi$ to $\psi$.) 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 '$\mathsf{Class\text{-}of\text{-}Form}$' 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, $\mathsf{true}$.) In more precise type-theoretic terms: suppose we have a proof-term $\mathsf{ok}$ with

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

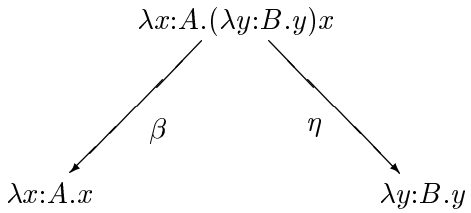
Then, to prove that a formula $T : \mathsf{Prop}$ from our class is provable, we only have to find its encoding $t : \mathsf{Class\text{-}of\text{-}Form}$ and then

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

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 $\mathsf{Check}$. 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 et al. [1996] – where it is called the 'two-level approach' and Oostdijk and Geuvers [2000]. 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 proof 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.

### 2.6. Equality

Note that we have not included $\eta$-reduction in the conversion rule, but just $\beta$, $\delta$ and $\iota$. This may seem remarkable, because for the untyped $\lambda$-calculus, many nice results of $\beta$-reduction (like confluence) extend to $\beta\eta$. This is however not the case for typed $\lambda$-calculus. The snag lies in the fact that our typed terms have a type attached to the bound variable in the $\lambda$-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 $\beta$ and $\eta$. For example consider $\lambda x{:}A.(\lambda y{:}B.y)x$,




These terms have a common reduct only if $A$ and $B$ have. This complication of $\eta$ 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 $\lambda$-terms *without* types attached to the bound variables, see Barthe and Sörensen [n.d.], 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 a *equality judgement* of the form

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

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

*Equality as a judgement or as a type*

As rules for deriving an equality judgement we would have $\beta$, $\delta$ and $\iota$ 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

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

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

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

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

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

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

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

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

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 judgement itself (as above), but often it is represented as a 'convertibility side condition', like in 2.5. 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 $\beta$, $\delta$ and, if present, also $\iota$. 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 ($M =_A N$ : 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:

$$t =_A q \;:=\; \Pi p{:}A \to \mathsf{Prop}(P\ t) \to (P\ q).$$

   Note that this equality looks assymmetric; 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

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

   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:

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

   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. the first requires impredicativity to be definable, while the second 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 2.1 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 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

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

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 $\Sigma$-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

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

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

### 2.7. 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 faithfull implementation of the type theory. The actual situation is as follows: the interactive part (where the proof-terms are *created*) is not fully explained in terms of the type theory; however, the *checking* (which is done after the proof-term has been completed) is completely faithfull 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, for $A : \mathsf{Prop}$, is $A$ true in $\mathcal{M}$ iff $A$ is inhabited?
- 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?

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 *formulas-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 represents proofs is affirmative: proof-terms represents natural deduction proofs. Of course, the situataion 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 formulas-as-types embedding:

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

where $\Sigma$ denotes the deduction of $\varphi$ in $\mathcal{L}$ and $[\![\Sigma]\!]$ its encoding as a term in $S(\mathcal{L})$. $\Gamma$ is a context in which the relevant variables are declared. In Section 3.2, we describe the formulas-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 $\varphi$ is inhabited in $S(\mathcal{L})$, then $\varphi$ is derivable in $\mathcal{L}$ (where $\varphi : \mathsf{Prop}$).

*Completeness* of the formulas-as-types embedding:

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

where $\Gamma$ 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 formulas-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 disctinction between 'sets' and 'propositions' – both are of type Set. We have already discussed this situation in Section 2.3, 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 $P\ a$, 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 $\Sigma$-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 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 $\Sigma$-type, which does not work well for higher order logic, but that is 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 built 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 $\mathsf{nat}$) 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 formulas-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 not be defined (whereas now it is an 'unspecified' element of $A$). In type theory there is a nice solution to this: construct

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

Then $\mathsf{recip}$ 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 to understand the dependency of this object (of type $A$) on the proof $p$ in terms of ordinary mathematics?

A 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. $\mathsf{mult}$ 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 nonzeros to $[A, =_A]$ itself. In this case, $\mathsf{recip}$ 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 $\mathsf{recip}$ now is a *setoid function*, which implies the following.

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

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 subterm 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 $\varphi$ 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 $\mathsf{absv}$ as follows.

$$|\ \ | := \lambda x{:}F.\ \mathsf{case}\ (px)\ \mathsf{of}\ \begin{array}{lcl} (\mathsf{inl}_) & \Rightarrow & x \\ (\mathsf{inr}_) & \Rightarrow & -x \end{array}$$

This function distinguishes cases according to the value of $px$. If it is of the form $\mathsf{inl}_$ (then $_ : x \geq 0$), we take $x$; if it is of the form $\mathsf{inr}_$ (then $_ : x \leq 0$), we take $-x$. Now, for $a : F$, the term $\mathsf{absv}a$ contains a proof-term $p$. We want to prove that the values of $\mathsf{absv}$ 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, $\mathsf{absv}_p$ and $\mathsf{absv}_q$, 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

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

Note that it may be the case that for some $x$, the value of $px$ is $\mathsf{inl}_$, while the value of $qx$ is $\mathsf{inr}_$. Then $\mathsf{absv}_p x$ has value $x$ and $\mathsf{absv}_q x$ 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 stiil 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 $\beta$, $\delta$, $\iota$ and $\eta$ rules as possible reduction rules. A very powerful extension of the reduction relation is obtained by adding a fixed-pointoperator $\mathsf{Y}{:}\forall A.(A{\rightarrow}A){\rightarrow}A$ satisfying

$$\mathsf{Y}f{\rightarrow}_{\mathsf{Y}} f(\mathsf{Y}f).$$

With this addition the reduction of the type system does not satisfy strong normalisation and proof-objects are *potential* ones. It has been shown in Geuvers et al. [1999] that under mild conditions the $\mathsf{Y}$-rules are conservative over the ones without a $\mathsf{Y}$.

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, $\lambda$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 $\lambda$HOL, both as a motivation for the definition of $\lambda$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 $\lambda$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 $\lambda$HOL 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 2.7. We also give no formal treatment of definitions here (the $\delta$-rule for unfolding definitions etc., see Section 2.6). 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 $\lambda$-terms are a faithfull 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 $\Omega$ represents the *domain of propositions*.

2. For every $\sigma \in D$, the set of *terms of type* $\sigma$, $\mathrm{Term}_\sigma$ is inductively defined as follows. (As usual we write $t : \sigma$ to denote that $t$ is a term of type $\sigma$.)

   (a) the constants $c_1^\sigma, c_2^\sigma, \ldots$ are in $\mathrm{Term}_\sigma$,

   (b) the variables $x_1^\sigma, x_2^\sigma, \ldots$ are in $\mathrm{Term}_\sigma$,

(c) if $\varphi : \Omega$ and $x^\sigma$ 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 $x^\sigma$ is a variable, then $(\lambda x^\sigma.M) : \sigma \to \tau$.

3. The set of terms of **HOL**, Term, is defined by $\text{Term} := \cup_{\sigma \in D} \text{Term}_\sigma$.

4. The set of formulas of **HOL**, $\mathsf{form}$, is defined by $\mathsf{form} := \text{Term}_\Omega$.

We adapt the well-known notions of *free* and *bound* variable, substitution, $\beta$-reduction and $\beta$-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 $\Omega$, 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 3.5 for more details.

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 $R(t, q)$.

3.2. Examples. Before giving the logical rules of $\lambda\mathsf{HOL}$, 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 $\mathsf{ind}$ defined as

$$\forall P^{\mathbb{N} \to \Omega}.(P0) \Rightarrow (\forall x^{\mathbb{N}}.(Px \Rightarrow P(S\ x))) \Rightarrow \forall x^{\mathbb{N}}.Px.$$

3. The formula $\mathsf{trans}(R)$, defined as $\forall x^A y^A z^A (R\ x\ y \Rightarrow Ryz \Rightarrow R\ x\ z)$ denotes the fact that $R$ is transitive. So, as a matter of fact, $\mathsf{trans} : (A \to A \to \Omega) \to \Omega$. Note that, as usual, we use $\forall x^A y^A z^A$ 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 $R \subseteq Q := \forall x^A y^A.(R\ x\ y \Rightarrow Q\ x\ y)$. (We informally use the infix notation $R \subseteq Q$ to denote $\subseteq RQ$.)

5. The term $\lambda x^A y^A.(\forall Q^{A \to A \to \Omega}.(\mathsf{trans}(Q) \Rightarrow (R \subseteq Q) \Rightarrow Q\ x\ y)$ 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.

3.3. DEFINITION. The notion of *provability*, $\Gamma \vdash \varphi$, for $\Gamma$ a finite set of formulas (terms of type $\mathsf{form}$) and $\varphi$ a formula, is defined inductively as follows.

| | | |
|---|---|---|
| (axiom) | $\dfrac{}{\Gamma \vdash \varphi}$ | 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 $\varphi =_\beta \psi$ |

3.4. REMARK. The rule (conversion) is an operationalization of the Poincaré principle discussed in Section 2.6. The rule says that we don't want to distinguish between $\beta$-equal propositions.

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

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

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 $\sigma$ 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.)

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

**Impredicativity** In the definition of the connectives (Example 3.5) and in the definition of equality, one makes use of *impredicativity*, i.e. 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 3.5 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. 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 can only base on the existence of some other object or collection of objects. 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 has already been constructed, before we construct binary relations. 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 $\lambda$-calculus

In type theory, one interprets formulas and proofs via the well-known 'formulas-as-types' and 'proofs-as-terms' embedding, originally due to Curry, Howard and de Bruijn. Under this interpretation, a formula is viewed as the type of its proofs. It turns out that one can define a typed $\lambda$-calculus $\lambda$HOL that represents **HOL** in a very precise way. What *very precise* means will not be defined here, but see e.g. Barendregt [1992]. Here, we just define the system $\lambda$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 $\lambda$HOL

2. In $\lambda$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 $\lambda$-terms. The discharging of hypotheses is done by $\lambda$-abstraction. The modus ponens rule is interpreted via application.

The *derivable judgements* of λHOL are of the form

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

where $\Gamma$ 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. In a context the variables that occur in $M$ and $A$ are given a type. If, in the judgement $\Gamma \vdash M : A$, 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 $A$.

3.6. DEFINITION. The typed $\lambda$ calculus λHOL, representing higher order predicate logic, is defined as follows. The set of *pseudoterms* $\mathcal{T}$ 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 pseudoterms, are as follows. Here, $s$ ranges over the set of sorts $\mathcal{S}$.

| | | |
|---|---|---|
| (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 $(s_1, s_2) \in$ { (Type, Type), (Prop, Prop), (Type, 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 $A =_\beta B$ |

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 $\Gamma$. The equality in the conversion rule (conv) is the $\beta$-equality on the set of pseudoterms $\mathcal{T}$.

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 pseudoterm $A$ is *legal* if there is a context $\Gamma$ and a pseudoterm $B$ such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable.

The set of legal terms of $\lambda$HOL is denoted by Term($\lambda$HOL). A context $\Gamma$ 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 $\Pi$, which comes in three flavours, 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 $\Pi$-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.

NOTATION. 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 $\Pi$-type that can be encountered in $\lambda$HOL.

3.7. EXAMPLES. 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 $\lambda$HOL, 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 $\lambda$HOL, 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 $\lambda$HOL.

We will not define formal interpretations from **HOL** to $\lambda$HOL 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.

3.8. EXAMPLES. 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 $\lambda$-abstraction to define predicates.

2. $\mathbb{N}{:}\mathsf{Type}, 0{:}\mathbb{N}, S{:}\mathbb{N}{\to}\mathbb{N} \vdash \Pi P{:}\mathbb{N}{\to}\mathsf{Prop}.(P0){\to}(\Pi x{:}\mathbb{N}(Px{\to}P(S\ x))){\to}\Pi x{:}\mathbb{N}Px : \mathsf{Prop}$. This is the formula for induction written down in $\lambda$HOL as a term of type Prop.

3. $A{:}\mathsf{Type}, R{:}A{\to}A{\to}\mathsf{Prop} \vdash \Pi x, y, z{:}A.R\ x\ y{\to}Ryz{\to}R\ x\ z : \mathsf{Prop}$. (Transitivity of $R$)

4. $A{:}\mathsf{Type} \vdash \lambda R, Q{:}A{\to}A{\to}\mathsf{Prop}.\Pi x, y{:}A.R\ x\ y{\to}Q\ x\ y : (A{\to}A{\to}\mathsf{Prop}){\to}(A{\to}A{\to}\mathsf{Prop}){\to}\mathsf{Prop}$. (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 relation is also called 'Leibniz equality' and is usually denoted by $=_L$ or $=_A$ if we want to denote the domain type explicitly.

6. $A{:}\mathsf{Type}, x, y{:}A \vdash \lambda r{:}(x =_A y).\lambda P{:}A{\to}\mathsf{Prop}.r(\lambda z{:}A.Pz{\to}P\ x)(\lambda q{:}P\ x.q) :$
   $(x =_A y){\to}(y =_A x)$. 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 $\lambda$HOL. 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 $\Pi$-types) are heavily 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. 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 e.g. a product type. A product type is very close to the conjunction (&), but the construction above for & can only be done at the Prop level.

*Formulas-as-types for higher order predicate logic*

The formulas-as-types interpretation from higher order predicate logic **HOL** into $\lambda$HOL maps a formula to a type and a derivation (in natural deduction of formula $\varphi$) to a typed $\lambda$-term (of the type associated with $\varphi$):

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

where $(\!|-|\!)$ denotes the interpretation of formulas as types and $[\![-]\!]$ denotes the interpretation of derivations as $\lambda$-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 $\Gamma_\Sigma$ is the context that declares all domains, constants and free variables that occur in $\Sigma$.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash \forall x^A y^A R\ x\ y \ \Rightarrow\ R\ y\ x \ \Rightarrow\ \bot}{\Gamma' \vdash \forall y^A R\ x\ y \ \Rightarrow\ R\ y\ x \ \Rightarrow\ \bot}}{\Gamma' \vdash R\ x\ x \ \Rightarrow\ R\ x\ x \ \Rightarrow\ \bot} \qquad \Gamma' \vdash R\ x\ x}}{\Gamma' \vdash R\ x\ x \ \Rightarrow\ \bot} \qquad \Gamma' \vdash R\ x\ x}}{\dfrac{\Gamma' \vdash \bot}{\Gamma \vdash R\ x\ x \ \Rightarrow\ \bot}}}{\Gamma \vdash \forall x^A.R\ x\ x \ \Rightarrow\ \bot}$$

This derivation is mapped to the typed $\lambda$-term $\lambda x{:}A.\lambda q : R\ x\ x.z\ x\ x\ q\ q$. This term is well-typed in the context $A : \mathsf{Type}, R : A{\to}A{\to}\mathsf{Prop}, z : \Pi x, y{:}A.(R\ x\ y{\to}R\ y\ x{\to}\bot)$, yielding the following judgement, derivable in $\lambda$HOL in context $\Gamma = \{A{:}\mathsf{Type}, R{:}A{\to}A{\to}\mathsf{Prop}, z{:}\Pi x, y{:}A.(R\ x\ y{\to}R\ y\ x{\to}\bot)\}$

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

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

Now one may wonder if the type system $\lambda$HOL is really faithful to higher order predicate logic **HOL**. Put differently, one can ask the question of *completness*: given a proposition of **HOL** such that $\Gamma_\varphi \vdash M : (\!|\varphi|\!)$ in $\lambda$HOL, is $\varphi$ derivable in **HOL**? It turns out that this is the case. Even though the number of rules of $\lambda$HOL is limited (where one rule serves several different purposes, e.g. the ($\lambda$)-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 $\mathsf{Prop}$) and the sets (terms of type $\mathsf{Type}$), we can completely *disambiguate* the syntax. This is stated by the following Lemma.

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

1. *$\Gamma_D, \Gamma_L, \Gamma_P$ is a* permutation *of $\Gamma$,*
2. *$\Gamma_D, \Gamma_L, \Gamma_P \vdash M : A$*
3. *$\Gamma_D$ consists only of declarations $A : \mathsf{Type}$,*

4. $\Gamma_L$ *consists only of declarations* $x : A$ *with* $\Gamma_D \vdash A : \mathsf{Type}$,

5. $\Gamma_P$ *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 $\psi$ with $\Gamma \vdash \psi : \mathsf{Prop}$ be given.

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

Here the $\varphi_1 \ldots \varphi_n$ are computed from $\Gamma$, using Lemma 3.9, 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 formulas-as-types interpretation: if $\Gamma \vdash M : \varphi$, then $\varphi$ is derivable in **HOL** from the assumptions listed in $\Gamma_P$.

*Type Checking*

An important property of a type system is *computability of types*, i.e. given $\Gamma$ 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 inference problem, TIP*. In this Section, a type inference algorithm for the system $\lambda\mathsf{HOL}$ is given, which is quite reminiscent for type inference algorithms in general. Before discussing the details we briefly recapitulate some generalities on type inference and type checking. See also Sections 2.1, 2.2.

A problem related to type inference is *decidability of typing*, i.e. given $\Gamma$, $M$ 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 inference 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 inference and type checking are closely entwined. (See Section 2.2 for an extended example.) The crucial algorithm to construct is an algorithm Type_(−), that takes a context $\Gamma$ 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 $\beta$-equality checking: Confluence for $\beta$-reduction and Strong Normalization for $\beta$-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.)

3.10. Proposition (Confluence). *On the set of pseudoterms $\mathcal{T}$, $\beta$-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 $\beta$ can be proved by following the well-known proofs for confluence for the untyped $\lambda$-calculus. Another important property of $\lambda$HOL is Subject Reduction.

3.11. Proposition (Subject Reduction). *The set of legal terms of a given type is closed under reduction. That is, for $\Gamma$ 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 2.4 for a discussion on Subject Reduction and Section 3.3 for a list of properties for $\lambda$HOL (among which Subject Reduction). The following is a consequence of confluence on $\mathcal{T}$ and Subject Reduction.

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

3.13. Proposition (Strong Normalization). *For any term $M$ legal in $\lambda$HOL, there are no infinite $\beta$-reduction paths starting from $M$. (Put differently: all reductions starting from a legal term terminate.)*

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

The *type inference 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 $(\lambda)$-rule. (It could also be a conclusio 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 $\Gamma$ and a term $M$, there is *at most* one rule can have as conclusion $\Gamma \vdash M : C$ (for some $C$). See Pollack [1995] and van Benthem Jutting et al. [1993] for more on syntax-directed sets of rules for type systems and their advantages. We will treat the (potential) probelm of non-syntax-directedness later when we discuss the soundnes 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 $\Gamma$ and 'false' otherwise. But, if $\Gamma$ is not a well-formed context, we want to return 'false' as well! So we have to check the well-formedness of $\Gamma$. A type inference algorithm consists of two mutually dependent recursive functions:

Type_(−), the real type inference algorithm, and the *context checking algorithm* Ok(−). The latter takes as input a context and returns 'true' if and only if the context is well-formed (and 'false' otherwise).

3.14. DEFINITION. We define the algorithms Ok(−), taking a context and returning 'true' or 'false', and Type_(−) taking a context and a term and returning a term or 'false' as follows. Here $x$ denotes a variable.

$$
\begin{array}{rcl}
\mathrm{Ok}(<>) & = & \text{'true' (the empty context),} \\
\mathrm{Ok}(\Gamma, x{:}A) & = & \mathrm{Type}_\Gamma(A) \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}, \\
\mathrm{Type}_\Gamma(x) & = & \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{and}\ x{:}A \in \Gamma\ \mathsf{then}\ A\ \mathsf{else}\ \text{'false'}, \\
\mathrm{Type}_\Gamma(\mathsf{Prop}) & = & \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{then}\ \mathsf{Type}\ \mathsf{else}\ \text{'false'}, \\
\mathrm{Type}_\Gamma(\mathsf{Type}) & = & \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{then}\ \mathsf{Type}'\ \mathsf{else}\ \text{'false'}, \\
\mathrm{Type}_\Gamma(\mathsf{Type}') & = & \text{'false'}, \\
\mathrm{Type}_\Gamma(MN) & = & \mathsf{if}\ \mathrm{Type}_\Gamma(M) = C\ \mathsf{and}\ \mathrm{Type}_\Gamma(N) = D \\
 & & \quad \mathsf{then} \quad \mathsf{if}\ C \twoheadrightarrow_\beta \Pi x{:}A.B\ \mathsf{and}\ A =_\beta D \\
 & & \qquad\qquad \mathsf{then}\ B[N/x]\ \mathsf{else}\ \text{'false'} \\
 & & \quad \mathsf{else} \quad \text{'false'}, \\
\mathrm{Type}_\Gamma(\lambda x{:}A.M) & = & \mathsf{if}\ \mathrm{Type}_{\Gamma, x:A}(M) = B \\
 & & \quad \mathsf{then} \quad \mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\} \\
 & & \qquad\qquad \mathsf{then}\ \Pi x{:}A.B\ \mathsf{else}\ \text{'false'} \\
 & & \quad \mathsf{else}\ \text{'false'}, \\
\mathrm{Type}_\Gamma(\Pi x{:}A.B) & = & \mathsf{if}\ \mathrm{Type}_\Gamma(A) = s_1\ \mathsf{and}\ \mathrm{Type}_{\Gamma, x:A}(B) = s_2\ \mathsf{and}\ s_1, s_2 \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\} \\
 & & \quad \mathsf{then} \quad \mathsf{if}\ (s_1, s_2) \in \{\ (\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})\ \} \\
 & & \qquad\qquad \mathsf{then}\ s_2\ \mathsf{else}\ \text{'false'} \\
 & & \quad \mathsf{else}\ \text{'false'},
\end{array}
$$

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

3.15. DEFINITION. The type inference algorithm Type_(−) is *sound* if for all $\Gamma$ and $M$,

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

The type inference algorithm Type_(−) is *complete* if for all $\Gamma$, $M$ and $A$,

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

Note that completeness of Type_(−) implies that if $\mathrm{Type}_\Gamma(M) =$ 'false', then $M$ is not typable in $\Gamma$. 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 λHOL. Without uniqueness of types, we would have to let Type_(−) generate a *set of possible types*, for otherwise it could happen that a valid type $A$ for $M$ in $\Gamma$ is not computed (upto $=_\beta$) by $\mathrm{Type}_\Gamma(M)$.

Besides soundness and completeness, we want to know that 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.

3.16. Proposition (Soundness of Type_(−)). *The* type inference algorithm Type_(−) *and the* context checking algorithm *are sound, i.e. if* $\mathrm{Type}_\Gamma(M) = A$, *then* $\Gamma \vdash M : A$ *and if* $\mathrm{Ok}(\Gamma) =$ *'true', then* $\Gamma$ *is well-formed.*

The proof of soundness of Type_(−) and 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 $\mathrm{Type}_\Gamma(MN)$, where one has to use the Subject Reduction property and Confluence.

The termination of Type_(−) and Ok(−) should also be proved simultaneously, by devising a measure that decreases with every recursive call. We define the measure m for a context $\Gamma$ or a pair of a context $\Gamma$ 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 Type_(−) or Ok(−), except for the case of $\mathrm{Type}_\Gamma(\lambda x{:}A.M)$, 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 $(\lambda)$-rule, where we have to verify whether $\Pi x{:}A.B$ is a legal type. This is a situation encountered very generally in type inference 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 $\lambda$HOL, there is a rather easy way out: we can replace the side-condition $\Gamma \vdash \Pi x{:}A.B : s$ in the $(\lambda)$-rule by an equivalent but simpler one.

3.17. 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.*

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

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

3.18. Definition. The new *type inference algorithm* Type_(−) and the *context checking algorithm* Ok(−) are defined by replacing in the case $\mathrm{Type}_\Gamma(\lambda x{:}A.B)$ the part
$\mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$ by

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

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 smalles measure. We remark that, this slight variation of the type inference algorithm is still sound.

To establish termination, we now only have to verify that all side conditions are decidable. Here the only work is in the application in case: in computing $\mathrm{Type}_\Gamma(MN)$, we have to check a $\beta$-equality and we have to check whether a term reduces to a $\Pi$-type. In general, checking $\beta$-equality on pseudoterms is *not decidable* beacuse we have the full expressive power of the untyped $\lambda$-calculus. However, due to the soundness of the algorithm (Proposition 3.16), we know that the intermediate results in the computation of $\mathrm{Type}_\Gamma(MN)$, $C$ and $D$, are typable terms. Now, $\beta$-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).

3.19. 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 inference algorithm.

3.20. Definition. The *modified derivation rules* of $\lambda$HOL are to derive two forms of judgement: $\Gamma \vdash^{\mathrm{tc}} M : A$ and $\Gamma \vdash^{\mathrm{tc}} \mathsf{ok}$. They are given by the original rules of $\lambda$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

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

We state the following properties for the modified derivation rules.

3.21. 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* $\mathrm{Ok}(-)$

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

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 $\lambda$ 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 generalisation of the $\lambda$-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 generalisation 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 $\Pi$-types can be varied. (In $\lambda\mathsf{HOL}$: $(\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})$.)

3.22. 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 lambda calculus

with the following deduction rules.

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

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 *pseudoterms* $\mathcal{T}$ defined by

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

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

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 $\lambda$HOL this amounts to the following.

| $\lambda$HOL | |
|---|---|
| $\mathcal{S}$ | Prop, Type, Type$'$ |
| $\mathcal{A}$ | Prop : Type, Type : Type$'$ |
| $\mathcal{R}$ | (Prop, Prop), (Type, Type), (Type, 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 $\lambda$PRED, representing first order predicate logic, is defined as follows.

| $\lambda$PRED | |
|---|---|
| $\mathcal{S}$ | Set, Type$^s$, Prop, Type |
| $\mathcal{A}$ | Set : Type$^s$, Prop : Type |
| $\mathcal{R}$ | (Set, Set), (Set, Type), (Prop, Prop), (Set, Prop) |

We briefly explain the rules. The rule (Prop, Prop) is the usual for forming the implication. With (Set, 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 (Set, 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 (Set, 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 $\lambda$PRED 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 $\lambda$PRED, 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. $\lambda$PRED without higher order functions). See Barendregt [1992] for details.

To regain all connectives, $\lambda$PRED 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 ($\lambda$PRED$\omega$) here and compare it with $\lambda$HOL.

| $\lambda$PRED$\omega$ | |
|---|---|
| $\mathcal{S}$ | $\mathsf{Set}, \mathsf{Type}^s, \mathsf{Prop}, \mathsf{Type}$ |
| $\mathcal{A}$ | $\mathsf{Set} : \mathsf{Type}^s, \mathsf{Prop} : \mathsf{Type}$ |
| $\mathcal{R}$ | $(\mathsf{Set}, \mathsf{Set}), (\mathsf{Set}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Set}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})$ |

The rule (Type, Prop) allows quantification over domains of type Type, which are $A \to \mathsf{Prop}$, $A \to A \to \mathsf{Prop}$ etcetera (with $A : \mathsf{Set}$). The addition of (Type, 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 $\lambda$-calculi that can be described as a PTS are simple typed $\lambda$-calculus, polymorphic typed $\lambda$-calculus (also known as system F Girard [1972], Girard et al. [1989]), higher order typed $\lambda$-calculus (also known as F$\omega$ 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 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$\lambda$calculi*, see Barendregt [1992] for further details. In view of higher order predicate logic, one can understand the Calculus of Constructions 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 $\lambda$PRED$\omega$ 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 $\lambda$HOL, we define a context $\Gamma$ to be *well-formed* if $\Gamma \vdash M : A$ for some $M$ and $A$.

Two basic properties are Thinning, saying that typing judgements remain valid in an extended context, and Substitution, saying that typing judgements remain valid if we substitute legal terms.

3.23. Proposition (Thinning). *For $\Gamma$ a context, $\Gamma'$ 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 $\Gamma$, also occur in $\Gamma'$.*

3.24. 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 Reducsction, saying that typing is closed under reduction.

3.25. 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 judgement (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 $\Gamma_1 \vdash N : B$.)

3.26. Proposition (Subject Reduction). *For $\Gamma$ 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.

3.27. 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.

3.28. PROPOSITION (Uniqueness of Types). *This property holds for*functional *PTSs only. For* $\Gamma$ *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.

3.29. 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 pseudoterms and contexts. Moreover, this mapping preseves reduction in a faithful way: $M \to_\beta N$ iff $f(M) \to_\beta f(N)$. We have the following property.

3.30. 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.

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

The proof is rather difficult and can be found in Geuvers and Nederhof [1991], Coquand and Gallier [1990], Berardi [1990].

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.)

3.32. COROLLARY. *The following PTSs are all Strongly Normalizing. All subsystems of CC;* $\lambda$PRED*;* $\lambda$PRED$\omega$*.*

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 the terminal object in the category of PTSs with PTS-morphisms as arrows.

As a matter of fact, we now have two formalisations of higher order predicate logic as a PTS: $\lambda$HOL and $\lambda$PRED$\omega$. We employ the notion of PTS-morphism to see that they are equivalent. From $\lambda$PRED$\omega$ to $\lambda$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

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

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

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

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

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

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, $\lambda$PRED$\omega$ and $\lambda$HOL are equivalent systems. This equivalence implies that the system $\lambda$HOL is Strongly Normalizing as well.

### 3.5. Extensions to Pure Type Systems

Several features are not present in PTSs. For example, it is possible to define data types (in a polymoprphic sort, e.g. Prop in $\lambda$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' have been added as an extra feature. The way we present them below, they were first defined by Coquand and Mohring 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, Lego and ALF.

Another feature that we will discuss is the notion of product and (strong) $\Sigma$-type. A $\Sigma$-type is a 'dependent product type' and therefore a generalisation of product type in the same way that a $\Pi$-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 $\Sigma$-type, we have projections to take a pair apart: if $t : \Sigma x{:}A.B$, then $\pi_1 t : A$ and $\pi_2 t : B[\pi_1 t/x]$. $\Sigma$-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 $\Sigma$-types cannot.

### 3.6. Products and Sums

We have already seen how to define conjunction and disjunction in $\lambda$HOL. These are very close to product-types and sum-types. Here is what one would want for a product-type.

$$\text{(products)} \quad \frac{\Gamma \vdash A_1 : s \;\; \Gamma \vdash A_2 : s}{\Gamma \vdash A_1 \times A_2 : s}$$

$$\text{(projection)} \quad \frac{\Gamma \vdash p : A_1 \times A_2}{\Gamma \vdash \pi_i p : A_i}$$

$$\text{(pairing)} \quad \frac{\Gamma \vdash t_1 : A_1 \;\; \Gamma \vdash t_2 : A_2}{\Gamma \vdash \langle t_1, t_2 \rangle : A_1 \times A_2}$$

with the computation rule

$$\pi_i \langle t_1, t_2 \rangle \to t_i.$$

In presence of polymorphism, these constructions are all definable. For example in $\lambda$HOL we have products in the sort $\mathsf{Prop}$. Let $A_1, A_2 : \mathsf{Prop}$ and define

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

For sum-types one would like to have the following.

$$\text{(sums)} \quad \frac{\Gamma \vdash A_1 : s \;\; \Gamma \vdash A_2 : s}{\Gamma \vdash A_1 + A_2 : s}$$

$$\text{(injection)} \quad \frac{\Gamma \vdash p : A_i}{\Gamma \vdash \mathsf{in}_i \; p : A_1 + A_2}$$

$$\text{(case)} \quad \frac{\Gamma \vdash f_1 : A_1 \to C \;\; \Gamma \vdash f_2 : A_2 \to C}{\Gamma \vdash \mathsf{case}(f_1, f_2) : (A_1 + A_2) \to C}$$

with the computation rule

$$\mathsf{case}(f_1, f_2)(\mathsf{in}_i \; p) \to f_i p.$$

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

$$\begin{aligned}
A_1 + A_2 &:= \Pi\alpha{:}\mathsf{Prop}.(A_1 \to \alpha) \to (A_2 \to \alpha) \to \alpha, \\
\mathsf{in}_1 &:= \lambda p{:}A_1.\lambda\alpha{:}\mathsf{Prop}.\lambda h_1{:}(A_1 \to \alpha).\lambda h_2{:}(A_2 \to \alpha).h_1 p, \\
\mathsf{in}_2 &:= \lambda p{:}A_2.\lambda\alpha{:}\mathsf{Prop}.\lambda h_1{:}(A_1 \to \alpha).\lambda h_2{:}(A_2 \to \alpha).h_2 p, \\
\mathsf{case}(f_1, f_2) &:= \lambda x{:}(A_1 + A_2).xCf_1f_2.
\end{aligned}$$

### 3.7. $\Sigma$-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 $\lambda$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 $\Sigma$-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: $\pi_1(\pi_2(\pi_2(\pi_2 t))) : \Pi x,y,z{:}A.\pi_1 t(\pi_1 t x y)z = \pi_1 t x(\pi_1 t y z)$, 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 $p_4$ proofterms 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 $\Sigma$-types in $\lambda$HOL are as follows.

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

$$(\langle -,- \rangle)\quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash p : \varphi[a/x] \quad \Gamma \vdash \Sigma x{:}A.\varphi : \mathsf{Type}}{\Gamma \vdash \langle a, p \rangle : \Sigma x{:}A.\varphi}$$

$$(\pi_1)\quad \frac{\Gamma \vdash t : \Sigma x{:}A.\varphi}{\Gamma \vdash \pi_1 t : A}$$

$$(\pi_2)\quad \frac{\Gamma \vdash t : \Sigma x{:}A.\varphi}{\Gamma \vdash \pi_2 t : \varphi[\pi_1 t/x]}$$

The reduction rules associated to the pairing and projection construction are the usual ones: $\pi_1\langle a, p\rangle \to a$ and $\pi_2\langle a, p\rangle \to p$.

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

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

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, we can not allow $\Sigma x{:}\mathsf{Type}.B : \mathsf{Type}$. Hence 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 $\Sigma$-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 $\varphi$ 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 $\Sigma$-types for the existential quantifier requires that $\Sigma x{:}A.\varphi : \mathsf{Prop}$ (not of type $\mathsf{Type}$) in $\lambda\mathsf{HOL}$. In order to achieve this we could modify the $\Sigma$-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 $\Sigma$-type that represents the existential quantifier, while remaining consistent, but only for $A : \mathsf{Set}$. On the other hand, one may wonder whether a $\Sigma$-type is the correct formaliztion 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 COQ [1999] and that were first defined in Coquand and Paulin-Mohring [1990].

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

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

to obtain the following rules.

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

$$(\mathrm{elim}_2)\quad \frac{\Gamma \vdash P : \mathsf{nat}{\to}\mathsf{Prop} \quad \Gamma \vdash f_1 : P0 \quad \Gamma \vdash f_2 : \Pi x{:}\mathsf{nat}.P\ x{\to}P(S\ x)}{\Gamma \vdash \mathsf{Rec}_{\,\mathsf{nat}} f_1 f_2 : \Pi x{:}\mathsf{nat}.P\ x}$$

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

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

It is understood that the additional $\iota$-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 $(\mathrm{elim}_1)$ 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 $d\ (S\ x) \twoheadrightarrow_{\beta\iota} S(S(dx))$. The predicate of 'being even', $\mathsf{even}(-)$, can also be defined by using $(\mathrm{elim}_1)$:

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

We obtain indeed that

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

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

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

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

$$\begin{array}{lll} \mathsf{Inductive\ \ definition} & \mathsf{List} : \mathsf{Type} := \\ & \mathsf{Nil} : \mathsf{List} \\ & \mathsf{Cons} : \mathsf{List} \to D \to \mathsf{List} \end{array}$$

with the following rules.

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

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

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

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

Of course, there is a more general pattern behind these two examples. The extension of $\lambda\mathsf{HOL}$ with inductive types is defined by adding the following scheme.

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

where the $\sigma_j^i(\mu)$ are all 'type schemes with a strictly positive occurence of $\mu$', i.e. each $\sigma_j^i(\mu)$ is of the form $A_1 \to \cdots A_n \to X$ with no occurrence of $\mu$ in the $A_k$ and either $X \equiv \mu$ or $\mu$ not in $X$. This inductive definition of $\mu$ generates the constructors 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.

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

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 $(\mathrm{elim}_1)$ rule is as follows.

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

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

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

It is an interesting exercise to define all kinds of standard functions on Tree, like the function that takes the $n$th 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 ($\mathrm{elim}_2$) rule.

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

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

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

As we do not have the (Prop, Type) rule in $\lambda$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 ($\mathrm{elim}_2$) 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 $\varphi$ (respectively $\psi$) 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 $\varphi$ (respectively $\psi$).

*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 want to discuss here 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 the 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.

$$\begin{array}{rl} \mathsf{Inductive\ \ definition} & \mathsf{List} : \mathsf{Type}\to\mathsf{Type} := \\ & \mathsf{Nil} : \Pi D{:}\mathsf{Type}.(\mathsf{List} D) \\ & \mathsf{Cons} : \Pi D{:}\mathsf{Type}.(\mathsf{List} D)\to D\to(\mathsf{List} D). \end{array}$$

Which would generate the following elimination rules and reduction rule.

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

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

$$\begin{array}{rcl}\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2(\mathsf{Nil}D) & \to_\iota & f_1\\ \mathsf{Rec}_{\,\mathsf{List}} f_1 f_2(\mathsf{Cons}Dtd) & \to_\iota & f_2 td(\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 t)\end{array}$$

To be able to write down the type of the constructors Nil and Cons, we need the rule $(\mathsf{Type}', \mathsf{Type})$ in $\lambda$HOL, which makes the system inconsistent. Therefore, this extension works much better in a system like $\lambda$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 defined using one dependent type: the $\Sigma$-type. Let $B : \mathsf{Type}$ and $Q : A\to\mathsf{Prop}$ and suppose we have added the rule $(Prop, \mathsf{Type})$ to our system.

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

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

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

The $\iota$-reduction rule is

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

Now, taking in $(\mathrm{elim}_1)$ $B$ 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 in $(\mathrm{elim}_2)$ $\lambda x{:}\mu.Q(\pi_1 x)$ for $P$ 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 $\mu$ together with In (as pairing constructor) and $\pi_1$ and $\pi_2$ (as projections) represents the $\Sigma$-type.

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

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

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 $\iota$-reduction rule is

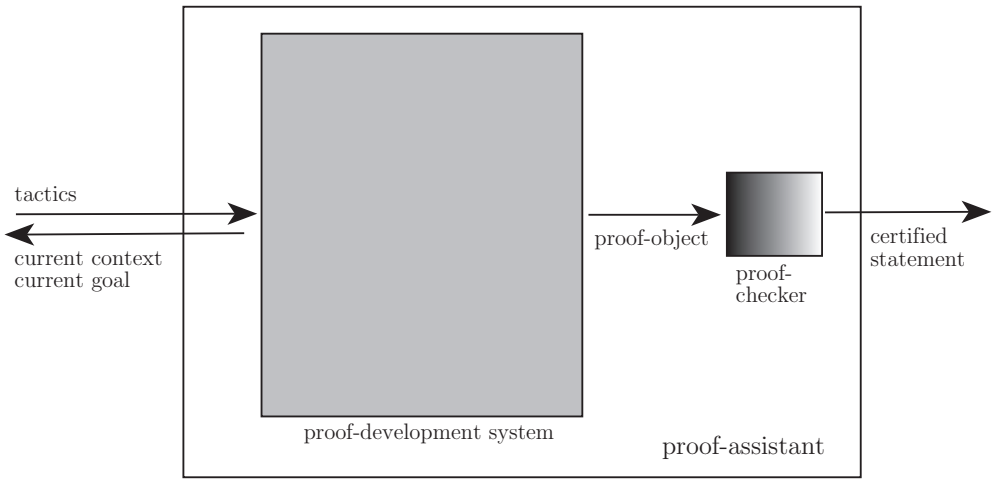
$$\mathsf{Rec}_{\,\mathsf{Eq}} xx f_1(\mathsf{Refl}x) \quad \to_\iota \quad f_1 x$$

## 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 interect 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 formalised theories.

### 4.1. Tactics

In §2.1 and §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—so-called *proof-objects*—is relatively involved. Therefore tools have been developed—so-called *proof-assistents*—that make this task more easy. A proof assistent consists of a proof-checker and an interactive proof-development system.




A proof-assistent and its components

In the proof-development system one chooses a context and formulates 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 how the proof-object looks like. For example, if one wants to prove

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

in context $\Gamma$, then there is a tactic ('intros') that changes the context by picking a fresh ('arbitrary') $x{:}A$ and assumes $Px$, the goal now becoming $Qx$. More precisely, the interaction takes place as follows.

| Step | User | | System | |
|---|---|---|---|---|
| u:1 | Context: | $\Gamma$ | | |
| | Goal: | $\forall x{:}A.(Px \Rightarrow Qx)$ | | |
| s:1 | | | Current context: | $\Gamma$ |
| | | | | ========== |
| | | | Current goal: | $\forall x{:}A.(Px \Rightarrow Qx)$ |
| u:2 | Intros | | | |
| s:2 | | | Current context: | $\Gamma, x{:}A, p{:}Px$ |
| | | | | ======= |
| | | | Current goal: | $Qx$ |
| u:3 | ..... | | | |

The '$p{:}Px$' means that we assume that $p$ is a proof of $Px$ (in order to construct a proof $q(p)$ of $Qx$, thereby providing a proof of $(Px \Rightarrow Qx)$, namely $\lambda p{:}Px.q(p)$, and hence of $\forall x{:}A.(Px \Rightarrow Qx)$, namely $\lambda x{:}A\lambda p{:}Px.q(p)$.

Another tactic is the command 'Apply'. If the current context is $\Gamma, a{:}A, p{:}\forall x{:}A.(Rx \Rightarrow Sx)$ and the current goal is $Sa$, then the command 'Apply p' will change the current goal into $Pa$.

| Step | User | System | |
|---|---|---|---|
| s:k | | Current context: | $\Gamma, a{:}A, p{:}\forall x{:}A.(Rx \Rightarrow Sx)$ |
| | | | ========== |
| | | Current goal: | $Sa$ |
| u:k+1 | Apply p | | |
| s:k+1 | | Current context: | $\Gamma, a{:}A, p{:}\forall x{:}A.(Rx \Rightarrow Sx)$ |
| | | | ========== |
| | | Current goal: | $Ra$ |

The idea behind this tactic is that if the original goal was to prove $Sa$, then it is sufficient to prove in the same context $Ra$. Indeed $p\,a$ will be a proof of $Ra \Rightarrow Sa$ and if we find a proof $r$ for $Ra$, then $p\,a\,r$ will be proof of $Sa$.

Another essential tactic is concerned with inductive types. For example the type nat is defined by

```
Inductive nat := O | S: nat→ nat
```

This type comes together with an induction principle

```
nat_ind :  ∀P:(nat→ Prop).[P(O)→ [∀n:nat.P(n)→ P(n+1)] → (∀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 $\iota$-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 the current goal becomes one of the assumptions in the current context, or an earlier proved result (from the same context. Then the system is satisfied and produces something like

System
QED!

One then can save the proof-object, for example using some mnemonic name:

User System
Save proposition_1.
proposition_1 saved.

Then later for $a{:}A$ one can use 'proposition_1 $a$' as a proof of $Pa \Rightarrow Qa$.

The set of tactics and its implementation may be a rather large user interface. For example, several techniques of automated deduction may be incorporated as tactics. 'even if the resulting proof-development system as subunit in general will be large, the reliability of the proof-assistent 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 $\Gamma$ 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.[3]

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

$$A \wedge B \;=\; \texttt{Inductive}\;\; \texttt{inl} : A{\rightarrow}A \vee B \mid \texttt{inr} : A{\rightarrow}A \vee B.$$

[3]From this statement Euclid's theorem that there are infinitely many primes is not far removed.: consider a primefactor of $n! + 1$ and show that it is necessarily $> n$. Thus one obtains $\forall n \exists p > n.\texttt{prime}\;\; p$. A slightly different formaliz is possible in type theory. The statement

$$A_n \equiv \neg\exists p_1, \ldots, p_n \forall x\, [\texttt{prime}\;\; x \;\Leftrightarrow\; x = p_1 \vee \ldots \vee x = p_n].$$

can be formally defined with $n$ as parameter and one can prove formally

$$\forall n{:}\mathbb{N}.A_n.$$

This is impossible in Peano Arithmetic, not so much because of its first-order aspect, but because there is no built in tool for generating and proving formulas like $A_n$.

This means that $A \vee B$ comes together with maps

$$\begin{array}{lcl} \texttt{inl} & : & A \to A \vee B; \\ \texttt{inr} & : & B \to A \vee B. \end{array}$$

We also need a map corresponding to the elimination principle for disjunction (for example to prove that $A \vee B \to B \vee A$):

$$\texttt{or.el} \;:\; (A \to C) \to (B \to C) \to C.$$

This map can be derived from the postulated induction principle for $A \vee B$

$$\begin{array}{lcl} \texttt{or.ind} & \equiv & \lambda A, B, P{:}\texttt{Prop} \lambda f{:}(A \to P)\; g{:}(B \to P)\; o{:}(A \vee B) \\ & & \texttt{Cases}\;\; o \;\; \texttt{of} \\ & & \quad (\texttt{inl}\; x) \;\; \Rightarrow \;\; (f x) \\ & & \quad \mid (\texttt{inr}\; x) \;\; \Rightarrow \;\; (g x) \\ & & \texttt{end} \end{array}$$

having type $\forall A, B, C{:}\texttt{Prop}.(A \to C) \to (B \to C) \to (A \vee B) \to C$.

It is also possible to define the operations $\wedge$, $\vee$, $\texttt{FF}$, $\tilde{}$ and $\exists$ without inductive types, but using higher order quantification, 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

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

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.

Another point of interest is that inductive types are founded in such a way that they become freely generated by their constructors. This has for example as consequence that if the type of natural numbers is defined by

$$\texttt{nat} = \texttt{Inductive zero} \mid \texttt{suc nat} : \texttt{nat},$$

then one can prove

$$\begin{array}{l} \forall n{:}\texttt{nat.suc}\; n \neq \texttt{zero}; \\ \forall n, m{:}\texttt{nat.suc}\; n = \texttt{suc}\; m \;\Rightarrow\; n = m. \end{array}$$

Thus we see that within typetheory with inductive types, Peano arithmetic can be formalized, without assuming additional axioms or rules. To quote Randy Pollack: "Type thery with inductive types is intuitinionistic: 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.[4] After the introductiontion 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[5] is used. The notation in Coq for $\Pi x{:}A.B$ is `(x:A)B` and for $\lambda x{:}A.b$ it is `[x:A].b`. Text written between `(* ... *)` parentheses serves as a comment.

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

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

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

(* Propositional connectives defined inductively. *)

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

Inductive or [A:Prop; B:Prop] :  Prop
                              := or_introl : A->(or A B)
                              |  or_intror : B->(or 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 parsing macros (not shown) are introduced that
   allow to make the following notations

  ~A := (not A)
A/\B := (and A B)
A\/B := (or A B)
A<-> := (iff A B)

*)

(* Introduction and elimination rules. *)

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:
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))
*)

Lemma  contra  : (a,b:Prop)(a->b)->~b->~a.
Lemma     dn_i : (a:Prop)a->~~a.


(* Predicate logic. *)

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

(*  A macro (not shown) is defined that allows one to state
"(EX x:A|(P x))"
for
"ex A [x:A](P x)"
as we are used to. *)

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).

Lemma contra_c : (a,b:Prop)(~b->~a)->(a->b).
```

```
Lemma contra_f : (a,b:Prop)(a->b)<->(~b->~a).
Lemma  non_and : (a,b:Prop)~(a/\b)->~a\/~b.
Lemma     or_c : (a,b:Prop)a\/b<->(~(~a/\~b)).
Lemma  non_arr : (a,b:Prop)~(a->b)->a/\~b.
Lemma    arr_c : (a,b:Prop)(a->b)<->(~a\/b).


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.
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 macro (not shown) is introduced in order to use the abbreviation
x=y for (eq nnat x y). *)

Lemma   sym_eq : (A:Set)(x,y:A)(eq A x y)->(eq A y x).
Lemma     leib : (A:Set)(P:A->Prop)(x,y:A)(eq 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 := zero : nat | suc : nat->nat.

Definition   one : nat := (suc zero).
Definition   two : nat := (suc one).
```

```
Definition Is_suc := [n:nat]Cases n of
                       zero   => FF
                   | (suc p) => TT
                  end.

Lemma no_conf : (n:nat)~(zero= (suc n)).

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

Definition lthan := [n,m:nat](leseq (suc 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 zero).
Lemma   suc_leseq : (n,m:nat)(leseq (suc n)(suc m))->(leseq n m).
Lemma        lt01 : (x:nat)(x=zero\/x=one\/(lthan one x)).
Lemma      n0n1lt : (n:nat)(~(n=zero)->~(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
                         zero    => m
                      |  (suc p) => (suc (plus p m))
                       end.

Lemma  plus_altsuc : (n,m:nat)(plus n (suc m))=(suc(plus n m)).
Lemma plus_altzero : (n:nat) (plus n zero)=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
                          zero    => zero
                       |  (suc 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 zero)=zero.
```

```
Lemma  timesaltsuc : (n,m:nat)(times n (suc 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).

Lemma  divtrans : (n,m,k:nat)(div n m)->(div m k)->(div n k).
Lemma propdless : (n,d:nat)(propdiv d n)->(lthan d n).
Lemma  dnotzero : (d,n:nat)(lthan zero n) ->(div d n)->~(d=zero).

Definition HPD (* has prime divisor*) :  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 ways how to formalise 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) $\texttt{Prime}(61)$;
(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.

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

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[6].

A way to handle (1) is to use the Poincaré principle extended to the reduction relation $\twoheadrightarrow_\iota$ for primitive recursion on the natural numbers. Operations like $f(n) = [\sqrt{n}\,]$ are primitive recursive and hence are lambda definable (using $\twoheadrightarrow_{\beta\iota}$) by $\mathsf{Rec}_{\,\mathsf{nat}}$ introduced in §3.8. Then, writing $\ulcorner 0 \urcorner = \mathtt{zero}, \ulcorner 1 \urcorner = \mathtt{succ\ zero}$, ... , 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 lambda defining term $K_{\mathtt{Prime}}$ for the characteristic function of the predicate $\mathtt{Prime}$. This term should satisfy the following statement

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

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 lambda 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

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

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

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

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

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

[6]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 so by a mathematician.

now since

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

we have by the Poincaré principle

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

Bureaucratic details how to treat free variables under $\mathsf{E}$ are omitted.

The use of inductive types like $\mathsf{Nat}$ 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. Fo example for polynomial expressions $p$ one has for (formal) differentiation and simplification the following.

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

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 $\iota$-reduction these computations can be captured as follows.

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

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

In Oostdijk [1996] 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 $K_P$. The resulting computations for $P = \mathtt{Prime}$ 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 Elbers [1996], a more efficient ad hoc lambda definition of the characteristic function of $\mathtt{Prime}$ is given, using Fermat's small theorem about primality. Also the required proof obligation has been given. So the statements in the beginning of this subsection can be obtained by computations.

Another use of the two level approach 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 reliable statements (within a given mathematical context). These statements 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 computable; on the other hand the problem whether a putative theorem is indeed a theorem is undecidable. Having said this, it is neverthelss 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, Boyer [1993]. A pure proof-checker, to which one has to present an entire fully formalised 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). 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 for classical predicate logic with equality, see Bibel and Schmitt [1998] the Boyer-Moore theorem prover Nqthm based upon primitive recursive arithmetic, see Boyer and Moore [1997], and the geometry theorem prover of Wu that was already mentioned, see Wu [1994].

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-assistents consists of a proof-development system together with a proof-checker.

### 5.1. Comparing proof-assistants

We will discuss several proof-assistants. All systems work with *proof scripts* that are a list of *tactics* needed to make the proof-assistant to accept a putative proof. 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 does generate and store 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 doubtfull whether a certain statement is valid, can download a proof-object via the internet and locally verify it using a 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 wich a trustworthy check can be performed. The second class consists of proof-assisants 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 userfriendlyness.

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. The logical power of this formal system has been described in [Aczel]. 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.

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

[Here one assumes to have defined $\text{Reflexive}(R) \equiv \forall x.R\ x\ x$, $\mathsf{I} \equiv \lambda x.x$ 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 normalising.

### Nuprl, Hol and Isabelle

The system Nuprl does have proof-objects, but a judgement

$$\vdash p : A,$$

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, but also in the following sense.

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

For this reason Nuprl is said to be based on an *extensional* type system. This implies that type checking $p : A?$ is no longer decidable and therefore proofs cannot be checked. However, there are 'second order' proof-objects $d$ that can establish that $p : A$. In fact, the $d$ takes into account that there was a $q : t = s$. So these $d$ serve as non-standard proof-objects.

The proof-assistant Hol is based on Church's simple type theory (being based on classical logic). This is a classical system with Higher Order Logic. That Hol does use 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 $\vdash \mathtt{fac}\ \mathbf{c}_n = \mathbf{c}_{n!}$. Therefore the design decision was made that proof-objects are not stored, only the proof-scripts. Even if a proof of $\mathtt{fac}\ \mathbf{c}_n = \mathbf{c}_{n!}$ may be long, it is possible to give it a a short description in the proof-script.

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 stated 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 formalising more easy, but one cannot use proof-objects for example to see details of the proof or for program extraction, see Paulin-Mohring and Werner [1993].

### Mizar, ACL2, PVS

Mizar is based on a form of set theory (Tarski-Grothendieck) that is ZFC extended with an axiom expressing the existence of a large cardinal. 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 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 is an extension of the theorem prover of Boyer-Moore. It is based on classical primitive recursive arithmetic and it is used in industry.

PVS 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 Lega are to the left, indicating reliability (Agda given the first place because it has only predicative logic; Coq coming second, since only strongly normalising 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 looses. Of course eventually proof-assistants should be developed that are both reliable and user-friendly. The judgements are based on some intuition and should not be taken too seriously.

| assistant | p.o. | reliability | PP | logic | dep.t. | ind.t | autom. | #users |
|---|---|---|---|---|---|---|---|---|
| Agda | yes | +++ | $\beta\delta\iota R_1$ | int.; pred. | yes | yes | none | (+) |
| 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 | no | ++ | ++ |
| Isabelle | n.s. | ++ | none | t.b.s. | no | no | ++ | ++ |
| Mizar | none | + | none | cl. | no | no | + | ++ |
| ACL2 | none | ? | none | cl. | no | no | +++ | +++ |
| PVS | none | − | none | cl. | no | no | +++ | +++ |

Legenda

| | |
|---|---|
| p.o. | proof-objects; |
| n.s. | non-standard; |
| 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; |
| t.b.s. | to be specified by the user; |
| $R_1$ | arbitrary notion of reduction; |
| $R_2$ | arbitrary strongly normalising notion of reduction; |
| $R_3$ | arbitrary provable equality. |

Many (often smaller) proof-assistants we have not mentioned. For a(n incomplete) but extended survey see `<www.cs.kun.nl/~freek/digimath>`.

### 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. 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 et al. [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 reliablity. In this appraoch 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 $\iota$-conversion has as consequence that primitive recursive computations within a proof come without proof obligations. The reduction relation for which the Poincaré principle is assumed is called the *computational strength* of the system. In fact pragmatic proof-assistants can be viewed as those systems, for which a strong computational power is 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 type system.

Rather than making a choice for the computational and logical strength one may think of a "universal[7]" system in which these two can be set according to the taste and application area of the user. One may think that the two parameters are determined by positions of a joystick. It is hoped (and expected) that it is possible to construct such a general proof-assistant that is sufficiently efficient. Also there is a considerable foundational interest in the enterprise to construct user-friendly proof-assistants. One has to realise what steps are obvious to the mathematician and provide suitable tools. Another point of interest is that a foundational *status* of a mathematical result may be defined as collection of positions of the joystick for which the theorem can be proved.

It is a (possibly long term) goal of the second approach to make the formalisation 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 formalising 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 formalisation has been reached not only spin-off in system design, but also in the development of mathematics is expected. First of all there will emerge a different system of refereeing. People will only submit papers that are correct. The referee can focus on the judgment wheter 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.

Also it is to be expected that eventually proof-assistants will help the working mathematician. Arbitray 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

[7]Of course there cannot be a universal proof-assistant, by 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 intuition and machines take over part of the craftmanship.

Next to these theoretical aspects, there is a potentially 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 from the proof. See Mohring [1986], Paulin-Mohring and Werner [1993] and Parent [1995].

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

## Acknowledgement

The European Workgroup Types, sponsored by grant xxxxxx, has provided contact with other research groups. In particular we thank the following people who, in various ways, contributed to the content of this chapter: the anonymous referees, Ana Bove, Thierry Coquand, Wolfgang Narachewski and Randy Pollack.

## References

Abramsky, S., Dov M. Gabbay and T.S.E. Maibaum (eds.) [1992]. *Handbook of Logic in Computer Science, Volume 2: Background: Computational Structures*, Oxford Science Publications.

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

Barendregt, Henk and Tobias Nipkow (eds.) [1994]. *Types for proofs and programs: international workshop TYPES '93, Nijmegen, The Netherlands, May 24–28, 1993: selected papers*, Lecture Notes in Computer Science 806, Springer-Verlag, Berlin.

Barendregt, H.P. [1992]. Lambda calculi with types, Abramsky et al. [1992], Oxford University Press, pp. 117–309.

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

Barthe, G. and M.H. Sörensen [n.d.]. Domain-free pure type systems, *Journal of Functional Programming*.

Barthe, G., M. Ruys and H. Barendregt [1996]. A two-level approach towards lean proof-checking, *Berardi and Coppo [1996]*, number 1158 in *Lecture Notes in Computer Science*, Springer Verlag, Berlin, pp. 16–35.

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

van Benthem Jutting, L.S. [1993]. Typing in pure type systems, *Information and Computation* **105**(1), pp. 30–41.

van Benthem Jutting, L.S., J. McKinna and R. Pollack [1993]. Checking algorithms for Pure Type Systems, *in:* Henk Barendregt and Tobias Nipkow (eds.), *Proceedings of the International Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands*, Lecture Notes in Computer Science 58, Springer, Berlin and New York, pp. 19–61.

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*, Dissertation, Dipartimento Matematica, Università di Torino.

Berardi, S. and M. Coppo (eds.) [1996]. *Blabla*, Vol. 1158, Springer-Verlag, Berlin.

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

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

Boyer, R.S. [1993]. Personal communication.

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

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

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

COQ [1999]. The coq proof assistant version 6.9, Internet: <pauillac.inria.fr/coq/assis-eng.html>. Paulin-Mohring, Chr. et al.

Coquand, T. [1985]. *Une théorie des constructions*, Dissertation, Université Paris VII, Thèse de troisième cycle.

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

Coquand, T. and J. Gallier [1990]. A proof of strong normalization for the theory of constructions using a kripke-like interpretation, *Technical report*, INRIA, ftp://ftp.inria.fr/INRIA/Projects/coq/types/Proceedings/book90.dvi.

Coquand, T. and G. Huet [1988]. The calculus of constructions, *Information and Computation* **76**, pp. 95–120. Available at URL :
http://pauillac.inria.fr/ coq/coq-eng.html.

Coquand, Th. and C. Paulin-Mohring [1990]. Inductively defined types, *Martin-Löf and Mints [1990]*, Lecture Notes in Computer Science 417, Springer Verlag, Berlin.

Coquand, Thierry [1996]. An algorithm for type-checking dependent types, *Science of Computer Programming* **26**(1–3), pp. 167–177.

Daalen, D.T. van [1980]. *The Language Theory of Automath*, Dissertation, Eindhoven University of Technology.

Dezani-Ciancaglini, M. and G. Plotkin (eds.) [1995]. *Second International Conference on Typed Lambda Calculi and Applications, TLCA'95*, Lecture Notes in Computer Science 902, Springer, Berlin.

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

Elbers, H. [1996]. Personal communication.

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

Geuvers, H., E. Poll and J. Zwanenburg [1999]. Safe proof checking in type theory with $y$, *in:* F. Flum and M. Rodriguez-Artalejo (eds.), *Computer Science Logic (CSL'99)*, LNCS 1683, Spinger-Verlag, pp. 439–452.

Geuvers, J.H. [1993]. *Logics and Type Systems*, Dissertation, Catholic University of Nijmegen.

Geuvers, J.H. and M.J. Nederhof [1991]. A modular proof of strong normalization for the Calculus of Constructions, *Journal of Functional Programming* **1**(2), pp. 155–189.

Girard, J.-Y. [1972]. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur, These D'Etat, Universite Paris VII.

Girard, J.-Y. (ed.) [1999]. *Typed Lambda Calculus and Applications, TLCA '99*, Lecture Notes in Computer Science 1581, Springer, Berlin.

Girard, Jean-Yves, Yves G. A. Lafont and Paul Taylor [1989]. *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press.

de Groote, Phillippe and J. Roger Hindley (eds.) [1997]. *Typed Lambda Calculus and Applications, TLCA '97*, Lecture Notes in Computer Science 1210, Springer, Berlin.

Hofmann, M. [1994]. Elimination of extensionality and quotient types in martin-löf type theory, *in:* H. Barendregt and T. Nipkow (eds.), *Proceedings of the International Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands*, Lecture Notes in Computer Science 806, Springer, Berlin and New York, pp. 166–190.

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.

JAPE [1997]. Jape - a framework for building interactive proof editors, Internet: <users.comlab.ox.ac.uk/bernard.sufrin/jape.html>. Sufrin, B. and R. Bornat.

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

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

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

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

Martin-Löf, P. and G. Mints (eds.) [1990]. *COLOG-88: International conference on computer logic*, Lecture Notes in Computer Science 417, Springer Verlag, Berlin.

Mathpert [1997]. Internet: <www.mathpert.com>. Beeson, Michael.

McCarthy, J. [1962]. Computer programs for checking mathematical proofs, *Proceedings of the Symposium in Pure Mathematics 5*, American Mathematical Society.

MIZAR [1989]. Internet: <www.mizar.org>. Architects: Andrzej Trybulec, Czeslaw Bylinski.

Mohring, Ch. [1986]. Algorithm development in the calculus of constructions, *Proceedings of the First Symposium on Logic in Computer Science, Cambridge, Mass.*, IEEE, Washington DC, pp. 84–91.

Nederpelt, R.P. [1973]. *Strong normalisation in a lambda calculus with lambda structured types*, Dissertation, Eindhoven University, Box 417, Eindhoven, The Netherlands.

Nederpelt, R.P., J.H. Geuvers and R.C. de Vrijer (eds.) [1994]. *Selected Papers on Automath*, Studies in Logic and the Foundations of Mathematics **133**, North-Holland, Amsterdam.

Nordström, B., K. Petersson and J.M. Smith [1990]. *Programming in Martin-Löf's Type Theory*, Oxford University Press.

Oostdijk, M. [1996].
*Proof by calculation*, Master's thesis, 385, Universitaire School voor Informatica, Catholic University Nijmegen.

Oostdijk, M. and H. Geuvers [2000]. Proof by computation in the coq system, *Theoretical Computer Sci.*

OpenMath [1998]. Internet: <www.nag.co.uk/projects/openmath/omsoc>.

Otter [1998]. Internet:<www.mcs.anl.gov/AR/otter>. Architect: William McCune.

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, 17–21 July 1995*, Lecture Notes in Computer Science 947, Springer-Verlag, Berlin, pp. 351–379.

Paulin-Mohring, C. [1994]. Inductive definitions in the system coq; rules and properties, *Bezem and Groote [1993]*, number 664 in *Lecture Notes in Computer Science*, Springer Verlag, Berlin, pp. 328–345.

Paulin-Mohring, Ch. and B. Werner [1993]. Synthesis of ml programs in the system coq, *Journal of Symbolic Computation* **15**, pp. 607–640.

Pollack, R. [1995]. A verified type checker, *in:* M. Dezani-Ciancaglini and G. Plotkin (eds.), *Second International Conference on Typed Lambda Calculi and Applications, TLCA'95*, number 902 in *Lecture Notes in Computer Science*, Springer, Berlin, pp. 365 – 380.

Ruess, H., N. Shankar and M.K. Srivas [1996]. Modular verification of srt division, *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA*, eds. R. Alur and T.A. Henzinger*, Lecture Notes in Computer Science 1102, Springer, pp. 123–134.

Rushby, J.M. and F. von Henke [1993]. Formal verification of algorithms for critical systems, *IEEE Transactions on Software Engineering* **19**(1), pp. 13–23.

Russell, B. [1903]. *The Principles of Mathematics*, Allen & Unwin, London.

Scott, D.S. [1970]. Constructive validity, *in:* D. Lacombe M. Laudet and M. Schuetzenberger (eds.), *Symposium on Automated Demonstration*, Lecture Notes in Mathematics 125, Springer, Berlin, pp. 237–275.

Severi, P. [1998]. Type inference for pure type systems, *Information and Computation* **143-1**, pp. 1–23.

Severi, P. and E. Poll [1994]. Pure type systems with definitions, *in:* A. Nerode and Yu.V. Matiyasevich (eds.), *Proceedings of LFCS'94 (LNCS* **813***)*, LFCS'94, St. Petersburg, Russia, Springer Verlag, New York, pp. 316–328.

Swaen, M. [1989]. *Weak and strong sum-elimination in intuitionistic type theory*, Dissertation, University of Amsterdam.

Terlouw, J. [1989]. Een nadere bewijstheoretische analyse van GSTT's, *Technical report*, Department of Computer Science, University of Nijmegen.

Wu, W. [1994]. *Mechanical Theorem Proving in Geometries*, Texts and Monographs in Symbolic Computation, Springer.