The Calculus of Constructions and Higher Order Logic

Herman Geuvers,
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1,
6525 ED Nijmegen,
The Netherlands

August 1992

Abstract

The Calculus of Constructions (CC) ([Coquand 1985]) is a typed lambda calculus for higher order intuitionistic logic: proofs of the higher order logic are interpreted as lambda terms and formulas as types. It is also the union of Girard's system F_ω ([Girard 1972]), a higher order typed lambda calculus, and a first order dependent typed lambda calculus in the style of de Bruijn's Automath ([de Bruijn 1980]) or Martin-Löf's intuitionistic theory of types ([Martin-Löf 1984]). Using the impredicative coding of data types in F_ω, the Calculus of Constructions thus becomes a higher order language for the typing of functional programs. We shall introduce and try to explain CC by exploiting especially the first point of view, by introducing a typed lambda calculus that faithfully represent higher order predicate logic (so for this system the Curry-Howard 'formulas-as-types isomorphism' is really an isomorphism.) Then we discuss some propositions that are provable in CC but not in the higher order logic, showing that the formulas-as-types embedding of higher order predicate logic into CC is not an isomorphism. It is our intention that this chapter can be read without any specialist knowledge of higher order logic or higher order typed lambda calculi.

1 Introduction

The so called Curry-Howard formulas-as-types embedding provides a formalization of the Brouwer-Heyting-Kolmogorov understanding of proofs as constructions. (See [Troelstra and Van Dalen 1988].) The first detailed description is in [Howard 1980], where also the terminology 'formulas-as-types' is first used. There it is shown how, in first order logic, types can be associated with formulas and lambda terms with proofs in such a way that there is a one-to-one correspondence between types and formulas and terms and proofs and further that cut-elimination in the logic corresponds to reduction in the term calculus. In view of the last point it would be correct to associate also Tait with

the formulas-as-types notion, as his ([Tait 1965])'discovery of the close correspondence between cut elimination and reduction of λ-terms provided half of the motivation' for [Howard 1980]. Also De Bruijn is often associated to the formulas-as-types notion, because the Automath project which was founded by De Bruijn, was the first to rigorously interpret mathematical structures and propositions as types and objects and proofs as λ-terms. So, from a wider perspective it is certainly justifiable to speak of the Curry-Howard-de Bruijn embedding (also because the earliest developments in Automath took place independent of the work of Howard.) Having said this we want to point out that there are essential differences between the two approaches. For one, because in the Automath systems the logic is *coded* into the system there is in general no reduction relation in the term calculus that corresponds to cut-elimination. Automath systems are intended to serve as a logical framework in which the user can work with any formal systems he or she desires. Application, λ-abstraction and conversion serve as tools for handling the basic mathematical manipulations like function application, function definition and substitution. Although the Calculus of Constructions can serve perfectly well as an Automath-like logical framework, from the literature about the system ([Coquand 1985], [Coquand and Huet 1988]) it clearly shows that the inventors aim at the formulas-as-types embedding in the first sense. In this paper we shall therefore look at the Curry-Howard formulas-as-types embedding of higher order predicate logic into CC. The embedding is not complete: CC proves more propositions than higher order predicate logic. This may seem quite harmful and for some purposes it is. However, we shall see that CC does not prove everything and is a conservative extension of higher order propositional logic. (These are more or less standard results by now, but we shall devote some attention to them as this text is meant to be introductory.) Further we shall discuss a recent result by Berardi([Berardi 199+]), showing that CC is still an adequate system for higher order reasoning about inductive data types, which is one of the main practical applications of the system. To understand this result, we have to devote some attention to data types and specifications in CC, a subject extensively studied in e.g. [Paulin 1989]. Finally we discuss some variants and extensions of the system.

2 Higher Order Predicate Logic as a typed lambda calculus

In the literature there are several systems of higher order predicate logic (e.g. [Church 1940], [Takeuti 1975], [Schütte 1977] and [Lambek and Scott 1986]), most of them aiming at the formalisation of higher order arithmetic. We shall not try to give an overview of all the different options, but introduce our own formalism (which of course heavily relies on the mentioned works) and pinpoint at some of the places where we essentially leave the standard paths. As usual we start by defining the domains that the logic is about in terms of the simple theory of types: There are countably many base types, one of which is a special that we denote here by Ω, to be understood as the type of proposi-

tions. The terms of the logic are the terms of the simply typed lambda calculus built from variables and some typed constants, among which are $\supset\colon \Omega\to\Omega$ and $\forall_\sigma : (\sigma\to\Omega)\to\Omega$ (for every type σ.) So the (essentially many-sorted) language doesn't start from a fixed similarity type, as is usual for first order logic, but any similarity type can be built in by using the base types and the constants. (For example for the natural numbers by starting from the base type N and the constants $z : N$ and $s : N\to N$, or for countable ordinals by adding the base type O and $z' : O$, $s' : O\to O$ and $l : (N\to O)\to O$.)
We shall now make the language and the derivation rules of our system of *higher order intuitionistic predicate logic* precise. We call the system HOPL.

Definition 2.1 *The language of* HOPL *is defined as follows.*

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

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

   where B is the set of (names of) basic domains *(in the syntax just a countable set of expressions.)*

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

   (a) *for each $\sigma \in D$, the variables $x^\sigma_1, x^\sigma_2, \ldots$ are in* TERM_σ,

   (b) *for each $\sigma \in D$, $\forall_\sigma : (\sigma\to\Omega)\to\Omega$,*

   (c) $\supset\colon \Omega\to\Omega$,

   (d) *if $M : \sigma\to\tau$ and $N : \sigma$, then $MN : \tau$,*

   (e) *if $M : \tau$ and x^σ is a variable, then $\lambda x^\sigma.M : \sigma\to\tau$.*

3. *The set of terms of* HOPL, TERM, *is defined by* $\mathsf{TERM} := \cup_{\sigma\in D}\mathsf{TERM}_\sigma$.

4. *The set of formulas of* HOPL, FORM, *is defined by* $\mathsf{FORM} := \mathsf{TERM}_\Omega$.

We adapt the well-known notions of *free* and *bound* variable, substitution, β-reduction and β-conversion to the terms of this system. If there is no ambiguity, we omit the subscript under the $\forall$. The terms $\supset \varphi\psi$ and $\forall_\sigma(\lambda x^\sigma.\varphi)$ are written as $\varphi \supset \psi$, respectively $\forall x^\sigma.\varphi$.

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

Definition 2.2 *The notion of* provability, $\Gamma \vdash \varphi$, *for Γ a finite set of formulas (terms of type* FORM*) and φ a formula, is defined inductively as follows.*

$$(axiom)\qquad \frac{}{\Gamma \vdash \varphi} \qquad if\ \varphi \in \Gamma$$

$$(\supset\text{-}introduction)\qquad \frac{\Gamma \cup \varphi \vdash \psi}{\Gamma \vdash \varphi \supset \psi}$$

$$(\supset\text{-}elimination)\qquad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash \varphi \supset \psi}{\Gamma \vdash \psi}$$

$$(\forall\text{-}introduction)\qquad \frac{\Gamma \vdash Px^\sigma}{\Gamma \vdash \forall P} \qquad if\ x^\sigma \notin FV(\Gamma)$$

$$(\forall\text{-}elimination)\qquad \frac{\Gamma \vdash \forall P}{\Gamma \vdash Pt} \qquad if\ t : \sigma$$

$$(conversion)\qquad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \psi} \qquad if\ \varphi =_\beta \psi$$

Another option (and maybe what one would expect) for the $\forall$ rules is the following.

$$(\forall\text{-introduction})\qquad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x^\sigma.\varphi} \qquad \text{if } x^\sigma \notin \mathrm{FV}(\Gamma)$$

$$(\forall\text{-elimination})\qquad \frac{\Gamma \vdash \forall x^\sigma.\varphi}{\Gamma \vdash \varphi[t/x^\sigma]} \qquad \text{if } t : \sigma$$

However, this is not convenient, because then in general $\not\vdash \forall(\lambda x^\sigma.Px^\sigma) \supset \forall P$ and $\not\vdash \forall P \supset \forall(\lambda x^\sigma.Px^\sigma)$. With our $\supset$-introduction and $\supset$-elimination rule of Definition 2.2, we even have them as derived rules:

$$\frac{\Gamma \vdash \forall P}{\Gamma \vdash \forall(\lambda x^\sigma.Px^\sigma)} \text{ and } \frac{\Gamma \vdash \forall_\sigma P}{\Gamma \vdash \forall(\lambda x^\sigma.Px^\sigma)}$$

Note that also ($\forall$-elimination$'$) and ($\forall$-introduction$'$) are derived rules.

A well-known fact about this logic is that the connectives $\&, \vee, \bot$ and $\exists$ are definable in terms of $\supset$ and $\forall$. (For $\varphi, \psi : \Omega$,

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

and the latter is the same as $\forall z^\Omega.(\forall x^\sigma.Px \supset z) \supset z$. It's not difficult to check that the intuitionistic elimination and introduction rules for these connectives are sound. (The elimination rules are even derived and if we would have formulated our syntax with a weakening rule and as axiom just

$$(\text{axiom}')\ \frac{}{\varphi \vdash \varphi}$$

then the introduction rules would all be derived too.) Further we shall use the abbreviation $\varphi \sim \psi$ for $(\varphi \supset \psi)\&(\psi \supset \varphi)$.

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

$$t =_\sigma t' :\equiv \forall P^{\sigma \to \Omega}(Pt \supset Pt'), \text{ for } t, t' : \sigma.$$

It's a standard exercise to show that this equality is symmetric.

Let's now say something about the relations between HOPL and the definitions of higher order predicate logic in [Church 1940], [Takeuti 1975], [Schütte 1977] and [Lambek and Scott 1986]. We try to restrict to the essential differences and not go into issues of notation. Most of them start from two basic domains (usually called types), i and o, letting $\sigma_1 \times \cdots \times \sigma_n \to o$ be a type if $\sigma_1, \cdots, \sigma_n$ are types, with o representing the type of formulas. (In [Takeuti 1975] the type o doesn't have an explicit name, [Lambek and Scott 1986] also have the 'singleton type' 1 as base type and a more fine grained syntax for types, allowing $\sigma_1 \times \cdots \times \sigma_n$ and $\sigma \to o$ (denoted by $P\sigma$) for $\sigma_1, \cdots, \sigma_n, \sigma$ types.) Only Church allows all arrow types, where the type $(i \to i) \to (i \to i)$, denoted by i' is used as the type of natural numbers and the types $(\sigma \to o) \to \sigma$ are types for choice operators $\iota_{(\sigma \to o) \to \sigma}$. The way we introduce the $\forall_\sigma$ (as constants of the language of type $(\sigma \to \Omega) \to \Omega$) is like in [Church 1940]. This is also the only version that formalises classical logic. It should be remarked here that Lambek and Scott do suggest the extension of the domains to include all arrow domains as a 'seemingly stronger version' of the theory. Only 'seemingly' because the extension is conservative, which can be formulated in our framework by the statement that HOPL is conservative over the version of the system with

$$D ::= B \mid \Omega \mid D \to \cdots \to D \to \Omega.$$

The conservativity can be shown syntactically by defining a mapping that sends terms of the extended system to terms of the restricted system such that derivability is preserved and the mapping is the identity on the restricted system.

The derivation rules are given in various ways (sequent calculus, natural deduction or with inference rules and axioms.) Our formulation is closest to [Lambek and Scott 1986]. Most of the systems have in addition to the derivation rules a list of axioms to include (among other things) arithmetic, extensionality and comprehension. Our system is very raw in the sense that most of these properties (except for comprehension) are not built in, but have to be added via the context. For example extensionality for functions and predicates:

$$\begin{aligned} \mathrm{EXT}_1 &:= \forall f^{\sigma \to \tau} \forall g^{\sigma \to \tau}(\forall x^\sigma fx =_\tau gx) \supset f =_{\sigma \to \tau} g, \\ \mathrm{EXT}_2 &:= \forall P^{\sigma \to \Omega} \forall Q^{\sigma \to \Omega}(\forall x^\sigma Px \sim Qx) \supset P =_{\sigma \to \Omega} Q. \end{aligned}$$

(Note that extensionality for predicates of higher arity follows from EXT_2 by EXT_1.) Comprehension states that, for φ a proposition with free variable x of type σ, there is a predicate P of type $\sigma \to \Omega$ such that

$$\forall x^\sigma(Px \sim \varphi).$$

In our system the rule of comprehension is valid by taking for the P above just $\lambda x^{\sigma}.\varphi$ and applying the rule (conversion). This is very similar to [Takeuti 1975], [Schütte 1977] and [Church 1940]. (In the latter this is not explicitly noted as a feature of the system.) In [Lambek and Scott 1986] comprehension has to be explicitly included as an axiom because, unlike the other systems, predicates and functions can not be formed by (λ-)abstraction. They form predicates as subsets (notation $\{x \in A|\varphi\}$ for φ a proposition possibly containing x) and the proposition stating that a term satisfies a predicate is denoted by set-membership ($t \in \{x \in A|\varphi\}$, so $\in$ denotes both 'of type' and 'element of'.)

The definition of HOPL above is convenient for describing subsystems of higher order predicate: First order predicate logic is obtained by restricting the set of domains to $D ::= B \mid B{\rightarrow}\Omega \mid B{\rightarrow}D$ and the set of constants of the form $\forall_{\sigma}$ to the ones for which $\sigma \in B$. (It is then also usance not to allow the construction of new functions or predicates using λ-abstraction. However, this is a conservative extension and the construction of predicates by λ-abstraction is necessary for our formulation of the $\forall$.) As another example, higher order propositional logic is obtained by removing in the definition of D the set of basic domains B.

Because the formalism for describing the sublogics of HOPL is quite uniform it provides a good framework for discussing conservativity questions. For example the conservativity of HOPL over higher order proposition logic is quite easily shown by defining a mapping on the terms of HOPL that forgets everything about the basic domains. The mapping preserves provability and is the identity on the terms of the subsystem of higher order proposition logic.

We shall now describe a typed lambda calculus that faithfully represents HOPL following the Curry-Howard isomorphism of formulas-as-types (and proofs-as-λ-terms.) It should be obvious from the definition of the system that there is a bijective mapping between the two systems. We shall not go into a detailed description of this bijection, but only give an example.

Definition 2.3 *1. The set of* types *of* λHOPL*, Type, is described by the following abstract syntax.*

$$Type ::= Prop \mid Var^{ty} \mid Type{\rightarrow}Type,$$

with Var^{ty} *a countable set of* type-variables*.*

2. The set of typable terms *is a subset of the set of* pseudoterms, T*, which is generated by the following abstract syntax.*

$$\mathsf{T} ::= Var^{te} \mid \mathsf{TT} \mid \lambda x{:}Type.\mathsf{T} \mid \mathsf{T} \supset \mathsf{T} \mid \forall Var^{te}{:}Type.\mathsf{T},$$

with Var^{te} *a countable set of* term-variables*. A term is of a certain type only under assumption of specific types for the free variables that occur in the term. That the term* t is of type A if x_i is of type A_i for $1 \leq i \leq n$*, is denoted by the judgement*

$$x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n \vdash t : A.$$

Here $x_1, \ldots, x_n$ are different term-variables and $A_1, \ldots, A_n$ are types. The rules for deriving these typing judgements are the following.

(*variable*) $\dfrac{}{\Gamma \vdash x : A}$ *if* $x{:}A$ *in* Γ

(*λ-abstraction*) $\dfrac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x{:}A.t : A{\to}B}$

(*application*) $\dfrac{\Gamma \vdash q : A{\to}B \;\; \Gamma \vdash t : A}{\Gamma \vdash qt : B}$

($\supset$) $\dfrac{\Gamma \vdash \varphi : \mathit{Prop} \;\; \Gamma \vdash \psi : \mathit{Prop}}{\Gamma \vdash \varphi \supset \psi : \mathit{Prop}}$

($\forall$) $\dfrac{\Gamma, x{:}A \vdash \varphi : \mathit{Prop}}{\Gamma \vdash \forall x{:}A.\varphi : \mathit{Prop}}$

3. *The set of proofs is a subset of the set of pseudoproofs,* P*, generated by the following abstract syntax.*

$$\mathsf{P} ::= \mathit{Var}^{pr} \mid \mathsf{PP} \mid \mathsf{PT} \mid \lambda x{:}\mathit{Type}.\mathsf{P} \mid \lambda x{:}\mathsf{T}.\mathsf{P},$$

where Var^{pr} is the set of proof-variables. The rules for generating statements of the form

$$x_1{:}A_1, \ldots, x_n{:}A_n; p_1{:}\varphi_1, \ldots, p_k{:}\varphi_k \vdash M : A,$$

where the $\vec{x}$ and $\vec{A}$ are as in 2, $p_1, \ldots p_k$ are different proof-variables and $x_1{:}A_1, \ldots, x_n{:}A_n \vdash \varphi_i : \mathit{Prop}$ (for $1 \le i \le k$), are the following.

(*axiom*) $\dfrac{}{\Gamma; \Delta \vdash p : \varphi}$ *if* $p{:}\varphi$ *in* Δ

(*$\supset$ -introduction*) $\dfrac{\Gamma; \Delta, p{:}\varphi \vdash M : \psi}{\Gamma; \Delta \vdash \lambda p{:}\varphi.M : \varphi \supset \psi}$

(*$\supset$ -elimination*) $\dfrac{\Gamma; \Delta \vdash M : \varphi \supset \psi \;\; \Gamma; \Delta \vdash N : \varphi}{\Gamma; \Delta \vdash MN : \psi}$

(*$\forall$-introduction*) $\dfrac{\Gamma, x{:}A; \Delta \vdash M : \varphi}{\Gamma; \Delta \vdash \lambda x{:}A.M : \forall x{:}A.\varphi}$ *if* $x \notin \mathit{FV}(\Delta)$,

(*$\forall$-elimination*) $\dfrac{\Gamma; \Delta \vdash M : \forall x{:}A.\varphi \;\; \Gamma \vdash t : A}{\Gamma; \Delta \vdash Mt : \varphi[t/x]}$

(*conversion*) $\dfrac{\Gamma; \Delta \vdash M : \varphi \;\; \Gamma \vdash \psi : \mathit{Prop}}{\Gamma; \Delta \vdash M : \psi}$ *if* $\varphi =_\beta \psi$.

It is not difficult to see that, apart from the differences in the treatment of constants, HOPL and λHOPL are essentially the same. (In the latter system

there are no constants, but because the types for which one can have constants in HOPL are the same as the types for which one can have variables, the variables can play the role of constants in the syntax.) To a deduction of $\varphi_1, \ldots, \varphi_n \vdash \psi$ in HOPL, we can associate a derivation of $\Gamma; p_1{:}\varphi_1, \ldots, p_n{:}\varphi_n \vdash M : \varphi$ in λHOPL, where M is a faithful coding of the deduction in HOPL and Γ assigns types to all the free term-variables in the deduction that are not bound by a $\forall$ at any later stage. (To be precise: Variables are not really 'bound by a $\forall$' in HOPL; we use this terminology to say that the variable has been removed from the proof by an application of the rule ($\forall$-introduction), as defined in 2.2.) Similarly we can associate to every derivation of a judgement $\Gamma; p_1{:}\varphi_1, \ldots, p_n{:}\varphi_n \vdash M : \varphi$ in λHOPL a derivation of $\varphi_1, \ldots, \varphi_n \vdash \varphi$ in HOPL. If we don't allow term-constants in HOPL these mappings from HOPL to λHOPL and vice versa can be made such that the composition of the two is the identity (up to the removal from Γ of those variables that do not play a role.) To stress how the context Γ is constructed from the deduction we treat two examples.

Examples 2.4 *1. Let $\Delta = \{\forall x{:}A.(Px \supset Q), \forall x{:}A.Px\}$, with P and Q variables. From the deduction*

$$\dfrac{\dfrac{\Delta \vdash \forall x{:}A.(Px \supset Q)}{\Delta \vdash Px \supset Q} \quad \dfrac{\Delta \vdash \forall x{:}A.Px}{\Delta \vdash Px}}{\Delta \vdash Q}$$

we obtain the judgement

$$P{:}A{\to}Prop, Q{:}Prop, x{:}A; p_1{:}\forall x{:}A.(Px \supset Q), p_2{:}\forall x{:}A.Px \vdash p_1 x(p_2 x) : Q.$$

Notice that the declaration of x is essential here for the construction of the proof. (λHOPL explicitly takes care of the so called free logic*, where domains are allowed to be empty.)*

2. Let $\Delta = \{\forall x{:}A.(Px \supset Qx), \forall x{:}A.Px\}$, with P and Q variables. From the deduction

$$\dfrac{\dfrac{\dfrac{\Delta \vdash \forall x{:}A.(Px \supset Q)}{\Delta \vdash Px \supset Q} \quad \dfrac{\Delta \vdash \forall x{:}A.Px}{\Delta \vdash Px}}{\Delta \vdash Q}}{\Delta \vdash \forall x{:}A.Qx}$$

we obtain the judgement

$$P{:}A{\to}Prop, Q{:}Prop; p_1{:}\forall x{:}A.(Px \supset Q), p_2{:}\forall x{:}A.Px \vdash \lambda x{:}A.p_1 x(p_2 x) : \forall x{:}A.Q.$$

Now it is not needed for the construction of the proof to declare x. By our restriction to the declaration of only those variables that are not bound by a $\forall$, the (superfluous) declaration of x is not done.

To restrict the huge number of rules and to be able to better treat the formulas-as-types embedding from higher order predicate logic to CC, we give another type-theoretic syntax for HOPL in the shape of a so called *Pure Type System*. Pure Type Systems (*PTS*'s) provide a general discription of a large class of typed lambda calculi and makes it possible to derive a lot of meta theoretic properties in a generic way. We shall not go into details about meta theory nor do we give a list of examples of systems in the form of a PTS but refer to [Barendregt 199+] and [Geuvers and Nederhof 1991]. Here we just repeat the definition and the main meta-theoretic properties.

Definition 2.5 *For $\mathcal{S}$ a set, $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$, $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is the typed lambda calculus with the following deduction rules.*

$$(sort) \quad \vdash s_1 : s_2 \qquad \text{if } (s_1, s_2) \in \mathcal{A}$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$$

$$(weak) \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\lambda) \quad \frac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$$

$$(app) \quad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

$$(conv_\beta) \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \qquad A =_\beta B$$

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 Γ. If $s_2 \equiv s_3$ in a triple $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$. The equality in the conversion rule ($conv_\beta$) is the β-equality on the set of pseudoterms T, *defined by*

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

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. The derivation rules above select the typable terms from the pseudoterms, a pseudoterm A being typable *if there is a context Γ and a pseudoterm B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. The set of typable terms of $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$is denoted by* $\mathsf{TERM}(\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R}))$.

A practical purpose for the use of the PTS framework is that many properties can be proved once and for all for the whole class of PTSs. We list the most important ones. (Proofs can be found in [Geuvers and Nederhof 1991]

or [Barendregt 199+]. In most cases the proofs are not essentially different from the proof for the Calculus of Constructions.) First, the *reduction relation* $\longrightarrow\!\!\!\!\!\rightarrow_\beta$ *is Church-Rosser on* T. (That is, if $M \longrightarrow\!\!\!\!\!\rightarrow_\beta M_1$ and $M \longrightarrow\!\!\!\!\!\rightarrow_\beta M_2$ then $M_1 \longrightarrow\!\!\!\!\!\rightarrow_\beta N$ and $M_2 \longrightarrow\!\!\!\!\!\rightarrow_\beta N$ for some $N \in \mathsf{T}$. (The proof is the same as the Church-Rosser proof of β-reduction on the untyped lambda terms in e.g. [Barendregt 1984].) One of the basic properties is the *Substitution property*, stating that if $\Gamma_1, x{:}A, \Gamma_2 \vdash M : B$ and $\Gamma_1 \vdash N : A$ then $\Gamma_1, \Gamma_2[N/x] \vdash M[N/x] : B[N/x]$. Another important property is that *Subject Reduction* holds for β. (That is, if $\Gamma \vdash M : A$ and $M \longrightarrow\!\!\!\!\!\rightarrow_\beta M'$ then $\Gamma \vdash M' : A$.) This property is sometimes called the Closure property, e.g. by the Automath community. A property which holds for all the typed lambda calculi in this paper (but not for all PTSs) is *Uniqueness of Types*: If $\Gamma \vdash M : A$, $\Gamma \vdash M : A'$, then $A =_\beta A'$. Uniqueness of types holds only for *functional* or *singly sorted* PTSs, which means that the relations $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subset (\mathcal{S} \times \mathcal{S}) \times \mathcal{S}$ of the Pure Type System are functions. The last property we want to mention here is the *Permutation property* which says that if $\Gamma \vdash M : A$, then $\Gamma' \vdash M : A$ for any Γ' which is a *sound* permutation of the declarations in Γ. ($x_1{:}A_1, \ldots, x_n{:}A_n$ is *sound* if $\mathrm{FV}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for all $i \leq n$.) This property is not a standard one in the literature but it follows immediately from the *Strengthening property*, stating that if $\Gamma_1, x{:}C, \Gamma_2 \vdash M : A$ with $x \notin \mathrm{FV}(\Gamma_2, M, A)$, then $\Gamma_1, \Gamma_2 \vdash M : A$.

The PTS framework yields a nice tool for describing a certain class of mappings between type systems, the so called PTS-morphisms. In general a *mapping from* $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ *to* $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ is a function that assigns *pseudojudgements* of $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ to judgements of $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$, a pseudojudgement being a sequent $x_1{:}A_1, \ldots, x_n{:}A_n \vdash M : B$ with $A_1, \ldots, A_n, M, B$ pseudoterms. We define a *morphism from the* PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ *to the* PTS $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ as a mapping f from $\mathcal{S}$ to $\mathcal{S}'$ that preserves axioms and rules (i.e. $s_1{:}s_2 \in \mathcal{S} \Rightarrow f(s_1){:}f(s_2) \in \mathcal{S}'$ and $(s_1, s_2, s_3) \in \mathcal{R} \Rightarrow (f(s_1), f(s_2), f(s_3)) \in \mathcal{R}'$.) A PTS-morphism from $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ to $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ immediately extends to a mapping f from the pseudoterms of $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ to the pseudoterms of $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ and hence to a mapping between the PTSs by induction on the structure of terms. This mapping preserves substitution and β-equality and also derivability, i.e.

$$\Gamma \vdash M : A \Rightarrow f(\Gamma) \vdash f(M) : f(A).$$

There are certainly many other interesting mappings between Pure Type Systems and we don't want to give the PTS-morphisms any priority. However they have some practical interest because they are easy do describe and share a lot of desirable properties. And of course the Pure Type Systems with the PTS morphisms form a category with products, coproducts and as terminal object the system with Type : Type, often referred to as $\lambda*$:

$$\begin{aligned} \mathcal{S} &= \mathrm{Type}, \\ \mathcal{A} &= \mathrm{Type} : \mathrm{Type}, \\ \mathcal{R} &= (\mathrm{Type}, \mathrm{Type}). \end{aligned}$$

The system λHOPL can now be poured in the form of a PTS as follows.

Definition 2.6 *The typed lambda calculus* λ**HOPL** *is the* PTS *with*

$$\begin{aligned}\mathcal{S} &= \mathit{Prop}, \mathit{Type}, \mathit{Type}',\\ \mathcal{A} &= \mathit{Prop} : \mathit{Type}, \mathit{Type} : \mathit{Type}',\\ \mathcal{R} &= (\mathit{Type}, \mathit{Type}),\\ &\quad (\mathit{Prop}, \mathit{Prop}), (\mathit{Type}, \mathit{Prop})\end{aligned}$$

The meaning of the components of the system should be clear from the system λHOPL. The sort Type$'$ is just there to be able to introduce variables of type Type, the basic domains of the logic. There is a heavy overloading of symbols: $\Pi x{:}A.B$ stands for logical implication ($\supset$) if A and B are both propositions (of type Prop), it stands for universal quantification ($\forall_A$) if A is a type and B a proposition (A:Type, B:Prop) and it stands for the domain $A \to B$ if both A and B are types (of type Type.) Further it is not immediately obvious that we can still see the higher order predicate logic as being built up in three stages. (First the domains, then the terms and finally the proofs.) It could well be the case that a term expression contains a proof expression or that a domain expression depends on a term. That this is not so is stated in the following proposition.

Proposition 2.7 *We work in* λ**HOPL**. *If* $\Gamma \vdash M : A$ *then* $\Gamma_D, \Gamma_T, \Gamma_P \vdash M : A$ *with*

- $\Gamma_D, \Gamma_T, \Gamma_P$ *is a sound permutation of* Γ,
- Γ_D *only contains declarations of the form* $x : \mathit{Type}$,
- Γ_T *only contains declarations of the form* $x : A$ *with* $\Gamma_D \vdash A : \mathit{Type}$,
- Γ_P *only contains declarations of the form* $x : \varphi$ *with* $\Gamma_D, \Gamma_T \vdash \varphi : \mathit{Prop}$,
- *if* $A \equiv \mathit{Type}$, *then* $\Gamma_D \vdash M : A$,
- *if* $\Gamma \vdash A : \mathit{Type}$, *then* $\Gamma_D, \Gamma_T \vdash M : A$.

The Proposition states (among other things) that the domains (terms of type Type) are just built up from domain-variables using Π, so no object- or proof-variables occur as subterms, so the domains are as in λHOPL. Further it states that the terms of the object-language are formed from the object-variables by λ-abstraction and application and (for terms of type Prop) by Π, so they don't contain proof-variables: $\Pi x{:}\varphi.\psi$ (φ, ψ : Prop) denotes $\varphi \supset \psi$, the logical implication.

Examples 2.8 *1. The first example of 2.4 becomes the following judgement in* λ**HOPL**.

$$\begin{aligned}&A{:}\mathit{Type}, P{:}A \to \mathit{Prop}, Q{:}\mathit{Prop}, x{:}A,\\ &\quad p_1{:}\Pi x{:}A.(Px \to Q), p_2{:}\Pi x{:}A.Px \quad \vdash \quad p_1 x (p_2 x) : Q.\end{aligned}$$

2. *Peano arithmetic can be done in the following context*

$$\begin{aligned}\Gamma_{PA} \;=\;& N{:}Type, 0{:}N, S{:}N{\to}N,\\ & cl{:}\Pi x{:}Prop.x \vee \neg x,\\ & z_1{:}\Pi x{:}N.(Sx =_N Sy){\to}(x =_N y),\\ & z_2{:}S0 \neq_N 0,\\ & z_3{:}\Pi P{:}N{\to}Prop.P0{\to}(\Pi y{:}N.Py{\to}P(Sy)){\to}(\Pi y{:}N.Py).\end{aligned}$$

One can prove (for readability we omit all type information)

$$\Gamma_{PA} \vdash z_3 Q z_2(\lambda y.\lambda p.\lambda a.p(z_1(Sy)a) : \Pi x{:}N.(Sx \neq x).$$

3. *For $A{:}Type$, a set of subsets of A is a predicate $F{:}(A{\to}Prop){\to}Prop$. The intersection and union of all subsets of F can now be described in* **λHOPL** *by $\cap_F = \lambda x{:}A.(\Pi P{:}A{\to}Prop.FP{\to}Px)$ and $\cup_F = \lambda x{:}A.(\exists P{:}A{\to}Prop.FP\&Px)$, where $\exists$ is defined in terms of Π, just asc it was defined in terms of $\forall$ earlier.*

Remark 2.9 *In the following we shall sometimes write $\forall$ where in fact we should write Π. This is to stress the (informal) semantics of the Π that we aim at at that specific point in the text.*

A disadvantage of our way of presenting higher order predicate logic as **λHOPL** is that we can not find e.g. second order predicate logic as a subsystem by an easy restriction on the rules. For the syntactic rules there is no distinction between the basic domains and the domain Prop. Further it doesn't allow a straightforward syntactical description of the formulas-as-types embedding of higher order predicate logic into CC. We therefore look at the following definition of higher order predicate logic, due to [Berardi 1988] (and defined for the purpose of describing the Curry-Howard embedding.)

Definition 2.10 *The system λPREDω is the following Pure Type System*

$$\begin{aligned}\mathcal{S} &= Prop, Set, Type^p, Type^s,\\ \mathcal{A} &= Prop : Type^p\, Set : Type^s,\\ \mathcal{R} &= (Set, Set), (Set, Type^p), (Type^p, Set), (Type^p, Type^p),\\ &= (Prop, Prop), (Set, Prop), (Type^p, Prop).\end{aligned}$$

The sort Prop is to be understood as the universe of propositions, the universes Set and Type^p together form the universe of domains (domains of the form $A_1{\to}\cdots{\to}A_n{\to}\alpha$ with α a variable are of type Set, domains of the form $A_1{\to}\cdots{\to}A_n{\to}\Omega$ are of type $\text{Type}^p (n \geq 0)$.) The sort Type^s allows the introduction of variables of type Set.

As a subsystem of λPREDω we easily find higher order predicate logic without functional domains by removing the rules (Set, Set) and $(\text{Type}^p, \text{Set})$, but also second order predicate logic by in addition removing the rule $(\text{Type}^p, \text{Type}^p)$. Before going further we state that λPREDω is really the same as λHOPL.

Proposition 2.11 *There are derivability-preserving mappings G from $\lambda PRED\omega$ to λHOPL and F from λHOPL to $\lambda PRED\omega$ such that $F \circ G = Id$ and $G \circ F = Id$.*

Proof Take for $G : \lambda\text{PRED}\omega \to \lambda\mathsf{HOPL}$ the PTS morphism

$$\begin{aligned} G(\text{Prop}) &= \text{Prop}, \\ G(\text{Set}) &= \text{Type}, \\ G(\text{Type}^p) &= \text{Type}, \\ G(\text{Type}^s) &= \text{Type}'. \end{aligned}$$

and for $F : \lambda\mathsf{HOPL} \to \lambda\text{PRED}\omega$ first define the mapping F from $\mathsf{TERM}(\lambda\mathsf{HOPL})\backslash \{\text{Type}'\}$ to $\mathsf{TERM}(\lambda\text{PRED}\omega)$ by

$$\begin{aligned} F(x) &= x, (x \text{ a variable}), \\ F(\text{Prop}) &= \text{Prop}, \\ F(\text{Type}) &= \text{Set}, \end{aligned}$$

and further by induction on the structure of the terms. G, being a PTS morphism, preserves derivations. F preserves substitution and β-equality and F extends to contexts straightforwardly by defining

$$F(x_1{:}A_1, \ldots, x_n{:}A_n) := x_1{:}F(A_1), \ldots, x_n{:}F(A_n).$$

(The sort Type$'$ does not appear in a context of λHOPL.) Now we extend F to derivable judgements of λHOPL by defining

$$\begin{aligned} F(\Gamma \vdash M : A) &= F(\Gamma) \vdash F(M) : F(A), \text{ if } A \neq \text{Type}, \text{Type}', \\ F(\Gamma \vdash M : \text{Type}) &= F(\Gamma) \vdash F(M) : \text{Set}, \text{ if } M \equiv M_1 \to \cdots \to \alpha, (\alpha \text{ a variable}), \\ F(\Gamma \vdash M : \text{Type}) &= F(\Gamma) \vdash F(M) : \text{Type}^p, \text{ if } M \equiv M_1 \to \cdots \to \text{Prop}, \\ F(\Gamma \vdash \text{Type} : \text{Type}') &= F(\Gamma) \vdash \text{Set} : \text{Type}^s. \end{aligned}$$

By easy induction one proves that F preserves derivations. Also $F(G(\Gamma \vdash M : A)) = \Gamma \vdash M : A$ and $G(F(\Gamma \vdash M : A)) = \Gamma \vdash M : A$.

3 The Calculus of Constructions

We may observe that the inductive data types have already become part of the systems λ**HOPL** and $\lambda\text{PRED}\omega$ on the logical level via the coding of data types as propositions in the polymorphic lambda calculus. There the proposition $\Pi\alpha : \text{Prop}.\alpha \to (\alpha \to \alpha) \to \alpha$ denotes the data type of natural numbers and the equivalence classes (equivalence under cut-elimination = β-conversion) of closed terms represent the numbers. This encoding can be done here because λ**HOPL** contains Girard's F_ω (λ**HOPL** without Type$'$ is just F_ω.)

Before introducing CC, let's first outline this impredicative coding of data types in polymorphic lambda calculus. We feel this is necessary for a good understanding of the system. Details of the encoding can be found in [Böhm and Berarducci 1985]

and [Girard et al. 1989]. First we define the polymorphic lambda calculus (Girard's system F) as the Pure Type System with

$$\begin{aligned} \mathcal{S} &= \text{Prop}, \text{Type}, \\ \mathcal{A} &= \text{Prop} : \text{Type}, \\ \mathcal{R} &= (\text{Prop}, \text{Prop}), (\text{Type}, \text{Prop}). \end{aligned}$$

This is a polymorphic language for the typing of functional programs. There are no basic data types, but most of what we need can be defined. We treat three examples.

Examples 3.1 *1. The natural numbers in F are defined by the type* $Nat := \Pi\alpha : Prop.\alpha\to(\alpha\to\alpha)\to\alpha$ *and we find zero and succesor by taking the constructors*

$$\begin{aligned} Z &:= \lambda\alpha{:}Prop.\lambda x{:}\alpha.\lambda f{:}\alpha\to\alpha.x, \\ S &:= \lambda n{:}Nat.\lambda\alpha{:}Prop.\lambda x{:}\alpha.\lambda f{:}\alpha\to\alpha.f(n\alpha x f). \end{aligned}$$

Now it is easy to define functions by iteration on Nat, by taking for $c{:}\sigma$ *and* $g{:}\sigma\to\sigma$, $Itcg := \lambda x{:}Nat.x\sigma cg : Nat\to\sigma$. *It is also possible to define functions by primitive recursion, but this is a bit more involved and also inefficient.*

2. For σ *a type, the type of list over* σ *is defined by the type* $List(\sigma) := \Pi\alpha{:}Prop.\alpha\to(\sigma\to\alpha\to\alpha)\to\alpha)$ *and we find the constructors*

$$\begin{aligned} Nil &:= \lambda\alpha{:}Prop.\lambda x{:}\alpha.\lambda f{:}\sigma\to\alpha\to\alpha.x, \\ Cons &:= \lambda t{:}\sigma\lambda l{:}List(\sigma).\lambda\alpha{:}Prop.\lambda x{:}\alpha.\lambda f{:}\sigma\to\alpha\to\alpha.ft(l\alpha x f). \end{aligned}$$

Again functions (like 'head' and 'tail') can be defined by iteration and primitive recursion over lists.

3. Also coinductive data types, which can be understood as greatest fixed points in a domain, can be defined in system F. (The inductive data types correspond to smallest fixed points.) As an example we treat the type of streams (infinite lists) of natural numbers. $Str := \exists\alpha.(\alpha\to Nat)\&(\alpha\to\alpha)\&\alpha$. *For convenience we write* $\langle f, g, x\rangle : (\alpha\to Nat)\&(\alpha\to\alpha)\&\alpha$ *for* $f{:}\alpha\to Nat$, $g{:}\alpha\to\alpha$ *and* $x{:}\alpha$, *with projections* π_1, π_2 *and* π_3. *Then we have destructors* $Head{:}Str\to Nat$ *and* $Tail{:}Str\to Str$ *defined by* $Head := \lambda s{:}Str.sNat(\lambda\alpha z.(\pi_1 z)(\pi_3 z)$ *and* $Tail := \lambda s{:}Str.sStr(\lambda\alpha z.\lambda\beta k.k\alpha(\pi_1 z)(\pi_2 z)(\pi_2 z(\pi_3 z))$. *It is possible to define functions from a type* σ *to Str by coiteration and corecursion.*

We see that the impredicative data types that are definable on the level of the propositions have a lot of structure already. (Girard has shown that on the type Nat one can represent all recursive functions that are provably total in higher order arithmetic.) It could therefore be fruitful to use them for the domains and to skip the variables of type Type′. This means that both the logical formulas and the data types are of type Prop. Then, because we want

to do *predicate* logic, we have to introduce the possibility of defining predicates on these new domains (which are in fact propositions) by adding the rule (Prop, Type) to $\mathcal{R}$. (The type $A \to$Prop then represents the type of predicates on A and we can declare variables of type $A \to$Prop in the context.) This is the Calculus of Constructions, sometimes referred to as the *Pure* Calculus of Constructions to distinguish it from its extensions and variations.

Definition 3.2 *The* Calculus of Constructions *is the Pure Type System with*

$$\begin{aligned} \mathcal{S} &= \mathit{Prop}, \mathit{Type}, \\ \mathcal{A} &= \mathit{Prop} : \mathit{Type}, \\ \mathcal{R} &= (\mathit{Prop}, \mathit{Prop}), (\mathit{Prop}, \mathit{Type}), (\mathit{Type}, \mathit{Type}), (\mathit{Type}, \mathit{Prop}). \end{aligned}$$

Using our understanding of higher order predicate logic, the sort Prop is the universe of both propositions and domains in which a whole range of (closed) data types is present.

Another way to see things is to understand Prop just as the universe of propositions (refraining from understanding the propositions as domains), in which case a type like $\varphi \to$Prop (φ : Prop) can be understood as the type of predicates on proofs of φ. This allows one to do predicate logic over the proofs of propositions. For practical purposes this latter approach doesn't seem to be so fruitful. For example one can not distinguish between proofs that are cut-free and proofs that are not. This is because lambda terms that are β-equal (proofs that are equal via cut-elimination) are identified: If Pt is provable and $t =_\beta t'$, then also Pt' is provable. If one is looking for these kind of applications, it is much more promising to use the 'coding' of a logic in a relatively weak framework like Automath or LF. There is however also the possibility to restrict the conversion rule of CC, such that only some convertible propositions are identified. (A system like this is described in [Coquand and Huet 1988].)

It should be clear that in any of the two approaches the distinction between domains, objects and proofs is blurred: propositions may contain proofs and there is no a priori distinction between domains and propositions. On the other hand it does take the formulas-as-types approach very seriously in the sense that formulas are not only treated *in the same way* as the types (domains) but just as if they were types, putting them in the same universe. Because of this mixing of formulas and domains, the Curry-Howard embedding from higher order predicate logic into CC (as described informally above) will not be complete. The embedding from higher order propositional logic into CC (i.e. if one refrains from understanding the propositions as domains) is complete. To see what is going on here we shall make the Curry-Howard embedding precise by describing it via the system λPREDω.

Before going into the syntactical formalisation of the formulas-as-types embedding, we want to treat some examples to get the flavour. In these examples, the impredicative coding of data types will be used as described in 3.1. First we want to discuss induction over the terms of type Nat and see to which extent Nat represents the free algebra of natural numbers. Then we treat two formulas that represent specifications of programs. This touches upon one of the most

interesting aspects of CC: To use it as a higher order constructive logic in which one can represent specifications as formulas (about data types.) From a proof of the formula the constructive content can then be extracted as a program (more precisely a lambda term typable in F_ω.) A lot of work on this subject has been done in [Paulin 1989]; we shall say a little bit more about this in Section 5.

Example 3.3 *We know (this can be proved by Theorem 3.5) that in CC each closed term of type Nat is β-equal to a term of the form $\lambda\alpha{:}Prop.\lambda x{:}\alpha.\lambda f{:}\alpha{\to}\alpha.f(\ldots(fx)\ldots)$. That is, modulo β-equality, the closed terms of type Nat are precisely the ones formed by S out of Z. This induction property can be expressed in CC, but is not provable in side it. To be precise, define*

$$Ind_{Nat} := \forall P{:}Nat{\to}Prop.PZ{\to}(\forall x{:}Nat.Px{\to}P(Sx)){\to}(\forall x{:}Nat.Px),$$

then Ind_{Nat} is not provable. If we assume Ind_{Nat}, we still can't prove that the type Nat is the free structure generated by Z and S. To establish this we have to add the premises $Z \neq_{Nat} SZ$ and $\forall x,y{:}Nat.(Sx = Sy){\to}(x = y)$. None of these two propositions is provable in CC. In higher order predicate logic (working in the natural numbers-signature $\langle N, Z, S\rangle$) these three assumptions are independent, so we would have to add all three of them to obtain the free algebra of natural numbers. In CC this is not so: The assumptions Ind_{Nat} and $Z \neq_{Nat} SZ$ suffice to prove the freeness of Nat. (This is so because one can define $P{:}Nat{\to}Nat$ with $Ind_{Nat} \vdash \forall x{:}Nat.P(Sx) =_{Nat} x$ in CC.)

Examples 3.4 *1. Abbreviate List(Nat) to List. The proposition stating that for every finite list of numbers there is a number that majorizes all its elements can be expressed by*

$$\forall l{:}List.\exists n{:}Nat.\forall m{:}Nat.\text{`}m \in l \to m \leq n',$$

where $m \in l$ represents

$$\forall P{:}List{\to}Prop.(\forall k{:}List.P(Consmk)) \to \forall k{:}List\forall r{:}Nat.(Pk{\to}P(Consrk)) \to Pl$$

and $m \leq n$ represents

$$\forall R{:}Nat{\to}Nat{\to}Prop.(\forall x{:}Nat.Rxx){\to}(\forall x,y{:}Nat.Rxy{\to}Rx(Sy)){\to}Rmn.$$

A proof of this proposition constructs for every list l a number n and a proof of the fact that n majorizes l. From it one can extract a program of type List→Nat that satisfies this specification.

2. Abbreviate Str(Nat) to Str. The proposition that every (infinite) stream that is majorizable has a maximal element can be expressed by

$$\forall s{:}Str.(\exists n{:}Nat.\forall m{:}Nat.m \in s \to m \leq n){\to}(\exists n{:}Nat.\text{`}n \text{ is maximum of } s\text{'},$$

where $m \in s$ now represents

$$\exists p{:}Nat.Head(pStrTails) = m,$$

and 'n is maximum of s' represents

$$(n \in s)\&(\forall m{:}Nat.m \in s \rightarrow m \leq n.$$

From a proof of this formula one would like to extract a term of type $Str \rightarrow Nat$ that computes the maximum of a stream, if it exists. In general this may however not be possible because the construction of the maximum of s will depend on the proof of the premise that s is majorizable. Part of the construction lies in the proof of this premise (and not just in the construction *of the majorant of s but also in the* proof *that it is the majorant.) As there is no notion of undefinedness in CC (all terms are normalising), the construction in the proof always gives a number as answer, even if there is no majorant of the stream s.*

We want to state some of the most important meta-properties of CC. In the examples we already came accross the normalization property.

Theorem 3.5 *CC is strongly normalizing. (All β-reduction sequences starting from an $M \in \mathsf{TERM}(CC)$ are finite.)*

A first proof of this theorem can be found in [Coquand 1985], but the proof contained a bug as remarked by Jutting, who then gave a proof of normalization for CC. (That is, every $M \in \mathsf{TERM}(\mathrm{CC})$ reduces to a term in normal form.) Coquand repaired his own proof in a preliminary version of [Coquand 1990]. All proofs use a higher order variant of the 'candidat de réducibilité' method as developped by Girard for proving strong normalisation for his system F and F_ω. (See [Girard et al. 1989] for the proof for system F.) The idea is to define a kind of realisability model in which propositions are interpreted as sets of lambda terms (the realisers). A detailed explanation of the method can be found in [Gallier 1990]. It is also possible to obtain the strong normalisation for CC more or less directly from the strong normalisation property of F_ω, as is shown in [Geuvers and Nederhof 1991]. The importance of the (strong) normalisation property lies in the fact that it gives a handle on the number of proofs of a proposition. (One can for example show that every closed term of type Nat is β-equal to a numeral (i.e. a term of the form $S(\ldots S(Z)\ldots)$.) Further, by using normalization one can prove the decidability of typing.

Theorem 3.6 *Given a context Γ and a pseudoterm M, it is decidable whether there exists a term A with $\Gamma \vdash M : A$. If such a term A exists, it can be computed effectively.*

Some hints towards a proof can be found in [Coquand and Huet 1988] and more details in [Coquand 1985] and especially in [Martin-Löf 1971]. See also [Harper and Pollack 1991] for an exposition on the decidability of typing for an extended version of CC, which also describes an algorithm for computing a type.

4 The formulas-as-types embedding from higher order predicate logic into CC

The Curry-Howard embedding from higher order predicate logic into CC makes an essential distinction between basic and functional domains on the one hand (including the definable data types) and higher order domains like $A \to \Omega$ on the other. The first are interpreted as propositions (the basic domains as variables of type Prop, the functional domains as implicational formulas and the definable data types via the embedding of data types in system F) and the higher order domains are interpreted as types, e.g. $A \to \mathrm{Prop} : \mathrm{Type}$.

Using the system $\lambda\mathrm{PRED}\omega$ we can now describe the Curry-Howard formulas-as-types embedding of higher order predicate logic into CC as a PTS morphism. In fact this is the whole reason for introducing $\lambda\mathrm{PRED}\omega$ here. In fact there are different ways of interpreting HOPL in CC, but the one we describe here is what the inventor(s) of CC aim at (see [Coquand 1985] and [Coquand and Huet 1988]), and which is sometimes called the 'canonical embedding' of higher order predicate logic into CC. In our setting this canonicity is partly forced upon by the syntax, therefore it is worthwile to also understand the embedding from a more semantical point of view.

It is well-known by now that the embedding is not complete, i.e. there are propositions that are not provable in HOPL that become provable when mapped into CC. We shall treat some examples of those formulas. This incompleteness result is sometimes referred to as the 'non-conservativity of CC over HOPL', but this terminology is a bit ambiguous because '(non-)conservativity' actually only applies if one system is a real subsystem of the other. Therefore we shall use the more correct terminology of '(in)completeness of the embedding' here.

Definition 4.1 *The formulas-as-types embedding from $\lambda PRED\omega$ to CC is the* PTS *morphism H with*

$$\begin{aligned} H(Prop) &= Prop, \\ H(Set) &= Prop, \\ H(Type^p) &= Type, \\ H(Type^s) &= Type. \end{aligned}$$

Let's first remark that there are terms of type Prop, typable in CC in a context that comes from $\lambda\mathrm{PRED}\omega$, that do not have an intuitive meaning in higher order predicate logic, for example $\alpha{:}\mathrm{Prop}, P{:}\alpha \to \mathrm{Prop}, x{:}\alpha \vdash Px \to \alpha : \mathrm{Prop}$. (Is $Px \to \alpha$ a domain or a proposition in $\lambda\mathrm{PRED}\omega$?)

As pointed out, one can also refrain from understanding CC as a system of predicate logic and view CC as a higher order propositional logic with propositions about (proofs of) propositions. Let's also make that embedding precise.

Definition 4.2 *The typed lambda calculus corresponding to higher order propositional logic $\lambda PROP\omega$ is the* PTS *with*

$$\mathcal{S} = Prop, Type,$$

$$\begin{aligned}\mathcal{A} &= Prop : Type,\\ \mathcal{R} &= (Prop, Prop), (Type, Prop), (Type, Type).\end{aligned}$$

The PTS morphisms from λPROPω into CC and from λPROPω into λPREDω are easily defined; the first is just the identity and the second maps Type to Type^p.

Now all kind of rather exotic propositions can be understood as meta-propositions about higher order propositional logic. For example

$$\alpha{:}\mathrm{Prop}, P{:}\alpha{\to}\mathrm{Prop}, x{:}\alpha \vdash Px{\to}\alpha : \mathrm{Prop}$$

can be seen as the statement that for α a proposition and x a proof of α, if P holds for x, then α holds. We can go to arbitrary high levels of meta-reasoning, for example

$$\alpha{:}\mathrm{Prop}, P{:}\alpha{\to}\mathrm{Prop}, x{:}\alpha, Q{:}Px{\to}\mathrm{Prop}, y{:}Px \vdash Px{\to}Qy : \mathrm{Prop}$$

but also

$$P{:}\Pi\alpha{:}\mathrm{Prop}.\alpha{\to}\mathrm{Prop}, \varphi{:}\mathrm{Prop}, x{:}\varphi, y{:}P\varphi x \vdash P(P\varphi x)y{:}\mathrm{Prop}.$$

Of course the typed lambda calculus λPROPω is just Girard's calculus $F\omega$, the extension of system F with higher order type constructors. Viewing it in this way (as a calculus that assigns types to programs), we see a very powerful language for typing functional programs that includes many basic data types. As Girard has shown, we can type all recursive functions on the natural numbers that are provably total in higher order arithmetic. ([Girard 1972]) This is also the power of the formulas-as-types formalism: it relates constructive proofs to functional programs in the sense that from a constructive proof of a formula that represents a certain specification we can derive the construction in the proof as a program that satisfies the specification. A lot of work has been done in this field by Paulin. (See [Paulin 1989].) In 5 we shall give an example to get an idea of the strength of the formalism.

4.1 Consistency of CC

As the described embedding from λPREDω into CC is not complete (CC proves more propositions than λPREDω), one may wonder whether there are propositions that CC can not prove, or to put the question differently, is CC consistent? That this is the case can be shown quite easily by giving a two-point model for CC. (Originally due to [Coquand 1990].) The type Prop is interpreted as $\{\emptyset, \{\emptyset\}\}$ (or $\{0, 1\}$ in ZF language) and if $\vdash M : A$, the interpretation of M is in the set A. This model is called the 'proof-irrelevance' model in [Coquand 1990] because in the model all proofs of a proposition are mapped to 0; the model doesn't distinguish between proofs. So the model also implies that in CC one can not prove $a \neq_A a'$ (for $\vdash a, a' : A$) in the empty context. The interpretation will be such that the proposition $\bot(:\equiv \Pi\alpha{:}\mathrm{Prop}.\alpha)$ is interpreted by 0, so there can be no term M with $\vdash M : \bot$ and so CC doesn't prove $\bot$. As this is an

introductory text we shall make the model construction a bit more precise here, in the meantime obtaining the result that CC is conservative over λPROPω. (A result first proved by Paulin (see [Paulin 1989]) and independently due to Berardi ([Berardi 1988]).)

Definition 4.3 *Define the mapping* $[-] : \mathsf{TERM}(CC) \to \mathsf{TERM}(\lambda PROP\omega)$ *as follows.*

$$\begin{array}{rcl}
[Type] & = & Type, \\
[Prop] & = & Prop, \\
[x] & = & x, \ \textit{for } x \ \textit{a variable}, \\
[\Pi x{:}A.B] & = & [B] \ \textit{if } A{:}Prop, B{:}Type, \\
 & = & \Pi x{:}[A].[B] \ \textit{else}, \\
[\lambda x{:}A.M] & = & [M] \ \textit{if } A{:}Prop, M{:}B{:}Type, \ \textit{(for some } B\textit{)}, \\
 & = & \lambda x{:}[A].[B] \ \textit{else}, \\
[PM] & = & [P] \ \textit{if } M{:}A{:}Prop, P{:}B{:}Type, \ \textit{(for some } A, B\textit{)}, \\
 & = & [P][M] \ \textit{else},
\end{array}$$

Remark 4.4 *One may wonder whether the side conditions 'A:Prop B:Type' can lead to ambiguities, making the definition incomplete. (It could be the case that PM is typable in both* Γ *and* Γ' *with* $\Gamma \vdash M{:}A{:}Prop$ *and* $\Gamma' \vdash M{:}B{:}Type$*.) In fact, there are such ambiguities, for example in xy, the type of y can be of type Prop (e.g. in* $\Gamma = \beta{:}Prop, y{:}\beta, x{:}\beta{\to}Prop$*) but also of type Type (e.g. in* $\Gamma = y{:}Prop, x{:}Prop{\to}Prop$*.) However, these ambiguities can easily be solved if we would have built up the syntax a little bit more carefully, namely by deviding the set of variables* V *into disjoint sets* V^s *for every* $s \in \mathcal{S}$*. In the (weak) and (var) rule we then put the restriction that the new variable that is added to the context (the x in 'x : A') should be in* V^s *(if* $\Gamma \vdash A : s$*.)*
The advantage of these small modifications is that we can distinguish propositions from types 'on the nose' and similarly distinguish inhabitants of propositions from inhabitants of types. To be precise, we have the following property. $(s, s' \in \{Prop, Type\}$*.)*

$$\begin{array}{rcl}
\Gamma \vdash A : s, \Gamma' \vdash A : s' & \Rightarrow & s \equiv s', \\
\Gamma \vdash M : A : s, \Gamma' \vdash M : B : s' & \Rightarrow & s \equiv s'.
\end{array}$$

This property is valid for a whole class of Pure Type Systems (provided we have made the sketched alterations to the syntax), which covers all PTS*s that are used in this paper. (See [Geuvers and Nederhof 1991] for more details.)*

The mapping $[-]$ straightforwardly extends to contexts. The following can be proved by an easy induction on derivations. ($\vdash_{CC}$ denotes derivability in CC, $\vdash_{\lambda\text{PROP}_\omega}$ denotes derivability in λPROPω.)

Proposition 4.5 *[Paulin 1989, Berardi 1988]*

$$\Gamma \vdash_{CC} M : A \Rightarrow [\Gamma] \vdash_{\lambda PROP_\omega} [M] : [A]$$

Corollary 4.6 *[Paulin 1989, Berardi 1988]) CC is conservative over $\lambda PROP\omega$*

Proof The only thing to check is that for $M \in \mathsf{TERM}(\lambda\mathrm{PROP}\omega)$, $[M] \equiv M$.

The consistency of CC now follows from the consistency of higher order propositional logic (and in fact a detailed verification that $\lambda\mathrm{PROP}\omega$ proves the same propositions as higher order propositional logic.) We sketch here a short proof of the consistency of $\lambda\mathrm{PROP}\omega$ by constructing the promised two point model, which is (by Proposition 4.5) also a model of CC. (It is not so easy to construct the model immediately for CC, a problem that is solved in [Coquand 1990] by describing the model for a variant of CC that we shall discuss in 6.3. Here we use the mapping $[-]$ from CC to $\lambda\mathrm{PROP}\omega$ for this purpose.)

Before constructing the model we want to state some properties of $\lambda\mathrm{PROP}\omega$ that will be used. First, the set of types of $\lambda\mathrm{PROP}\omega$ (those terms A for which $\Gamma \vdash A : \mathrm{Type}$ for some Γ) can be described by K, where

$$K ::= \mathrm{Prop} \mid K \to K.$$

Second, no proposition-variables are subterms of propositions or constructors, that is

$$\Gamma \vdash M : A : \mathrm{Type} \Rightarrow \Gamma' \vdash M : A : \mathrm{Type},$$

where Γ' consists just of those declarations $x{:}B$ in Γ for which $\Gamma \vdash B : \mathrm{Type}$.

These two properties imply that we can build the interpretation in three stages by first giving a meaning to the types, then to the propositions and constructors and then to the proofs. It will be convenient to seperate the variables, as was discussed in Remark 4.4, into two sets, $\mathsf{V}^{\mathrm{Prop}}$ for proof-variables and $\mathsf{V}^{\mathrm{Type}}$ for constructor-variables. The first will be denoted by Latin characters, the latter by Greek characters. In general, an interpretation of terms of $\lambda\mathrm{PROP}\omega$ uses a valuation ξ of constructor-variables and a valuation ρ of proof-variables. In our simple model all free proof-variables will have the value 0, so we only need ξ. For convenience we think of contexts of $\lambda\mathrm{PROP}\omega$ as being split up in a Γ_1, containing the declarations of constructor variables, and a Γ_2, containing the declarations of proof-variables. The valuation ξ *satisfies* Γ_1 (notation $\xi \models \Gamma_1$) if for all $\alpha : A \in \Gamma_1$, $\xi(\alpha)$ is in the interpretation of A. ($A : \mathrm{Type}$, so A doesn't contain any free variables.) The valuation ξ *satisfies* Γ (notation $\xi \models \Gamma$) if ξ satisfies Γ_1 and for all $x{:}A \in \Gamma_2$, the interpretation of A under ξ is not empty. ($A : \mathrm{Prop}$, so A can only contain free constructor-variables.)

Definition 4.7 *For $\Gamma \vdash M{:}A$ we define the interpretation function $[\![-]\!] : \mathsf{TERM}(\lambda PROP\omega) \to$ Sets as follows.*

1. *For types, $[\![Prop]\!] = 2$ and $[\![k_1 \to k_2]\!] = [\![k_1]\!] \to [\![k_2]\!]$ (for k_1, $k_2 \in K$), where the latter arrow denotes set-theoretic function space.*

2. *For constructors, let ξ be a valuation of constructor-variables such that $\xi \models \Gamma_1$,*

$$[\![\alpha]\!]_\xi \quad = \quad \xi(\alpha),$$

$$
\begin{array}{rcl}
[\![\Pi x{:}A.B]\!]_\xi & = & 1 \textit{ if } \forall a \in [\![A]\!][[\![B]\!]_{\xi(x:=a)} = 1], \\
 & = & 0 \textit{ else, (for } A : \textit{Type}, B : \textit{Prop)}, \\
[\![A{\to}B]\!]_\xi & = & [\![A]\!]_\xi \to [\![B]\!]_\xi, \textit{ (for } A, B : \textit{Prop)}, \\
[\![PQ]\!]_\xi & = & [\![P]\!]_\xi [\![Q]\!]_\xi, \\
[\![\lambda\alpha{:}A.P]\!]_\xi & = & \lambda\!\!\lambda a \in [\![A]\!]_\xi . [\![P]\!]_{\xi(x:=a)}.
\end{array}
$$

3. All proofs are interpreted as 0.

Here, $\lambda\!\!\lambda a \in U.V(a)$ *denotes a set-theoretic function. Further we identify all singleton sets (like e.g.* $[\![A]\!]_\xi \to [\![A]\!]_\xi$*) with* 1 *and we use the fact that no proof-variables occur in propositions.*

By induction on derivations one can prove the following property.

Proposition 4.8 *If* $\Gamma \vdash M : A$*, then for all valuations* ξ *with* $\xi \models \Gamma$*,* $[\![M]\!]_\xi \in [\![A]\!]_\xi$*.*

It is good to realise here that for example for $\Gamma = x{:}\bot(\equiv \Pi\alpha{:}\mathrm{Prop}.\alpha)$, there is no ξ with $\xi \models \Gamma$, so in this case the conclusion of the proposition is trivially satisfied.

Corollary 4.9 *$\lambda PROP\omega$, and hence CC, is consistent.*

Proof For all valuations ξ, $[\![\bot]\!]_\xi = 0$. All valuations satisfy the empty context, so if $\vdash M : \bot$, then $0 \in 0$, quod non.

4.2 Incompleteness of the formulas-as-types embedding

As already pointed out, the formulas-as-types embedding from higher order predicate logic in CC is not complete. In this section we want to discuss some examples of propositions that are not provable in the logic but become inhabited when mapped into CC. At the same time one obtains a better understanding of the logical merits of CC. First we show that if one allows empty domains in the logic, the incompleteness is quite easy.

Remark 4.10 *In CC, the existential quantifier has a first projection, similar to Martin-Löf's understanding of the existential quantifier as a strong* Σ*-type. (See e.g. [Martin-Löf 1984].) To be precise, there is a projection function* $p : (\exists x{:}A.\varphi){\to}A$*, for* $A, \varphi{:}\mathit{Prop}$ *in CC: Remember that* $\exists x{:}A.\varphi \equiv \Pi\alpha{:}\mathit{Prop}.(\Pi x{:}A.\varphi{\to}\alpha){\to}\alpha$ *and take* $p :\equiv \lambda z{:}(\exists x{:}A.\varphi).zA(\lambda x{:}A.\lambda y{:}\varphi.x)$*. So, if* $\exists x{:}A.\varphi$ *is provable one immediately obtains a closed term of type A by applying p. In general there is no second projection, so the* $\exists$ *is not a strong* Σ*. (If, for example,* $\exists x{:}A.\varphi$ *is assumed in the context, say by* $z{:}\exists x{:}A.\varphi$*,* $\varphi[pz/x]$ *is not provable.)*

Lemma 4.11 *In* λHOPL*, for* $x \notin FV(\varphi)$*,*

$$P{:}A{\to}\mathit{Prop}, \varphi{:}\mathit{Prop} \not\vdash (\exists x{:}A.Px) \supset (\forall x{:}A.\varphi) \supset \varphi),$$

but in CC there is a term M with

$$A{:}\mathit{Prop}, P{:}A{\to}\mathit{Prop}, \varphi{:}\mathit{Prop} \vdash M : (\exists x{:}A.Px){\to}(A{\to}\varphi){\to}\varphi).$$

Proof Because the λHOPL-context doesn't contain a declaration of a variable to A, we can't construct a term of type A, so we have no proof. In CC, take $M \equiv \lambda z{:}(\exists x{:}A.Px).\lambda y{:}(A{\to}\varphi).y(px)$, with p as in Remark 4.10.

Also without using empty domains the embedding is not complete, as was first independently shown by [Berardi 1989] and [Geuvers 1989]. We treat both counterexamples, starting with the latter as it is very short (but syntactic.) Both proofs give a counterexample already for the completeness of the embedding of third order predicate logic in so called third order dependent typed lambda calculus. (In this terminology, CC is higher order dependent typed lambda calculus and the system λP2 of [Barendregt 1992] is second order dependent typed lambda calculus.) The counterexample with empty domains above already works for second order dependent typed lambda calculus; it is not known whether one can find a counterexample without allowing empty domains.

Proposition 4.12 *The formulas-as-types embedding of higher order predicate logic into CC is not complete.*

Proof [Geuvers 1989] We use the fact that if $x \notin \mathrm{FV}(\varphi)$, then $\forall x{:}A.\varphi$ and $A \supset \varphi$ can not be distinguished in CC. Take

$$\Gamma := A{:}\mathrm{Set}, a{:}A, \varphi{:}\mathrm{Prop}, \alpha{:}\mathrm{Prop}{\to}\mathrm{Prop}, z{:}P(\forall x{:}A.\varphi),$$

and we try to find a proof t of $\exists\beta{:}\mathrm{Prop}.P(\beta{\to}\varphi)$. As no extensionality has been assumed in the context, such t can't be found. (Supposing there is such t, one easily shows that it can't be in normal form.) However, in CC one can take the domain A for β because domains and propositions are not distinguished. More precisely, in $\Gamma' = A{:}\mathrm{Prop}, a{:}A, \varphi{:}\mathrm{Prop}, \alpha{:}\mathrm{Prop}{\to}\mathrm{Prop}, z{:}P(\Pi x{:}A.\varphi)$,

$$\Gamma' \vdash \lambda\gamma{:}\mathrm{Prop}.\lambda h{:}(\Pi\beta{:}\mathrm{Prop}.P(\gamma{\to}\varphi){\to}\beta).hAz : \exists\beta{:}\mathrm{Prop}.P(\beta{\to}\varphi).$$

Proof [Berardi 1989] Define

$$\mathrm{EXT} := \forall\alpha,\beta{:}\mathrm{Prop}.(\alpha{\leftrightarrow}\beta) \to (\alpha =_{\mathrm{Prop}} \beta),$$

where $\alpha{\leftrightarrow}\beta$ denotes $(\alpha{\to}\beta)\&(\beta{\to}\alpha)$ and $=_{\mathrm{Prop}}$ denotes the Leibniz equality on Prop, $\alpha =_{\mathrm{Prop}} \beta :\equiv \forall P{:}\mathrm{Prop}{\to}\mathrm{Prop}.P\alpha{\to}P\beta$. This 'EXT' is the extensionality axiom for propositions; note that it is a consequence of EXT_2 as defined on page 5. In CC this axiom has some unexpected consequences, because for non-empty domains A one has $A \leftrightarrow A{\to}A$ and so, by EXT, all generic properties that hold for A, hold for $A{\to}A$ and vice versa. This can be used to construct in CC a proof p with

$$A{:}\mathrm{Prop}, a{:}A, z{:}\mathrm{EXT} \vdash p : A \text{ is a } \lambda\text{-model},$$

where

$$\begin{aligned} A \text{ is a } \lambda\text{-model} \;:=\; & \exists\Lambda{:}(A{\to}A){\to}A.\exists\mathrm{App}{:}A{\to}A{\to}A. \\ & \mathrm{App}\circ\Lambda =_{\mathrm{Prop}} \mathrm{Id}_{A\to A}\& \\ & \Lambda\circ\mathrm{App} =_{\mathrm{Prop}} \mathrm{Id}_A. \end{aligned}$$

This implies (among other things) that every term of type $A \to A$ has a fixed point. Of course, in higher order predicate logic, from EXT it doesn't follow that every function on a non-empty domain has a fixed point.

If we look for example at a context for Heyting arithmetic,

$$\begin{aligned}\Gamma_{HA} \quad := \quad & N{:}\mathrm{Prop}, 0{:}N, S{:}N \to N, \\ & z_1{:}\forall x{:}N.(Sx =_N Sy) \to (x =_N y), \\ & z_2{:}S0 \neq_N 0, \\ & z_3{:}\forall P{:}N \to \mathrm{Prop}.P0 \to (\forall y{:}N.Py \to P(Sy)) \to (\forall y{:}N.Py),\end{aligned}$$

then there is a term t with

$$\Gamma_{HA}, z{:}\mathrm{EXT} \vdash t : \bot.$$

4.3 Consistency of contexts in CC

One may wonder whether $\mathrm{EXT} := \forall \alpha, \beta{:}\mathrm{Prop}.(\alpha \leftrightarrow \beta) \to (\alpha =_{\mathrm{Prop}} \beta)$, is *consistent* in CC. That this is the case can be seen by using the proof-irrelevance model of Definition 4.7. The interpretation of EXT in the model is 1, so if EXT were inconsistent, CC itself would be inconsistent, quod non. The same argument applies to show that CC with classical logic is consistent. Define

$$\mathrm{CL} := \forall \alpha : \mathrm{Prop}.\alpha \vee \neg\alpha,$$

where $\neg\alpha$ denotes $\alpha \to \bot$. Then $[\![\mathrm{CL}]\!] = 1$, so CL is consistent. A more interesting example is the Axiom of Choice. Let

$$\mathrm{AC} := \forall P{:}A \to B \to \mathrm{Prop}.(\forall x{:}A.\exists y{:}B.Pxy) \to (\exists f{:}A \to B.\forall x{:}A.Px(fx)).$$

Applying the mapping of Definition 4.3 we obtain $[\mathrm{AC}] = \forall P{:}\mathrm{Prop}.(A \to B \,\&\, P) \to (A \to B)\,\&\,(A \to P)$. Now $[\mathrm{AC}]$ is provable in $\lambda\mathrm{PROP}\omega$, so AC is not inconsistent in CC (by the consistency of $\lambda\mathrm{PROP}\omega$.)

We may notice that in all these cases the proof of consistency of an assumption is done by giving a model in which the assumption is satisfied; for EXT and CL the proof-irrelevance model and for AC the system $\lambda\mathrm{PROP}\omega$. In some (quite trivial) cases it is even possible to use CC itself as model: If the context Γ consists only of declarations $x : A$ with $A : \mathrm{Type}$ or $A =_\beta zt_1 \ldots t_p$ with z a variable, then Γ is consistent. Contexts of this kind are called *strongly consistent* in [Seldin 1990]. To verify the consistency we let $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ be a strongly consistent context and suppose that $\Gamma \vdash M : \bot$ for some M. Now we consecutively substitute all free variables that are declared in Γ by a closed term, such that all the assumed propositions become $\top (= \forall \alpha.\alpha \to \alpha)$, as follows: If $x_i : A_i \in \Gamma$ with $\Gamma \vdash A_i : \mathrm{Type}$, then $A_i =_\beta \Pi \vec{y}{:}\vec{B}.\mathrm{Prop}$, (with $\mathrm{FV}(\vec{B}) \subseteq \{x_1, \ldots, x_{i-1}\}$) and we substitute x_i by $\lambda \vec{y}{:}\vec{B^*}.\top$, where the B^* are the terms in which the substitution for $x_1, \ldots, x_{i-1}$ has already been done. If $x : zt_1 \ldots t_p (: \mathrm{Prop})$ with z a variable, we substitute x by $\lambda \alpha{:}\mathrm{Prop}.\lambda x{:}\alpha.x$, which is of type $\top$. If we denote this substitution by *, we can conclude from $\Gamma \vdash M : \bot$ and the Substitution property (page 10) that $\vdash M^* : \bot$. So Γ is consistent by the consistency of CC.

The techniques described above to show that a context is consistent are not sufficient to handle the more interesting examples. For mere proof theoretic reasons it will for example not be possible to show the consistency of Γ_{HA} (defined in the second proof of Proposition 4.12) with these techniques, because this would give us a first order consistency proof of higher order arithmetic. These kind of contexts have to be handled by a normalization argument; assuming the inconsistency of Γ_{HA}, show that a proof of $\bot$ in Γ_{HA} can not be in normal form, and so there is no such proof. In [Seldin 1990] one can find a detailed proof of the consistency of a context that represents Peano Arithmetic in a system that is a slight extension of CC. Coquand shows in [Coquand 1990] by a normalization argument that the context

$$\begin{aligned}\mathrm{Inf} \;=\; & A{:}\mathrm{Prop}, a{:}A, f{:}A{\to}A, R{:}A{\to}A{\to}\mathrm{Prop} \\ & z_1{:}\forall x{:}A.(Rxx){\to}\bot, z_2{:}\forall x, y, z{:}A.Rxy{\to}Ryz{\to}Rxz, z_3{:}\forall x{:}A.Rx(fx)\end{aligned}$$

is consistent. When contexts become larger, a consistency proof by the normalization argument can of course get very involved. Semantics is then a very helpful tool for showing that contexts are consistent and in general to show the non-derivability of a formula from a specific set of assumptions. Of course one has to use more interesting models then the one of 4.7 to establish this. In [Streicher 1991] there are some examples of this technique using the realisability semantics.

Knowing that a certain context is consistent is of course not enough to be able to use it safely for doing proofs. Due to the incompleteness of the formulas-as-types embedding, a well-understood context that is beyond suspicion in higher order predicate logic, may have unexpected side-effects when embedded in CC. Further, CC has a greater expressibility then higher order predicate logic so we may also put in the context axioms which do have a meaning but can not be expressed in the logic, for example an axiom that makes a statement about all domains. An example of this is the axiom of definite descriptions as described in [Pottinger 1989],

$$\mathrm{DD} := \forall\alpha{:}\mathrm{Prop}.\forall P{:}\alpha{\to}\mathrm{Prop}.\forall z{:}(\exists! x{:}\alpha.Px).P(\iota\alpha Pz),$$

where

$$\exists! x{:}\alpha.Px := (\exists x{:}\alpha.Px)\&(\forall x, y{:}\alpha.Px{\to}Py{\to}(x =_\alpha y))$$

and ι is a term of type $\forall\alpha{:}\mathrm{Prop}.\forall P{:}\alpha{\to}\mathrm{Prop}.(\exists! x{:}\alpha.Px){\to}\alpha$. (One can take some fixed closed term for ι but also declare it as variable in the context.) We assume the intended meaning of DD in HOPL to be clear. Together with classical logic, the axiom of definite descriptions has an unexpected side-effect in CC.

Proposition 4.13 *[Pottinger 1989] 'Classical logic' and 'definite descriptions' yield proof irrelevance in CC*

We have already encountered the semantical notion of proof irrelevance in the discussion of the model in 4.7. It can also be expressed in purely syntactical

terms as the phenomenon that for all propositions φ, all proofs of φ are Leibniz-equal. It is then formalised in CC by the proposition

$$\mathrm{PI} := \forall\alpha{:}\mathrm{Prop}.\forall x, y{:}\alpha.(x =_\alpha y).$$

Of course, PI holds in the proof-irrelevance model of 4.7 (the interpretation of PI is 1), so PI doesn't imply inconsistency. However, if we intend to use CC for predicate logic it is clearly undesirable: if Γ proves PI, then any assumption $a \neq a'$ makes Γ inconsistent. We see that PI, which is a very useful principle for proofs, is a very odd principle when applied to domain-objects. Because of the treatment of domains and propositions at the same level, principles about (proofs of) propositions have unwanted applications to the domains.

The proof of Proposition 4.13 in [Pottinger 1989] uses an adapted form of a proof by Coquand ([Coquand 1990]), showing that CC with classical logic and a derivation rule for a strong version of disjoint sum yields proof irrelevance. Let's also state this result, but not by adding a derivation rule but by adding an axiom, which really amounts to the same as the rule used in [Coquand 1990]. (Using the result by Reynolds that polymorphism is not set-theoretic, Berardi has proved that in CC, classical logic with a stronger form of definite descriptions (replacing the $\exists!$ by $\exists$) implies PI. See [LEGO-examples] for details.)

Proposition 4.14 *[Coquand 1990] 'Classical logic' with 'disjunction property for classical proofs' implies proof irrelevance in CC.*

Here we mean by 'disjunction property for classical proofs', that for $c : \mathrm{CL}$ in the context and $\varphi : \mathrm{Prop}$, $c\varphi$ is in the smallest set of proofs of $\varphi \vee \neg\varphi$ that contains all proofs that are obtained by $\vee$-introduction from a proof of φ or a proof of $\neg\varphi$. Put in syntactical terms this says that, for i and j the injections from A to $A \vee B$, respectively from B to $A \vee B$, the proposition $\forall P{:}(A \vee B){\to}\mathrm{Prop}.(\forall x{:}A.P(ix)){\to}(\forall x{:}B.P(jx)){\to}P(c\varphi)$ holds. So proof irrelevance follows from the context

$$cl{:}\mathrm{CL}, z{:}\forall\alpha{:}\mathrm{Prop}.(\alpha + \neg\alpha)(cl\alpha),$$

where for $A, B{:}\mathrm{Prop}$, $A{+}B := \lambda y{:}A{\vee}B.\forall P{:}(A{\vee}B){\to}\mathrm{Prop}.(\forall x{:}A.P(ix)){\to}(\forall x{:}B.P(jx)){\to}Py$. In presence of CL also the reverse can be proved, so we can construct a proof p with

$$cl{:}\mathrm{CL} \vdash p : PI \leftrightarrow (\forall\alpha{:}\mathrm{Prop}.(\alpha + \neg\alpha)(cl\alpha)).$$

The implication from right to left is the most interesting. The proof (in [Coquand 1990]) uses the fact that if in Γ one can construct $A : \mathrm{Prop}$, $E : A{\to}\mathrm{Prop}$, $\epsilon : \mathrm{Prop}{\to}A$ and a proof of $\forall\alpha{:}\mathrm{Prop}.\alpha \leftrightarrow E(\epsilon\alpha)$, then Γ proves $\bot$.

5 Formulas about data-types in CC

Having seen the incompleteness of the formulas-as-types embedding of higher order predicate logic in CC, we shall now see that the distance between CC and HOPL is not so large when it comes to propositions about inductive data types.

This follows from a recent result by Berardi, which we shall discuss it here only for what concerns the implications for the formulas-as-types embedding. For details and proofs we refer to [Berardi 199+]. The point is that for purposes of deriving programs from proofs, it doesn't seem to make sense to declare a theory in the context. Instead one uses the definable impredicative data types and inductive predicates on them, as is done in the examples of 3.4. This is not the place to discuss in detail the topic of extracting programs from proofs in CC, for which we refer to [Paulin 1989], but to get some flavor we do want to treat the first example of 3.4. Roughly, the program extracted from the proof is the F_{omega}-term obtained by the mapping $[-]$, as defined in Definition 4.3.

Let's consider the first example of 3.4. Suppose t is a proof of

$$\forall l{:}\mathrm{List}.\exists n{:}\mathrm{Nat}.\forall m{:}\mathrm{Nat}.`m \in l \rightarrow m \leq n'$$

in the context $a{:}\mathrm{Ind}_{\mathrm{Nat}}$. Then in F_ω we have $a{:}\forall P{:}\mathrm{Prop}.P{\rightarrow}(\mathrm{Nat}{\rightarrow}P{\rightarrow}P){\rightarrow}(\mathrm{Nat}{\rightarrow}P) \vdash [t] : \mathrm{List}{\rightarrow}(\mathrm{Nat}{\times}\mathrm{Nat}{\rightarrow}\mathrm{True}_1{\rightarrow}\mathrm{True}_2)$, where True_1 and True_2 are some trivially provable propositions. Now $[t]$ still contains computationally irrelevant information; the real program to be extracted should be something like $\lambda x{:}\mathrm{Nat}.\pi_1([t]^*x) : \mathrm{List}{\rightarrow}\mathrm{Nat}$, where * substitutes some closed term for a in $[t]$. Of course it is not irrelevant what we substitute for a, but the general picture should be clear: From the proof of the specification one can obtain the program that staisfies the specification. In [Paulin 1989] it is also shown how to extract from the proof the logical content which is a proof that the extracted program satisfies the specification. Some parts of the proof have computational content while others don't. Therefore, to mechanize the extraction proces, in [Paulin 1989] the type Prop is divided in Prop, Data and Spec, the first consisting of the propositions with purely logical content, the second consisting of the propositions with purely computaional content and the third consisting of propositions containing both logical and computaional content.

In view of the discussion of the example above it is an interesting question whether CC proves more propositions *about inductive data types* then higher order predicate logic does. It is clear that we have to be more precise if we want to have a negative answer, because in general the answer will be positive. (E.g. in CC we can still prove $\mathrm{EXT} \rightarrow \exists x{:}\mathrm{Nat}.Sx =_{\mathrm{Nat}} x$ (see the second proof of Proposition 4.12) and $\mathrm{Ind}_{\mathrm{Nat}}\&(Z \neq_{\mathrm{Nat}} SZ) \rightarrow \forall x,y{:}\mathrm{Nat}.(Sx =_{\mathrm{Nat}} Sy){\rightarrow}(x =_{\mathrm{Nat}} y)$ (see Example 3.3.)) First we have to consider only the strongest version of inductive data types, called *parametric data types* in [Berardi 199+]. A parametric data type is in set-theoretic terms the smallest set X closed under some fixed operators (functions of type $A_1{\rightarrow}A_2{\rightarrow}\ldots{\rightarrow}A_n{\rightarrow}X$, where $n \geq 0$ and each A_i is X or an already defined parametric data type.) If D is a parametric data type this implies that the induction and uniqueness properties for D are satisfied. In algebraic terms, a parametric data type is just a free (or initial) algebra. Further we have to restrict ourselves to a specific class of propositions, what Berardi calls the *propositions on functional types*. The functional types are the ones obtained by putting arrows between the types; further there are the so called logical types, which is the class of (higher order) predicate types on functional types. The

propositions on functional types are the propositions obtained from the basic propositions by the usual logical connectives $\supset, \vee\&, \neg, \forall_L$ and $\exists_L$, where L is a logical type. The *basic propositions* are those propositions obtained by applying an inductive predicate to the right number of terms (of the right type), so this class is already quite big. (Inductive predicates are minimal subsets among those closed under some fixed monotone constructors; they can be defined in higher order predicate logic by the higher order quantification over all such predicates. For example $\leq \subseteq$ Nat $\times$ Nat and $\in \subseteq$ Nat $\times$ List of the Examples in 3.4 are inductive predicates.) In [Berardi 199+] all this is defined in set-theoretic terms and then translated into CC. As is done there, we shall not denote this translation explicitly (but there are no ambiguities about this.)

The main result of [Berardi 199+] is now saying that for φ a proposition in the set Pos, if $\Gamma \vdash M{:}\varphi$ in CC for some term M, and Γ is satisfied in the model PER, then φ is provable in Set theory. Here PER is some model based on the interpretation of propositions of CC as partial equivalence realtions on Λ (the set of untyped lambda terms.) The model-construction is in [Berardi 199+]; we will not go into it here but state the important facts that for all parametric data type D, the interpretation of Ind_D in PER is not empty, which means that $z{:}\mathrm{Ind}_D$ is satisfied. The set of propositions Pos consists of those propositions on functional types that are built up from the basic propositions using $\supset, \vee, \&, \neg$ and $\forall x{:}D, \exists x{:}D$ (for D a parametric data type) with the restriction that a $\forall x{:}D$ that is not bound may only occur in a positive place. (The $\forall x{:}$Nat for example, is bound if it appears as $\forall x{:}\mathrm{Nat}.(\leq (x,n) \rightarrow \ldots)$.)

One of the obvious examples where the result applies is the first of 3.4. Berardi shows that also the statement of Girard's theorem, saying that all typable terms in system F are strongly normalizable. It is of the form

$$\forall t{:}Te.\forall A{:}Ty.\forall c{:}Co.\exists n{:}\mathrm{Nat}.\forall t'{:}Te.\forall m{:}\mathrm{Nat}.\mathrm{Redd}(t,t',m) \supset m \leq n,$$

where the type of pseudoterms Te, the type of types Ty and the type of contexts Co are parametric data types and $\mathrm{Redd} \subseteq Te \times Te \times \mathrm{Nat}$ is an inductive predicate with $Redd(t,t',m)$ if t reduces to t' in m steps. We see that the restrictions on the form of the propositions is not very serious; a specification will usually be of the form $\forall x{:}D.\exists y{:}D'.P(x,y)$ with $P(x,y) \in$ Pos. Further the result is very general, as there are no restrictions at all on the shape of Γ or M. So Γ may even contain assumptions that can not be expressed in set-theoretical terms: As long as the assumptions are satisfied in PER, the conclusion is valid.

It would be interesting to see whether the result discussed above can be rephrased syntactically by extending λPREDω with inductive data types and describing a formulas-as-types embedding from the extended higher order predicate logic to CC. This extension of λPREDω can be defined by adding a scheme for inductive types (by allowing a kind of least fixed point construction for positive type constructors), but also by extending λPREDω with polymorphic domains. As we know how to define inductive data types in polymorphic lambda calculus and the formulas-as-types embedding from λPREDω to CC immediately extends to λPREDω with polymorphic domains, we want to say a bit more about the latter possibility. Let λPREDω^p be the following Pure Type

System.

$$\begin{aligned}
\mathcal{S} &= \mathrm{Prop}, \mathrm{Set}, \mathrm{Type}^p, \mathrm{Type}^s, \\
\mathcal{A} &= \mathrm{Prop} : \mathrm{Type}^p \mathrm{Set} : \mathrm{Type}^s, \\
\mathcal{R} &= (\mathrm{Set}, \mathrm{Set}), (\mathrm{Type}^s, \mathrm{Set}), (\mathrm{Type}^p, \mathrm{Set}) \\
&= (\mathrm{Set}, \mathrm{Type}^p), (\mathrm{Type}^p, \mathrm{Type}^p), \\
&= (\mathrm{Prop}, \mathrm{Prop}), (\mathrm{Set}, \mathrm{Prop}), (\mathrm{Type}^p, \mathrm{Prop}).
\end{aligned}$$

(So this is λPREDω with $(\mathrm{Type}^s, \mathrm{Set})$: a higher order predicate logic built on the polymorphic lambda calculus in stead of the simple theory of types. In view of the description of parametric data types in the beginning of this section it is natural to leave the rule $(\mathrm{Type}^p, \mathrm{Set})$ out of the system to eliminate things like $\Pi\alpha{:}\mathrm{Set}.(\alpha{\to}\mathrm{Prop}){\to}\alpha : \mathrm{Set}$. This is an option that we want to leave open.)

The formulas-as-types embedding from λPREDω^p into CC is now induced by the formulas-as-types embedding from λPREDω into CC of Definition 4.1, so it is the PTS-morphism H with

$$\begin{aligned}
H(\mathrm{Prop}) &= \mathrm{Prop}, \\
H(\mathrm{Set}) &= \mathrm{Prop}, \\
H(\mathrm{Type}^p) &= \mathrm{Type}, \\
H(\mathrm{Type}^s) &= \mathrm{Type}.
\end{aligned}$$

This immediately shows that λPREDω^p is consistent. (In fact the mapping H shows that all extensions of λPREDω with rules of the form (s, s'), $s, s' \in \{\mathrm{Prop}, \mathrm{Set}, \mathrm{Type}^p, \mathrm{Type}^s\}$, are consistent.) The embedding H is not complete; the same counterexamples as for λPREDω do the job. However, if we restrict ourselves to propositions in the set Pos, we may still be able to prove that if $z_1{:}\mathrm{Ind}_{D_1}, \ldots, z_n{:}\mathrm{Ind}_{D_n}, a{:}\mathrm{Ind}_{\mathrm{Nat}}, b{:}Z \neq_{\mathrm{Nat}} SZ \vdash M : \varphi$ in CC, then there is a proof P in λPREDω^p with $z_1{:}\mathrm{Ind}_{D_1}, \ldots, z_n{:}\mathrm{Ind}_{D_n}, a{:}\mathrm{Ind}_{\mathrm{Nat}}, b{:}Z \neq_{\mathrm{Nat}} SZ \vdash P : \varphi$, where $D_1, \ldots, D_n$ are the parametric data types that occur in φ. (We omit the mapping H for reasons of readability.) In view of the proof of the original result in [Berardi 199+], we have a strong feeling that this adapted completeness of the formulas-as-types embedding from λPREDω^p into CC holds. However, it is not as general as the original result; one would like to allow more assumptions then just those stating the parametricity of the data types. Still we think that the matter is interesting for further investigations, because it may give a more syntactical handle as to which propositions about data types are provable in CC.

Let's end this section with a few remarks on the system λPREDω^p. As it is a higher order predicate logic built on 'polymorphic domains', it may be more readily understood then CC, where things are more interwoven. A straightforward semantics is given by an arbitrary model for the polymorphic lambda calculus (to interpret the Set-part) with a higher order predicate logic on top of it (giving the Prop-part the Tarskian semantics). It may then be more natural to do without the rule $(\mathrm{Type}^p, \mathrm{Set})$. An arbitrary model for the polymorphic lambda calculus has alot of specific structure and this may raise the question

whether the extension $\lambda\text{PRED}\omega^p$ is conservative over $\lambda\text{PRED}\omega$. We don't have a definite answer to this, but we do have reasons to believe that the extension is not conservative. The idea comes from an argument by Berardi, suggesting a possible method to show that the formulas-as-types embedding from second order predicate logic into second order dependent typed lambda calculus (λP2 in [Barendregt 1992]) is not complete. (The proofs of incompleteness for Proposition 4.12 also work to show the incompleteness of the formulas-as-types embedding from nth order predicate logic into nth order dependent typed lambda calculus, but only for for $n > 2$.) We look at the context

$$\Gamma := A{:}\text{Set}, a, a'{:}A, z{:}a \neq_A a'.$$

This context has a finite model (without going into details about models for higher order predicate logic, it will be clear that if we take for A the two element set, for $A{\to}A$ the set-theoretic function space, for $A{\to}\text{Prop}$ the set of subsets of A and so forth, this yields a model.) If we now look at a model for this context in $\lambda\text{PRED}\omega^p$, we see that there are a lot of (new) closed domains (types of type Set) which will have an interpretation in the model. For example the domain $\text{Nat} := \Pi\alpha{:}\text{Set}.\alpha{\to}(\alpha{\to}\alpha){\to}\alpha$. In the proof-irrelevance model of CC, Nat could consistently be interpreted by a one element set (because $Z \neq SZ$ isn't provable in CC in the empty context). However, here the interpretation of Nat has to be an infinite set, which makes it impossible for Γ to have a finite model in $\lambda\text{PRED}\omega^p$. The point is that from $a \neq a'$ one can prove $Z \neq_{\text{Nat}} SZ$ and hence $S^n(Z) \neq_{\text{Nat}} S^{n+1}(Z)$ (for n a natural number), viz. Suppose $Z =_{\text{Nat}} SZ$, then $ZAa(\lambda x{:}A.a') =_A SZAa(\lambda x{:}A.a')$ so $a =_A a'$, quod non. We want to stress here that we don't know how to use this fact (syntactically or semantically) to show the non-conservativity; it may still be possible that, although Γ has essentially only infinite models in $\lambda\text{PRED}\omega^p$, it still doesn't prove more $\lambda\text{PRED}\omega$-propositions then those provable in $\lambda\text{PRED}\omega$ from Γ.

6 Some extensions of the Calculus of Constructions

There are many ways in which CC has been extended, to capture a stronger notions of type, to capture an extended logic, for reasons of operational or denotational semantics or for implementational reasons. We also briefly discuss two extensions of higher order predicate logic that are of interest mainly because they are inconsistent. We feel this expopsition of CC would be incomplete if we wouldn't discuss some of these extensions or variations. The order in which we treat them is arbitrary.

6.1 Inconsistent extensions of higher order predicate logic

In the previous section we have extended higher order predicate logic with polymorphic domains by extending the system $\lambda\text{PRED}\omega$ with the rule $(\text{Type}^s, \text{Set})$ (which is consistent.) We may wonder whether we can do the same with λ**HOPL** by extending it with the rule $(\text{Type}', \text{Type})$. We call this system λU^-, in analogy with [Girard 1972], where the same system formulated as a logic is

called U^-. There it is also shown that the system λU, which is λU^- extended with the rule $(\mathrm{Type}^s, \mathrm{Prop})$ (quantification over the collection of all domains), is inconsistent, which was the original statement referred to as 'Girard's paradox'. (For details about the paradox see [Coquand 1986].) It is not so difficult to see that the extension of λ**HOPL** with only the rule $(\mathrm{Type}^s, \mathrm{Prop})$ is consistent, but only in [CC-documentation] it is shown by Coquand that the system λU^- is inconsistent, which was left as an open question in [Girard 1972]. The proof by Coquand is done by internalising Reynolds result about the non-existence of a set-theoretic model for the polymorphic lambda calculus. At first sight one may think that the inconsistency arises from the formalization of an easy cardinality argument like: $\Pi\alpha{:}\mathrm{Type}.\alpha{\to}\alpha$ is the collection of all functions from a 'small' set A to itself (in set-theoretic terms $\Pi_{A\in\mathrm{Type}}(A{\to}A)$.) This can not itself be a 'small' set as the cardinality of $\Pi_{A\in\mathrm{Type}}(A{\to}A)$ is larger then Type itself. Such an (intuitive) argument will not work, for one thing because it would also imply the inconsistency of $\lambda\mathrm{PRED}\omega^p$ of the previous section, but more importantly because the interpretation of $\Pi\alpha{:}\mathrm{Type}.\alpha{\to}\alpha$ is the set of functions from a type to itself that do *not make any specific assumption on the shape of the type*, which is much closer to the *intersection* of all function spaces from a type A to itself then the *union*.

6.2 Some extensions and variations for practical purposes

For an implementation of CC to use it as an interactive system for proof verification it is of course necessary to add some new mechanisms to the calculus. One point is how to represent variables (bound or free) in such a way that one doesn't have to take care of α-conversion. This may be solved by representing variables with De Bruijn indices ([de Bruijn 1980]). An extended exposition about this technique for the case of CC is in [Coquand and Huet 1988].

Another practical issue is how to introduce definitions: To release the burden of writing the same λ-term several times and for reasons of readability one wants to abbreviate terms by a smaller expression that can replace it. In the syntax this can be done by introducing some extra rules for introducing variables as abbreviations as follows (we follow [Pollack 1989].)

$$(defvar)\quad \frac{\Gamma \vdash M : A}{\Gamma, x = M \vdash x : A} \qquad \text{if } x \text{ doesn't occur free in } \Gamma$$

$$(def)\quad \frac{\Gamma, x = M \vdash N : B}{\Gamma, \langle x = M\rangle \vdash \langle x = M\rangle : \langle x = M\rangle B}$$

with the extra reduction rules

$$\begin{aligned} \langle x = M\rangle N &\longrightarrow_\delta N, \text{ if } x \notin (\mathrm{FV}(N)),\\ \langle x = M\rangle(N(x)) &\longrightarrow_\delta \langle x = M\rangle(N(M)). \end{aligned}$$

Here $N(x)$ denotes a term N with one specified free occurrence of x. The idea will be clear: $\langle x = M\rangle N$ represents the term N in which for every free occurrence of x one should read M. So definitions can be on the global level in

the context, but also purely local inside terms. From the informal reading of terms with definitions and the reduction rules we immediately get the criterion for typability of a term, relating the system to the original version of CC. To do this it is convenient to also describe a more general reduction rule on sequents:

$$\begin{array}{rcl} \Gamma_1, x = M, \Gamma_2 \vdash N : B & \longrightarrow_\Delta & \Gamma_1, \Gamma_2[M/x] \vdash N[M/x] : B[M/x], \\ \text{if } \Gamma \longrightarrow\!\!\!\!\!\rightarrow_\delta \Gamma', N \longrightarrow\!\!\!\!\!\rightarrow_\delta N', B \longrightarrow\!\!\!\!\!\rightarrow_\delta B' & & \text{(at least one of them nonempty) then} \\ \Gamma \vdash N : B & \longrightarrow_\Delta & \Gamma' \vdash N' : B'. \end{array}$$

Now if $\Gamma \vdash M : A$ is derivable in CC with definitions, then the Δ normal form of $\Gamma \vdash M : A$ exists and is derivable in CC. This shows the conservativity of the extension with definitions.

6.3 CC with equality, η-reduction or T-operator

For semantical reasons it is often inconvenient to describe a typed lambda calculus by starting from pseudoterms and especially to use the equality on the set of pseudoterms in the rules (as is done in the conversion rule.) The reason is that pseudoterms have no real meaning (even for the syntax) and so they are not intended to denote anything in the model. Therefore a more 'semantical' description of CC would have, in addition to the typing judgement, a (typed) equality judgement of the form $\Gamma \vdash M = M' : A$. The reasons for using the set of pseudoterms T in the syntax are of meta-theoretical nature: For T one can prove the Church-Rosser property quite easily (it is completely similar to the proof for the untyped lambda calculus), which implies the Church-Rosser property for a 'semantical' version of the system and at the same time proves that the two versions are equivalent. To get the picture clear we discuss the variant of CC with equality judgement and state the important properties. (The syntax is very close to the one given in [Scedrov 1990], where also a semantics for this system is discussed.)

Definition 6.1 *The system $CC_=$ is a typed lambda calculus with a typing judgement and an equality judgement. The typing rules are (axiom), (weakening), (variable), (Π-rule), (λ-rule), and (application) as for CC. (To denote that we are in $CC_=$ in stead of CC we write $\vdash_=$ in the rules.) The conversion rule of $CC_=$ is*

$$(conv_\beta\text{'}) \quad \frac{\Gamma \vdash_= M : A \quad \Gamma \vdash_= A = B : Prop/Type}{\Gamma \vdash_= M : B}$$

The judgement $\Gamma \vdash_= A = B : s$ is generated by

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

$$(\textit{eq-axiom})\qquad \frac{\Gamma \vdash_= M : A}{\Gamma \vdash_= M = M : A}$$

$$(\textit{sym})\qquad \frac{\Gamma \vdash_= M = N : A}{\Gamma \vdash_= N = M : A}$$

$$(\textit{trans})\qquad \frac{\Gamma \vdash_= M = N : A \quad \Gamma \vdash_= N = Q : A}{\Gamma \vdash_= M = Q : A}$$

$$(\Pi\text{-}\textit{eq})\qquad \frac{\Gamma \vdash_= A = A' : s \quad \Gamma, x{:}A \vdash_= B = B' : s'}{\Gamma \vdash_= \Pi x{:}A.B = \Pi x{:}A'.B' : s'} \textit{ for } s, s' \in \{\textit{Prop}, \textit{Type}\}$$

$$(\lambda\text{-}\textit{eq})\qquad \frac{\Gamma \vdash_= A = A' : s \quad \Gamma, x{:}A \vdash_= M = M' : B \quad \Gamma \vdash_= \Pi x{:}A.B : \textit{Prop}/\textit{Type}}{\Gamma \vdash_= \lambda x{:}A.M = \lambda x{:}A'.M' : \Pi x{:}A.B}$$

$$(\textit{app-eq})\qquad \frac{\Gamma \vdash_= M = M' : \Pi x{:}A.B \quad \Gamma \vdash_= N = N' : A}{\Gamma \vdash_= MN = M'N' : B[N/x]}$$

$$(\textit{conv-eq})\qquad \frac{\Gamma \vdash_= M = M' : A \quad \Gamma \vdash_= A = B : \textit{Prop}/\textit{Type}}{\Gamma \vdash_= M = M' : B}$$

In this version of the system the conversion rule can only be applied to two equal types if they are equal via a path through the typable terms. In the original PTS version the types only have to be equal as pseudoterms. The two versions are equivalent: If $M, M' \in \mathsf{TERM}$ with $M =_\beta M'$, then there is a path between them through TERM. (A proof uses CR for β-reduction on T and Subject Reduction.) This equivalence is expressed by the following theorem.

Theorem 6.2

$$\left.\begin{array}{r} \Gamma \vdash M : A \\ \Gamma \vdash M' : A \\ M =_\beta M \end{array}\right\} \Leftrightarrow \Gamma \vdash_= M = M' : A.$$

As a corollary to the theorem one finds the Church-Rosser property for the system $\mathrm{CC}_=$: If $\Gamma \vdash_= M = M' : A$, then there is an N with $\Gamma \vdash_= N : A$ and $M \longrightarrow\!\!\!\!\!\rightarrow_\beta N$, $M' \longrightarrow\!\!\!\!\!\rightarrow_\beta N$. The Church-Rosser property can be stated more 'semantically' by introducing a judgement for typed reduction, $\Gamma \vdash_= M \longrightarrow\!\!\!\!\!\rightarrow N : A$. The required equivalences follow easily from the Subject Reduction property.

We want to point out here that the equivalence implies that the system with equality on the pseudoterms is 'sound': There would really be something wrong if $\Gamma \vdash M, M' : A$ and $M =_\beta M$ as pseudoterms, without there being a path from M to M' through the collection of terms of type A in Γ.

A second extension which comes in quite naturally is the one with η-conversion. In almost all of the models the η rule ($\lambda x{:}A.Mx = M$ if $x \notin \mathrm{FV}(M)$) holds, so it is quite natural to consider the extension of the system with η. For the sematical version of the syntax ($\mathrm{CC}_=$ as defined in 6.1) this amounts to adding the following rule.

$$(\eta)\quad \frac{\Gamma, x{:}A \vdash_= Mx : B}{\Gamma \vdash_= \lambda x{:}A.Mx = Mx : \Pi x{:}A.B} \quad \text{if } x \notin \mathrm{FV}(M)$$

For our version of the system the extension with η means replacing the (conv_β) rule by the ($\mathrm{conv}_{\beta\eta}$) rule defined as follows.

$$(\mathrm{conv}_{\beta\eta})\quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : \mathrm{Prop/Type}}{\Gamma \vdash M : B} \quad \text{if } A =_{\beta\eta} B (\text{in } \mathsf{T})$$

To show the decidability of equality and the equivalence of the two systems it is now convenient to represent η-equality via a reduction rule

$$\lambda x{:}A.Mx \longrightarrow_\eta M \text{ (if } x \notin \mathrm{FV}(M)).$$

Then the decidability of equality follows from normalization of $\beta\eta$-reduction on typable terms and the Church-Rosser property. The equivalence of the two versions follows from the Church-Rosser property for typable terms in CC with ($\mathrm{conv}_{\beta\eta}$). However, with $\beta\eta$-reduction the Church-Rosser property on the pseudoterms T is invalid. (The well-known counterexample is due to [Nederpelt 1973]: For $A \neq_{\beta\eta} B$ and $x \notin \mathrm{FV}(M)$, $\lambda x{:}A.(\lambda y{:}B.My)x$ can be reduced by a β step and an η step to two terms that have no common reduct.) This complicates matters quite a lot because some meta theorems depend on the Church-Rosser property (normalization proofs usually require it.) Further it' not clear now how to prove the equivalence between the two versions of the system, which makes the syntactical system a bit suspect. (The most we may still hope for is that the Church-Rosser property holds for the set of typable terms of a fixed type in a fixed context; this suffices to prove the equivalence as in Theorem 6.2.) A discussion of and a solution to the problem of Church-Rosser (of $\beta\eta$-reduction) for CC with $\beta\eta$-conversion can be found in [Salvesen1991] and [Geuvers 1992], the first proving the property for the semantical version of CC and the second proving Church-Rosser for the syntactical version of CC (and hence the equivalence of the two versions.) Both proofs rely on the assumption that $\beta\eta$-reduction is normalizing and in both cases the proof is given for a large collection of Pure Type Systems. Strong normalization of $\beta\eta$-reduction for CC with $\beta\eta$-conversion can be proved by adapting the proof for the β-case in [Geuvers and Nederhof 1991].

In [Coquand 1990] and in [Streicher 1988] the syntax is built up more explicitly using a T-operator. This is done for semantical reasons (the latter therefore discusses even more explicit versions of the calculus), to be better able to describe the interpretation of the syntax in the model. The idea is to view Prop as a special base type and to put all the typing on the type level. The T comes in to lift propositions (terms of type Prop) to the type of its proofs, $T(\varphi)$. To be more precise the system has only one sort, Type, the known rules (weak), (var), (Π) (for types), (λ), (app) and (conv) and as the extra rules

$$\text{(lift)}\quad \frac{\Gamma \vdash A : \mathrm{Prop}}{\Gamma \vdash T(A) : \mathrm{Type}}$$

$$(\forall)\quad \frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x{:}A \vdash B : \mathrm{Prop}}{\Gamma \vdash \forall x{:}A.B : \mathrm{Prop}}$$

$$(\Lambda)\quad \frac{\Gamma \vdash A : \mathrm{Type} \;\; \Gamma, x{:}A \vdash M : T(B) \;\; \Gamma, x{:}A \vdash B : \mathrm{Prop}}{\Gamma \vdash \;\Lambda x{:}A.M : T(\forall x{:}A.B)}$$

$$(\mathrm{App})\quad \frac{\Gamma \vdash M : T(\forall x{:}A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash \mathrm{App}(M, N) : T(B[N/x])}$$

with additional β rules for redexes of the form $\mathrm{App}(\Lambda x{:}A.M, N)$. It is not difficult to define a mapping from the explicit syntax to the one we have been using here so far, that preserves the derivation rules. Similarly, a derivation in our version of CC can easily be translated to a derivation in the explicit system such that the mapping back yields the derivation we started with.

6.4 Further extensions

There are many other extensions and variations for CC in the literature that we want to discuss in some short detail. (Most of them are treated extensively in other texts.) First there is the system ECC of [Luo 1989], which is an extension of CC with strong Sigma types, universes and universe inclusion. The Sigma types are a kind of generalised sum types, $\Sigma x{:}A.B$ representing the type of pairs $\langle a, b\rangle$ with $b{:}B[a/x]$; there are projections $\pi_1 : (\Sigma x{:}A.B) \to A$ and $\pi_2 : \Pi z{:}(\Sigma x{:}A.B).B[\pi_1 z/x]$. (These projections distinguish the 'strong' Sigma types from weaker versions with different elimination rules.) The Sigma types are well-known to be useful for describing theories (see [Coquand 1990] for a discussion), especially in combination with universes (ECC has sorts (universes) Type_i for all natural numbers i with the axiom $\mathrm{Type}_i : \mathrm{Type}_{i+1}$ (the Type of CC is just Type_0) and further an inclusion rule for these universes: if $\Gamma \vdash A : \mathrm{Type}_i$ then $\Gamma \vdash A : \mathrm{Type}_{i+1}$ and similar for Prop and Type_0.) The theory of groups for example, can be denoted by $\Sigma A{:}\mathrm{Type}_0.\Sigma f{:}A{\to}A{\to}A.\Sigma e{:}A.\Sigma i{:}A{\to}A.\mathrm{groupax}(A, f, e, i)$, where $\mathrm{groupax}(A, e, f, i)$ is the group axiom for carrier A, group operation f, neutral element e, and inverse operation i. The 'theory of groups' is of type Type_1. It can't be of type Type_0, because this would lead to inconsistency of the system. Similarly one can't allow an impredicative Σ-type like $\Sigma\alpha{:}\mathrm{Prop}.\varphi : \mathrm{Prop}$. This means it is not possible, as can be done in the first order case, to represent the higher order existential quantification by a Σ-type. See [Coquand 1986] for a discussion on inconsistent extensions of CC, and [Harper and Pollack 1991] for a description of CC^ω, CC extended with universes and universe inclusion.

As a final extension of CC we want to point at the possibility of adding inductive types to the system. This can be useful because, although the system is very powerful from an extensional point of view (all recursive functions that are

provably total in higher order arithmetic can be represented on the polymorphic Church numerals), the system doesn't have such good intensional properties. For example, the recursion over data types has to be coded in terms of iteration. This does not only complicate matters quite a bit, but has as a consequence that the recursion equations only hold 'locally' and not 'globally'. That is, for the type of natural numbers, Nat, one can define a function Reccg for c : Nat and g : Nat→Nat→Nat such that Rec$cg(\underline{n+1}) = g\underline{n}$(Rec$cg(\underline{n})$) and Rec$cg(\underline{0}) = c$ (where $\underline{n}$ is the nth Church numeral), but not Rec$cg(\underline{Sx}) = g\underline{x}$(Rec$cg(\underline{x})$) for x a variable. Another consequence of this coding is that the algorithms that represent the recursive functions have a very bad evaluation behaviour. For example the λ-term that represents the predecessor computes $\underline{n}$ from $\underline{n+1}$ in anumber of steps of order n.

For these reasons there are several suggestions for extending CC with inductive types which yield for the natural numbers a recursor like the one in Gödels T. The problem of the inefficiency of recursion over data types already appears in system F and therefore a the suggested extensions to system F can immediately be adapted to CC. (For example the ones in [Mendler 1987] or [Parigot 1992].) An essentially different approach is taken in [Coquand and Mohring 1990], where inductive types as well as inductive predicates can be constructed by a scheme and the scheme not only allows to define functions by recursion, but also to do proofs by induction. The latter system is implemented as 'Coq'. (See [Dowek e.a. 1991].)

References

[Barendregt 1984] H.P. Barendregt, *The lambda calculus: its syntax and semantics*, revised edition. Studies in Logic and the Foundations of Mathematics, North Holland.

[Barendregt 199+] H.P. Barendregt, Typed lambda calculi. In Abramski et al. (eds.), *Handbook of Logic in Computer Science*, Oxford Univ. Press, to appear.

[Barendregt 1992] H.P. Barendregt, The cube of typed lambda calculi. This volume.

[van Benthem Jutting 199+] L.S. van Benthem Jutting, Typing in Pure Type Systems. To appear in *Information and Computation*.

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

[Berardi 1989] S. Berardi, Talk given at the 'Jumelage meeting on typed lambda calculus', Edinburgh, September 1989.

[Berardi 199+] S. Berardi, Encoding of data types in Pure Construction Calculus: a semantic justification. To appear in the Proceedings of the second BRA meeting on Logical Frameworks, Edinburgh, May 1991.

[Böhm and Berarducci 1985] C. Böhm and A. Berarducci, Automatic synthesis of typed Λ-programs on term algebras *Theor. Comput. Science*, 39, pp 135-154.

[de Bruijn 1980] N.G. de Bruijn, A survey of the project Automath, In J.P. Seldin, J.R. Hindley, eds. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, New York, pp 580-606.

[CC-documentation] The Calculus of Constructions, documentation and users guide, version 4.10, Technical report, INRIA, August 1989.

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

[Coquand 1985] Th. Coquand, Une théorie des constructions, Thèse de troisième cycle, Université Paris VII, France.

[Coquand 1986] Th. Coquand, An analysis of Girard's paradox, *Proceedings of the first symposium on Logic in Computer Science, Cambridge Mass.*, IEEE, pp 227-236.

[Coquand 1990] Th. Coquand, Metamathematical investigations of a calculus of constructions. In *Logic and Computer Science*, ed. P.G. Odifreddi, APIC series, vol. 31, Academic Press, pp 91-122.

[Coquand and Huet 1988] Th. Coquand and G. Huet, The calculus of constructions, *Information and Computation*, 76, pp 95-120.

[Coquand and Mohring 1990] Th. Coquand and Ch. Paulin-Mohring Inductively defined types, In P. Martin-Löf and G. Mints editors. *COLOG-88 : International conference on computer logic, LNCS 417.*

[Dowek e.a. 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin-Mohring, B. Werner, The Coq proof assistant version 5.6, user's guide. INRIA Rocquencourt - CNRS ENS Lyon.

[Geuvers 1989] J.H. Geuvers, Talk given at the 'Jumelage meeting on typed lambda calculus', Edinburgh, September 1989.

[Geuvers and Nederhof 1991] J.H. Geuvers and M.J. Nederhof, A modular proof of strong normalisation for the calculus of constructions. *Journal of Functional Programming*, vol 1 (2), pp 155-189.

[Geuvers 1992] J.H. Geuvers, 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.

[Gallier 1990] On Girard's "Candidats de Reductibilité". In *Logic and Computer Science*, ed. P.G. Odifreddi, APIC series, vol. 31, Academic Press, pp 123-204.

[Girard 1972] J.-Y. Girard, Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VII, France.

[Girard 1986] J.-Y. Girard, The system F of variable types, fifteen years later. TCS 45, pp 159-192.

[Girard et al. 1989] J.-Y. Girard, Y. Lafont and P. Taylor, *Proofs and types*, Camb. Tracts in Theoretical Computer Science 7, Cambridge University Press.

[Harper and Pollack 1991] R. Harper and R. Pollack, Type checking with universes, TCS 89, pp 107-136.

[Howard 1980] W.A. Howard, The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds. J.P. Seldin, J.R. Hindley, Academic Press, New York, pp 479-490.

[Hyland and Pitts 1988] The theory of constructions: categorical semantics and topos-theoretic models. In *Categories in computer science and logic, Proc. of the AMS Research Conference, Boulder, Col.*, eds. J.W. Gray and A.S. Scedrov, Contemporary Math., vol 92, AMS, pp 137-199.

[Lambek and Scott 1986] J. Lambek and P.J. Scott, *Introduction to higher order Categorical Logic*, Cambridge studies in advanced mathematics 7, Camb. Univ.Press.

[LEGO-examples] R. Pollack and others, Examples of formalised proofs in LEGO, Edinburgh.

[Luo 1989] Z. Luo, ECC: An extended Calculus of Constructions. *Proc. of the fourth ann. symp. on Logic in Comp. Science, Asilomar, Cal.* IEEE, pp 386-395.

[Martin-Löf 1971] P. Martin-Löf, A theory of types, manuscript, October 1971.

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

[Mendler 1987] N.P. Mendler, Inductive types and type constraints in second-order lambda calculus. *Proceedings of the Second Symposium of Logic in Computer Science.* Ithaca, N.Y., IEEE, pp 30-36.

[Mohring 1986] Ch. Mohring, Algorithm development in the calculus of constructions. In *Proceedings of the first symposium on Logic in Computer Science, Cambridge, Mass.* IEEE, pp 84-91.

[Nederpelt 1973] R.P. Nederpelt, Strong normalization in a typed lambda calculus with lambda structured types. Ph.D. thesis, Eindhoven Technological University, The Netherlands.

[Paulin 1989] Ch. Paulin-Mohring, Extraction des programmes dans le calcul des constructions, Thèse, Université Paris VII, France.

[Parigot 1992] M. Parigot, Recursive programming with proofs. *Theor. Comp. Science* 94, pp 335-356.

[Pollack 1989] R. Pollack, Talk given at the 'Jumelage meeting on typed lambda calculus', Edinburgh, September 1989.

[Pottinger 1989] G. Pottinger, Definite descriptions and excluded middle in the theory of constructions, TYPES network, November 1989.

[Salvesen1991] A. Salvesen, The Church-Rosser property for $\beta\eta$-reduction, manuscript.

[Scedrov 1990] A guide to polymorphic types. In *Logic and Computer Science*, ed. P.G. Odifreddi, APIC series, vol. 31, Academic Press, pp 387-420.

[Seldin 1990] J. Seldin, Excluded middle without definite descriptions in the theory of constructions, TYPES network, September 1990.

[Schütte 1977] K. Schütte, *Proof Theory*, Grundlehren der mathematischen Wissenschaften 225, Springer-Verlag.

[Streicher 1988] T. Streicher, Correctness and completeness of a categorical semantics of the calculus of constructions, Ph.D. Thesis, Passau University, Germany.

[Streicher 1991] T. Streicher, Independence of the induction principle and the axiom of choice in the pure calculus of constructions, TCS 103(2), pp 395 - 409.

[Tait 1965] W.W. Tait, Infinitely long terms of transfinite type. In *Formal Systems and Recursive Functions*, eds. J.N. Crossley and M.A.E. Dummett, North-Holland.

[Takeuti 1975] G. Takeuti, *Proof Theory*, Studies in Logic, vol. 81, North-Holland.

[Troelstra and Van Dalen 1988] A. Troelstra and D. van Dalen, *Constructivism in mathematics, an introduction, Volume I/II*, Studies in logic and the foundations of mathematics, vol 121 and volume 123, North-Holland.