Introduction to Pure Type Systems

Herman Geuvers*

Notes for the Types Summer School, Giens (F), 1999

1 Introduction

We introduce the notion of 'Pure Type System' by showing how logic (in particular higher order predicate logic) is interpreted as a typed λ-calculus, via the formulas-as-types embedding. Therefore we start with defining constructive higher order predicate logic precisely. Then we show how the formulas-as-types embedding gives us the associated type system for higher order predicate logic. Then we define the general notion of Pure Type System and show how typed λ-calculi.

Before discussing proof-checking in detail, we first want to define our logic. In a system for proof-checking we want to be able to interactively construct formal proofs. These formal proofs will be derivations in a system of formal logic. So, we have to set the logic that we will be working with. For the choice of logic there are various possibilities: first order, second order, higher order. It is also possible to choose between either classical or intuitionistic logic, or between natural deduction and sequent calculus.

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

In higher order predicate logic we have a set of *basic domains* B, for which we don't make any special assumptions, except for that B is countably infinite. Furthermore there is one special domain, Ω, the domain of propositions. One may think of Ω as the set of truth values (or booleans). However, it is more intuitive to think of Ω as the set of expressions that represent a formula. (In a standard model these expressions will of course be interpreted as a truth value, but for the syntax they are 'just' expressions.) As we

*Faculty of Mathematics and Computer Science, Eindhoven University of Technology and Faculty of Mathematics and Computer Science, University of Nijmegen, The Netherlands

are in higher order logic, we want to be able to form *functions domains* like $A \to A'$, for $A, A' \in B$, but also *higher order function domains* (often called 'function domains of *higher type*') like $(A \to A') \to A$. Functions in the latter domain take a function from A to A' and return an element of domain A. Finally we want to be able to form *predicate domains* like the domain of predicates on A or the domain of binary relations on A. It is more or less standard by now to identify a predicate on A with a function from A to Ω and similarly identify a binary relation on A with a (binary) function from A^2 to Ω. Hence, the domain of predicates on A is represented by $A \to \Omega$ and the domain of binary relations on A is represented by $A \to (A \to \Omega)$. In the representation of the domain of binary relations we avoid the use of product domains, but use a *curried* version of the binary function. To be able to construct expressions we use application and λ-abstraction.

2 Higher order predicate logic

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 constants $c_1^\sigma, c_2^\sigma, \ldots$ are in* TERM_σ,

   (b) *for each $\sigma \in D$, the variables $x_1^\sigma, x_2^\sigma, \ldots$ are in* TERM_σ,

   (c) *if $\varphi : \Omega$ and x^σ is a variable, then $\forall x^\sigma \varphi : \Omega$,*

   (d) *if $\varphi : \Omega$ and $\psi : \Omega$, then $\varphi \supset \psi : \Omega$,*

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

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

3. *The set of terms of* 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.

Remarks 2.2 1. *There are no 'product' domains $(D \times D)$ in our logic. We present functions of higher arity by* currying*: a binary function on D is represented as a term in the domain $D \to (D \to D)$.*

2. *A predicate is represented as a function to Ω, following the idea (probably due to Church) that a predicate can be seen as a function that takes a value as input and returns a formula. So, a binary relation over D is represented as a term in the domain $D\to(D\to\Omega)$. (If $R : D\to(D\to\Omega)$ and $t, q : D$, then $((Rt)q) : \Omega$.)*

3. *The logical connectives are just implication and universal quantification. Due to the fact that we have* higher order *universal quantification, we can express all other quantifiers using just $\supset$ and $\forall$. See 2.8.*

Notation 2.3 *We fix the following notational conventions. Outside brackets are omitted. In the domains we omit the brackets by letting them associate to the right, so $D\to D\to\Omega$ denotes $D\to(D\to\Omega)$. In terms we omit brackets by associating them to the left, so Rtq denotes $(Rt)q$. Note that in ordinary mathematics, this is usually written as $R(t, q)$.*

Examples 2.4 *Before giving the logical rules of λHOPL, we treat some examples of terms and formulas that can be written down in this language. Let the domains* nat *and A, the relation-constant $>: \mathsf{nat}\to\mathsf{nat}\to\Omega$, the relation-variables $R, Q : A\to A\to\Omega$ and the function-constants $0 : \mathsf{nat}$ and $S : \mathsf{nat}\to\mathsf{nat}$ be given.*

1. *The predicate 'being larger than 0' is expressed by the term $\lambda x^{\mathsf{nat}}.x > 0 : \mathsf{nat}\to\Omega$.*

2. *Induction over* nat *can be expressed by the (second order) formula* ind *defined as $\forall P^{\mathsf{nat}\to\Omega}(P0) \supset (\forall x^{\mathsf{nat}}(Px \supset P(Sx))) \supset \forall x^{\mathsf{nat}}Px$.*

3. *The formula $\mathsf{trans}(R)$, defined as $\forall x^A, y^A, z^A(Rxy\to Ryz\to Rxz)$ denotes the fact that R is transitive. So, as a matter of fact, $\mathsf{trans} : (A\to A\to\Omega)\to\Omega$.*

4. *The terms $\subseteq: (A\to A\to\Omega)\to(A\to A\to\Omega)\to\Omega$ is defined by $R \subseteq Q :=\forall x^A, y^A(Rxy \supset Qxy)$. (We informally use the infix notation $R \subseteq Q$ to denote $\subseteq RQ$.)*

5. *The term $\lambda x^A, y^A(\forall Q{:}A\to A\to\Omega(\mathsf{trans}(Q) \supset (R \subseteq Q) \supset Qxy)$ is of type $A\to A\to\Omega$. It denotes the transitive closure of R.*

Remark 2.5 *As was already noted by [Church 1940], the language of* HOPL *is just the simple typed λ-calculus. (As a matter of fact the main motivation for Church to introduce the simple typed λ-calculus was to give a precise description of the language of higher order predicate logic was; the definition of* HOPL *here is roughly Church's.) We can see this more clearly if we remove cases (c) and (d) in the Definition 2.1 and instead add the special constants*

$$\begin{array}{rcl}\supset & : & \Omega\to\Omega\to\Omega,\\ \forall_\sigma & : & (\sigma\to\Omega)\to\Omega,\ \textit{for } \sigma \in D.\end{array}$$

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

Definition 2.6 *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.*

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

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

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

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

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

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

Remark 2.7 *The rule (conversion) may look strange, but it is merely present to take care of the substitution process in predicates that are defined by* λ*-abstraction. An example of this situation occurs when we want to prove* $\forall y^{\mathsf{nat}}(y \geq 0)$ *from the induction axiom* ind *(see Example 2.4). We proceed by instantiating* ind *with the predicate* $(\lambda y^{\mathsf{nat}}.y \geq 0)$ *and then applying the conversion rule.(For reasons of space we omit the* $\mathsf{ind} \vdash$ *at the beginning of each line.)*

$$\frac{\forall P^{\mathsf{nat}\to\Omega}(P0) \supset (\forall x^{\mathsf{nat}}(Px \supset P(Sx))) \supset \forall x^{\mathsf{nat}} Px}{\dfrac{(\lambda y^{\mathsf{nat}}.y \geq 0)0 \supset (\forall x^{\mathsf{nat}}((\lambda y^{\mathsf{nat}}.y \geq 0)x \supset (\lambda y^{\mathsf{nat}}.y \geq 0)(Sx))) \supset \forall x^{\mathsf{nat}}(\lambda y^{\mathsf{nat}}.y \geq 0)Px}{(0 \geq 0 \supset (\forall x^{\mathsf{nat}}(x \geq 0) \supset (Sx \geq 0)) \supset \forall x^{\mathsf{nat}}(x \geq 0)}}$$

Examples 2.8 *A well-known fact about this logic is that the connectives* $\&, \vee, \bot, \neg$ *and* $\exists$ *are definable in terms of* $\supset$ *and* $\forall$*: for* $\varphi, \psi : \Omega$*, define*

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

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

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

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

It's a well-known fact that this equality is symmetric. The idea is to take $\lambda y^A.Qy \supset Qt$ for P. The precise proof is as follows. (The double horizontal line indicates an application of the ($\forall$-elim) rule followed by an application of the (conv) rule.)

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

3 Higher order typed λ calculus

In type theory one interprets formulas and proofs via the well-known 'formulas-as-types' and 'proofs-as-terms' embedding, originally due to Curry, Howard and De Bruijn. Under this interpretation, a formula is viewed as the type of its proofs. It turns out that one can define a typed λ-calculus λHOL that represents HOPL in a very precise way. What *very precise* means will be defined later. Here, we just define the system λHOL, using the intuitions of HOPL. In order to get a better understanding we note a few things.

1. The language of HOPL as presented in 2.1 is a typed language already. This language will be a part of λHOL

2. In λHOL, formulas like $\varphi \supset \psi$ and $\forall x{:}A.\varphi$ will become types. However, these 'propositional' types are not the same as the 'set' types like e.g. nat. Hence there will be two 'universes', Prop, containing the 'propositional' types, and Type, containing the 'set' types. Prop itself is a 'set' type.

3. The deductions are represented as typed λ-terms. The discharging of hypotheses is done by λ-abstraction. The modus ponens rule is interpreted via application.

The *derivable judgements* of λHOL are of the form

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

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

Definition 3.1 *The typed λ calculus λHOL, representing higher order predicate logic, is defined as follows. The set of* pseudoterms T *is defined by*

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

Here, V *is a set of variables. The set of* sorts, $\mathcal{S}$ *is* $\{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$. *A* context *is a sequence* $x_1{:}A_1, \ldots, x_n{:}A_n$, *where the* $\vec{x}$ *are in* V *and the* $\vec{A}$ *are in* T.

The typing rules, that select the well-typed *terms from the pseudoterms, are as follows. Here,* s *ranges over the set of sorts* $\mathcal{S}$.

$(axiom)$ $\quad \vdash \mathsf{Prop} : \mathsf{Type} \qquad\qquad \vdash \mathsf{Type} : \mathsf{Type}'$

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

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

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2} \qquad \textit{if } (s_1, s_2) \in \{ (\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop}) \}$$

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

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

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

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

We see that there is no distinction between types and terms in the sense that the types are formed first and then the terms are formed using the types.

Definition 3.2 *A pseudoterm* A *is* 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* λHOL *is denoted by* $\mathsf{TERM}(\lambda\mathsf{HOL})$.

The only type-forming operator in this language is the Π, which comes in three flavours, depending on the type of the *domain* (the A in $\Pi x{:}A.B$) and the type of the *range* (the

B in $\Pi x{:}A.B$). Intuitively, a Π-type should be read as a set of functions. If we depict the occurrences of x in B explicitly by writing $B(x)$, the intuition is:

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

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

Notation 3.3 *In case $x \notin \mathrm{FV}(B)$, we write $A{\to}B$ for $\Pi x{:}A.B$. We call this a* non-dependent *function type.*

By examples we list all instances of the Π-type that can be encountered in $\lambda\mathsf{HOL}$.

Examples 3.4 *1. Using the combination (*Type,Type*), we can form the function type $A{\to}B$ for $A, B{:}\mathsf{Type}$. This also comprises the types of unary predicates and binary relations: $A{\to}\mathsf{Prop}$ and $A{\to}A{\to}\mathsf{Prop}$. Furthermore, it also extends to higher order predicate types like $(A{\to}A{\to}\mathsf{Prop}){\to}\mathsf{Prop}$.*
If $\Gamma \vdash A{:}\mathsf{Type}$ and $\Gamma, x{:}A \vdash B{:}\mathsf{Type}$, then $x \notin \mathrm{FV}(B)$ in $\lambda\mathsf{HOL}$, so all types formed by (Type,Type*) are non-dependent function types.*

2. *Using the combination (*Prop,Prop*), we can form the propositional type $\varphi{\to}\psi$ for $\varphi, \psi{:}\mathsf{Prop}$. This is to be read as an* implicational formula.
If $\Gamma \vdash \varphi{:}\mathsf{Prop}$ and $\Gamma, x{:}\varphi \vdash \psi{:}\mathsf{Prop}$, then $x \notin \mathrm{FV}(\psi)$ in $\lambda\mathsf{HOL}$, so all types formed by (Prop,Prop*) are non-dependent types.*

3. *Using the combination (*Type,Prop*), we can form the dependent propositional type $\Pi x{:}A.\varphi$ for $A{:}\mathsf{Type}$, $\varphi{:}\mathsf{Prop}$. This is to be read as a* universally quantified formula.
This quantification can also range over higher order domains, like in $\Pi P{:}A{\to}A{\to}\mathsf{Prop}.\varphi$.
If $\Gamma \vdash A{:}\mathsf{Type}$ and $\Gamma, x{:}A \vdash \varphi{:}\mathsf{Prop}$, then it can *occur that $x \in \mathrm{FV}(\varphi)$ in $\lambda\mathsf{HOL}$.*

We will not define formal interpretations from HOPL to $\lambda\mathsf{HOL}$ and back. In the end of this section, we list some properties relating the two systems; here we motivate these properties by listing some examples of typing statements in $\lambda\mathsf{HOL}$.

Examples 3.5 *1.* $\mathsf{nat}{:}\mathsf{Type}, 0{:}\mathsf{nat}, >{:}\mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{Prop} \vdash \lambda x{:}\mathsf{nat}.x{>}0 : \mathsf{nat}{\to}\mathsf{Prop}$. *Here we see the use of λ-abstraction to define predicates.*

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

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

4. $A{:}\mathsf{Type} \vdash \lambda R, Q{:}A{\to}A{\to}\mathsf{Prop}.\Pi x, y{:}A.Rxy{\to}Qxy : (A{\to}A{\to}\mathsf{Prop}){\to}(A{\to}A{\to}\mathsf{Prop}){\to}\mathsf{Prop}$. *(Inclusion of relations)*

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

6. $A{:}\mathsf{Type}, x, y{:}A \vdash \lambda r{:}x =_A y.\lambda P{:}A{\to}\mathsf{Prop}.r(\lambda z{:}A.Pz \supset Px)(\lambda q{:}Px.q)$:
   $x =_A y{\to}y =_A x$. *The proof of the fact that Leibniz equality is symmetric.*

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

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

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

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

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

then these terms are of the right type. Hence the introduction and elimination rules for the connective & are definable. Similarly for the other connectives, the introduction and elimination rules can be defined.

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

4 Pure Type Systems

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

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

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

Definition 4.1 *For $\mathcal{S}$ a set, $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$, the* Pure Type System *$\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is the typed lambda calculus with the following deduction rules.*

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

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

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

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

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

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

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

If $s_2 \equiv s_3$ in a triple $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$. In these rules, the expressions are taken from the set of pseudoterms T, *defined by*

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

The pseudoterm A is 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* $\mathsf{TERM}(\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R}))$.

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

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

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

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

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

The system λPRED, as described above, captures quite a lot of first order predicate logic. As a matter of fact it precisely captures *minimal* first order predicate logic with *higher order functions.* The minimality means that there are only two connectives implication and first order universal quantification. As we are in a first order frameork, thye other connectives can not be defined. This makes the expressibility rather low, as one can not write down negative formulas. On the other hand, we do have higher order function types. Those can easily be removed by restricting the rule (Set, Set) as is done in the system λPRED^f:

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

Here, Fun represents the sort of *function types*, while Set represents the sort of basic set types. the only basic set types are variables in the context. Predicates can be formed over these basic set types only and quantification is only allowed over basic set types. The rule (Set, Set, Fun) allows the formation of $A \to A$: Fun and the rule (Set, Fun, Fun) allows the formation of function types of higher arity, like $A \to A \to A$: Fun.

To regain all connectives, λPRED can be extended to the second order system λPRED2, (where all connectives are definable):

λPRED2	
$\mathcal{S}$	Set, Type^s, Prop, Type
$\mathcal{A}$	Set, Type^s, Prop, Type
$\mathcal{R}$	(Set, Set), (Set, Type), (Prop, Prop), (Set, Prop), (Type, Prop)

Only the rule (Type, Prop). has been added. This rule allows quantification over domains of type Type, which are $A \to$Prop, $A \to A \to$Prop etcetera (with A : Set).

Higher order predicate logic (λPREDω) can be constructed out of λPRED2 as follows.

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

The addition of (Type, Type) implies that now $(A \to \mathsf{Prop}) \to \mathsf{Prop}$: Type and $((A \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$: Type as well. Quantification is over Type, which now includes all higher order domains.

Other well-known typed λ-calculi that can be described as a PTS are simple typed λ-calculus, polymorphic typed λ-calculus (also known as system F) and higher order typed λ-calculus (also known as Fω). They are described as follows.

$\lambda\to$	
$\mathcal{S}$	$\star, \Box$
$\mathcal{A}$	$\star, \Box$
$\mathcal{R}$	$(\star, \star)$

$\lambda 2$	
$\mathcal{S}$	$\star, \Box$
$\mathcal{A}$	$\star, \Box$
$\mathcal{R}$	$(\star, \star), (\Box, \star)$

$\lambda\omega$	
$\mathcal{S}$	$\star, \Box$
$\mathcal{A}$	$\star, \Box$
$\mathcal{R}$	$(\star, \star), (\Box, \star), (\Box, \Box)$

In $\lambda\to$, the only types are type variables and arrow-types and the only λ-abstractions are abstrcations over types ($\lambda x{:}\sigma.M$ with σ a type and $M : \tau$ for some type τ). It takes some effort to show that $\lambda\to$ is really equivalent to the ordinary simple typed λ-calculus. In $\lambda 2$, there are polymorphic types, like $\Pi\alpha{:}\star.\alpha\to\alpha$, besides the arrow types. It is also possible to abstract over the collection of all types $\star$: $\lambda\alpha{:}\star.\lambda x{:}\alpha.x : \Pi\alpha{:}\star.\alpha{\to}\alpha$. In the usual definition of $\lambda 2$ (or system F) the Π is denoted by $\forall$. In $\lambda\omega$ the possibility of forming higher order types, like $\star{\to}\star$ is added. Due to the $(\Box, \star)$ rule, abstraction can also be done over the higher order types.

The three systems presented above extend straightforwardly to a 'cube of typed λ-calculi', which contains eight different systems. It is defined as follows.

	$\star, \Box$
	$\star : \Box$
$\lambda{\to}$	$(\star,\star)$
$\lambda 2$	$(\star,\star), (\Box,\star)$
$\lambda\underline{\omega}$	$(\star,\star), (\Box,\Box)$
$\lambda\omega$	$(\star,\star), (\Box,\star), (\Box,\Box)$
λP	$(\star,\star), (\star,\Box)$
$\lambda P2$	$(\star,\star), (\star,\Box), (\Box,\star)$
$\lambda P\underline{\omega}$	$(\star,\star), (\star,\Box), (\Box,\Box)$
$\lambda P\omega$	$(\star,\star), (\star,\Box), (\Box,\star), (\Box,\Box)$

The system $\lambda P\omega$ is the Calculus of Constructions, sometimes called the *Pure* Calculus of Constructions to distinguish it from its variants and extensions. It is usually referred to as CC. The systems of the cube are presented as follows. An arrow denotes inclusion of one system in another.

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega\ (= \mathrm{CC}) \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

The systems containing the rule $(\star, \Box)$ are so called systems with *dependent types*, i.e. types can depend on terms (that inhabit other types). An example in λP is

$$A{:}\star, x{:}A, P{:}A{\to}\star \vdash Px{:}\star,$$

where the type Px contains a term (x) of type A. Note that the rule $(\star, \Box)$ is needed to form the 'kind' $A{\to}\star$. The system λP is the core of many Automath systems ([Nederpelt et al. 1994]) and of many of Martin Löf's type systems ([Martin-Löf 1984], [Nordström et al. 1990]). In the way it is defined as a PTS, it is very close to tye logical framework LF [Harper et al. 1987]. Even though λP looks not very powerful as a *logic*, compared to e.g. $\lambda\mathsf{HOL}$, it is very powerful as a *logical framework*. Many known logical systems (and formal systems in general) can be defined as a λP-context in a very flexible way. Here, we see a different approach to the formulas-as-types interpretation, originally due to de Bruijn [de Bruijn 1980], where the type system is used as the *meta language* in which to *define* your own logic. as an example we give the context that describes one-sorted minimal predicate logic. (We have added a relation symbol, a function symbol and a constant with some axioms for reasons of exposition.)

$$\begin{array}{rcl}\Gamma_{\mathrm{PRED}} & := & A:\star, \mathsf{prop}:\star, R:A{\to}A{\to}\mathsf{prop}, f:A{\to}A, c:A,\\ & & \supset: \mathsf{prop}{\to}\mathsf{prop}{\to}\mathsf{prop}, \forall:(A{\to}\mathsf{prop}){\to}\mathsf{prop},\\ & & T:\mathsf{prop}{\to}\star,\\ & & \supset -i : \Pi x,y{:}\mathsf{prop}.(Tx{\to}Ty){\to}T(x\supset y), \supset -e : \Pi x,y{:}\mathsf{prop}.T(x\supset y){\to}Tx{\to}Ty,\\ & & \forall - i : \Pi p{:}A{\to}\mathsf{prop}.(\Pi x{:}A.T(px)){\to}T(\forall p), \forall - e : \Pi p{:}A{\to}\mathsf{prop}.T(\forall p){\to}\Pi x{:}A.T(px),\\ & & a_1:\Pi x{:}A.T(Rx(fx)), a_2:\Pi x,y{:}A.T(Rxy){\to}T(R(fx)(fy)).\end{array}$$

5 Properties of Pure Type Systems

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

The first two propositions are very basic properties, saying that anything stated in a *correct* context is derivable (as one would hope) and that all free variables of a well-typed term are indeed declared in the context.

Here, Γ is a 'correct' context means that it is not just an arbitrary sequence of variables declared to pseudoterms. We define a context to be *correct* if $\Gamma \vdash M : A$ for some M and A.

Proposition 5.1 (Projection) *For* $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ *a* correct *context,* $\Gamma \vdash x_i : A_i$ *for all* $i \in \{1, \ldots n\}$.

Proposition 5.2 (Free variables) *For* $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ *a context and* $M, A \in \mathsf{T}$*, if* $\Gamma \vdash M : A$*, then* $\mathrm{FV}(M, A) \subseteq \{x_1, \ldots, x_n\}$.

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

Proposition 5.3 (Thinning) *For Γ a context, Γ' a* correct *context and M and A in T, if $\Gamma \vdash M : A$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash M : A$.*
Here, $\Gamma \subseteq \Gamma'$ denotes that al declarations that occur in Γ, also occur in Γ'.

Proposition 5.4 (Substitution) *For $\Gamma_1, x{:}B, \Gamma_2$ a context, and M, N and A in T, if $\Gamma_1, x{:}B, \Gamma_2 \vdash M : A$ and $\Gamma_1 \vdash N : B$, then $\Gamma_1, \Gamma_2[N/x] \vdash M[N/x] : A[N/x]$.*
Here $M[N/x]$ denotes the substirution of N for x in M, which is straightforwardly extended to contexts by substituting in all types in the declarations.

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

Proposition 5.5 (Strengthening) *For $\Gamma, x{:}B, \Gamma_2$ a context, and M and A in T, if $\Gamma_1, x{:}B, \Gamma_2 \vdash M : A$ and $x \notin \mathrm{FV}(\Gamma_2, M, A)$, then $\Gamma_1, \Gamma_2 \vdash M : A$.*

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

Proposition 5.6 (Subject Reduction) *For Γ a context and M, N and A in T, if $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash N : A$.*

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

Definition 5.7 *A PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is* functional, *also called* singly sorted, *if the relations A and R are functions, i.e. if the following two properties hold*

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

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

Proposition 5.8 (Unicity of Types) *For* functional *PTSs, if Γ is a context and M, A and B are in T, then, if $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.*

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

Definition 5.9 *For* $T = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ *and* $T' = \lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ *PTSs, a* PTS-morphism from T to T' *is a mapping* $f : \mathcal{S} \to \mathcal{S}'$ *that preserves the axioms and rules. That is, for every* $(s_1, s_2) \in \mathcal{A}$, $(f(s_1), f(s_2) \in \mathcal{A}'$ *and for every* $(s_1, s_2, s_3) \in \mathcal{R}$, $(f(s_1), f(s_2), f(s_3)) \in \mathcal{R}'$.

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

Proposition 5.10 *For* T *and* T' *PTS s and* f *a PTS-morphism from* T *to* T'*, if* $\Gamma \vdash M : A$ *in* T*, then* $f(\Gamma) \vdash f(M) : f(A)$ *in* T'.

An important property of a type system is *computability of types*, i.e. given Γ and M compute an A for which $\Gamma \vdash M : A$ holds, and if there is no such A, return 'false'. This is usually referred to as the *type synthesis problem*, TS. A related problem is *decidability of typing*, i.e. given Γ, M and A, decide whether $\Gamma \vdash M : A$ holds. This is usually referred to as the *type checking problem*, TC. Both problems are very much related, because in the process of type checking, one has to solve type synthesis problems as well: for example when *checking* whether $MN : C$, one will have to *synthesize* a type for N, say A, and a type for M, say D, and then to check whether for some B, $D =_\beta \Pi x{:}A.B$ with $B[N/x] =_\beta C$. Of course this is not a *proof* that decidability oft TC and decidability of TS are equivalent, but it should be clear from this example that in practice TS and TC are closely entwined. Furthermore it should be noted from this example that, for general PTSs, one will need an algorithm for *β-equality checking* to decide typing.

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

Confluence and Strong Normalization will be discussed in the next paragraph. Let us say a little bit more about decidability of TC and TS. Except for decidability of β-equality on types, one needs that the axioms ($\mathcal{A}$) and rules ($\mathcal{R}$) are decidable. (Obviously: given two sorts s_1 and s_2 one has to be able to decide whether $s_1 : s_2$.) But then still, designing a type checking algorithm and – especially – proving its correctness are not straightforward, because the derivation rules of PTSs are not *syntax directed*. (They would be *syntax directed* if given Γ and M there is at most one possible rule that applies to derive $\Gamma \vdash M : \_$.) General TC algorithms are defined (and proved correct) for the class of *functional* normalizing PTSs with decidable $\mathcal{A}$ and $\mathcal{R}$. See [van Benthem Jutting et al. 1993, Pollack 1995, Severi 1998] for more details.

Proposition 5.11 (Confluence) *On the set of pseudoterms* T*,* β*-reduction is* confluent *i.e. for all* $M, N_1, N_2 \in \mathsf{T}$*, if* $M \twoheadrightarrow_\beta N_1$ *and* $M \twoheadrightarrow_\beta N_2$*, then there exists a* $P \in \mathsf{T}$ *such that* $N_1 \twoheadrightarrow_\beta P$ *and* $N_2 \twoheadrightarrow_\beta P$.

Confluence for β can be proved by following the well-known proofs for confluence for the untyped λ-calculus. The following is a consequence of conflunece on T and Subject Reduction.

Corollary 5.12 (Confluence on well-typed terms) *For $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ a PTS, β-reduction is confluent on* $\mathsf{TERM}(\lambda(\mathcal{S},\mathcal{A},\mathcal{R}))$.

Definition 5.13 *A PTS $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ is* Strongly Normalizing *if for all $M \in \mathsf{TERM}(\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$, M is Strongly Normalizing, i.e. there are no infinite β-reduction paths starting from M. (Put differently: all reductions starting from a typable term terminate.)*

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

Theorem 5.14 *The Calculus of Constructions, CC, (the system $\lambda P\omega$ in the cube) is Strongly Normalizing.*

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

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

Corollary 5.15 *The following PTSs are all Strongly Normalizing. $\lambda{\to}$, $\lambda 2$, $\lambda\underline{\omega}$, $\lambda\omega$, λP, $\lambda P2$, $\lambda P\underline{\omega}$, λPRED, λPRED^f λPRED2 λPREDω.*

A well-known example of a PTS that is not Strongly Normalizing is $\lambda\star$.

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

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

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

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

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

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

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

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

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

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

By easy induction one proves that g preserves derivations. Furthermore, $f(g(\Gamma \vdash M : A)) = \Gamma \vdash M : A$ and $g(f(\Gamma \vdash M : A)) =$ '$\Gamma \vdash M : A$. Hence, $\lambda P\omega$ and $\lambda\mathsf{HOL}$ are equivalent systems.

This equivalence also implies that the system $\lambda\mathsf{HOL}$ is Strongly Normalizing.

6 Extensions to Pure Type Systems

Several features are not present in PTSs. For example, it is possible to define data types (in a system containing $\lambda 2$), but one does not get induction over these data types for free. (It is possible to define functions by recursion, but induction has to be assumed as an axiom.) Therefore, 'inductive types' have been added as an extra feature. The way we present them below, they were first defined by Coquand and Mohring in [Coquand and Mohring 1990]. Inductive types are present in all widely used type-theoretic theorem provers, like Coq, Lego and ALF.

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

6.1 Products and Sums

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

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

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

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

with the computation rule

$$\pi_i(\langle t_1, t_2\rangle) \longrightarrow t_i.$$

In presence of polymorphism, these constructions are all definable. For example in the Calculus of Constructions we have products in the sort $\star$. Let $A_1, A_2 : \star$ and define

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

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

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

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

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

with the computation rule

$$\mathsf{case}(f_1, f_2)(inj_i p) \longrightarrow f_i p.$$

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

$$\begin{aligned}
A_1 + A_2 &:= \Pi\alpha{:}\star.(A_1{\to}\alpha){\to}(A_2{\to}\alpha){\to}\alpha,\\
\mathsf{in}_{\,1} &:= \lambda p{:}A_1.\lambda\alpha{:}\star.\lambda h_1{:}(A_1{\to}\alpha).\lambda h_2{:}(A_2{\to}\alpha).h_1 p,\\
\mathsf{in}_{\,2} &:= \lambda p{:}A_2.\lambda\alpha{:}\star.\lambda h_1{:}(A_1{\to}\alpha).\lambda h_2{:}(A_2{\to}\alpha).h_2 p,\\
\mathsf{case}(f_1, f_2) &:= \lambda x{:}(A_1 + A_2).xCf_1 f_2.
\end{aligned}$$

A straightforward way to have product and sum types in the system $\lambda\mathsf{HOL}$ is by extending $\lambda\mathsf{HOL}$ with the rule $(\mathsf{Type}', \mathsf{Type})$, so that Type is a polymorphic domain. This system is known as λU^-.

λU^-	
$\mathcal{S}$	$\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$
$\mathcal{A}$	$\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$
$\mathcal{R}$	$(\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Type}), (\mathsf{Type}', \mathsf{Type}), (\mathsf{Type}, \mathsf{Prop})$

The system λU^- is a restriction of the systems λU, which has additional feature quantification over the sort Type.

λU	
$\mathcal{S}$	$\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$
$\mathcal{A}$	$\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$
$\mathcal{R}$	$(\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Type}), (\mathsf{Type}', \mathsf{Type}), (\mathsf{Type}', \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})$

In λU one can write down, e.g. $\Pi A{:}\mathsf{Type}.\Pi P{:}A{\to}\mathsf{Prop}.\Pi x{:}A.Px{\to}Px$, which is of type Prop.

However, both λU and λU^- are logically inconsistent. For λU, this was first proven by Girard in [Girard 1972]. This became known as the famous Girard's Paradox. For λU^-, the inconsistency was first proven by Coquand [Coquand 199+]. A short proof of the inconsistency of both systems can be found in [Hurkens 1995].

It should be noted here, that it is possible to have some form of polymorphic domains in higher order predicate logic, but only if we restrict ourselves to polymorphism over 'basic domains' (excluding Prop as a domain). This can be achieved by extending $\lambda\mathrm{PRED}\omega$ with the rule $(\mathsf{Type}^s, \mathsf{Set})$ (and also $(\mathsf{Type}^s, \mathsf{Prop})$ if one would like). This system is consistent, because the PTS-morphism from $\lambda P\omega$ to CC extends immediately to these extra rules.

6.2 Σ-types

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

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

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

$$\Pi x, y, z{:}A.(x \circ y) \circ z \ = \ x \circ (y \circ z),$$

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

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

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

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

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

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

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

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

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

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

These rules allow the formation of the 'dependent tuples' we need for formalizing notions like Group and Ring. An even more general approach towards the theoey of groups would be to also abstract over the carrier type, obtaining

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

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

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

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

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

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

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

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 1992] H.P. Barendregt, Lambda calculi with Types. In *Handbook of Logic in Computer Science*, eds. Abramski et al., Oxford Univ. Press, pp. 117 – 309.

[van Benthem Jutting 1993] L.S. van Benthem Jutting, Typing in Pure Type Systems. In *Information and Computation* Vol. 105-1, pp. 30–41.

[van Benthem Jutting et al. 1993] L.S. van Benthem Jutting, J. McKinna and R. Pollack, Checking algorithms for Pure Type Systems. In *Types for Proofs and Programs*, eds. H. Barendregt and T. Nipkow, LNCS 806.

[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 1990] S. Berardi, Type dependence and constructive mathematics, Ph.D. thesis, Universita di Torino, Italy.

[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.

[Boyer and Moore 1988] R.S. Boyer and J.S. Moore, *A Computational Logic Handbook*. Academic Press, Boston.

[de Bruijn 1980] N.G. de Bruijn, A survey of the project Automath, 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 580-606.

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

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

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

[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.

[Coquand 199+] Th. Coquand, A new paradox in type theory, to appear in *Proceedings of the 9th International Congress of Logic, Methodology and Philosophy of Science, Uppsala, Sweden 1991.*

[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.*

[van Dalen 1983] D. van Dalen, *Logic and Structure*, second edition. Springer Verlag.

[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.

[Girard 1971] J.-Y. Girard, Une extension de l'interprétation fonctionnelle de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. *Proceedings of the Second Scandinavian Logic Symposium*, ed. J.E. Fenstad, North-Holland.

[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 et al. 1987] R. Harper, F. Honsell and G. Plotkin, A framework for defining logics. *Proceedings Second Symposium on Logic in Computer Science*, (Ithaca, N.Y.), IEEE, Washington DC, pp 194-204.

[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.

[Hurkens 1995] T. Hurkens, A simplification of Girard's Paradox. TLCA'95, LNCS 902.

[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.

[Longo and Moggi 1988] G. Longo and E. Moggi, Constructive Natural Deduction and its "Modest" Interpretation. Report CMU-CS-88-131.

[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.

[Luo and Pollack 1992] Z. Luo, R. Pollack, Lego proof development system: User's Manual, Dept. of Computer Science, University of Edinburgh, April 1992.

[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.

[Nederpelt et al. 1994] R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer (editors), *Selected Papers on Automath*, Volume 133 in Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1994, pp 1024.

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

[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.

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

[Pollack 1995] R. Pollack, A verified type checker, In *TLCA'95*, eds. M. Dezani and G. Plotkin, LNCS 902, pp. 365 – 380.

[Severi 1998] P. Severi, Type inference for Pure Type Systems, *Information and Computation vol. 143 no. 1, 1998*, pp. 1 – 23.

[Terlouw 1989a] J. Terlouw, Een nadere bewijstheoretische analyse van GSTT's (incl. appendix), Manuscript, Faculty of Mathematics and Computer Science, University of Nijmegen, Netherlands, March, April 1989. (In Dutch)