Some logical and syntactical observations concerning the first order dependent type system $\lambda P$

HERMAN GEUVERS$^1$ and ERIK BARENDSEN$^2$

$^1$ Computing Science Department, Eindhoven Univ of Technology, NL
$^1,^2$ Computing Science Department, Nijmegen University, NL

Received 31 March 1999

We look at two different ways of interpreting logic into the dependent type system $\lambda P$. The first is by a direct formulas-as-types interpretation à la Howard where the logical derivation rules are mapped to derivation rules in the type system. The second is by viewing $\lambda P$ as a Logical Framework following (Harper et al. 1987) and (Harper et al. 1993). Then the type system is used as the meta-language in which various logics can be coded.

We give a (brief) overview of known (syntactical) results about $\lambda P$. Then we discuss two issues in some more detail. The first is the completeness of the formulas-as-types embedding of minimal first-order predicate logic into $\lambda P$. This is a remarkably complicated issue, a first proof of which appeared in (Geuvers 1993), following ideas in (Barendsen and Geuvers 1989) and (Swaen 1989). The second issue is the minimality of $\lambda P$ as a logical framework. We will show that some of the rules are actually superfluous (even though they contribute nicely to the generality of presentation of $\lambda P$).

At the same time we will attempt to provide a gentle introduction to $\lambda P$ and its various aspects and we will try to use little inside knowledge.

1. Introduction and motivation

The typed $\lambda$-calculus $\lambda P$ is the extension of simple typed $\lambda$-calculus with (first order) dependent types. It occurs in different variants in the work of (Martin-Löf 1975) (intuitionistic type theory), (van Daalen 1973) (Automath), (Harper et al. 1987) (Logical Framework) and also in (Hindley and Seldin 1986) (Generalized Type Assignment). The precise definition that we will be using is the one in (Barendregt 1992), which is – of the aforementioned – closest to (Harper et al. 1987). In the literature we find several ways of motivating the definition of $\lambda P$ and explaining its use. These can be devided in two: $\lambda P$ can be seen as a system for interpreting minimal first order predicate logic and $\lambda P$ can be seen as a logical framework. In both views one uses a kind of formulas-as-types interpretation to interpret (in the case of a logical framework one would say ‘encode’) the logic, the difference being that in the first case there is one logic, the internal logic of $\lambda P$, whereas in the second case almost any formal system can be encoded. We will not
give a complete overview of the different possible interpretations, but instead motivate
the definition of AP by explaining the two interpretations by examples.

Using AP to represent logic, either by a direct interpretation or an encoding, raises a
major question, namely: is the interpretation complete? Or, if one takes the view of AP
as a logical framework: is the encoding adequate? If we take the point of view of a direct
encoding of minimal first order predicate logic, L, into AP, then the question would be
whether the implication

$$\Gamma_\Sigma \vdash_{AP} M : \varphi \Rightarrow \vdash_L \varphi$$

holds for all formulas \( \varphi \) of minimal first order predicate logic. (Here \( \Gamma_\Sigma \) represents
the context that declares the constants of the first order signature \( \Sigma; \varphi \) is a formula over this
signature \( \Sigma \).) So the question is whether, if \( \varphi \), considered as a type in AP, is inhabited,
then \( \varphi \) is derivable in L. The same question obviously applies to AP seen as a logical
framework, with the difference that there is not just one logic, but that for every logic
L we have to define a context \( \Gamma_L \) that codes the logic. How this works in detail will be
discussed later by treating some examples.

One may wonder whether the soundness of the interpretation is not an issue. Well, it
is an issue: we have to prove that

$$\vdash_L \varphi \Rightarrow \exists M [\Gamma_\Sigma \vdash_{AP} M : \varphi]$$

holds for all formulas \( \varphi \) of minimal first order predicate logic L. However, soundness
is not a major issue, because it is easily satisfied. Also for the logical framework view,
soundness is usually relatively easy: it boils down to choosing the ‘correct’ \( \Gamma_L \) as an
encoding of the logic L.

The question of adequacy of the encoding of a logic L into AP, as a logical framework,
was first dealt with by (Harper et al. 1987). (A full version of this paper has appeared as
(Harper et al. 1993.) As a matter of fact, they were the first to actually state the problem.
To prove adequacy of an encoding (Harper et al. 1987) devise a general technique that
applies to many different logics L. The idea is to construct, out of a proof term \( M : \varphi \),
a canonical proof term \( M' \) (technically: the so called \( \eta \)-\( \beta \)-normal form of \( M \)). From
such a canonical proof term a proof of \( \varphi \) in the logic L is immediately constructed. We
will illustrate this technique briefly in Section 4 by an example. (For proving adequacy
of the encoding it is convenient to extend AP with \( \eta \)-conversion. This yields the actual
type system of LF as defined in (Harper et al. 1987). However, for the adequacy result
this is not needed – as we will also argue in 4 – because AP is a subsystem of the LF
type system.) In Section 4 we will also show how we can define a minimal version of AP
that can serve as a logical framework.

The question of completeness of the interpretation of minimal first order predicate logic
into AP will be treated in Section 3. This issue was already raised by Martin-Löf in the
seventies. A proof of completeness was first sketched in (Barendsen and Geuvers 1989).
A precise proof – based on this proof – occurs in (Geuvers 1993). We give it in Section 3
with some more explanation and examples. Independently, (Berardi 1990) proved the
same completeness result. It’s maybe most remarkable that the completeness is such an
intricate problem. To grasp this we have to understand how exactly minimal first order
First order dependent type theory, λP

Predicate logic is interpreted in λP. This is done by interpreting both sets and formulas as types. A predicate P on a set A is then interpreted as a function from (the type) A to the collection of all types, type, so P : A → type. From this one constructs e.g. the type \( \Pi x:A.Px \to Px \), representing the formula \( \forall x:a.Px \to Px \), but one can also construct types like \( \Pi x:A.Px \to A \), which does not represent any set or formula of the logic. In the completeness proof one has to take care of all these ‘meaningless’ types and it is not at all clear whether these meaningless types can somehow spoil the completeness.

2. The system λP

We begin by defining the system λP. Then we give some examples of well-typed terms and list some of the general issues (and properties) of type systems. Finally we give a brief list of some meta-theoretic properties.

**Definition 2.1.** λP (Harper et al. 1987) is a system for deriving judgements of the following two forms

\[
\Gamma \vdash M : B \quad M \text{ is of type } B \text{ in context } \Gamma, \\
\Gamma \vdash \quad \Gamma \text{ is a correct context.}
\]

Here Γ is called the context and M and B are terms, which are taken from the set of pseudoterms

\[
\Gamma := \text{Var} | \text{type} | \text{kind} | (\Pi \text{T:T}) | (\lambda x:T.T) | (\Pi x:T.T).
\]

The derivation rules for deriving the judgements \( \Gamma \vdash M : B \) and \( \Gamma \vdash \) are the following. (s ranges over \{type, kind\}.)

- **(base)** \( \emptyset \vdash \)

- **(ctx)** \( \frac{\Gamma \vdash A : s \quad \text{if } x \text{ not in } \Gamma}{\Gamma, x:A \vdash} \)

- **(ax)** \( \frac{\Gamma \vdash \text{type} : \text{kind}}{\Gamma \vdash} \)

- **(proj)** \( \frac{\Gamma \vdash x : A \quad \text{if } x : A \in \Gamma}{\Gamma \vdash} \)

- **(Π)** \( \frac{\Gamma \vdash A : \text{type} \quad \Gamma, x:A \vdash B : s}{\Gamma \vdash \Pi x:A.B : s} \)

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

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

- **(conv)** \( \frac{\Gamma \vdash M : B \quad \Gamma \vdash A : s \quad A =_\beta B}{\Gamma \vdash M : A} \)

As usual, we omit brackets in sequences of applications by associating them to the left and we omit brackets in sequences of abstraction terms by associating them to the right. So \( MN P \) denotes \((MN)P\), \( \lambda x:A.\lambda y:B.M \) denotes \( \lambda x:A.(\lambda y:B.M) \) and \( \Pi x:A.\Pi y:B.C \) denotes \( (\Pi x:A.\Pi y:B.C) \).

We use the convention of writing \( A \to B \) for \( \Pi x:A.B \) if \( x \notin \text{FV}(B) \).
In (Harper et al. 1987), a system called LF is defined (‘Logical Framework’), specifically geared towards the coding of logics (and formal systems in general) in this typed \( \lambda \)-calculus. The system \( \lambda \mathcal{P} \) can be seen as the ‘basic underlying type system’ of LF. In the definition of \( \lambda \mathcal{P} \) we have ignored some features of LF, the main one being the use of so called ‘signatures’. These are special contexts in which constants are declared. In our definition a signature is part of the context. Another difference is that we do not consider \( \eta \)-conversion. In LF, \( \eta \)-conversion is used for proving the adequacy of the encoding. However, we don’t need \( \eta \)-conversion for adequacy, as will be argued in Section 4.

The system \( \lambda \mathcal{P} \) is an extension of the simply typed \( \lambda \)-calculus \( \lambda \rightarrow \). This is not entirely obvious, as the types (and contexts) in \( \lambda \rightarrow \) are usually defined separately from the terms, whereas in \( \lambda \mathcal{P} \) these definitions are interwoven. We therefore treat some examples of well-typed terms in \( \lambda \rightarrow \).

**Example 2.2.**

1. \( \alpha : \text{type, } \beta : \text{type, } \gamma : \text{type} \vdash \lambda x : \alpha \rightarrow \beta \rightarrow \gamma, \lambda z : \alpha, \lambda x. x(zy) : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma. \)

2. \( \alpha : \text{type, } \beta : \text{type, } y : \beta \vdash \lambda x : (\alpha \rightarrow \beta) \rightarrow \alpha, x(\lambda z : \alpha. y) : ((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha. \)

It is well-known that \( \lambda \rightarrow \) is isomorphic with minimal propositional logic (logic with just implication) via the formulas-as-types embedding. In the example above, the first \( \lambda \)-term represents a proof (natural deduction derivation) of \( (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \), whereas the second represents a proof of \( ((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha \) from the assumption \( \beta \).

In a similar fashion one can interpret in \( \lambda \mathcal{P} \) minimal first order predicate logic (logic with just implication and universal quantification). To be able to do this we have to follow one basic principle:

A formula is associated with the type of its proofs, hence a formula is provable if and only if the associated type is not empty (‘inhabited’).

As a consequence, we associate with a predicate over the set \( \{ \text{type} \} A \) a term of type \( A \rightarrow \text{type} \), the idea being that for \( a : A \),

\( Pa \) holds if and only if the type \( Pa \) is inhabited.

This amounts to an interpretation of minimal predicate logic in \( \lambda \mathcal{P} \) where both sets and formulas are interpreted as types. We will define this interpretation precisely later and restrict to some motivating examples now.

**Example 2.3.**

1. \( \alpha : \text{type, } P : \alpha \rightarrow \text{type} \vdash \lambda x : \alpha. \lambda y : P x. (\Pi x : \alpha. P x) \rightarrow P x. \)

2. \( \alpha : \text{type, } f : \alpha \rightarrow \alpha, R : \alpha \rightarrow \alpha \rightarrow \text{type}, \)

\( h_1 : \Pi x : \alpha. Rx(f(x)), h_2 : \Pi x, y, z : \alpha. (Rxy) \rightarrow (Ryz) \rightarrow (Rxz) \vdash \)

\( \lambda x : \alpha. h_2(x, f(x))(h_1(x))(h_1(f(x)) : \Pi x : \alpha. Rx(f(x)). \)

In a predicate logical interpretation, the first term is a proof of \( \forall x \in A. P(x) \rightarrow P(x) \) and the second is a proof of \( \forall x \in A. R(x, y) \rightarrow R(x, z) \rightarrow R(y, z) \) from the hypotheses \( \forall x \in A. R(x, f(x)) \) and \( \forall x, y, z \in A. R(x, y) \rightarrow R(y, z) \rightarrow R(x, z). \)

As a third type of example we treat a coding of minimal propositional logic in \( \lambda \mathcal{P} \). The idea is to declare a type \( \text{prop} \) in the context to represent the type of (names) of
propositions and to add a $T : \text{prop} \rightarrow \text{type}$, taking a name of a proposition to the type of its proofs. (By adding suitable declarations for the derivations we establish that for $\alpha : \text{prop}$, the terms of type $Ta$ indeed represent natural deduction proofs.)

**Example 2.4.** Define

$$
\Gamma := \text{prop} : \text{type}. T : \text{prop} \rightarrow \text{type}, \\
\vdash: \text{prop} \rightarrow \text{prop} \rightarrow \text{prop}, \\
\vdash t : \Pi x, y : \text{prop}. (Tx \rightarrow Ty) \rightarrow T(\forall xy), \\
\vdash e : \Pi x, y : \text{prop}. T(\forall xy) \rightarrow Tx \rightarrow Ty.
$$

Then we have the following typings.

1. $\Gamma, x : \text{prop} \vdash t(\forall xx(\lambda p : Tx.p)) : T(\forall xx)$.
   This term codes a proof of `$x \rightarrow x$'.

2. For the purpose of presentation we write $\forall xy$ as $x \forall y$ and we omit the first two arguments of $\vdash t$ and $\vdash e$. We then find

   $\Gamma, x, y : \text{prop}, h : Ty \vdash t(\lambda p : T((x \forall y) \forall x) \vdash e(\forall x (\lambda q : Tx.h))) : T((x \forall y) \forall x) \forall x$.

   This term codes a derivation of $((x \rightarrow y) \rightarrow x) \rightarrow x$ from the hypothesis $y$.

2.1. Properties of $\lambda \Pi$

In the examples above we saw that the general use of $\lambda \Pi$ is to code (or represent directly) either terms (programs) or derivations (proofs). The $\beta$-reduction relation then corresponds to evaluation (of programs) or cut-elimination (of proofs). Important and natural properties to have are then that the typing is preserved by evaluation, that evaluation is confluent and that typing is decidable.

We list the main properties of the typed $\lambda$-calculus $\lambda \Pi$. Proofs can be found in (Harper et al. 1987), (Barendregt 1992) or in (Geuvers and Nederhof 1991). (The proofs in (Harper et al. 1987) are for $\lambda \Pi$ with $\eta$; the proofs here are roughly the same, sometimes a bit simpler due to the absence of $\eta$.)

**Proposition 2.5.** *(Subject Reduction)* If $\Gamma \vdash M : A$ and $M \rightarrow_\beta N$, then $\Gamma \vdash N : A$.

The following Proposition follows from Subject Reduction and the fact that $\beta$-reduction is confluent on the pseudo-terms $T$.

**Proposition 2.6.** *(Typed Confluence)* If $\Gamma \vdash M : A$, $\Gamma \vdash N : A$ and $M \equiv_\beta N$, then there is a term $P$ with $M \rightarrow_\beta P, N \rightarrow_\beta P$ and $\Gamma \vdash P : A$.

**Proposition 2.7.** *(Uniqueness of Types)* If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A \equiv_\beta B$.

**Proposition 2.8.** *(Strong Normalization)* All well-typed expressions of $\lambda \Pi$ are Strongly Normalizing with respect to $\beta$-reduction.

This Proposition was first proved in (Harper et al. 1987), by defining a reduction preserving mapping to the simple typed $\lambda$-calculus.

In view of Example 2.3, strong normalization of $\beta$-reduction is very natural, as it corresponds to cut-elimination in the logic (evaluation of natural deductions), which we know to be terminating for first order predicate logic.
Proposition 2.9. (Decidability of Typing) Given a context $\Gamma$ and a pseudo-term $M$, it is decidable whether $M$ has a type in $\Gamma$ and if $M$ has a type it can be computed.

As a matter of fact, ‘typechecking a term’ (i.e. find out whether a given pseudoterm has a type in a given context) proceeds by just computing it (with a ‘fail’ option if the pseudoterm does not have a type). Similarly, checking whether a given pseudoterm $M$ has a given type $A$ proceeds by computing the type of $M$ and checking whether it is $\beta$-equal to $A$.

Decidability of Typing holds because the abstracted variables come with their types. So, if we want to typecheck $\lambda x : A. M$ in $\Gamma$, we can proceed by typechecking $M$ in $\Gamma$ $\times A$. If we want to typecheck $MN$ in $\Gamma$, we have to typecheck $M$ in $\Gamma$ and see whether its type reduces to something of the form $\Pi x : A. B$. Then we typecheck $N$ in $\Gamma$ and see whether its type is $\beta$-equal to $A$. In the typechecking algorithm, an algorithm for checking $\beta$-equality is used. As all terms in $\lambda P$ are Strongly Normalizing, this equality-checking algorithm terminates, and hence the type-checking algorithm does.

If we pursue the view of $\lambda P$ as a logic, one may wonder whether the inhabitation problem is decidable, that is, given a context $\Gamma$ and a type $A$, is it decidable whether there exists a term of type $A$ in $\Gamma$?

Proposition 2.10. (Undecidability of Inhabitation) (Bezem and Springintveld 1996) In general it is undecidable whether, given a context $\Gamma$ and a type $A$ in $\Gamma$, there exists an $M$ such that $\Gamma \vdash M : A$.

The proof proceeds by translating register machine programs $P$ into a pair of a context $\Gamma_P$ and a type $A_P$ in such a way that $A_P$ is inhabited in $\Gamma_P$ iff $P$ terminates.

From a more computational perspective, viewing the terms as programs, Strong Normalization may not seem very natural to have. In the computational view, one may wonder whether types can be reconstructed automatically, i.e. if they are not given in the $\lambda$-abstraction. This is known as the Type Assignment Problem, which is known to be decidable for simple typed $\lambda$-calculus. To be more precise, we define the erasure, $\| - \|$, of a pseudoterm by $\| \lambda x : A. M \| := \lambda x : \| M \|$ and for the other cases by structural recursion. Then, if $\Gamma \vdash M : A$ (type), the term $\| M \|$ is an ordinary untyped $\lambda$-term. Now, given an untyped $\lambda$-term $N$ one may wish to find out whether $N$ is typable, i.e. whether a well-typed term $M$ exists such that $\| M \| \equiv N$.

Proposition 2.11. (Undecidability of Type Assignment) (Dowek 1993) It is in general undecidable whether, given a context $\Gamma$ and an untyped $\lambda$ term $N$, there is a term $M$ such that $M$ is well-typed in $\Gamma$ and $\| M \| \equiv N$.

The proof proceeds by defining a special context $\Gamma$ such that for every Post correspondence problem $P$ there is a (untyped) term $t_P$ such that $t_P$ is typable in $\Gamma$ iff the problem $P$ has a solution.

2.2. Minimal $\lambda P$

It turns out that for many practical uses, there is no need to be able to abstract over a type to form a constructor (i.e. form $\lambda x : A. M$ for $\Pi x : A. B : \text{kind}$). We therefore define a minimal version of $\lambda P$. 
Definition 2.12. In \( \lambda P \) we split the rule \((\lambda)\) in two, a \((\lambda_0)\) and a \((\lambda_P)\) rule. For convenience we attach a label to the abstraction that we introduce with the rule, so

\[
\frac{\Gamma, x:A \vdash M : B \quad \Gamma \vdash \Pi x : A. \, B : \text{type}}{\Gamma \vdash \lambda_0 x : A. \, M : \Pi x : A. \, B}
\]

\[
\frac{\Gamma, x:A \vdash M : B \quad \Gamma \vdash \Pi x : A. \, B : \text{kind}}{\Gamma \vdash \lambda_P x : A. \, M : \Pi x : A. \, B}
\]

The system \( \lambda P \) without the rule \((\lambda_P)\) we call \( \lambda P^- \), and we write \( \vdash^- \) for judgements in \( \lambda P^- \). On the terms of \( \lambda P \) we now distinguish \( \beta_0 \)-reduction from \( \beta_P \)-reduction in the obvious way:

\[
(\lambda_0 x : A. M)N \rightarrow_{\beta_0} M[N/x].
\]

\[
(\lambda_P x : A. M)N \rightarrow_{\beta_P} M[N/x].
\]

Similarly we can now talk about \( \beta_P \)-normal forms etcetera.

All the nice properties that we know from \( \lambda P \) remain to hold for \( \lambda P^- \). In Section 4 we show that this minimal version of \( \lambda P \) is adequate as a logical framework. It is also adequate for interpreting minimal first order predicate logic, as we will see in the next Section.

3. \( \lambda P \) as minimal first order predicate logic

We want to make a precise study of the embedding of minimal first order predicate logic into \( \lambda P \), following the so called formulas-as-types embedding, originally due to Curry, Howard and De Bruijn. The embedded logic, minimal first order predicate logic, has as connectives only implication \( \to \) and first-order universal quantification. There is no negation and (hence) there is no double negation rules, so the logic is constructive. A peculiarity of the embedding is that both domains (of the logic) and formulas are interpreted as types, which makes the completeness of the formulas-as-types embedding not at all obvious. Moreover, there are two constructions possible in \( \lambda P \) that are – although not alien to predicate logic – not in the realm of first order logic. These are:

1. The possibility to define new predicates by abstraction. E.g. if \( R : A \to A \to \text{type} \) is a binary predicate on \( A \), then \( \lambda x : A. \, R x x \) is of type \( A \to \text{type} \), a unary predicate on \( A \).
2. The possibility to declare and construct higher order functions. E.g. one can declare \( f : (A \to A) \to A \) in the context, a function from \( A \to A \) (functions from \( A \) to \( A \)) to \( A \). Also one can construct higher order functions, like \( \lambda f : (A \to A) \to A. \, f(\lambda x : A. \, x) \) which is of type \( ((A \to A) \to A) \to A \).

In the following we first give a precise definition of the logical systems: minimal first order predicate logic, \( \text{PRED} \) and its extension with definable predicates and higher order functions, \( \text{PRED}^f \). Then we define a typed \( \lambda \) calculus, \( \lambda \text{PRED}^f \), that we show to be isomorphic to \( \text{PRED}^f \) via the formulas-as-types embedding. Then we prove in two phases that the embedding of minimal first order predicate logic (\( \text{PRED} \)) into \( \lambda P \) is indeed complete. In the first phase we show completeness of the embedding of \( \lambda \text{PRED}^f \) into \( \lambda P \). In the second phase we show conservativity of \( \lambda \text{PRED}^f \) over \( \text{PRED} \).
In the following diagram the steps in the proof of completeness of the formulas-as-typed embedding from PRED into $\lambda P$ are depicted. An arrow denotes the inclusion embedding of one system in another.

\[
\text{PRED} \xrightarrow{3.17} \text{PRED}^r \\
\cong \xrightarrow{3.16} \\
\lambda \text{PRED}^r \xrightarrow{3.36} \lambda P
\]

3.1. Minimal first order predicate logic

**Definition 3.1.** The language of the system PRED is defined as follows.

1. The **domains** are given by

\[ D ::= B | P | F \]

where $B$ is a set of basic domains, $P$ is the set of predicate domains, defined by

\[ P ::= \text{Prop} | B \rightarrow P, \]

and $F$ is the set of functional domains, defined by

\[ F ::= B \rightarrow \cdots \rightarrow B. \]

(We assume every functional domain to be built up from at least two basic domains.) The intention is that $B_1 \rightarrow B_2 \rightarrow B_3$ should be read as $B_1 \rightarrow (B_2 \rightarrow B_3)$, the set of functions taking an argument in $B_1$, an argument in $B_2$ and returning a value in $B_3$. Similarly, $B_1 \rightarrow B_2 \rightarrow \text{Prop}$ is intended to represent the set of relations on $B_1 \times B_2$.

2. The terms of the language of PRED are described as follows.

- There are countably many constants $c^D_i$ for every domain $D \in D$.
- There are countably many variables of each basic domain $B \in B$.
- If $c^D_i$ is a constant of domain $D \equiv B_1 \rightarrow \cdots \rightarrow B_i \rightarrow C$, where $C \in B \cup \text{Prop}$ and for $1 \leq i \leq p, t_i$ is a term of domain $B_i$, then $c^D_1 t_1 \ldots t_p$ is a term of domain $C$.
- If $\varphi \in \text{Prop}$ and $x$ is a variable of basic domain $B$, then $\forall x \in B. \varphi$ is a term of domain $\text{Prop}$.
- If $\varphi$ and $\psi$ are terms of domain $\text{Prop}$, then $\varphi \supset \psi$ is a term of domain $\text{Prop}$.

If $t$ is a term of domain $D$, we shall just write $t \in D$.

The derivation rules of PRED are the following. (The usual rules for $\supset$ and $\forall$, where
quantification is restricted to the basic domains.)

\[
\begin{align*}
\vdash & \quad [\varphi]^i \\
\vdash & \quad \psi \quad \text{(\forall I)} \\
\hline
\varphi \supset \psi \quad \text{if } t \in B
\end{align*}
\]

(*) in the \forall I rule we make the usual restriction that the variable \(x\) may not occur free in a non-discharged assumption of the derivation. In the \forall I rule, \(i\) is an index to label the formulas that are discharged with that rule.

For \(\Gamma\) a set of formulas of PRED and \(\varphi\) a formula of PRED, we say that \(\varphi\) is derivable from \(\Gamma\) in PRED, notation \(\Gamma \vdash_{\text{PRED}} \varphi\), if there is a derivation with root \(\varphi\) and all non-discharged formulas in \(\Gamma\).

This is the usual system of minimal first order predicate logic. We extend this definition to allow for definable predicates (by \(\lambda\)-abstraction) and higher order functions.

**Definition 3.2.** The system PRED\(^\alpha\) is PRED plus the following extra clauses. The functional domains \(\mathcal{F}\) are defined by

\[
\mathcal{F} ::= B \to B \mid \mathcal{F} \to B.
\]

In the rules for term-formation we add

- If \(t \in D_2\), \(D_2 \in \mathcal{D}\) (an arbitrary domain) and \(x\) a variable of domain \(D_1 \in B \cup \mathcal{F}\), then \(\lambda x.\varphi. t \in D_1 \to D_2\).
- If \(t \in D_1 \to D_2\) and \(q \in D_1\), then \(tq \in D_2\).
- If \(\varphi \in \text{Prop}\) and \(x\) a variable of domain \(D_1 \in B \cup \mathcal{F}\), then \(\forall x.\varphi\) \(\in \text{Prop}\).

The last rule replaces the application rule in the terms of PRED. The \(\lambda\)-abstraction (and application) on terms comes together with the usual notion of \(\beta\)-equality: two terms are \(\beta\)-equal if they are equal via the transitive, symmetric, reflexive closure, compatible with application and abstraction. of the one step \(\beta\)-reduction \((\lambda x.\varphi. t)q \to_{\beta} t[q/x]\). In the derivation rules we add the rule

\[
(\text{conv}) \quad \frac{\psi}{\varphi} \quad \text{if } \varphi = \psi
\]

So in PRED\(^\alpha\) there is no real distinction – in treatment – between basic domains and functional domains. For clarity and to be able to compare the systems, we keep the distinction.

The need for a conversion rule, establishing that \(\beta\)-equal formulas are equivalent, is felt in the following examples.

**Examples 3.3.** We work in the system PRED\(^\alpha\). Let \(A\) be a basic domain, \(R \in A \to A \to \text{Prop}\), \(f \in A \to A \to A\), \(a \in A\).
— We can define a new predicate \( \lambda x: A. R x x \in A \rightarrow \text{Prop} \) by \( \lambda \)-abstraction. Let’s abbreviate it to \( Q \). Now \( R a a \vdash Q a \) by the conversion rule.
— We can define a new function \( \lambda x: A. f x (f x a) \in A \rightarrow A \) by \( \lambda \)-abstraction. Let’s abbreviate it to \( g \). Now \( \forall x: A. R x (f x (f x u)) \vdash \forall x: A. R x (g x) \).

3.2. A typed \( \lambda \)-calculus for minimal first order predicate logic

**Definition 3.4.** \( \lambda \text{PRED}^{fr} \) is the typed \( \lambda \)-calculus defined as follows. Just like in \( \lambda \text{P} \), there are two forms of judgement.

\[
\Gamma \vdash M : B \quad M \text{ is of type } B \text{ in context } \Gamma, \\
\Gamma \vdash \Gamma \text{ is a correct context.}
\]

The set of pseudoterms is now defined by

\[
T ::= \text{Var} | \text{Set} \mid \text{Prop} \mid \text{Type}^s \mid \text{Type}^p \mid (T T) \mid (\lambda x: T . T) \mid (\Pi x: T . T).
\]

The intended interpretations are that \( \text{Set} \) represents the universe of basic sets (including sets of higher type). \( \text{Prop} \) represents the universe of formulas, \( \text{Type}^p \) represents the universe of predicate sets and \( \text{Type}^s \) contains just \( \text{Set} \). The derivation rules for deriving the judgements \( \Gamma \vdash M : B \) and \( \Gamma \vdash \) are the following. Here, \( s,s_1 \) and \( s_2 \) range over \{\text{Set, Prop, Type}^p, \text{Type}^s\} and \( \mathcal{R} = \{(\text{Set, Set}), (\text{Prop, Prop}), (\text{Set, Prop}), (\text{Set, Type}^p)\} \).

\[
\begin{align*}
\text{(base)} & \quad \emptyset \vdash & \quad \text{(ctx)} & \quad \frac{}{\Gamma \vdash A : s} \\
& \quad \frac{}{\Gamma, x : A \vdash} & \quad \text{(proj)} & \quad \frac{}{\Gamma \vdash x : A} \\
\text{(ax)} & \quad \frac{}{\Gamma \vdash \text{Set} : \text{Type}^s} & \quad \text{(ax)} & \quad \frac{}{\Gamma \vdash \text{Prop} : \text{Type}^p} \\
\text{(II)} & \quad \frac{}{\Gamma \vdash \Pi x : A . B : s_2} & \quad \text{(app)} & \quad \frac{}{\Gamma \vdash M : A . B . \Pi x : A . B : s} \\
& \quad \frac{}{\Gamma \vdash \lambda x : \Pi x : A . B : s_1} & \quad \frac{}{\Gamma \vdash M : B} & \quad \frac{}{\Gamma \vdash A : s \quad \Gamma \vdash M : A} \\
& \quad \frac{}{\Pi x : A . M : \Pi x : A . B} & \quad \frac{}{\Gamma \vdash M : A} & \quad \frac{}{\Gamma \vdash M : A} \\
\end{align*}
\]

We use the same notational conventions as for \( \lambda \text{P} \).

**Example 3.5.** \( A : \text{Set}, R : A \rightarrow A \rightarrow \text{Prop}, s : \Pi x, y : A . R x y \rightarrow R y x, t : \Pi x, y, z : A . R x y \rightarrow R y z \rightarrow R z z \vdash \lambda x, y : A . \lambda h : R x y . (t x y z h (s y x z)) : \Pi x, y : A . R x y \rightarrow R x x. \)

This is a proof of the fact that if \( R \) is symmetric and transitive, then \( R \) is reflexive on its domain (the \( x \) for which \( R x y \) for some \( y \)).

We list the main properties of the \( \lambda \text{PRED}^{fr} \). They are a consequence of the fact
that $\text{APRED}^{fr}$ is a Pure Type System. Unless stated otherwise, proofs can be found in (Barendregt 1992) or in (Geuvers and Nederhof 1991).

**Proposition 3.6. (Subject Reduction)** If $\Gamma \vdash M : A$ and $M \rightarrow^\beta N$, then $\Gamma \vdash N : A$.

**Proposition 3.7. (Typed Confluence)** If $\Gamma \vdash M : A$, $\Gamma \vdash N : A$ and $M \equiv^\beta N$, then there is a term $P$ with $M \rightarrow^\beta P$, $N \rightarrow^\beta P$ and $\Gamma \vdash P : A$.

**Proposition 3.8. (Uniqueness of Types)** If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A \equiv^\beta B$.

**Proposition 3.9. (Decidability of Typing)** Given a context $\Gamma$ and a pseudo-term $M$, it is decidable whether $M$ has a type in $\Gamma$ and if $M$ has a type it can be computed.

**Proposition 3.10. (Strong Normalization)** All well-typed expressions of $\text{APRED}^{fr}$ are Strongly Normalizing with respect to $\beta$-reduction.

**Proposition 3.11. (Undecidability of Inhabitation)** In general it is undecidable whether, given a context $\Gamma$ and a type $A$ in $\Gamma$, there exists an $M$ such that $\Gamma \vdash M : A$.

The proof of Undecidability of Inhabitation for $\text{AP}$ in (Bezem and Springintveld 1996) (see also Proposition 2.10) applies immediately to $\text{APRED}^{fr}$. As a matter of fact, the proof in (Bezem and Springintveld 1996) already shows that minimal first order predicate logic $\text{PRED}$ is undecidable.

**Proposition 3.12. (Undecidability of Type Assignment)** It is in general undecidable whether, given a context $\Gamma$ and an untyped term $N$, there is a term $M$ such that $M$ is well-typed in $\Gamma$ and $\|M\| \equiv N$.

Modulo some small changes, the proof of Undecidability of Type Assignment for $\text{AP}$ (see 2.11) in (Dowek 1993) can be applied to $\text{APRED}^{fr}$ as well.

The following is a specific property of $\text{APRED}^{fr}$, that does not hold for $\text{AP}$. In fact it states that $\text{APRED}^{fr}$ is really a logical system, where first the terms are built up, then the propositions and then the proofs.

**Proposition 3.13.** In $\text{APRED}^{fr}$, if $\Gamma \vdash M : A$, then

$$\Gamma_D, \Gamma_T, \Gamma_P \vdash M : A$$

where

- $\Gamma_D, \Gamma_T, \Gamma_P$ is a permutation of $\Gamma$,
- $\Gamma_D$ only contains declarations $\alpha : \text{Set}$,
- $\Gamma_T$ only contains declarations $x : A$ with $\Gamma_D \vdash A : \text{Set}/\text{Type}^0$,
- $\Gamma_P$ only contains declarations $p : \varphi$ with $\Gamma_D, \Gamma_T \vdash \varphi : \text{Prop}$.

The $\Gamma_D, \Gamma_T$ and $\Gamma_P$ are determined uniquely up to permutation. We refer to $\Gamma_D$ as the set-context of $\Gamma$, to $\Gamma_T$ as the object-context of $\Gamma$ and to $\Gamma_P$ as the proof-context of $\Gamma$ and to the concatenation $\Gamma_D, \Gamma_T$ as a language-context. Furthermore, if $\Gamma \vdash M : A$, then

- if $A \equiv \text{Set}/\text{Type}^0$, then $\Gamma_D \vdash M : A$,
- if $\Gamma \vdash A : \text{Set}/\text{Type}^0$, then $\Gamma_D, \Gamma_T \vdash M : A$.

We will refer to the $M$ such that $\Gamma_D, \Gamma_T \vdash M : A$ with $A : \text{Set}$ as the object-language.

The proof of this Proposition is not difficult and proceeds by induction on the derivation of $\Gamma \vdash M : A$. A detailed proof can be found in (Geuvers 1993).
By restricting the rules we obtain a system we call \( \lambda \text{PRED} \) which corresponds to PRED in the same way that \( \lambda \text{PRED}^{fr} \) corresponds to PRED\(^{fr} \), via the formulas-as-types embedding.

**Definition 3.14.** The system \( \lambda \text{PRED} \) is defined by
- Restricting in the rules of \( \lambda \text{PRED}^{fr} \) the \( (\lambda) \)-rule to the case where \( s = \text{Prop} \).
- Adding a universe \( \text{Fun} \).
- Removing \( (\text{Set}; \text{Set}) \) from \( \mathcal{R} \).
- Allowing in triples \( (s_1; s_2; s_3) \) of universes in \( \mathcal{R} \), where \( s_3 \) is the type in the conclusion of the \( (\Pi) \)-rule, and adding the triples \( (\text{Set}; \text{Set}; \text{Fun}) \) and \( (\text{Set}; \text{Fun}; \text{Fun}) \).

The intention of \( \text{Fun} \) is to represent the universe of function types. (The \( \mathcal{F} \) in Definition 3.1.) We can now only form basic sets of type \( \text{Set} \) and \( A \to A : \text{Fun}, A \to A \to A : \text{Fun}, \) etcetera.

### 3.3. The conservativity results

The formulas-as-types embedding from PRED\(^{fr} \) into \( \lambda \text{PRED}^{fr} \) is defined as follows.
- A basic domain \( B \) is mapped to a declaration \( B : \text{Set} \) in the context. Hence, the functional domains are mapped to terms of type \( \text{Set} \) and the predicate domains are mapped to terms of type \( \text{Type}^n \). For convenience, we don’t distinguish between the notation of the domain in the PRED\(^{fr} \) and in \( \lambda \text{PRED}^{fr} \).
- A constant \( c^D \) is mapped to a declaration \( c : D \) in the context. Hence, a term \( f(t_1, \ldots, t_n) \) is translated to \( ft_1 \cdots t_n \).
- A proposition is mapped to a term of type \( \text{Prop} \) by translating \( \forall x : A. \varphi \) to \( \Pi x : A. \varphi \) and \( \varphi \supset \psi \) to \( \varphi \to \psi \).
- An assumption \( \varphi \) is translated into a declaration \( p : \varphi \) in the context.
- A derivation is mapped to a typed term translating an \( \text{I} \)-rule into a \( \lambda \)-abstraction and an \( \text{E} \)-rule into an application.

As the translation alters so little for domains, terms and propositions, we usually don’t write it and identify e.g. \( \forall x : A. P x \supset Q x \) with \( \Pi x : A. P x \to Q x \). The translation from derivations to proof-terms will be denoted by \( \{ - \} \). So if \( \Delta \vdash^{\Theta} \varphi \), denoting that \( \Theta \) is a derivation of \( \varphi \) from \( \Delta \), then \( [\Theta] \) is the associated proof-term.

We do not define the translation of derivations into proof-terms precisely, as it is rather well-known and involves quite a lot of syntactical detail. Instead we give an example of a derivation and how it is (inductively) translated to a term in \( \lambda \text{PRED}^{fr} \). Moreover, this example should provide enough evidence for the fact that there is also a translation back from proof-terms (in \( \lambda \text{PRED}^{fr} \)) to derivations in PRED\(^{fr} \). The translation from terms to derivations will be denoted by \( \Sigma(-) \): if \( t \) is a proof-term, then \( \Sigma(t) \) is the associated derivation.

The isomorphism between PRED\(^{fr} \) and PRED\(^{fr} \) arises from the fact that these two translations \( \{ - \} \) and \( \Sigma(-) \) are eachother’s inverses. Details can be found in (Geuvers 1993).

**Example 3.15.** Let a domain \( A \), a predicate \( P \) on \( A \) and a binary relation \( Q \) on \( A \) be given. The following is a derivation of \( (\forall x : A. P x) \supset (\forall x : A. Q x) \) from the hypothesis
$\forall x, y \in A. P x \supset P y \supset Q x y$. Let’s call this derivation $\Theta$.

\[
\begin{align*}
\forall x, y \in A. P x \supset P y \supset Q x y & \quad \forall y \in A. P x \supset P y \supset Q x y \\
P x \supset P x \supset Q x x & \quad [\forall x \in A. P x] \\
P x \supset Q x x & \quad \frac{P x}{P x}
\end{align*}
\]

We decorate $\Theta$ by $\lambda$-terms, thus inductively defining the $\lambda$-term $[\Theta]$ from the derivation $\Theta$.

\[
g : \forall x, y \in A. P x \supset P y \supset Q x y
\]

\[
g x : P x \supset P x \supset Q x x
\]

\[
h x : \forall x \in A. P x
\]

\[
g x x (h x) : P x \supset Q x x
\]

\[
\lambda h : (\forall x \in A. P x). \forall x A. g x x (h x) : (\forall x \in A. Q x x)
\]

So $[\Theta] := \lambda h : (\Pi x : A. P x) \lambda x : A. g x x (h x) : (\forall x \in A. Q x x)$. In $\lambda\text{PRED}^f$, we can derive

\[
A : \text{Set}, P : A \Rightarrow \text{Prop}, Q : A \Rightarrow A \Rightarrow \text{Prop}, g : \Pi x, y : A. P x \Rightarrow P y \Rightarrow Q x y
\]

\[
\vdash \lambda h : (\Pi x : A. P x) \lambda x : A. g x x (h x) : (\Pi x : A. Q x x).
\]

**Proposition 3.16.** Given a $\text{PRED}^f$-signature $\Sigma$ (containing a finite number of domains and relations and constants over these domains), a finite set of formulas (over $\Sigma$) $\Delta$ and a formula $\varphi$,

\[
\Delta \vdash^\Theta \varphi \Rightarrow \Gamma, \varphi, \Theta, \varphi : \Delta \vdash^\text{PRED}^f \varphi,
\]

where $\Gamma, \Delta, \varphi, \Theta$ is the canonically defined language context containing declarations for all the constants and free variables in $\Sigma, \Delta, \varphi, \Theta$.

Furthermore, if $\Gamma, \Delta \vdash \psi : \text{Prop}$ for all $\psi \in \Delta \cup \{\varphi\}$, then

\[
\Gamma, \Delta \vdash^\text{PRED}^f M : \varphi \Rightarrow \Delta \vdash^\Sigma(\text{M}) \text{PRED}^f \varphi.
\]

The mappings $\{\} \text{ and } \Sigma(\cdot)$ constitute a bijection between derivations in $\text{PRED}^f$ and proof-terms in $\lambda\text{PRED}^f$.

For a detailed proof see (Geuvers 1993).

**Proposition 3.17.** $\text{PRED}^f$ is conservative over $\text{PRED}$, that is, for $\Delta$ a set of formulas and $\varphi$ a formula of $\text{PRED}$,

\[
\Delta \vdash^\text{PRED}^f \varphi \Rightarrow \Delta \vdash^\text{PRED} \varphi.
\]

**Proof.** The proof is by eliminating cuts (in derivations) and normalizing the terms as follows. In $\text{PRED}^f$ all terms are Strongly Normalizing and all cuts in derivations can be eliminated. Moreover if $\Delta$ and $\varphi$ are taken from $\text{PRED}$ and $\Delta \vdash^\text{PRED} \varphi$ by a cut-free derivation in which only normal terms occur, then this is already a derivation in $\text{PRED}$. 


The easiest way to treat the normalization and cut-elimination argument is by looking at $\text{APRED}^{fr}$ and $\text{APRED}$ instead. The argument then runs as follows.

1. The system $\text{APRED}^{fr}$ is Strongly Normalizing (e.g. there is a reduction-preserving embedding into the Calculus of Constructions, which is known to be SN).

2. Suppose that the term-context $\Gamma_T$ is actually a $\text{APRED}$-context (i.e. there occur no Set-types of higher type in it). Suppose furthermore that $\Delta \cup \{ \varphi \}$ contains only $\text{APRED}$-formula.

3. If $\Gamma_D, \Gamma_T \vdash_{\text{APRED}_{fr}} t : A$ with $t$ in normal form and $A : \text{Set} \in \Gamma_D$ (i.e. $A$ is a basic domain), then $\Gamma_D, \Gamma_T \vdash_{\text{APRED}} t : A$

4. If $\Gamma_D, \Gamma_T, \Gamma_P \vdash_{\text{APRED}_{fr}} q : \varphi$ with $q$ in normal form, then $\Gamma_D, \Gamma_T, \Gamma_P \vdash_{\text{APRED}} q : \varphi$.

The third and fourth step are proved by induction on the structure of terms ($t$ and $q$). Together, these four steps prove the proposition.

Now to show the completeness of the formulas-as-types embedding from first order predicative logic (PRED) into $\lambda P$, we only have to show the completeness of the embedding of $\text{APRED}^{fr}$ into $\lambda P$. Both $\text{APRED}^{fr}$ and $\lambda P$ are Pure Type Systems (PTS) and the embedding of $\text{APRED}^{fr}$ into $\lambda P$ that we are looking at is a $\text{PTS}$-morphism $H$ given by

$$H := \begin{cases} 
\text{Set} & \mapsto \text{type} \\
\text{Prop} & \mapsto \text{type} \\
\text{Type}^s & \mapsto \text{kind} \\
\text{Type}^p & \mapsto \text{kind} \end{cases}$$

A PTS-embedding extends immediately to all terms and contexts and it preserves typing, i.e.

$$\Gamma \vdash_{\text{APRED}_{fr}} M : A \Rightarrow H(\Gamma) \vdash_{\lambda P} H(M) : H(A).$$

We now give the technical details of the proof of completeness of $H : \text{APRED}^{fr} \rightarrow \lambda P$. The proof uses techniques developed in (Swan 1989) who shows completeness of the formulas-as-types embedding from first order predicate logic into Martin-Löf’s intuitionistic theory of types. A different proof of our result can be found in (Berardi 1990).

The question of completeness is whether for any $\text{APRED}^{fr}$-context $\Gamma_D, \Gamma_T, \Gamma_P$ and proposition $\varphi$ with $\Gamma_D, \Gamma_T \vdash_{\text{APRED}_{fr}} \varphi : \text{Prop}$, if

$$H(\Gamma_D, \Gamma_T, \Gamma_P) \vdash M : H(\varphi) \text{ in } \lambda P,$$

then there exists a term $N$ with

$$\Gamma_D, \Gamma_T, \Gamma_P \vdash N : \varphi \text{ in } \text{APRED}_{fr}.$$
The first clause is validated by Proposition 3.13. If the second clause were not satisfied we would in fact be working in propositional logic. The third and fourth clause are added for convenience. In case there are empty domains in the logic, the completeness result would still hold with a slightly adapted argument.

We fix a language-context of $\lambda$PRED$^\eta \Gamma_D, \Gamma_T$. We first want to define a map $\lbrack - \rbrack^p$ from $\lambda$P-terms, typable in $H(\Gamma_D, \Gamma_T)$, to the object language of $\lambda$PRED$^\eta$. This map $\lambda$ should be the inverse of $H$ on the object language. That is, if $\Gamma_D, \Gamma_T \vdash_{\lambda$PRED$^\eta} M : A$ with $A : \text{Set}$, then $\lbrack H(M) \rbrack^p \equiv M$. We need to define $\lbrack - \rbrack^p$ not just on terms typable in $H(\Gamma_D, \Gamma_T)$, but in a elementary extension of $H(\Gamma_D, \Gamma_T)$.

**Definition 3.19.** For $\Gamma_D, \Gamma_T$ a language-context of $\lambda$PRED$^\eta$ and $\Delta$ a context of $\lambda$P, we say that $\Delta$ is an elementary extension of $H(\Gamma_D, \Gamma_T)$, notation $\Delta \succ H(\Gamma_D, \Gamma_T)$, if $\Delta \supseteq H(\Gamma_D, \Gamma_T)$ and the extra declarations in $\Delta$ are all of the form $x : \sigma$ with $H(\Gamma_D, \Gamma_T) \vdash_{\lambda$P}$\sigma : \text{type}$.

For example $H(\Gamma_D, \Gamma_T, \Gamma)$ is always an elementary extension of $H(\Gamma_D, \Gamma_T)$.

**Definition 3.20.** Let $\Delta \succ H(\Gamma_D, \Gamma_T)$. The mapping $\lbrack - \rbrack^p$ from $\lambda$P-terms in the context $\Delta$ to terms of $\lambda$PRED$^\eta$ is defined as follows.

\begin{enumerate}
\item $\lbrack \text{type} \rbrack^p := \text{Set},$
\item $\lbrack \text{kind} \rbrack^p := \text{Type}^\ddagger,$
\item $\lbrack x \rbrack^p := x$, if $x \in \Delta \setminus (\text{so } x : \text{Set}),$
\item $\lbrack O \rbrack^p := O$, if $x \in \Gamma_T$ and $x : \cdots \to \text{Prop},$
\item $\lbrack x \rbrack^p := x$, for $x$ another variable,
\item $\lbrack \Pi x : A. B \rbrack^p := [B]^p$ if $A : \text{type}, B : \text{kind},$
\item $\lbrack \Pi x : A. B \rbrack^p := [A]^p, [B]^p$ else,
\item $\lbrack \lambda x : A. M \rbrack^p := [M]^p$ if $A : \text{type}, M : B : \text{kind}$ (for some $B$),
\item $\lbrack \lambda x : A. M \rbrack^p := [A]^p, [M]^p$ else,
\item $\lbrack P M \rbrack^p := [P]^p$ if $M : A : \text{type}, P : B : \text{kind}$ (for some $A, B$),
\item $\lbrack P M \rbrack^p := [P]^p, [M]^p$ else
\end{enumerate}

The definition extends immediately to the context $\Delta$ itself, where a declaration coming from $x : \cdots \to \text{Prop} \in \Gamma_T$ (case (iv)) is removed.

**Example 3.21.** As a running example throughout this Section, we will use the following $\lambda$PRED$^\eta$-context.

\[
\begin{align*}
O &: \text{Set}, A &: \text{Set} \\
\text{True} &: \text{Prop}, c_0 &: O, c_A &: A, f &: O \to A, g &: (O \to A) \to A, R &: O \to A \to \text{Prop}, \\
t &: \text{True}, p_1 &: \Pi x : O. R x(f x), p_2 &: \Pi x : A. (\Pi y : O. R y(g(\lambda z : O. z))) \to R_{Col}. \\
\end{align*}
\]

So here
\[
\begin{align*}
\Gamma_D &= O : \text{Set}, A : \text{Set}, \\
\Gamma_T &= \text{True} : \text{Prop}, c_0 &: O, c_A &: A, f &: O \to A, g &: (O \to A) \to A, R &: O \to A \to \text{Prop}, \\
\Gamma_P &= t : \text{True}, p_1 &: \Pi x : O. R x(f x), p_2 &: \Pi x : A. (\Pi y : O. R y(g(\lambda z : O. z))) \to R_{Col}. \\
\end{align*}
\]

(In the examples of this Section we will use these abbreviations. Note that in the Definitions, Lemmas and Propositions, $\Gamma_D, \Gamma_T$ is some fixed language context.) Now consider
the following elementary extension $\Delta$ of $H(\Gamma_D, \Gamma_T)$:

$$\Delta := H(\Gamma_D, \Gamma_T, \Gamma_P), \quad z_1 : O \to A, \quad z_2 : \Pi x O. Rx(x) \to A, \quad z_3 : \Pi x A. RcOx.$$  

Note that the type of $z_2$ does not correspond to a logical formula (it is not in the image of $H$). We compute the $|-|^p$-image of this context. It is

$$O : \text{Set}, \ A : \text{Set}, \ c_0 : O, \ c_A : A, \ f : O \to A, \ g : (O \to A) \to A, \ t : O, \ p_1 : O \to O, \ p_2 : A \to (O \to O) \to O, \ z_1 : O \to A, \ z_2 : O \to O \to A, \ z_3 : A \to O.$$  

That the mapping $|-|^p$ is indeed from $\lambda\text{P}$-terms in the context $\Delta$ to $\lambda\text{PRED}^{fr}$ is justified by the following Proposition.

**Proposition 3.22.** Let $\Delta \succ H(\Gamma_D, \Gamma_T)$.

$$\Delta \vdash_{\text{AP}} M : A \Rightarrow |\Delta|^p \vdash_{\text{APRED}^{fr}} |M|^p : |A|^p.$$  

**Proof.** By induction on the derivation of $\Delta \vdash M : A$ in $\lambda\text{P}$. There are no difficult cases. In the conversion rule, it is used that, if $M =_{\beta} N$, then $|M|^p \equiv |N|^p$. \qed

**Example 3.23.** (Definitions are as in Example 3.21.) In $\lambda\text{P}$ we have $\Delta \vdash_{\text{AP}} z_3 c_A : RcOc_A$, which is mapped to $|\Delta|^p \vdash_{\text{APRED}^{fr}} z_3 c_A : O$. Similarly, $\Delta \vdash_{\text{AP}} z_3 c_O(z_3(gz_1)) : A$ is mapped to $|\Delta|^p \vdash_{\text{APRED}^{fr}} z_3 c_O(z_3(gz_1)) : A$.

**Fact 3.24.** If $\Gamma_D, \Gamma_T \vdash M : A(: \text{Set})$, then $|H(A)|^p \equiv A$ and $|H(M)|^p \equiv M$. (Note that $H$ is the identity on these kinds of terms.)

**Corollary 3.25.** For $\Delta \succ H(\Gamma_D, \Gamma_T)$, say $\Delta \equiv H(\Gamma_D, \Gamma_T)$, $\Delta'$ we have

$$\Delta \vdash_{\text{AP}} M : A(: \text{type}) \Rightarrow \Gamma_D, \Gamma_T, |\Delta'|^p \vdash_{\text{APRED}^{fr}} |M|^p : |A|^p.$$  

**Proof.** We know by Proposition 3.22 that $|\Delta|^p \vdash_{\text{APRED}^{fr}} |M|^p : |A|^p$. Now note that $|H(\Gamma_D)|^p \equiv \Gamma_D$. Furthermore, for a declaration $x : A$ in $\Gamma_T$, if $A : \text{Set}$, then $|x|A|^p \equiv x:A$ and if $A : \text{Type}^p$, then $|x|^p \equiv O$ and the declaration of $x$ is removed from the context by $|-|^p$. So $\Gamma_D, \Gamma_T, |\Delta'|^p \supseteq |\Delta|^p$ and so $\Gamma_D, \Gamma_T, |\Delta'|^p \vdash |M|^p : |A|^p$ by Thinning. \qed

All this means that $|-|^p$ is a mapping back from $\lambda\text{P}$-terms (typable in $\Delta \succ H(\Gamma_D, \Gamma_T)$) to the object-language of $\lambda\text{PRED}^{fr}$ that does not change the terms that originated from the object-language.

Now we define a mapping $\text{Tr}$ back from $\lambda\text{P}$ to the proof-language of $\lambda\text{PRED}^{fr}$. So types in $\lambda\text{P}$ will become propositions and objects will become proofs of $\lambda\text{PRED}^{fr}$: If $\Delta \succ H(\Gamma_D, \Gamma_T)$ and $\Delta \vdash_{\text{AP}} A : \text{type}$, then $|\Delta|^p \vdash_{\text{APRED}^{fr}} \text{Tr}(A) : \text{Prop}$. Moreover, if $\Delta, \Gamma_T \vdash_{\text{APRED}^{fr}} \varphi : \text{Prop}$, then $\text{Tr}(\varphi)$ should be equivalent to $\varphi$, i.e., $\text{Tr}(\varphi)$ should be inhabitable if $\varphi$ is. We prove that for such $\varphi$ we can find terms $M_1, M_2$ such that $\Gamma_D, \Gamma_T \vdash_{\text{APRED}^{fr}} M_1 : \varphi \to \text{Tr}(\varphi)$ and $\Gamma_D, \Gamma_T \vdash_{\text{APRED}^{fr}} M_2 : \text{Tr}(\varphi) \to \varphi$. We introduce some notation.

**Notation 3.26.** For a context in $\lambda\text{PRED}^{fr}$ and $\Gamma \vdash_{\text{APRED}^{fr}} \varphi, \psi : \text{Prop}$, we write

$$\Gamma \vdash_{\text{APRED}^{fr}} \langle M_1, M_2 : \varphi \leftrightarrow \psi \rangle$$

if $\Gamma \vdash_{\text{APRED}^{fr}} M_1 : \varphi \to \psi$ and $\Gamma \vdash_{\text{APRED}^{fr}} M_2 : \psi \to \varphi$. 

**Proof.**
The main trick in the definition of $\text{Tr}$ is to decompose a type $\Pi x : A. B$ ($A, B : \text{type}$) into a quantification over $|A|^p$ and an implication $\text{Tr}(A) \rightarrow \text{Tr}(B)$. This idea occurs in different places in the literature. Our main inspiration is (Swaen 1989), who uses it for first-order predicate logic and Martin Löf’s type theory, but one can also find it in (Mohring 1986), who uses it for defining a realizability interpretation for the Calculus of Constructions.

and in (Geuvers 1995), where it is used to prove Strong Normalization for the Calculus of Constructions.

**Definition 3.27.** Let $\Delta \vdash H(\Gamma_D, \Gamma_T)$. The map $\text{Tr}$ on constructors of $\mathbb{L}P$ in $\Delta$ is defined as follows.

(i) $\text{Tr}(\alpha) := \text{True}$, if $\alpha : \text{Set} \in \Gamma_D$.
(ii) $\text{Tr}(\rho) := \alpha$, if $\alpha : \cdots \rightarrow \text{Prop} \in \Gamma_T$.
(iii) $\text{Tr}(\lambda x : A. M) := \lambda x : |A|^p. \text{Tr}(M)$.
(iv) $\text{Tr}(Q) := \text{Tr}(Q)|^p$.
(v) $\text{Tr}(\Pi x : A. B) := \Pi x : |A|^p. \text{Tr}(A) \rightarrow \text{Tr}(B)$.

**Example 3.28.** With the $\mathbb{L}P$-context $\Delta$ defined as in Example 3.21, we find the following $\text{Tr}$ translations for types in this context.

\[
\begin{align*}
\text{Tr}(\Pi x : O. Rx(f x)) &= \Pi x : O. \text{True} \rightarrow Rx(f x), \\
\text{Tr}(\Pi x : O. Rx(g z_1) \rightarrow A) &= \Pi x : O. \text{True} \rightarrow Rx(g z_1) \rightarrow \text{True}.
\end{align*}
\]

**Proposition 3.29.** For $\Delta \vdash H(\Gamma_D, \Gamma_T)$, say $\Delta \equiv H(\Gamma_D, \Gamma_T)$, $\Delta'$ we have

\[
\Delta \vdash_{\mathbb{L}P} C : \Pi z_1 : A_1 \cdot \cdots \cdot \Pi z_n : A_n. \text{type}
\]

\[
\Rightarrow \quad \Gamma_D, \Gamma_T, |\Delta'|^p \vdash_{\text{APRED}\text{fr}} \text{Tr}(C) : |A_1|^p \rightarrow \cdots \rightarrow |A_n|^p \rightarrow \text{Prop}.
\]

**Proof.** By induction on the derivation. Note that if $A : \text{type}$ in $\mathbb{L}P$, then $|A|^p$ contains no object-variables. Furthermore, if $\Delta \vdash_{\mathbb{L}P} M : A(: \text{type})$, then $\Gamma_D, \Gamma_T, |\Delta'|^p \vdash_{\text{APRED}\text{fr}} |M|^p : |A|^p$ by Corollary 3.25.

All $\sigma : \text{type}$ in $\mathbb{L}P$ are mapped to a $\text{Tr}(\sigma) : \text{Prop}$ in $\text{APRED}\text{fr}$. If $\sigma \equiv H(A)$ with $A : \text{Set}$ in $\text{APRED}\text{fr}$, then $\text{Tr}(\sigma)$ should be inhabited (‘true’).

**Lemma 3.30.** If $\Gamma_D \vdash_{\text{APRED}\text{fr}} A : \text{Set}$, then

\[
\exists M_1, M_2 : \Gamma_D, \Gamma_T \vdash_{\text{APRED}\text{fr}} (M_1, M_2) : \text{True} \leftrightarrow \text{Tr}(A).
\]

(To be precise we would have to write $\text{Tr}(H(A))$ instead of $\text{Tr}(A)$, but $H$ is the identity on terms of type $\text{Set}$.)

**Proof.** Immediate from the definition of $\text{Tr}$: if $\Gamma_D \vdash A : \text{Set}$, then $A \equiv \cdots \equiv \alpha$ with $\alpha : \text{Set} \in \Gamma_D$. Hence $\text{Tr}(A) \equiv \cdots \equiv \text{True}$, which is logically equivalent to True.

The mapping $\text{Tr}$ preserves $=_{\beta}$. This is proved using a substitution Lemma for $\text{Tr}$.

**Lemma 3.31.** For $\Delta \vdash H(\Gamma_D, \Gamma_T)$, say $\Delta \equiv H(\Gamma_D, \Gamma_T)$, $\Delta'$, with $\Delta \vdash_{\mathbb{L}P} A, B : \text{type}$ and $\Delta \vdash_{\mathbb{L}P} t : B$ we have

\[
\text{Tr}(A)[t/x] \equiv \text{Tr}(A[t/x]).
\]

If $\Delta \vdash_{\mathbb{L}P} A, A' : \text{type}$ and $A =_{\beta} A'$ then

\[
\text{Tr}(A) =_{\beta} \text{Tr}(A').
\]
Proof. The first is easily proved by induction on the structure of $A$. The second follows from the fact that, if $A \rightarrow_{\beta} A'$, then $\text{Tr}(A) =_{\beta} \text{Tr}(A')$. □

**Proposition 3.32.** For each language-context $\Gamma_D, \Gamma_T$ and $\varphi$ with $\Gamma_D, \Gamma_T \vdash_{\text{APRED}^{Fr}} \varphi : \text{Prop}$ we have

$$\exists M_1, M_2[\Gamma_D, \Gamma_T \vdash_{\text{APRED}^{Fr}} (M_1, M_2) : \varphi \leftrightarrow \text{Tr}(H(\varphi))]$$

(Note that $H$ is the identity on expressions of type Prop, so we can skip it.)

Proof. By induction on the structure of $\varphi$, assuming that $\varphi$ is in normal form. (By Lemma 3.31, $\text{Tr}(\varphi) =_{\beta} \text{Tr}(\text{nff}(\varphi)).$)

**base** If $\varphi \equiv \alpha_1 \cdots t_n$ with $\alpha$ a variable, then $\text{Tr}(\varphi) \equiv \varphi$ by the fact that $|t| \equiv t_i$. (Fact 3.24.)

**say** $\varphi \equiv \psi \rightarrow \chi$ with $\psi, \chi : \text{Prop}$. Then $\text{Tr}(\varphi \rightarrow \psi) \equiv \Pi x : A, \chi : \text{Prop}$. Now we are done by IH: The variable $x$ will not occur free in $\varphi \rightarrow \psi$ and one easily constructs the required proof-terms.

**∀** say $\varphi \equiv \Pi x : A, \psi$ with $A : \text{Set}$. Then $\text{Tr}(\Pi x : A, \psi) \equiv \Pi x : A, \text{Tr}(A) \rightarrow \text{Tr}(\psi)$. Now by Fact 3.24 and Lemma 3.30, $\Pi x : A, \text{Tr}(A) \rightarrow \text{Tr}(\psi)$ is equivalent to $\Pi x : A, \text{Tr}(\psi)$, so we are done by IH. □

**Definition 3.33.** For $\Delta >_\alpha H(\Gamma_D, \Gamma_T)$, say $\Delta \equiv H(\Gamma_D, \Gamma_T), \Delta'$, we define the context

$$\text{TR}(\Delta) := \Gamma_D, \Gamma_T, |\Delta|, |\Delta'|, \text{Tr}(\Delta),$$

where $\text{Tr}(\Delta)$ is defined by replacing every declaration $z: A$ in $\Delta'$ by $z' : \text{Tr}(A)$. (Of course we make sure that the declared variables in $\text{Tr}(\Delta)$ are different from the ones in $|\Delta'|$.)

**Example 3.34.** We look at the translation of $\Delta$ as in Example 3.21.

$$\text{TR}(\Delta) = \hfill O : \text{Set,} \quad A : \text{Set,} \hfill$$

$$\hfill \text{True} : \text{Prop}, c_0 : O, c_1 : A, f : A \rightarrow O, g : (O \rightarrow A) \rightarrow A, R : O \rightarrow A \rightarrow \text{Prop}. \hfill$$

$$\hfill t : O, p_1 : O \rightarrow O, p_2 : A \rightarrow (O \rightarrow O) \rightarrow O, z_1 : O \rightarrow A, z_2 : O \rightarrow O \rightarrow A, z_3 : A \rightarrow O, \hfill$$

$$\hfill t' : \text{True}, p'_1 : \Pi x : O, \text{True} \rightarrow R_x(f x), \hfill$$

$$\hfill p'_2 : \Pi x : A, \text{True} \rightarrow (\Pi y : O, \text{True} \rightarrow R_y(g(\lambda z : O, x))) \rightarrow R_{O, x}. \hfill$$

**Proposition 3.35.** Let $\Delta >_\alpha \Gamma_D, \Gamma_T$, then

$$\Delta \vdash_{\text{AP}} M : A (: \text{type}) \text{ in } \lambda P \Rightarrow \exists N[\text{TR}(\Delta) \vdash_{\text{APRED}^{Fr}} N : \text{Tr}(A)] \text{ in } \lambda P^{Fr}.$$ 

Proof. By induction on the derivation of $\Delta \vdash_{\text{AP}} M : A$ in $\lambda P$.

**var** $M \equiv x$ then either $x : A$ in $\Gamma_T$ or in $\Delta'$. In the first case $\text{Tr}(A) \leftrightarrow \text{True}$ and in the second case $x : \text{Tr}(A) \in \text{TR}(\Delta)$.

**app** Say

$$\Delta \vdash_{\text{AP}} M : \Pi x : A, B \quad \Delta \vdash_{\text{AP}} t : A$$

$$\Delta \vdash_{\text{AP}} M t : B[t/x]$$

By IH, $\text{TR}(\Delta) \vdash_{\text{APRED}^{Fr}} N : \text{Tr}(\Pi x : A, B) \equiv \Pi x : A, \text{Tr}(A) \rightarrow \text{Tr}(B)$ and
First order dependent type theory, $\lambda P$

$\text{TR}(\Delta) \vdash_{\text{APRED}^{fr}} Q : \text{Tr}(A)$. We also have $\text{TR}(\Delta) \vdash_{\text{APRED}^{fr}} |t|^p : |A|^p$, by Corollary 3.25. So we may conclude $\text{TR}(\Delta) \vdash_{\text{APRED}^{fr}} N[|t|^p/x] : \text{Tr}(B)|[|t|^p/x] \equiv \text{Tr}(B[t/x])$.

($\lambda$) Say

$$\Delta, x:B \vdash_{\text{AP}} M : C \quad \Delta \vdash_{\text{AP}} \Pi x:B. C : \text{type}$$

$$\Delta \vdash_{\text{AP}} \lambda x:B.M : \Pi x:B. C$$

By III. $\text{TR}(\Delta, x:B) \vdash_{\text{APRED}^{fr}} N : \text{Tr}(C)$. $\text{TR}(\Delta, x:B) \equiv \text{TR}((\Delta), x:B|B|^p, x':\text{Tr}(B))$, so we have

$$\text{TR}(\Delta) \vdash_{\text{APRED}^{fr}} \lambda x:B|B|^p, \lambda x':\text{Tr}(B). N : \Pi x:B|B|^p. \text{Tr}(B) \rightarrow \text{Tr}(C) \equiv \text{Tr}(\Pi x:B. C).$$

(conv) We are immediately done by Lemma 3.31. 

\[\square\]

**Corollary 3.36.** The embedding $H$ from $\lambda \text{PRED}^{fr}$ into $\lambda P$ is complete, i.e. if $\Gamma_D, \Gamma_T \vdash_{\text{APRED}^{fr}} \varphi : \text{Prop}$ and $\Gamma_P$ a proof-context, then

$$H(\Gamma_D, \Gamma_T, \Gamma_P) \vdash_{\text{AP}} M : H(\varphi) \Rightarrow \exists N[\Gamma_D, \Gamma_T, \Gamma_P \vdash_{\text{APRED}^{fr}} N : \varphi].$$

**Proof.** $H(\Gamma_D, \Gamma_T, \Gamma_P)$ is an elementary extension of $\Gamma_D, \Gamma_T$, so by the Proposition we have

$$\Gamma_D, \Gamma_T, [\Gamma_P]^p, \text{Tr}(\Gamma_P) \vdash_{\text{APRED}^{fr}} N : \text{Tr}(\varphi)$$

for some term $N$. Now every declaration in $[\Gamma_P]^p$ is of the form $y:B$ where $B : \text{Set}$, so we can substitute for such a $y$ a term of type $B$ in the context $\Gamma_D, \Gamma_T$. Furthermore, if $z:B \in \text{Tr}(\Gamma_P)$, then $\exists M_1, M_2[\Gamma_D, \Gamma_T \vdash_{\text{APRED}^{fr}} (M_1, M_2) : B \leftrightarrow \text{Tr}(B)]$ by Proposition 3.32. So we can replace each $z: \text{Tr}(B)$ by $\hat{z}:B$, at the same time substituting $M_1 \hat{z}$ for $z$ inside $N$. (Such a variable $z$ does not occur in $\text{Tr}(\varphi)$.) We obtain a term $N'$ such that

$$\Gamma_D, \Gamma_T, \Gamma_P \vdash_{\text{APRED}^{fr}} N' : \text{Tr}(\varphi).$$

By again applying Proposition 3.32 we can transform this $N'$ into a $N''$ with

$$\Gamma_D, \Gamma_T, \Gamma_P \vdash_{\text{APRED}^{fr}} N'' : \varphi.$$

\[\square\]

As a Corollary to the proof of completeness of $H : \lambda \text{PRED}^{fr} \rightarrow \lambda P$, we get completeness of $H : \lambda \text{PRED} \rightarrow \lambda P^+$: the same Definitions apply and the same results can be proven.

**Corollary 3.37.** The embedding $H$ from $\lambda \text{PRED}$ into $\lambda P^+$ is complete, i.e. if $\Gamma_D, \Gamma_T$ is a language-context with $\Gamma_D, \Gamma_T \vdash_{\text{APRED}} \varphi : \text{Prop}$ and $\Gamma_P$ a proof-context, then

$$H(\Gamma_D, \Gamma_T, \Gamma_P) \vdash_{\text{AP}} M : H(\varphi) \Rightarrow \exists N[\Gamma_D, \Gamma_T, \Gamma_P \vdash_{\text{APRED}} N : \varphi].$$

### 3.4. Some comments on the completeness result

The system PRED is too minimal to be of real interest for practical mathematics, also because a system like $\lambda P$ is usually seen as a logical framework (like $\lambda P$ that will be discussed in Section 4). However, the completeness result can be extended a little bit to systems with a bottom type. We are then considering the formulas-as-types embedding from $\text{PRED}^+$ to $\lambda P^+$, where $\text{PRED}^+$ is PRED, defined in 3.1, extended with a constant
$\bot \in \text{Prop}$ and the ‘ex falso sequitur quodlibet’ rule, saying that $\bot \rightarrow \varphi$ is always inhabited. The system $\lambda \mathcal{P}^\bot$ is $\lambda \mathcal{P}$ extended with a constant type $\bot : \text{type}$ and a constant term $\mathcal{E}_\bot$ with an extra rule

\[
\Gamma \vdash M : \bot \quad \Gamma \vdash A : \text{type} \\
\hline
\Gamma \vdash \mathcal{E}_\bot M : A
\]

The system $\text{PRE}^\bot$ is more interesting because the full classical first order predicate logic is a subsystem of it. More precisely, there is a faithful embedding of classical first order predicate logic into $\text{PRE}^\bot$ by a double negation translation. The embedding of classical first order predicate logic in to $\lambda \mathcal{P}^\bot$ via the system $\text{PRE}^\bot$ is now complete, due to the completeness of the embedding of $\text{PRE}^\bot$ into $\lambda \mathcal{P}$.

4. $\lambda \mathcal{P}$ as a Logical Framework

The idea of using a first order dependent type theory as a Logical Framework originates from De Bruijn and the Automath project. (See (de Bruijn 1980), (van Daalen 1973) or (Nederpelt et al. 1994).) However, the definition of $\lambda \mathcal{P}$, roughly as it is given here, and its use as a Logical Framework, originate from (Harper et al. 1987). It is also in (Harper et al. 1987) that the issue of adequacy of the encoding of a logic is first raised. Given a logic $L$ and its encoding as a $\lambda \mathcal{P}$-context $\Gamma_L$ (see 2.4 for an example) we may wonder whether this encoding is adequate, i.e. whether for $\Delta = \{ \psi_1, \ldots, \psi_n, \varphi \}$ a set of formulas of $L$,

\[
\text{if } \Gamma_L, p_1 : T \psi_1, \ldots, p_n : T \psi_n \vdash_{\lambda \mathcal{P}} M : T \varphi, \text{ then } \Delta \vdash_L \varphi ?
\]

Of course, soundness of the encoding is also an important issue, but usually that is a rather straightforward induction on the derivation in $L$. As a matter of fact, the soundness lies in constructing the right context $\Gamma_L$ (that represents the logic $L$ appropriately). So we assume that we have chosen the right $\Gamma_L$ and that the encoding of $L$ in $\Gamma_L$ is sound:

\[
\text{If } \Delta \vdash_L \varphi \text{ then } \Gamma_L, p_1 : T \psi_1, \ldots, p_n : T \psi_n \vdash_{\lambda \mathcal{P}} M : T \varphi, \text{ for some } M.
\]

(This $M$ will usually be a direct encoding of the derivation of $\Delta \vdash_L \varphi$.)

4.1. Adequacy of the LF encoding

The way to prove adequacy of the interpretation is by using so called ‘long-$\eta$-normal forms’. A long-$\eta$-normal form is obtained by first taking the $\beta$-normal form and then doing $\eta$-expansion: a term $C[M]$ in $\beta$-normal form $\eta$-expands to $C[\lambda x : A M x]$ if $x \notin \text{FV}(M)$. $M : \Pi x : A B$ and $C[\lambda x : A M x]$ is again in $\beta$-normal form. We write $I/\beta(\eta)(M)$ for the long-$\beta\eta$-normal form of the term $M$.

So, for proving adequacy of the encoding, it is most convenient to replace the $\beta$-conversion rule in $\lambda \mathcal{P}$ with a $\beta\eta$-conversion rule:

\[
\frac{\Gamma \vdash M : B \quad \Gamma \vdash A : \text{type}}{\Gamma \vdash M : A =_{\beta\eta} B}
\]

This is also what is done in LF: The type system of LF is $\lambda \mathcal{P}$ extended with $\eta$ in the
conversion rule. We will not go into this extension very deeply, but just remark that
adding this rule complicates the meta-theory quite a bit. Normalization is relatively
easy (shown in (Harper et al. 1987)), but confluence (of $\beta\eta$-reduction) is surprisingly
complicated and was first proved by (Salvesen 1989) and (Geuvers 1992).

We note that for proving adequacy of an encoding, the extension with $\eta$ is not strictly
required, but it does make the proof easier. (Maybe it is virtually the only way to prove it.) The argument runs as follows. Suppose we have a $\beta\eta$-conversion rule, so we work in

\[
\begin{align*}
\text{If } \Gamma_L, \vec{\beta} : T, \bar{\Delta} & \vdash M : T, \varphi, \\
\text{then } \Gamma_L, \vec{\beta} : T(\lambda\eta(\bar{\Delta})) & \vdash \beta\eta(M) : T(\lambda\eta(\varphi)) \text{ using } =_{\beta\eta},
\end{align*}
\]

then $\Delta' \vdash_{\varphi'}$ by a canonical translation from long-$\beta\eta$-nfs in the context $\Gamma_L$ to $L$. Now, by the fact
that the encoding of a formula of $L$ as a a term of type $\text{prop}$ yields a long-$\beta\eta$-nf, we are
done: $\Delta$, resp. $\varphi$ are exactly the LF-encoding of $\Delta'$, resp. $\varphi'$.

Having proved adequacy of the encoding in LF, we immediately conclude that the encoding in $\lambda\Pi$ is adequate as well, because $\lambda\Pi \subseteq LF$. More precisely: If $\Gamma_L, \vec{\beta} : T, \bar{\Delta} \vdash_{\lambda\Pi} M : T, \varphi$, then also $\Gamma_L, \vec{\beta} : T, \bar{\Delta} \vdash_{LF} M : T, \varphi$, and the rest of the argument runs as above.

We treat the adequacy for the encoding of minimal proposition logic (Example 2.4) in some more detail. Here we don’t need the $\beta\eta$-conversion rule at all. Technically: because there are no functions of higher type in the context $\Gamma$ of 2.4; in terms of the argument above: if $\Gamma \vdash_{\lambda\Pi} A : \text{prop}$, then $A \equiv \lambda\eta(A)$. So, we only need that $\Gamma \vdash_{\lambda\Pi} M : A \Rightarrow \Gamma \vdash_{\lambda\Pi} \lambda\eta(M) : A$, which is easily proved in $\lambda\Pi$.

Now, consider the context $\Gamma$ of 2.4. The proof of adequacy of the encoding proceeds
in three steps.

1. If $\Gamma, \bar{x} : \text{prop} \vdash_{\lambda\Pi} A : \text{prop}$, then either one of the following two is the case.

   \[
   A =_{\beta} \exists x. C \\
   \text{with } \Gamma, \bar{x} : \text{prop} \vdash_{\lambda\Pi} B, C : \text{prop}, \\
   A =_{\beta} x \text{ in the context.}
   \]

2. If $\Gamma, \bar{x} : \text{prop}, \vec{\beta} : T(\bar{A}) \vdash_{\lambda\Pi} t : TB$ (with $\bar{A} : \text{prop}$ and $B : \text{prop}$), then either one of the
   following three is the case.

   \[
   \begin{align*}
   t & =_{\beta} p \text{ with } B =_{\beta} A, p : A \text{ in the context, for some } A, \\
   t & =_{\beta} \exists x CDar \text{ with } \Gamma, \bar{x} : \text{prop}, \vec{\beta} : T(\bar{A}) \vdash_{\lambda\Pi} q : T(CD), r : TC \\
   & \text{and } B =_{\beta} D, \\
   t & =_{\beta} z CD(\lambda x : TC, q) \text{ with } \Gamma, \bar{x} : \text{prop}, \vec{\beta} : T(\bar{A}), z : TC \vdash_{\lambda\Pi} q : TD \\
   & \text{and } B =_{\beta} CD.
   \end{align*}
   \]

3. From the long-$\beta\eta$-nf of a term $t$ with $\Gamma, \bar{x} : \text{prop}, \vec{\beta} : T(\bar{A}) \vdash_{\lambda\Pi} t : TB$ (as in (2)) one can
   inductively define a derivation of the associated proposition in minimal first order
   proposition logic.

So there is an isomorphism between $\beta\eta$-equivalence classes of terms of type $TA$ in $\Gamma$ and
derivations of $A$ in the logic. The isomorphism is defined on the long-$\beta\eta$-normal forms,
which form a complete set of representants for the $\beta\eta$-equivalence classes.
4.2 A minimal version of $\lambda P$

Although the number of rules is limited, $\lambda P$ (or LF) is very powerful in interpreting a wide variety of formal systems. (See (Harper et al. 1987) or (Avron et al. 1987) for examples.) It is however not minimal yet: we can do without part of the $(\lambda)$-rule without weakening the power of the system. This is partly due to the way in which the system is being used. Once the context $\Gamma_L$ that represents the formal system has been established, one is only interested in judgements of the form

$$\Gamma_L \vdash_{\lambda P} M : A, \text{ with } A : \text{type}$$

On the other hand there is no reason to let the context $\Gamma_L$ not be in normal form. From these two principles we can show that half of the rule $(\lambda)$ is superfluous: there is no need to be able to form $\lambda x.A M : \Pi x.A B$ in case $\Pi x.A B : \text{kind}$. That is: everything that can be encoded in $\lambda P$ can already be encoded in $\lambda P^-$. See Definition 2.12.

We show that a $\beta_P$-normal form of a relevant judgement contains no $\lambda_P$ and that if a judgement contains no $\lambda_P$, it can be derived without the rule $(\lambda_P)$.

Lemma 4.1. If $\Gamma \vdash_{\lambda P} M : \text{type}$ or $\Gamma \vdash_{\lambda P} M : A(: \text{type})$, then $\beta_P$-nf$(M)$ contains no $\lambda_P$.

Proof. Suppose that $M$ is in $\beta_P$-nf and that $\Gamma \vdash_{\lambda P} M : \text{type}$ or $\Gamma \vdash_{\lambda P} M : A(: \text{type})$. We prove by induction on the structure of $M$ that it contains no $\lambda_P$. Note that $M \equiv \lambda x:B N$ is not possible, because then $M : A : \text{kind}$.

$M \equiv x i$ Then for all $i$, $\Gamma \vdash t_i : B_i(: \text{type})$ and $t_i$ in $\beta_P$-nf, so by induction hypothesis, $t_i$ contains no $\lambda_P$ and we are done.

$M \equiv \lambda x:B N$ Then $\Gamma, x : B \vdash N : C(: \text{type})$ and $\Gamma \vdash B : \text{type}$. Both are in $\beta_P$-nf, so we are done by using the induction hypothesis.

$M \equiv \Pi x:B.C$ Then $\Gamma, x : B \vdash C : \text{type}$ and $\Gamma \vdash B : \text{type}$. Both are in $\beta_P$-nf, so we are done by using the induction hypothesis.

$\square$

Proposition 4.2. If $\Gamma \vdash M : A$, $\Gamma$ and $M$ are in $\beta_P$-nf and contain no $\lambda_P$, then $\Gamma \vdash^- M : B$ for some $B =_{\beta} A$.

Proof. By induction on the length of $\Gamma + M$.

$M \equiv x$ Then $\Gamma \equiv \Gamma_1, x:B, \Gamma_2$ with $A =_{\beta} B$. Now $\Gamma_1 \vdash B : \text{type/kind}$ and the induction hypothesis applies, so $\Gamma \vdash^- B : \text{type/kind}$, so $\Gamma_1, x : B \vdash^- x : B$. Similarly for the declarations in $\Gamma_2$, so we find $\Gamma \vdash^- x : B$.

$M \equiv \lambda x:C.N$ Then $\Gamma, x : C \vdash N : D$, so the induction hypothesis applies and we find that $\Gamma, x : C \vdash^- N : D'$ for some $D =_{\beta} D'$. Now it must be the case that $\Gamma \vdash^- C : \text{type}$ and $\Gamma, x : C \vdash^- D' : \text{type}$, so $\Gamma \vdash^- \lambda x:C.N : \Pi x:C.D'$.

$M \equiv P N$ Then $\Gamma \vdash P : \Pi x:C.D$ and $\Gamma \vdash N : C$, so by induction hypothesis we find that $\Gamma \vdash^- P : \Pi x:C'.D'$ and $\Gamma \vdash^- N : C''$ with $C =_{\beta} C' =_{\beta} C''$. But then $\Gamma \vdash^- P N : D'[N/x]$ and $D''[N/x] =_{\beta} A$.

$M \equiv \Pi x:C.N$ Easy.

$\square$
Corollary 4.3. If $\Gamma \vdash M : \text{type}$, all in $\beta\nu$-normal form, then
\[ \Gamma \vdash M : A : \text{type}. \]

Proof. Suppose $\Gamma \vdash M : A$ with $\Gamma \vdash A : \text{type}$, all in $\beta\nu$-normal form. Then $\Gamma, M$ and $A$ do not contain any $\lambda\nu$ (Lemma 4.1), so we can apply the Proposition and find a $B =_{\beta} A$ such that $\Gamma \vdash M : B$. Moreover, $\Gamma \vdash A : \text{type}$, so $\Gamma \vdash M : A$ by the conversion rule.

Now, if $\Gamma$ is a $\lambda\nu$ context representing some system of logic and $A$ is a type that represents some formula of this logic, then we can assume $\Gamma$ and $A$ to be in $\beta\nu$-normal form. Now, when looking for a proof of $A$ in $\lambda\nu$, one only has to look at terms that do not contain a $\lambda\nu$: the (\nu\nu) rule can totally be ignored.

The previous Proposition says that the only real need for $\Pi x : A. B : \text{kind}$ is to be able to declare a variable in it. Even this use is usually of the most simple form where $x \notin \text{FV}(B)$. The standard application of it in both Automath systems and $\lambda\nu$ (certainly for logical systems) is the declaration of $T : \text{prop}\rightarrow\text{type}$, where $\text{prop} : \text{type}$ is another declaration.

We could even be more ‘minimal’ and not allow function types of the form $\Pi x_1 : A_1 \ldots \Pi x_n : A_n : \text{type}$, but instead add rules for ‘parametric constants’:
\[ \begin{align*}
\Gamma, x_1 : A_1, \ldots, x_n : A_n & \vdash c : \Pi(x_1 : A_1, \ldots, x_n : A_n). \text{type} \\
\Gamma \vdash t_1 : A_1[t_1/x_1] \ldots [t_{i-1}/x_{i-1}] & \quad (\forall i \leq n)
\end{align*} \]
\[ \Gamma \vdash c t_1 : \text{type} \]

Then a declared constant of type $\Pi x_1 : A_1 \ldots \Pi x_n : A_n : \text{type}$ can only be used in its ‘fully applied’ form, i.e. by applying it to $n$ values.

5. Concluding Remarks

We have studied first order dependent typed $\lambda$-calculus from two perspectives: first as a logical systems itself (via the formulas-as-types embedding) and second as a framework for defining logical systems. From the first perspective, we have proved the completeness of the formulas-as-types embedding of minimal first order predicate logic into $\lambda\nu$, which turned out to be remarkably intricate. It is known that this result does not extend to the embedding of higher order predicate into the the Calculus of Constructions (see Berardi 1990 and (Geuvers 1993)): completeness fails for the third order case and higher; for the second order case, the question of completeness is still open.

We have also seen that the rule that allows to $\lambda$-abstract over a type ($A : \text{type}$) to create a term of a kind ($B : \text{kind}$), does not contribute to the power of the system $\lambda\nu$. (This is defined as $\lambda\nu$-abstraction in Definition 2.12.) If we look at the formulas-as-types embedding, this addition is a conservative extension: see Corollary 3.37. If we look at $\lambda\nu$ as a logical framework, Corollary 4.3 shows that $\lambda\nu$-abstraction is superfluous.
References

A. Avron, F. Honsell and I. Mason, Using typed lambda calculus to implement formal systems on a machine. Report 87-31. LFCS Edingburgh, UK.

H.P. Barendregt, Typed lambda calculus. In Handbook of Logic in Computer Science, eds. Abramski et al., Oxford Univ. Press.

E. Barendsen and H. Geuvers, A guarantee for the first order predicate logic, Manuscript, Faculty of Mathematics and Computer Science, University of Nijmegen, Netherlands.

S. Benardi, Type dependence and constructive mathematics, Ph.D. thesis, Universita di Torino, Italy.


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


J.H. Geuvers, Logics and Type systems, PhD. Thesis, University of Nijmegen, Netherlands.


First order dependent type theory, λP

