

Chapter 8

Polymorphic λ -calculus

This chapter is concerned with a presentation of polymorphic λ -calculus: $\lambda 2$ or F . Typical for $\lambda 2$ is the presence of polymorphic types like for instance $\Pi a:*. a \rightarrow a$. The intuitive meaning of this type is ' $a \rightarrow a$ for every type a '. This is the type of the polymorphic identity. We discuss the correspondence between $\lambda 2$ and **prop2**, second-order propositional logic, via the Curry-Howard-De Bruijn isomorphism.

8.1 Polymorphic types: introduction

In simply typed λ -calculus, the identity on natural numbers

$$\lambda x:\text{nat}. x : \text{nat} \rightarrow \text{nat}$$

and the identity on booleans

$$\lambda x:\text{bool}. x : \text{bool} \rightarrow \text{bool}$$

are two different terms. The difference is only in the type of the function. The behaviour of both identity functions is the same: they both take one input and return it unchanged as an output. It may be useful to have just one identity function for all types, that can be instantiated to yield an identity function for a particular type. To start with, we write instead of `nat` or `bool` a type variable a , and obtain the identity on a :

$$\lambda x:a. x : a \rightarrow a.$$

We want to be able to instantiate the type variable a by means of β -reduction. This is done in $\lambda 2$ by an abstraction over the type variable a , which yields:

$$\lambda a:*. \lambda x:a. x : \Pi a:*. a \rightarrow a.$$

Here we use $*$ as notation for the set of all types. It corresponds to both **Set** and **Prop** in Coq. The function $\lambda a:*. \lambda x:a. x$ is called the polymorphic identity. The

type of this function is $\Pi a : * . a \rightarrow a$. This type can intuitively be seen as ‘for all types a : $a \rightarrow a$ ’. It is called a polymorphic type. The type constructor Π is used to build arrow types as in simply typed λ -calculus, and to build polymorphic types.

The abstraction over type variables as in the type $\Pi a : * . a \rightarrow a$ of the polymorphic identity is the paradigmatic property of $\lambda 2$. It is not present in simply typed λ -calculus.

Application is used to instantiate the polymorphic identity to yield an identity function of a particular type. For example:

$$\frac{\lambda a : * . \lambda x : a . x : \Pi a : * . a \rightarrow a \quad \text{nat} : *}{(\lambda a : * . \lambda x : a . x) \text{ nat} : \text{nat} \rightarrow \text{nat}}$$

and

$$\frac{\lambda a : * . \lambda x : a . x : \Pi a : * . a \rightarrow a \quad \text{bool} : *}{(\lambda a : * . \lambda x : a . x) \text{ bool} : \text{bool} \rightarrow \text{bool}}$$

As a second example we consider lists. Suppose we have a type `natlist` of finite lists of natural numbers, and a type `boollist` of finite lists of booleans. Suppose further for both sorts of lists we have a function that computes the length of a list:

$$\text{NLength} : \text{natlist} \rightarrow \text{nat}$$

and

$$\text{BLength} : \text{boollist} \rightarrow \text{nat}.$$

The behaviour of both functions is the same, and counting of elements does not depend on the type of the elements. Therefore also in this case it may be useful to abstract from the type of the elements of the list. To that end we first introduce the constructor `polylist` with $(\text{polylist } a)$ the type of finite lists of terms of type a , for every a . Then we can define the polymorphic length function

$$\text{polylength} : \Pi a : * . \text{polylist } a \rightarrow \text{nat}$$

Again we use application to instantiate the polymorphic length function to yield a length function on a particular type. We have for instance

$$\frac{\text{polylength} : \Pi a : * . (\text{polylist } a) \rightarrow \text{nat} \quad \text{nat} : *}{\text{polylength nat} : \text{polylist nat} \rightarrow \text{nat}}$$

and

$$\frac{\text{polylength} : \Pi a : * . (\text{polylist } a) \rightarrow \text{nat} \quad \text{bool} : *}{\text{polylength bool} : \text{polylist bool} \rightarrow \text{nat}}$$

8.2 $\lambda 2$

The presentation of $\lambda 2$ is along the lines of the presentation of λP in Chapter 6. We first build a set of pseudo-terms, and select the *terms* from the pseudo-terms using a typing system.

Symbols. We use the ingredients to build terms as for λP . We assume the following:

- a set Var consisting of infinitely many variables, written as x, y, z, \dots ,
- a symbol $*$,
- a symbol \square .

Besides these symbols, ingredients to build pseudo-terms are:

- an operator $\lambda_ : _ _$ for λ -abstraction,
- an operator $\Pi_ : _ _$ for product formation,
- an operator $(_ _)$ for application.

We often omit the outermost parentheses in an application.

Pseudo-terms. Also the definition of pseudo-terms is the same as for λP . The set of *pseudo-terms* of λP is defined by induction according to the following grammar:

$$P ::= \text{Var} \mid * \mid \square \mid \Pi \text{Var} : P . P \mid \lambda \text{Var} : P . P \mid (P P)$$

Some intuition about pseudo-terms:

- $*$ both represents the set of data-types and the set of propositions.
In Coq $*$ is split into two kinds, **Set** and **Prop**. In λP and $\lambda 2$ both of these are identified. This means that λP and $\lambda 2$ do not make the distinction between **Set** and **Prop** that Coq makes.
- \square represents the set of kinds.
- A product $\Pi x : A . B$ is the type of a function that takes as input an argument of type A , and gives back an output of type B where all occurrences of x are replaced by the argument.
If a product is not dependent, that is, if x does not occur in B , then we also write $A \rightarrow B$ instead of $\Pi x : A . B$.
- An abstraction $\lambda x : A . M$ is a function that takes as input something of type A . Here M is sometimes called the *body* of the abstraction.
- An application $(F M)$ is the application of a function F to an argument M .

Not all pseudo-terms are terms. For instance $(**)$ is not a term.

Environments. As before, an environment is a finite list of type declarations for distinct variables of the form $x : A$ with A a pseudo-term. An environment can be empty. Environments are denoted by Γ, Δ, \dots

Substitution. The substitution of P for x in M , notation $M[x := P]$ is defined by induction on the structure of M as follows:

1. $*[x := P] = *$,
2. $\square[x := P] = \square$,
3. $x[x := P] = P$,
4. $y[x := P] = y$,
5. $(\Pi y:A. B)[x := P] = \Pi y:A[x := P]. B[x := P]$,
6. $(\lambda x:A. M)[x := P] = \lambda x:A[x := P]. M[x := P]$,
7. $(MN)[x := P] = (M[x := P])(N[x := P])$.

We assume that bound variables are renamed whenever necessary in order to avoid unintended capturing of variables.

Typing system. In the typing system we can see the difference between $\lambda \rightarrow$ (simply typed λ -calculus), λP , and $\lambda 2$.

The typing system is used to select the terms from the pseudo-terms. **In these rules the parameter s can be either $*$ or \square .**

1. The *start rule*.

$$\overline{\vdash * : \square}$$

2. The *variable rule*.

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

3. The *weakening rule*.

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$$

4. The *product rule*.

$$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x:A. B : *}$$

5. The *abstraction rule*.

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

6. The *application rule*.

$$\frac{\Gamma \vdash F : \Pi x:A. B \quad \Gamma \vdash M : A}{\Gamma \vdash (FM) : B[x := M]}$$

7. The *conversion rule*.

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{with } B =_{\beta} B'$$

Some remarks concerning the typing system:

- As before, we write $A \rightarrow B$ instead of $\Pi x:A. B$ if x does not occur in A .
- Both $\lambda 2$ and λP are extensions of $\lambda \rightarrow$. In $\lambda \rightarrow$ we only have the product rules with both A and B in $*$.
- The typical products that are present in λP but not in $\lambda \rightarrow$ are the ones using the rule

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : \square}{\Gamma \vdash \Pi x:A. B : \square}$$

This rule is used to build for instance $\text{nat} \rightarrow *$, the type of the dependent lists.

This rule is present neither in $\lambda \rightarrow$ nor in $\lambda 2$.

- The typical products that are present in $\lambda 2$ but not in $\lambda \rightarrow$ are the ones using the rule

$$\frac{\Gamma \vdash A : \square \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x:A. B : *}$$

This rule is used to build for instance $\Pi a:*. a \rightarrow a$.

This rule is present neither in $\lambda \rightarrow$ nor in λP .

- Instantiation of a polymorphic function is done by means of application. For instance:

$$\frac{\Gamma \vdash \text{polyid} : \Pi a:*. a \rightarrow a \quad \Gamma \vdash \text{nat} : *}{\Gamma \vdash (\text{polyid nat}) : \text{nat} \rightarrow \text{nat}}$$

- An important difference between $\lambda 2$ and λP is that in $\lambda 2$ we can distinguish between terms and types whereas in λP we cannot.

Terms. If we can derive $\Gamma \vdash M : A$ for some environment Γ , then both pseudo-terms M and A are *terms*. Note that not all pseudo-terms are terms.

8.3 Properties of $\lambda 2$

We consider some properties of $\lambda 2$.

- The *Type Checking Problem* (TCP) is the problem whether $\Gamma \vdash M : A$ (given Γ , M , and A). The TCS is decidable for $\lambda 2$.
- The *Type Synthesis Problem* (TSP) is the problem $\Gamma \vdash M : ?$. So given Γ and M , does an A exist such that $\Gamma \vdash M : A$. The TSP is equivalent to the TCP and hence decidable.
- The *Type Inhabitation Problem* (TIP) is the problem $\Gamma \vdash ? : A$. So given an A , is there a *closed* inhabitant of A . The TIP is undecidable for $\lambda 2$.
- *Uniqueness of types* is the property that a term has at most one type up to β -conversion. $\lambda 2$ has the uniqueness of types property.
- Further, $\lambda 2$ has *subject reduction*. That is, if $\Gamma \vdash M : A$ and $M \rightarrow_{\beta} M'$, then $\Gamma \vdash M' : A$.

Note that in $\lambda 2$ types do not contain β -redexes.

- All β -reduction sequences in $\lambda 2$ are finite. That is, $\lambda 2$ is terminating, also called strongly normalizing.

8.4 Expressiveness of $\lambda 2$

Logic. In practical work 10 we have seen the definition of false, conjunction and disjunction in $\lambda 2$. The definitions are as follows:

$$\begin{aligned} \text{new_false} &= \Pi a : *. a \\ (\text{new_and } A B) &= \Pi c : *. (A \rightarrow B \rightarrow c) \rightarrow c \\ (\text{new_or } A B) &= \Pi c : *. (A \rightarrow c) \rightarrow (A \rightarrow c) \rightarrow c \end{aligned}$$

The idea is a connective is defined as an encoding of its elimination rule. Now we indeed have that all propositions follow from `new_false`:

$$\frac{\Gamma \vdash P : \text{new_false} \quad \Gamma \vdash A : *}{\Gamma \vdash (P A) : A}$$

As soon as we have an inhabitant (proof) of `new_false`, we can build an inhabitant (proof) of any proposition A , namely $(P A)$.

Suppose that $\Gamma \vdash P : (\text{new_and } A B)$ with $A : *$ and $B : *$. So P is an inhabitant (proof) of “ A and B ”, where conjunction is encoded using `new_and`. Using $L = \lambda l : A. \lambda r : B. l$ we find an inhabitant (proof) of A as follows:

$$\frac{\frac{\Gamma \vdash P : \Pi a : *. (A \rightarrow B \rightarrow a) \rightarrow a \quad \Gamma \vdash A : *}{\Gamma \vdash (P A) : (A \rightarrow B \rightarrow A) \rightarrow A} \quad \Gamma \vdash L : A \rightarrow B \rightarrow A}{\Gamma \vdash (P A L) : A}$$

Data-types. In Coq many datatypes like for instance `nat` and `bool` are defined as an inductive type. Often these datatypes can also be directly defined in polymorphic λ -calculus. This usually yields a less efficient representation. Here we consider as an example representations of the natural numbers and the booleans in $\lambda 2$. The examples do not quite show the full power of polymorphism.

Natural numbers. The type `N` of natural numbers is represented as follows;

$$\mathbf{N} = \Pi a:*. a \rightarrow (a \rightarrow a) \rightarrow a$$

The natural numbers are defined as follows:

$$\begin{aligned} 0 &= \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. o \\ 1 &= \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. (s\ o) \\ 2 &= \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. (s\ (s\ o)) \\ &\vdots \end{aligned}$$

That is, a natural number `n` is represented as follows:

$$\mathbf{n} = \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. s^n o$$

with the abbreviation n defined as:

$$\begin{aligned} F^0 M &= M \\ F^{n+1} M &= F(F^n M) \end{aligned}$$

This is the polymorphic version of the *Church numerals* as for instance considered in the ITI-course. Note that every natural number is indeed of type `N`. Now we can define a term for *successor* as follows:

$$\mathbf{S} = \lambda n:\mathbf{N}. \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. s\ (n\ a\ o\ s)$$

We have for instance the following:

$$\begin{aligned} \mathbf{S}\ 0 &= \\ (\lambda n:\mathbf{N}. \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. s\ (n\ a\ o\ s))\ 0 &\rightarrow_{\beta} \\ \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. s\ (0\ a\ o\ s) &= \\ \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. s\ ((\lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. o)\ a\ o\ s) &\rightarrow_{\beta}^* \\ \lambda a:*. \lambda o:a. \lambda s:a \rightarrow a. s\ o &= \\ 1. & \end{aligned}$$

Note that $n\ a\ o\ s \rightarrow^* s^n o$. An alternative definition for successor is $\lambda n:\mathbf{N}. \lambda o:a. \lambda s:a \rightarrow a. n\ a\ (s\ z)\ s$.

Booleans. The booleans are represented by a type `B`, and *true* and *false* are represented by terms `T` and `F` as follows:

$$\begin{aligned} \mathbf{B} &= \Pi a:*. a \rightarrow a \rightarrow a \\ \mathbf{T} &= \lambda a:*. \lambda x:a. \lambda y:a. x \\ \mathbf{F} &= \lambda a:*. \lambda x:a. \lambda y:a. y \end{aligned}$$

Note that we have

$$\begin{array}{l} \mathbf{T} : \mathbf{B} \\ \mathbf{F} : \mathbf{B} \end{array}$$

Now we can define a term for *conditional* as follows:

$$\mathbf{C} = \lambda a : *. \lambda b : \mathbf{B}. \lambda x : a. \lambda y : a. b a x y$$

with $\mathbf{C} : \Pi a : *. \mathbf{B} \rightarrow a \rightarrow a \rightarrow a$. The operator \mathbf{C} is similar to the conditional \mathbf{c} of system \mathbf{T} but there the polymorphism is implicit whereas here it is explicit.

Let $A : *$ and let $M : A$ and $N : A$. We have the following:

$$\begin{array}{l} \mathbf{C} \mathbf{A} \mathbf{T} M N \\ (\lambda a : *. \lambda b : \mathbf{B}. \lambda x : a. \lambda y : a. b a x y) \mathbf{A} \mathbf{T} M N \\ \mathbf{T} \mathbf{A} M N \\ (\lambda a : *. \lambda x : a. \lambda y : a. x) \mathbf{A} M N \\ M \end{array} \quad \begin{array}{l} = \\ \rightarrow_{\beta}^* \\ = \\ \rightarrow_{\beta}^* \end{array}$$

and

$$\begin{array}{l} \mathbf{C} \mathbf{A} \mathbf{F} M N \\ (\lambda a : *. \lambda b : \mathbf{B}. \lambda x : a. \lambda y : a. b a x y) \mathbf{A} \mathbf{F} M N \\ \mathbf{F} \mathbf{A} M N \\ (\lambda a : *. \lambda x : a. \lambda y : a. y) \mathbf{A} M N \\ N. \end{array} \quad \begin{array}{l} = \\ \rightarrow_{\beta}^* \\ = \\ \rightarrow_{\beta}^* \end{array}$$

We can also define negation, conjunction, disjunction, as follows:

$$\begin{array}{l} \text{not} = \lambda b : \mathbf{B}. \lambda a : *. \lambda x : a. \lambda y : a. b a y x \\ \text{and} = \lambda b : \mathbf{B}. \lambda b' : \mathbf{B}. \lambda a : *. \lambda x : a. \lambda y : a. b a (b' a x y) y \\ \text{or} = \lambda b : \mathbf{B}. \lambda b' : \mathbf{B}. \lambda a : *. \lambda x : a. \lambda y : a. b a x (b' a x y) \end{array}$$

8.5 Curry-Howard-De Bruijn isomorphism

This section is concerned with the static and the dynamic part of the Curry-Howard-De Bruijn isomorphism between second-order propositional logic and λ -calculus with polymorphic types ($\lambda 2$).

Formulas.

- A propositional variable a corresponds to a type variable a .
- A formula $A \rightarrow B$ corresponds to a type $\Pi x : A. B$, also written as $A \rightarrow B$.
- A formula of the form $\forall a. A$ corresponds to a type $\Pi a : *. B$.

Proofs.

- An assumption A corresponds to a variable declaration $x : A$.
- The implication introduction rule corresponds to the abstraction rule where a type of the form $\Pi x:A. B$ is introduced with $A : *$ and $B : *$.
- The implication elimination rule corresponds to the application rule.
- The universal quantification introduction rule corresponds to the abstraction rule where a type of the form $\Pi a:*. B$ is introduced.
- The universal quantification elimination rule corresponds to the application rule.

Questions.

- The question *is A provable?* in second-order propositional logic corresponds to the question *is A inhabited?* in $\lambda 2$.
- The question *is A a tautology?* in second-order propositional logic corresponds to the question *is A inhabited by a closed normal form?* in $\lambda 2$. That is the Type Inhabitation Problem (TIP). That is, provability corresponds to inhabitation.
- The question *is P a proof of A ?* in second-order propositional logic corresponds to the question *is P a term of type A ?* in $\lambda 2$. That is, proof checking corresponds to type checking (TCP).

The Dynamic Part.

- A \rightarrow -detour corresponds to a β -redex in $\lambda 2$.
- A \forall -detour corresponds to a β -redex in $\lambda 2$.
- A proof normalization step in second-order propositional logic corresponds to a β -reduction step in $\lambda 2$.

Examples. A proof of $(\forall b. b) \rightarrow a$ in prop2:

$$\frac{\frac{[(\forall b. b)^x]}{a}}{(\forall b. b) \rightarrow a} E\forall \quad I[x] \rightarrow$$

The type $(\Pi b : *. b) \rightarrow a$ corresponds to the formula $(\forall b. b) \rightarrow a$. We assume $a : *$. Then an inhabitant of $(\Pi b : *. b) \rightarrow a$ that corresponds to the proof given above is:

$$\lambda x : (\Pi b : *. b). (x a)$$