Logical Verification
Course Notes

Femke van Raamsdonk
`femke@cs.vu.nl`
Vrije Universiteit Amsterdam

autumn 2008

Contents

Chapter 1

1st-order propositional logic

This chapter is concerned with first-order propositional logic. First-order here means that there is no quantification over propositions, and propositional means that there is no quantification over terms. We consider mainly intuitionistic logic, where $A \vee \neg A$ is not valid in general. We also study classical logic. In addition, we consider minimal logic, which is the subset of intuitionistic logic with implication as only connective. In this setting we study detours and detour elimination.

1.1 Formulas

In this section we introduce the formulas of first-order propositional logic.

Symbols. We assume a set of *propositional variables*, written as $a, b, c, d, \ldots$. A *formula* or *proposition* in first-order propositional logic is built from propositional variables and logical connectives. We have the following binary connectives: $\rightarrow$ for implication, $\wedge$ for conjunction, and $\vee$ for disjunction. Further there is a constant (or zero-ary connective) falsum, denoted by $\perp$, and one for true, denoted by $\top$.

Formulas. The set of *formulas* is inductively defined by the following clauses:

1. a propositional variable a is a formula,
2. the constant $\perp$ is a formula,
3. the constant $\top$ is a formula,
4. if A and B are formulas, then $(A \rightarrow B)$ is a formula,
5. if A and B are formulas, then $(A \wedge B)$ is a formula,
6. if A and B are formulas, then $(A \vee B)$ is a formula.

Notation. We write arbitrary, unspecified, formulas as $A, B, C, D, \ldots$. Examples of formulas are: $(a \to a)$, $(a \to (b \to c))$, $((a \vee b) \wedge c)$. A formula in which we use some unspecified formulas, like $(A \to A)$, is also called a *formula scheme*. Usually we call a formula scheme somewhat sloppily also a formula.

As usual, we adopt some conventions that permit to write as few parentheses as possible, in order to make reading and writing of formulas easier:

1. We omit outermost parentheses.
2. Implication is supposed to be right associative.
3. $\wedge$ binds stronger than $\vee$ and $\to$, and $\vee$ binds stronger than $\to$.

For example: instead of $(A \to B)$ we write in the sequel $A \to B$, instead of $(A \to (B \to C))$ we write $A \to B \to C$, and instead of $((A \to B) \to C)$ we write $(A \to B) \to C$. Further, we write $A \wedge B \to C$ instead of $((A \wedge B) \to C)$ and sometimes $A \wedge B \vee C$ instead of $((A \wedge B) \vee C)$.

Negation. We have one defined connective which takes one argument: *negation*, written as $\neg$. Its definition is as follows: $\neg A := A \to \bot$.

1.2 Natural deduction for intuitionistic logic

In this section we consider a natural deduction proof system for intuitionistic first-order propositional logic.

There are various proof systems for natural deduction, that differ mainly in notation. Here we write proofs as trees. Using the tree notation, an unspecified proof of the formula A is written as follows:

$$\begin{array}{c} \vdots \\ A \end{array}$$

The most important characteristic of a natural deduction proof system, which all systems have in common, is that for every logical connective there are introduction and elimination rules. An *introduction rule* for a connective describes how a formula with that connective can be derived (or introduced). An *elimination rule* for a connective describes how a formula with that connective can be used (or eliminated). A proof further makes use of *assumptions* also called *hypotheses*. The intuition is that if there is a proof of B using assumptions $A_1, \ldots, A_n$, then B is a logical consequence of $A_1, \ldots, A_n$.

Proof rules. Below the rules for a natural deduction proof system for intuitionistic first-order propositional logic are given. In the implication introduction rule the phenomenon of *cancelling assumptions* occurs. We label the assumptions in a proof with labels $x, y, z, \ldots$ in order to make explicit which assumption is cancelled in an application of the implication introduction rule.

1. The *assumption rule.*

   A labelled formula A^x is a proof.

   $$A^x$$

   Such a part of a proof is called an *assumption.*

2. The *implication introduction rule.*

   If

   $$\begin{array}{c} \vdots \\ B \end{array}$$

   is a proof, then we can introduce an implication by cancelling all assumptions of the form A^x. In this way we obtain the following proof:

   $$\cfrac{\begin{array}{c} \vdots \\ B \end{array}}{A \to B}\ I[x]{\to}$$

   Note that we mention explicitly the label indicating which occurrences of assumption A are cancelled. All occurrences of the assumption A having label x are cancelled. Assumptions A with a different label are not cancelled.

3. The *implication elimination rule.*

   If we have a proof of $A \to B$ and one of A, as follows:

   $$\begin{array}{cc} \vdots & \vdots \\ A \to B & A \end{array}$$

   then we can combine both proofs by applying the implication elimination rule. This yields a (one) proof of B, as follows:

   $$\cfrac{\begin{array}{cc} \vdots & \vdots \\ A \to B & A \end{array}}{B}\ E{\to}$$

4. The *conjunction introduction rule.*

   If we have a proof of A and a proof of B, then those two proofs can be combined to form a proof of $A \wedge B$:

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}\; I\wedge$$

5. The *conjunction elimination rules.*

   There are two elimination rules for the conjunction: one used to turn a proof of $A \wedge B$ into a proof of A, and one to turn a proof of $A \wedge B$ into a proof of B. The first one is called the left conjunction elimination rule and the second one is called the right conjunction elimination rule. They are as follows:

$$\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{A}\, El\wedge \qquad\qquad \dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{B}\, Er\wedge$$

6. The *disjunction introduction rules.*

   For disjunction, there are two introduction rules. The first one is used to turn a proof of A into a proof of $A \vee B$, and the second one is used to turn a proof of B into a proof of $A \vee B$. They are called the left and right disjunction introduction rule. They are as follows:

$$\dfrac{\begin{array}{c}\vdots \\ A\end{array}}{A \vee B}\, Il\vee \qquad\qquad \dfrac{\begin{array}{c}\vdots \\ B\end{array}}{A \vee B}\, Ir\vee$$

7. The *disjunction elimination rule.*

   The disjunction elimination rule expresses how a formula of the form $A \vee B$ can be used. If both from A and from B we can conclude C, and in addition we have $A \vee B$, then we can conclude C. This is expressed by the disjunction elimination rule:

$$\dfrac{\begin{matrix}\vdots & \vdots & \vdots \\ A \vee B & A \to C & B \to C\end{matrix}}{C}\ E\vee$$

8. The *falsum rule.*

   There is no introduction rule for falsum. The falsum rule expresses how a proof with conclusion falsum can be transformed into a proof of an arbitrary formula A, so it is fact an elimination rule. It is as follows:

$$\dfrac{\begin{matrix}\vdots \\ \bot\end{matrix}}{A}$$

9. The *true rule.*

   There is only one rule for $\top$, namely the following introduction rule without premisses:

$$\top$$

Comments.

- The last formula of a proof is called its *conclusion.*
- The implication elimination rule, the conjunction introduction rule, and the disjunction elimination rule combine two or three proofs into one. It is due to these rules that the proofs have a tree-like structure.
- The most complicated aspect of proofs in natural deduction is the management of cancelling assumptions. If an assumption is cancelled, this is indicated by putting brackets ($[,]$) around it. See also the examples.
- Assumptions are labelled in order to be able to make explicit which assumptions are cancelled using an implication introduction rule. This is the only reason of using labels; that is, labels are not carried around elsewhere in the proof. For instance in the proof

$$\dfrac{[A^x]}{A \to A}\ I[x]{\to}$$

the occurrences of A in the conclusion $A \to A$ are not labelled.

- We assume that distinct assumptions have distinct labels. So it is not possible to have assumptions A^x and B^x in one and the same proof.
- It is possible to have a certain formula A more than once as an assumption in a proof. Those assumptions do not necessarily have the same label. For instance, we can have a proof with assumptions A^x, A^x, A^y, or A^x, A^y, A^z, or A^x, A^x, A^x, etcetera.

Tautologies. If there is a proof of A using zero assumptions, that is, all assumptions are cancelled (using the implication introduction rule), then A is said to be a *tautology* or *theorem*. Note that if a formula A is a tautology, then $B \to A$ is a tautology as well: we can extend the proof of A

$$\begin{array}{c} \vdots \\ A \end{array}$$

by adding an implication introduction rule that cancels zero assumptions B:

$$\dfrac{\begin{array}{c} \vdots \\ A \end{array}}{B \to A}\; I[x]{\to}$$

Here the label x must be fresh, that is, not used elsewhere in the proof of A.

Examples. We consider a few proofs in first-order propositional logic.

1. A proof showing that $A \to A$ is a tautology:

$$\dfrac{[A^x]}{A \to A}\; I[x]{\to}$$

2. A proof showing that $A \to B \to A$ is a tautology:

$$\dfrac{\dfrac{[A^x]}{B \to A}\; I[y]{\to}}{A \to B \to A}\; I[x]{\to}$$

3. A proof showing that $(A \to B \to C) \to (A \to B) \to A \to C$ is a tautology:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[(A\to B\to C)^x] \qquad [A^z]}{B\to C}\,E{\to} \qquad \dfrac{[(A\to B)^y] \qquad [A^z]}{B}\,E{\to}}{C}\,E{\to}}{A\to C}\,I[z]{\to}}{(A\to B)\to A\to C}\,I[y]{\to}}{(A\to B\to C)\to(A\to B)\to A\to C}\,I[x]{\to}$$

4. We give two different proofs showing that $A\to A\to A$ is a tautology. The difference is in the cancelling of assumptions. In the next chapter we will see that the two different proofs correspond to two different λ-terms.

   A first proof:

$$\dfrac{\dfrac{[A^x]}{A\to A}\,I[y]{\to}}{A\to A\to A}\,I[x]{\to}$$

   A second proof:

$$\dfrac{\dfrac{[A^x]}{A\to A}\,I[x]{\to}}{A\to A\to A}\,I[y]{\to}$$

5. (permutation)
   $(A\to B\to C)\to(B\to A\to C)$.

6. (weak law of Peirce)
   $(((((A\to B)\to A)\to A)\to B)\to B)$.

7. (contrapositive)
   A proof showing that $(A\to B)\to\neg B\to\neg A$ is a tautology:

$$\dfrac{\dfrac{\dfrac{\dfrac{[B\to\bot^y] \qquad \dfrac{[A\to B^x] \qquad [A^z]}{B}\,E\to}{\bot}\,E\to}{A\to\bot}\,I[z]\to}{\neg B\to\neg A}\,I[y]\to}{(A\to B)\to(\neg B\to\neg A)}\,I[x]\to$$

8. A proof showing that $A \to \neg\neg A$ is a tautology:

$$\dfrac{\dfrac{\dfrac{[A \to \bot^{y}] \qquad [A^{x}]}{\bot}\; E \to}{(\neg A) \to \bot}\; I[y] \to}{A \to \neg\neg A}\; I[x] \to$$

   NB: the implication $\neg\neg A \to A$ is not valid in intuitionistic logic.

9. $\neg\neg(\neg\neg A \to A)$.

Intuitionism. Proof checkers based on type theory, like for instance Coq, work with intuitionistic logic, sometimes also called constructive logic. This is the logic of the natural deduction proof system discussed so far. The intuition is that truth in intuitionistic logic corresponds to the existence of a proof. This is particularly striking for disjunctions: we can only conclude $A \vee B$ if we have a proof of A or a proof of B. Therefore, $A \vee \neg A$ is not a tautology of intuitionistic logic: it is not the case that for every proposition A we have either a proof of A or a proof of $\neg A$.

In classical logic, the situation is different. Here the notion of truth that is absolute, in the sense that it is independent of whether this truth can be observed. In classical logic, $A \vee \neg A$ is a tautology because every proposition is either true or false.

Interpretation. There is an intuitive semantics of intuitionistic logic due to Brouwer, Heyting and Kolmogorov. It is also called the BHK-interpretation. This interpretation explains what it means to prove a formula in terms of what it means to prove its components. It is as follows:

- A proof of $A \to B$ is a method that transforms a proof of A into a proof of B.
- A proof of $A \wedge B$ consists of a proof of A and a proof of B.
- A proof of $A \vee B$ consists of first, either a proof of A or a proof of B, and second, something indicating whether it is A or B that is proved.
- (There is no proof of $\bot$.)

1.3 Detours in minimal logic

We study a subset of intuitionistic logic which is called *minimal logic.* In minimal logic, the only logical connective is the one for implication. Also the proof rules are restricted accordingly: there are only the rules for assumption and implication. We are interested in minimal logic because, as we will see in the next chapter, it corresponds to simply typed λ-calculus.

Detours. We have seen already that the formula $A \to B \to A$ is a tautology:

$$\dfrac{\dfrac{[A^x]}{B \to A}\, I[y]{\to}}{A \to B \to A}\, I[x]{\to}$$

This is in fact in some sense the easiest way to show that $A \to B \to A$ is a tautology. An example of a more complicated way is the following:

$$\dfrac{\dfrac{\dfrac{[A^x] \qquad \dfrac{[A^z]}{A \to A}\, I[z]{\to}}{A}\, E{\to}}{B \to A}\, I[y]{\to}}{A \to B \to A}\, I[x]{\to}$$

This proof contains an application of the implication introduction rule immediately followed by an application of the implication elimination rule:

$$\dfrac{[A^x] \qquad \dfrac{[A^z]}{A \to A}\, I[z]{\to}}{A}\, E{\to}$$

Such a part is called a detour.

More generally, a *detour* in minimal first-order propositional logic is a part of a proof consisting of the introduction of an implication immediately followed by the elimination of that implication. A detour has the following form:

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ B\end{matrix}}{A \to B}\, I[x]{\to} \qquad \begin{matrix}\vdots\\ A\end{matrix}}{B}\, E{\to}$$

Detour elimination. In the example above, instead of the fragment

$$\dfrac{[A^x] \qquad \dfrac{[A^z]}{A \to A}\, I[z]{\to}}{A}\, E{\to}$$

we can use the smaller fragment

$$[A^x]$$

The replacement of the fragment with the detour by the more simple fragment is called *detour elimination* or *proof normalization.*

The general rule for detour elimination or proof normalization in minimal first-order propositional logic is as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{B}}{A \to B}\,I[x]{\to} \qquad \dfrac{\vdots}{A}}{B}\,E{\to} \qquad\qquad \to \qquad\qquad \dfrac{\vdots}{B}$$

A proof that does not contain a detour is said to be a *normal proof.* As another example, consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{[A^x]}{B \to A}\,I[y]{\to}}{A \to B \to A}\,I[x]{\to} \qquad \dfrac{\vdots}{A}}{B \to A}\,E{\to} \qquad \dfrac{\vdots}{B}}{A}\,E{\to}$$

One step of proof normalization results in this proof:

$$\dfrac{\dfrac{\dfrac{\vdots}{A}}{B \to A}\,I[y]{\to} \qquad \dfrac{\vdots}{B}}{A}\,E{\to}$$

In another step we obtain:

$$\dfrac{\vdots}{A}$$

Every proof can be transformed into a proof without detours by detour elimination.

1.4 From intuitionistic to classical logic

Intuitively, a formula in intuitionistic logic is valid if it has a proof, and a formula in classical logic is valid if it is true. There are several ways of extending intuitionistic logic to classical logic. One way is by adding the axiom scheme of the law of excluded middle (tertium non datur), which is as follows:

$$A \vee \neg A$$

This means that for any formula A we have $A \vee \neg A$. Here A can be instantiated by any formula.

Illustration. As an illustration (which uses predicates) of the fact that in intuitionistic logic less can be proved than in classical logic, we consider the following question:
Are there real numbers m and n such that m and n are irrational and m^n is rational?
In classical logic, we can prove that the answer is yes. In intuitionistic logic, this is not possible. The non-constructive proof, making use of the $A \vee \neg A$ axiom, is as follows. Let $m = \sqrt{2}$ and $n = \sqrt{2}^{\sqrt{2}}$. Two cases are distinguished: either n is irrational, or not.

- If n is irrational, then consider

$$n^m = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^2 = 2$$

  which is rational.
- If n is rational, then we are done immediately.

Here we use the elimination rule for $\vee$ in the following form:

- If $\sqrt{2}^{\sqrt{2}}$ is rational, then the answer is yes.
- If $\sqrt{2}^{\sqrt{2}}$ is not rational (that is, irrational), then the answer is yes.
- We have that $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational (because of the axiom scheme $A \vee \neg A$).

The conclusion is then that the answer is yes.

This proof doesn't yield a concrete example of two irrational numbers that have a rational exponentiation. In constructive logic this would be required.

Alternative approaches. It turns out that there are several alternative approaches to obtain classical logic from intuitionistic logic. Here we mention three of them:

- Add $A \vee \neg A$ (the law of excluded middle, EM) as an axiom.
- Add $\neg\neg A \to A$ (the rule for double negation, DN) as an axiom.
- Add $((A \to B) \to A) \to A$ (Peirce's law, PL) as an axiom.

These three approaches are equivalent.

In the practical work, we prove EM $\to$ PL (Lemma one), PL $\to$ DN (Lemma two), DN $\to$ EM (Lemma three).

Examples. We give a few examples of proofs in classical first-order propositional logic.

We show EM $\to$ DN. Recall that $\neg\neg A$ is defined as $(A \to \bot) \to \bot$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[A^z]}{A \to A}\,I[z]\to \qquad A \vee \neg A \qquad \dfrac{\dfrac{\dfrac{[\neg\neg A^x] \qquad [\neg A^y]}{\bot}\,E\to}{A}}{\neg A \to A}\,I[y]\to}{A}\,E\vee}{\neg\neg A \to A}\,I[x]\to}{}$$

Note that $((A \to \bot) \to \bot) \to A$ is *not* a tautology in intuitionistic logic. However, $A \to \neg\neg A$ is a tautology of intuitionistic logic.

Another example: Using both the law of the excluded middle, and the double negation rule derived above, we can show that the formula $((A \to B) \to A) \to A$ is a tautology of classical logic. This formula is known as *Peirce's Law.* A proof showing that Peirce's law is a tautology of classical propositional logic:

$$\dfrac{\dfrac{\dfrac{[(A \to \bot)^y] \qquad \dfrac{[((A \to B) \to A)^x] \qquad \dfrac{\dfrac{\dfrac{[(A \to \bot)^y] \qquad [A^z]}{\bot}\,E\to}{B}}{A \to B}\,I[z]\to}{A}\,E\to}{\bot}\,E\to}{\neg\neg A = (A \to \bot) \to \bot}\,I[y]\to$$

$$\dfrac{\dfrac{\neg\neg A = (A \to \bot) \to \bot}{A}\,nn}{((A \to B) \to A) \to A}\,I[x]\to$$

Here nn stands for double negation.

Peirce's Law is another example of a formula that is a tautology of classical logic but not of intuitionistic logic.

1.5 1st-order propositional logic in Coq

Coq can be used to prove that a formula in first-order propositional logic is a tautology. First-order propositional logic is decidable, and indeed we could use just one Coq command, namely `tauto`, to do this. However, we will use instead commands that more or less correspond to the natural deduction proof system given above. This also helps to realize what the internal representation of proofs in Coq is, a subject that will be treated in Chapters 2 and further.

A proof in Coq is built bottom-up. In every step there are one or more current goals, and a list of current assumptions. The current goals are the formulas we wish to prove, and the assumptions are the formulas that may be used. The first of the current goals is written below, and the assumptions are written above the line. The remaining subgoals are also mentioned.

The first step is to specify the lemma we wish to prove. This is done using the syntax `Lemma` from the specification language of Coq, called Galina. (There are alternatives; one can for instance also use `Goal`.) Then the statement of the lemma becomes the current goal, and the current list of assumptions is empty.

In the following steps, the user specifies how to transform the current goal into zero, one or more new goals. This is done using tactics. If there is no current subgoal anymore, the original goal is proved and we are done. All what remains is to save the proof, using `Save` or `Qed`.

In the Coq session spelled out below, we are concerned with formulas of minimal logic, only using $\to$. The main two tactics to use are then `intro`, which corresponds to the implication introduction rule, and `apply`, which corresponds to the implication elimination rule. All unknown formulas in the lemma have to be declared as variables of type `Prop`.

If the first of the current goals is a formula of the form $A \to B$, then there are two possibilities. We can introduce new assumption which is done by applying the tactic `intro`. As argument we can give a fresh label, say x. In this way the list of assumptions is extended with an assumption A with label x. If the formula $A \to B$ happens to be in the current list of assumptions, say with label x, then we can apply the tactic `assumption`, or `exact` with argument x.

If the first of the current goals is a formula B without $\to$, then we look in the list of assumptions whether there is an assumption of the form $A_1 \to \ldots \to A_n \to B$, with $n \geq 0$. If there is one, say with label x, then we apply the tactic `apply` with argument x. This yields n new goals, namely $A_1, \ldots, A_n$. If $n = 0$ then the goal is proved, and we continue with the remaining ones, if any.

Examples. This is an example of a Coq session in which two formulas are proved to be tautologies. First we prove $A \to A$ in Coq:

```
Coq < Parameter A B C : Prop .
A is assumed
B is assumed
C is assumed

Coq < Lemma I : A -> A .
1 subgoal

  ============================
   A -> A

I < intro x .
1 subgoal

  x : A
  ============================
   A
```

```
I < exact x .
Proof completed.

I < Qed .
intro x.
exact x.
I is defined
```

We continue the session and prove $(A \to B \to C) \to (A \to B) \to A \to C$:

```
Coq < Lemma S : (A -> B -> C) -> (A -> B) -> A -> C .
1 subgoal

  ============================
   (A -> B -> C) -> (A -> B) -> A -> C

S < intro x .
1 subgoal

  x : A -> B -> C
  ============================
   (A -> B) -> A -> C

S < intro y .
1 subgoal

  x : A -> B -> C
  y : A -> B
  ============================
   A -> C

S < intro z .
1 subgoal

  x : A -> B -> C
  y : A -> B
  z : A
  ============================
   C

S < apply x .
2 subgoals

  x : A -> B -> C
  y : A -> B
  z : A
  ============================
   A

subgoal 2 is:
 B

S < exact z .
1 subgoal

  x : A -> B -> C
  y : A -> B
  z : A
  ============================
   B

S < apply y .
1 subgoal

  x : A -> B -> C
  y : A -> B
  z : A
  ============================
   A

S < exact z .
Proof completed.

S < Qed .
intro x.
intro y.
intro z.
apply x.
 exact z.
 apply y.
   exact z.
S is defined
```

Example. As another example we consider a proof of the formula $(A \to B) \to (C \to A) \to C \to B$. On the left we have the different stages of constructing a proof in minimal logic, and on the right are the corresponding steps in Coq. To save space we do not indicate which rule is used in the minimal logic part.

Lemma
$(A \to B) \to (C \to A) \to C \to B.$

```
Lemma example :
(A->B)->(C->A)->C->B.
```

$$\frac{}{(A \to B) \to (C \to A) \to C \to B}$$

Goal:

```
(A->B)->(C->A)->C->B.
```

```
intro x.
```

Assumption: `x:A->B`

$$\frac{(C \to A) \to C \to B}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `(C->A)->C->B`

```
intro y.
```

Assumptions:

```
   x:A->B
   y:C->A
```

$$\dfrac{\dfrac{C \to B}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `C->B`

```
intro z.
```

Assumptions:

```
   x:A->B
   y:C->A
   z:C
```

$$\dfrac{\dfrac{\dfrac{B}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `B`

```
apply x.
```

Assumptions do not change

$$\dfrac{\dfrac{\dfrac{\dfrac{A \to B \quad A}{B}}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `A`

```
apply y.
```

Assumptions do not change

$$\dfrac{\dfrac{\dfrac{\dfrac{A \to B \quad \dfrac{C \to A \quad C}{A}}{B}}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `C`

```
exact z.
```

done!

Proof completed.

Notation.

- $\perp$ is in Coq written as `False`.
- $A \to B$ is in Coq written as `A -> B`.
- $A \wedge B$ is in Coq written as `A /\ B`.
- $A \vee B$ is in Coq written as `A \/ B`.
- $\neg A$ is in Coq written as `~A` or as `not A`.

  Negation is defined in Coq as in propositional logic: an expression `not A` in Coq is an abbreviation for `A -> False`.

Tactics.

- `assumption .`

  This tactic applies to any goal. It looks in the list of assumptions for an assumption which equals the current goal. If there is such an assumption, the current goal is proved, and otherwise it fails.

- `exact` *id* `.`

  This tactic applies to any goal. It succeeds if the current goal and the assumption with label *id* are convertible (to be explained later).

- `intro` *label* `.`

  This tactic is applicable if the first of the current goals is of the form $A \to B$. It extends the list of assumptions with an assumption A with label *label*, and transforms the first of the current goals into B.

  This tactic corresponds to the implication introduction rule.

  There are some variants of the tactic `intro` of which we mention some.

- `intro .`

  This does the same as `intro` *label* `.`, except that Coq chooses a label itself.

- `intros` *label1* ... *labeln* `.`

  This tactic repeats `intro` n times. It is applicable if the current goal is of the form $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$ with $n \geq 1$. It transforms this goal into the new goal B, and extents the list of assumptions with $A_1 \ldots, A_n$ with labels *label1* ... *labeln*.

- `intros .`

  This tactic repeats `intro` as often as possible, and works further in the same way as `Intros` *label1* ... *labeln* `.`, except that Coq chooses the labels itself.

- `apply` *label* `.`

  This tactic applies to any goal. It tries to match the current goal with the conclusion of the assumption with label *label*. If the current goal is B, and the assumption with label *label* is $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$, then the result of applying `apply` is that the new goals are $A_1, \ldots, A_n$. If $n = 0$, then there are no new goals.

  If the matching fails the goal is not changed, and an error message is given.

- `split .`

  For the moment we use this tactic only in the situation that the current goal is of the form `A /\ B`. Applying the tactic `Split` then yields that the goal is transformed into two new goals: `A` and `B`.

  In this use the tactic corresponds to the conjunction introduction rule.

- `elim` *label* `.`

  This tactic applies to any goal; *label* is the label of a hypothesis in the current list of hypotheses. In general the tactic `Elim` is used for reasoning by cases. Here we consider two uses.

  If *label* is the label of a hypothesis of the form `A /\ B` and the current goal is `C`, then `elim` *label* transforms the goal into `A -> B -> C`.

  In this use the tactic corresponds roughly to the conjunction elimination rules.

  If *label* is the label of a hypothesis of the form `A \/ B` and the current goal is `C`, then `elim` *label* transforms the goal into two goals: `A -> C` and `B -> C`.

  In this use the tactic corresponds to the disjunction elimination rule.

  If *label* is the label of a hypothesis of the form `False` then the current goal can be solved using `elim` *label*.

  In this use the tactic corresponds to the falsum elimination rule.

- `left .`

  If the current goal is of the form `A \/ B`, then this tactic transforms it into the new goal `A`. (There are other uses of this tactic.)

  In this use the tactic corresponds to the left disjunction introduction rule.

- `right .`

  If the current goal is of the form `A \/ B`, then this tactic transforms it into the new goal `B`. (There are other uses of this tactic.)

  In this use the tactic corresponds to the right disjunction introduction rule.

- `tauto .`

  This tactic succeeds if the current goal is an intuitionistic propositional tautology.

Proof Handling.

- `Goal` *form* `.`

  This command switches Coq to interactive editing proof-mode. It set *form* as the current goal.

- `Lemma` *id* `:` *form* `.`

  This command also switches Coq to interactive editing proof-mode. The current goal is set to *form* and the name associated to it is *id*.

- `Theorem` *id* `:` *form* `.`

  Same as `Lemma` except that now a theorem instead of a lemma is declared.

- `Abort .`

  This command aborts the current proof session and switches back to Coq top-level, or to the previous proof session if there was one.

- `Undo .`

  This command cancels the effect of the last tactics command.

- `Restart .`

  This command restarts the proof session from scratch.

- `Qed .`

  This command can be used if the proof is completed. If the name of the proof session is *id*, then a proof with name *id* is stored (and can be used later).

- `Save .`

  Equivalent to `Qed`. If an argument is specified, the proof is stored with that argument as name.

Miscellaneous.

- `Load` *id* `.`

  This command feeds the contents of the file *id*.v to Coq. The effect is the same as if the contents of the file *id*.v is typed in interactive proof editing mode in Coq.

- `Quit .`

  This command permits to quit Coq.

- Comments can be written between (* and *).

Chapter 2

Simply typed λ-calculus

The λ-calculus is a language to express functions, and the evaluation of functions applied to arguments. It was developed by Church in the 1930s. The untyped λ-calculus provides a formalization of the intuitive notion of effective computability. This is expressed by what is now known as *Church's Thesis*: all computable functions are definable in the λ-calculus. The expressivity of untyped λ-calculus is equivalent to that of Turing Machines or recursive functions.

The λ-calculus can be considered as a functional programming language in its purest form. It indeed forms the basis of functional programming languages like for instance Lisp, Scheme, ML, Miranda, and Haskell.

This chapter is concerned with simply typed λ-calculus. We discuss the correspondence between simply typed λ-calculus and minimal first-order propositional logic. This correspondence is called the Curry-Howard-De Bruijn isomorphism and forms the basis of Coq and other proof checkers that are based on type theory.

2.1 Types

Simple types. We assume a set of *type variables*, written as $a, b, c, d, \ldots$. A *simple type* is either a type variable or an expression of the form $A{\to}B$, with A and B simple types. So the set of simple types is inductively defined as follows:

1. a type variable a is a simple type,
2. if A and B are simple types, then $(A{\to}B)$ is a simple type.

We write arbitrary simple types as $A, B, C, D, \ldots$. If it is clear that we work in simply typed λ-calculus, then we often say *type* instead of simple type. Examples of simple types are $(A{\to}A)$, $(A{\to}(B{\to}C))$, and $((A{\to}B){\to}C)$.

As in the case of formulas, we adopt two conventions concerning the notation of simple types that make reading and writing them easier:

- Outermost parentheses are omitted.

- The type constructor $\to$ is assumed to be associative to the right.

For example: instead of $(A{\to}B)$ we write $A{\to}B$, instead of $(A{\to}(B{\to}C))$ we write $A{\to}B{\to}C$, and instead of $((A{\to}B){\to}C)$ we write $(A{\to}B){\to}C$.

A type of the form $A{\to}B$ is called a *function type*. For example, $\mathsf{nat}{\to}\mathsf{bool}$ is the type of a function that takes an argument of type nat and yields a result of type bool. Further, $\mathsf{nat}{\to}\mathsf{nat}{\to}\mathsf{bool}$ is the type of a function that takes an argument of type nat and yields as a result a function of type $\mathsf{nat}{\to}\mathsf{bool}$.

2.2 Terms

Abstraction and application. The λ-calculus can be used to represent functions, and the application of a function to its argument.

A function that assigns the value M to a variable x, with x of type A, is written as the *abstraction* $\lambda x{:}A.\, M$. Since we consider only typed terms here, the term M has a type, say B. The function $\lambda x{:}A.\, M$ then has type $A{\to}B$. For instance, the identity function on natural numbers is written as $\lambda x{:}\mathsf{nat}.\, x$.

If F is a function of type $A{\to}B$, and P is of type A, then we can form the *application* of F to P, written as $(F\, P)$. The application of the identity function to the argument 5 is written as $((\lambda x{:}\mathsf{nat}.\, x)\, 5)$.

Environments. An environment is a finite set of type declarations for distinct variables of the form $x : A$. Examples of environments are $x : A, y : B$ and $x : A, y : A$. The set $x : A, x : B$ is not an environment since it contains two declarations for x. Environments are denoted by $\Gamma, \Delta, \ldots$. The order of the declarations in an environment is irrelevant. If we write $\Gamma, x : A$ we assume that Γ does not contain a declaration for x.

Simply typed λ-terms. We assume a countably infinite set of *variables* written as $x, y, z, \ldots$. The set of variables is denoted by Var. Simply typed λ-terms are built from variables, abstraction and application. The set of *simply typed λ-terms* consists of the terms M for which we can derive $\Gamma \vdash M : A$ for some environment Γ and some type A using the following three rules:

1. The *variable rule.*

   If a variable is declared to be of type A, then it is a λ-term of type A.

   $$\Gamma, x : A \vdash x : A$$

2. The *abstraction rule.*

   If M is a simply typed λ-term of type B, and x is a variable of type A, then $(\lambda x{:}A.\, M)$ is a simply typed λ-term of type $A \to B$.

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

3. The *application rule.*

   If F is a simply typed λ-term of type $A \to B$, and N is a simply typed λ-term of type A, then $(F\,N)$ is a simply typed λ-term of type B.

$$\frac{\Gamma \vdash F : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (F\,N) : B}$$

Parentheses. Again we adopt some conventions concerning parentheses:

- We write $(M\,N\,P)$ instead of $((M\,N)\,P)$, so application is assumed to be *associative to the left.*
- We write $(\lambda x{:}A.\, \lambda y{:}B.\, M)$ instead of $(\lambda x{:}A.\, (\lambda y{:}B.\, M))$.
- We write $(\lambda x{:}A.\, M\,N)$ instead of $(\lambda x{:}A.\, (M\,N))$.
- We write $(M\,\lambda x{:}A.\, N)$ instead of $(M\,(\lambda x{:}A.\, N))$.

Then, we also adopt the following convention:

- Outermost parentheses are omitted.

Of course, we may add parentheses for the sake of clarity.

Comments.

- We denote the set consisting of simply typed λ-terms of type A by $\Lambda_A^{\to}$, and the set consisting of all simply typed λ-terms by $\Lambda^{\to}$.
- Note that the type of a variable in a given environment is unique, because of the definition of environment.
- Instead of $\emptyset \vdash M : A$ we write $\vdash M : A$.
- If there is a term M and an environment Γ such that $\Gamma \vdash M : A$, then the type A is said to be *inhabited.* In that case we also say that M is an *inhabitant* of the type A.
- In a term of the form $\lambda x{:}A.\, M$, the sub-term M is the *scope* of the abstraction over the variable x. Every occurrence of x in M is said to be *bound* (by the abstraction over x). If all variable occurrences in a term are bound, then the term is said to be *closed.*

Examples.

1.

$$\frac{x : A \vdash x : A}{\vdash \lambda x : A.\, x : A \to A}$$

2.

$$\dfrac{\dfrac{x : A, y : B \vdash x : A}{x : A \vdash \lambda y{:}B.\, x : B \to A}}{\vdash \lambda x{:}A.\, \lambda y{:}B.\, x : A \to B \to A}$$

3. Here we use $\Gamma = x : A \to B \to C, y : A \to B, z : A$, $\Gamma_1 = x : A \to B \to C, y : A \to B$, and $\Gamma_2 = x : A \to B \to C$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : A \to B \to C \qquad \Gamma \vdash z : A}{\Gamma \vdash xz : B \to C} \qquad \dfrac{\Gamma \vdash y : A \to B \qquad \Gamma \vdash z : A}{\Gamma \vdash yz : B}}{\Gamma \vdash xz(yz) : C}}{\Gamma_1 \vdash \lambda z{:}A.\, xz(yz) : A \to C}}{\Gamma_2 \vdash \lambda y{:}A \to B.\, \lambda z{:}A.\, xz(yz) : (A \to B) \to A \to C}}{\vdash \lambda x{:}A \to B \to C.\, \lambda y{:}A \to B.\, \lambda z{:}A.\, xz(yz) : (A \to B \to C) \to (A \to B) \to A \to C}$$

2.3 Substitution

Lambda terms are evaluated by means of β-reduction, and the crucial notion in the definition of β-reduction is the one of substitution. This section is concerned with substitution. We consider some problems that may arise and how they can be avoided.

Renaming of bound variables. The aim is to define the substitution of a term N for all free occurrences of x in a term M. This is denoted by $M[x := N]$. The rough idea of such a substitution is as follows: Imagine the tree corresponding to the term M. The leaves are variables. Replace all leaves that are labeled with x by the tree corresponding to the term N. Before giving the formal definition we discuss some subtleties.

A first important thing to notice is that the intention of a substitution $M[x := N]$ is that only *free* occurrences of x in M are replaced by N. Recall that a variable x occurs *free* if it is not in the scope of a λx. It occurs *bound* otherwise. Note that a variable may occur both free and bound in a term. This is for example the case for the variable x in the term $z\,(\lambda x{:}A.\, x)\, x$. The following two 'substitutions' are *not correct*:

- $(\lambda x{:}A.\, x)[x := y] = \lambda x{:}A.\, y$,
- $(z\,(\lambda x{:}A.\, x)\, x)[x := y] = z\,(\lambda x{:}A.\, y)\, y$.

A second important point is that it is not the intention that by calculating a substitution $M[x := N]$ a free variable x in N is captured by a λx in M. For instance the following 'substitutions' are *not correct*:

- $(\lambda y{:}A.\, x)[x := y] = \lambda y{:}A.\, y$,

- $(z\,(\lambda y{:}A.\,x)\,x)[x := y] = z\,(\lambda y{:}A.\,y)\,y$.

Third, notice that of course the variables in bindings like λx have nothing to do with substitution. For instance the following 'substitution' is *nonsense*:

- $(\lambda x{:}A.\,z)[x := y] = \lambda y{:}A.\,z$.

The third issue is only mentioned for curiosity. The usual solution to the first two problems is to rename bound variables in such a way that bound and free variables have different names, and such that free variables are not captured by a substitution. Renaming of bound variables works as follows: in a term M we replace a part of the form $\lambda x{:}A.\,P$ by $\lambda x'{:}A.\,P'$, where P' is obtained from P by replacing all occurrences of x by x'. Here x' is supposed to be *fresh*, that is, not used in building P. For instance, $\lambda x{:}A.\,x$ can be renamed to $\lambda x'{:}A.\,x'$.

Note that bound variables occur also in predicate logic ($\forall x.\,R(x,y)$), and calculus ($\int f(x)dx$). There it is also usual to rename bound variables: we identify for instance $\forall x.\,R(x,y)$ with $\forall x'.\,R(x',y)$, and $\int f(x)dx$ with $\int f(x')dx'$.

If a term M' is obtained from a term M by renaming bound variables, then we say that M and M' are *α-equivalent*. This is explicitly denoted by $M \equiv M'$. However, sometimes we will sloppily write $M = M'$ for two α-equivalent terms.

Some examples:

- $\lambda x{:}A.\,x \equiv \lambda y{:}A.\,y$,
- $\lambda x{:}A.\,z\,x\,x \equiv \lambda y{:}A.\,z\,y\,y$,
- $\lambda x{:}A.\,z\,x\,x \not\equiv \lambda y{:}A.\,z\,y\,x$,
- $x\,(\lambda x{:}A.\,x) \equiv x\,(\lambda y{:}A.\,y)$,
- $x\,(\lambda x{:}A.\,x) \not\equiv y\,(\lambda y{:}A.\,y)$.

In order to avoid the problems mentioned above, we adopt the following:

- Terms that are equal up to a renaming of bound variables are identified.
- Bound variables are renamed whenever necessary in order to prevent unintended capturing of free variables.

The second point is an informal formulation to what is called the *Variable Convention* which is more precisely stated as follows: if terms $M_1, \ldots, M_n$ occur in some context (for instance a proof), then in these terms all bound variables are chosen to be different form the free variables.

Substitution. Assuming the variable convention, the definition of the substitution of N for the free occurrences of $x : A$ in M, notation $M[x := N]$, is defined by induction on the structure of M as follows:

1. (a) $x[x := N] = N$,
   (b) $y[x := N] = y$,

2. $(\lambda y{:}B.\, M_0)[x := N] = \lambda y{:}B.\,(M_0[x := N])$,
3. $(M_1\, M_2)[x := N] = (M_1[x := N])\,(M_2[x := N])$.

Note that in the second clause by the variable convention we have that y is not free in N, and that $x \neq y$.

2.4 Beta-reduction

The β-reduction relation expresses how the application of a function to an argument is evaluated. An example of a β-reduction step is for example: $(\lambda x{:}\mathsf{nat}.\, x)\, 5 \to_\beta 5$. Here the application of the identity function on natural numbers applied to the argument 5 is evaluated. In general, the β-reduction rule is as follows:

$$(\lambda x{:}A.\, M)\, N \to_\beta M[x := N].$$

A β-reduction step, or β-rewrite step, is obtained by applying the β-reduction rule somewhere in a term. So a β-reduction step looks like the following:

$$\ldots (\lambda x{:}A.\, M)\, N \ldots \to_\beta \ldots M[x := N] \ldots$$

The more technical definition is as follows. The β-reduction relation, denoted by $\to_\beta$, is defined as the smallest relation on the set of λ-terms that satisfies

$$(\lambda x{:}A.\, M)\, N \to_\beta M[x := N]$$

and that is moreover closed under the following three rules:

$$\frac{M \to_\beta M'}{\lambda x{:}A.\, M \to_\beta \lambda x{:}A.\, M'}$$

$$\frac{M \to_\beta M'}{M\, P \to_\beta M'\, P}$$

$$\frac{M \to_\beta M'}{P\, M \to_\beta P\, M'}$$

A sub-term of the form $(\lambda x{:}A.\, M)\, N$ is called a *β-redex.* Applying the β-reduction rule to such a sub-term is called *contracting* the redex $(\lambda x{:}A.\, M)\, N$. A *β-reduction* or *β-rewrite sequence* is a finite or infinite sequence

$$M_0 \to_\beta M_1 \to_\beta \ldots$$

obtained by performing zero, one or more β-rewrite steps. A β-reduction can be seen as a computation. A term where no β-reduction step is possible, so a term where nothing remains to be computed, is said to be a *β-normal form* (or shortly normal form). We can think of a β-normal form as the result of a computation.

Examples.

- $(\lambda x{:}A.\,x)\,a \rightarrow_\beta x[x := a] = a$
- $(\lambda x{:}A.\,\lambda y{:}B.\,x)\,a\,b \rightarrow_\beta ((\lambda y{:}B.\,x)[x := a])\,b = (\lambda y{:}B.\,a)\,b \rightarrow_\beta a[y := b] = a$
- $(\lambda f{:}A \rightarrow A.\,f)\,(\lambda x{:}A.\,x)\,a' \rightarrow_\beta (\lambda x{:}A.\,x)\,a' \rightarrow_\beta a'$
- $\lambda x{:}A.\,(\lambda y{:}A.\,\lambda z{:}B.\,y)\,x \rightarrow_\beta \lambda x{:}A.\,\lambda z{:}B.\,x$
- $$\begin{array}{lll} (\lambda f{:}A{\rightarrow}A.\,\lambda x{:}A.\,f\,(f\,x))\,(\lambda y{:}A.\,y)\,5 & \rightarrow_\beta & (\lambda x{:}A.\,(\lambda y{:}A.\,y)\,((\lambda y{:}A.\,y)\,x))\,5 \\ & \rightarrow_\beta & (\lambda y{:}A.\,y)\,((\lambda y{:}A.\,y)\,5) \\ & \rightarrow_\beta & (\lambda y{:}A.\,y)\,5 \\ & \rightarrow_\beta & 5 \end{array}$$

Note that in a β-reduction step, an argument can be erased or multiplied. Further, the argument is an arbitrary λ-term. In particular, it can be a function itself. This higher-order behavior also occurs in programming languages where a procedure can receive another procedure as argument.

Notation. We write $\twoheadrightarrow_\beta$ or $\rightarrow_\beta^*$ for the reflexive-transitive closure of $\rightarrow_\beta$. We sometimes omit the subscript β. If $M \twoheadrightarrow_\beta M'$, then M reduces to M' in zero, one or more steps.

Normal forms. A term is said to be in *normal form* if it does not contain a redex. That is, it does not contain a subterm of the form $(\lambda x{:}A.\,M)\,N$. Examples of normal forms: x, $\lambda x{:}A.\,x$, $z\,(\lambda x{:}A.\,x)\,y$. The term $a\,((\lambda x{:}A.\,x)\,y)$ is not a normal form.

Shape of the normal forms. We define the set NF inductively by the following clauses:

1. if $x : A_1{\rightarrow}\ldots{\rightarrow}A_n{\rightarrow}B$, and $M_1, \ldots, M_n \in \mathsf{NF}$ with $M_1 : A_1, \ldots, M_n : A_n$, then $x\,M_1\,\ldots M_n \in \mathsf{NF}$,
2. if $M \in \mathsf{NF}$, then $\lambda x{:}A.\,M \in \mathsf{NF}$.

We can show the following:

- If M is a normal form, then $M \in \mathsf{NF}$.
- If $M \in \mathsf{NF}$, then M is a normal form.

That is, NF is a characterization of the β-normal forms.

Weak normalization. A term is said to be *weakly normalizing* if it has a normal form.

Untyped λ-calculus is not weakly normalizing: for instance the term $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ does not have a normal form.

The simply typed λ-calculus is weakly normalizing. A proof analyzing the complexity of the types in a term along a reduction is due to Turing and Prawitz.

Termination or strong normalization. A term M is said to be *terminating* or *strongly normalizing* if all rewrite sequences starting in M are finite. If a term is strongly normalizing then it is also weakly normalizing. If a term is weakly normalizing it is not necessarily strongly normalizing: the untyped λ-term $(\lambda x.\lambda y.\, y)\Omega$ is weakly normalizing but not strongly normalizing.

The simply typed λ-calculus is terminating. (And hence also weakly normalizing.) This means that there are no infinite reduction sequences. So every computation eventually ends in a normal form.

Confluence. A term M is said to be *confluent* if whenever we have $M \rightarrow^*_\beta N$ and $M \rightarrow^*_\beta N'$, there exists a term P such that $N \rightarrow^*_\beta P$ and $N' \rightarrow^*_\beta P$. A rewriting system is said to be confluent if all its terms are confluent. A consequence of confluence is that a term has at most one normal form. Both untyped and simply typed λ-calculus are confluent. Since simply typed λ-calculus is also terminating, we have that every term has a unique normal form.

Subject reduction. Subject reduction is the property that types are preserved under computation. More formally: if $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta M'$, then $\Gamma \vdash M' : A$. Simply typed λ-calculus has the property subject reduction.

2.5 Curry-Howard-De Bruijn isomorphism

There exist amazing correspondences between various typed λ-calculi on the one hand, and various logics on the other hand. These correspondences are known as the *Curry-Howard-De Bruijn isomorphism.* This isomorphism forms the basis of proof checkers based on type theory, like Coq. Here we focus on the correspondence between simply typed λ-calculus and first-order minimal propositional logic. Recall that in first-order *minimal* propositional logic there is only one logical connective: $\rightarrow$ for implication. So we do not consider conjunction, disjunction, and negation.

The static part of the isomorphism states that formulas correspond to types, and proofs to derivations showing that a term is of a certain type. This is summarized by the slogan *formulas as types and proofs as terms.*

Formulas as types. The first part of the Curry-Howard-De Bruijn isomorphism concerns the correspondence between formulas in minimal logic on the one hand, and types of simply typed λ-calculus on the other hand. Here the correspondence is literally: we can choose to see an expression A either as a formula in minimal logic or as a simple type. This *formulas as types* part can be summarized as follow:

propositional variable	$\sim$	type variable
the connective $\to$	$\sim$	the type constructor $\to$
formula	$\sim$	type

Proofs as terms. The second part of the isomorphism concerns the correspondence between proofs in natural deduction of first-order propositional minimal logic on the one hand, and derivations showing that a term has a certain type on the other hand. Since there is moreover an exact correspondence between a simply typed term M of type A and the derivation showing that $M : A$, we can identify a *term* M of type A with the *derivation* showing that $M : A$. This part is summarized by the slogan *proofs as terms* and consists of the following correspondences:

assumption	$\sim$	term variable
implication introduction	$\sim$	abstraction
implication elimination	$\sim$	application
proof	$\sim$	term

The examples of type derivations given in the previous section correspond to the proofs given in Chapter 1. Further, the two different proofs of $A{\to}A{\to}A$ given there correspond to type derivations of $\lambda x{:}A.\, \lambda y{:}A.\, x$ and $\lambda x{:}A.\, \lambda y{:}A.\, y$.

Provability and inhabitation. The type inhabitation problem is the question whether, given a type A, there is a closed inhabitant of A. Sometimes this is abbreviated as $\vdash ? : A$. This problem corresponds to the provability question: this is the question whether, given a formula A, there is a proof showing that A is a tautology.

Note that the provability question becomes trivial if we remove the word 'tautology': an assumption A is a proof of A (but not a proof that A is a *tautology*). Similarly, the inhabitation problem becomes trivial if we remove the word 'closed': a variable of type A is an inhabitant of the type A (but not a *closed* inhabitant).

Proof checking and type checking. The type checking problem is the question whether, given an environment Γ, a term P and a type A, the term M is of type A. Sometimes this is abbreviated as $\Gamma \vdash P : A$?. This problem corresponds to the proof checking problem: given a proof P and a formula A, is P a proof of A?

Note that we are a bit sloppy, because whereas A-seen-as-a-formula is literally the same as A-seen-as-a-type, this is not the case for proofs on the one hand and type derivations on the other hand. It is however possible to see a proof of the formula A as a derivation showing that a certain term has type A, even if the two expressions are not literally the same.

Coq. The Curry-Howard-De Bruijn isomorphism forms the essence of proof checkers like Coq. The user who wants to show that a formula A is a tautology construct, using the proof-development system, a closed inhabitant of A now viewed as a type. A type checking algorithm verifies whether the found term is indeed of type A. If this is the case, then we indeed have a proof of A.

Example. We recall the example that was already partly given in Chapter 1. On the left is a we have the different stages of constructing a proof in minimal logic, and on the right are the corresponding steps in Coq. To save space we do not indicate which rule is used in the minimal logic part.

Lemma
$(A \to B) \to (C \to A) \to C \to B.$

```
Lemma DrieA :
(A->B)->(C->A)->C->B.
```

$$\frac{}{(A \to B) \to (C \to A) \to C \to B}$$

Goal:
```
(A->B)->(C->A)->C->B.
```

```
intro x.
```
Assumption: `x:A->B`

$$\frac{(C \to A) \to C \to B}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `(C->A)->C->B`

```
intro y.
```
Assumptions:
```
   x:A->B
   y:C->A
```

$$\frac{\dfrac{C \to B}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `C->B`
```
intro z.
```
Assumptions:
```
   x:A->B
   y:C->A
   z:C
```

$$\frac{\dfrac{\dfrac{B}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `B`

```
apply x.
```
Assumptions do not change

$$\frac{\dfrac{\dfrac{\dfrac{A \to B \quad A}{B}}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `A`

```
apply y.
```
Assumptions do not change

$$\frac{\dfrac{\dfrac{\dfrac{A \to B \quad \dfrac{C \to A \quad C}{A}}{B}}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

Goal: `C`

```
exact z.
```

done!

Subtree proved!

The corresponding type derivation is as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash x : A \to B \qquad \cfrac{\Gamma \vdash y : C \to A \quad \Gamma \vdash z : C}{\Gamma \vdash (y\,z) : A}}{\Gamma \vdash (x\,(y\,z)) : B}}{\Gamma_1 \vdash \lambda z{:}C.\,(x\,(y\,z)) : C \to B}}{\Gamma_2 \vdash \lambda y{:}C \to A.\,\lambda z{:}C.\,(x\,(y\,z)) : (C \to A) \to C \to B}}{\vdash \lambda x{:}A \to B.\,\lambda y{:}C \to A.\,\lambda z{:}C.\,(x\,(y\,z)) : (A \to B) \to (C \to A) \to C \to B}$$

Here we use $\Gamma = x : A \to B, y : C \to A, z : C$, $\Gamma_1 = x : A \to B, y : C \to A$, and $\Gamma_2 = x : A \to B$.

Now we turn to the dynamic part of the Curry-Howard-De Bruijn isomorphism: β-reduction in the λ-calculus corresponds to proof normalization or detour elimination in minimal logic.

The key observation here is that a β-redex

$$(\lambda x{:}A.\,M)\,N$$

in some λ-term corresponds to a detour (an introduction rule immediately followed by an elimination rule) in some proof:

$$\cfrac{\cfrac{\cfrac{\vdots}{B}}{A \to B}\, I[x]\!\to \qquad \cfrac{\vdots}{A}}{B}\, E\!\to$$

This correspondence is more clearly visible if we consider the relevant part of a typing derivation for a redex:

$$\cfrac{\cfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\,M : A \to B} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x{:}A.\,M)\,N : B}$$

We consider β-reduction and proof normalization, by listing first the important notions for β-reduction and then the corresponding ones for proof normalization.

Beta Reduction.

- The β-reduction rule is

$$(\lambda x{:}A.\,M)\,N \to_\beta M[x := N].$$

- The *substitution* of a term $N{:}A$ for a variable $x{:}A$ in a term M is defined by induction on the structure of M.

- A *β-redex* is a term of the form $(\lambda x{:}A.\,M)\,N$.

  A sub-term of P of the form $(\lambda x{:}A.\,M)\,N$ is called a β-redex in P.

- A *β-reduction step* $P \to_\beta P'$ is obtained by replacing in P a β-redex $(\lambda x{:}A.\, M)\, N$ by $M[x := N]$. We then say that this β-redex is *contracted.*

  The formal definition of the β-reduction relation is given by an axiom (for β-reduction) and three rules that express how β-reduction takes place in a context.

- A *β-normal form* is a term that cannot perform a β-reduction step, or equivalently, that does not contain a β-redex.

Proof Normalization.

- The proof normalization rule is as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{B}}{A \to B}\, I[x]\to \qquad \dfrac{\vdots}{A}}{B}\, E\to$$

  is replaced by

$$\dfrac{\vdots}{B}$$

  where every occurrence of the assumption A^x is replaced by the proof

$$\dfrac{\vdots}{A}$$

- The *substitution operation* 'replace all assumptions A^x by a proof of A' remains informal.

- A *detour* is a proof of the form

$$\dfrac{\dfrac{\dfrac{\vdots}{B}}{A \to B}\, I[x]\to \qquad \dfrac{\vdots}{A}}{B}\, E\to$$

  A proof contains a detour if a sub-proof (also this notion remains informal) is a detour as above, that is, an introduction rule immediately followed by an elimination rule.

- A *proof normalization step* is obtained by replacing a detour

$$\dfrac{\dfrac{\dfrac{\vdots}{B}}{A \to B}\; I[x] \to \qquad \dfrac{\vdots}{A}}{B}\; E \to$$

in a proof by

$$\dfrac{\vdots}{B}$$

that is, by applying the proof normalization rule.

- A *normal proof* is a proof that cannot perform a proof normalization step, or equivalently, that does not contain a detour.

Summary. So in summary, one can say that the dynamic part of the Curry-Howard-De Bruijn isomorphism consists of the following ingredients:

β-redex	$\sim$	detour
β-reduction step	$\sim$	proof normalization step
β-normal form	$\sim$	normal proof

2.6 Coq

- Notation of λ-terms in Coq:

  An abstraction $\lambda x{:}A.\, M$ is written in Coq as `fun x:A => M`.

  An application $F\,N$ is written in Coq as `F N`.

  The λ-term $\lambda x{:}A.\, \lambda x'{:}A.\, \lambda y{:}B.\, M$ can be written in Coq in the following ways:

```
fun (x:A) (x':A) (y:B) => M
  fun (x x':A) (y:B) => M
```

- Reduction in Coq.

  Coq uses β-reduction, but also other reduction relations.

  We can use the combination of tactics `Eval cbv beta in` with argument a term to β-reduce the term using the call-by-value evaluation strategy.

- `Print` *id* `.`

  This command displays on the screen the information concerning *id*.

- `Check` *term* `.`

  This command displays the type of *term*.

- `Load` *id* `.`

  If you start Coq in unix without using Proof General, by typing `coqtop`, then you can easily check whether a file `id.v` is accepted by Coq: use the command `Load id.` (without the `v`).

- `Proof .`

  It is nice to start a proof with this command.

- `Qed .`

  This closes a finished proof.

- `Section` *id* `.`

  This opens a section. It is closed by

  `End` *id* `.`

- Local declarations.

  A declaration gives the type of an identifier but not its value. The scope of a local declaration is the section where it occurs. Outside the section the declaration is unknown. Local declarations are given using `Variable`, for example `Variable A B C : Prop`.

  Instead of `Variable` we can also use `Hypothesis`. The idea of `Hypothesis p : A` is to declare `p` as representing an arbitrary (unknown) proof of `A`.

- Global declarations. The syntax for a global declaration is `Parameter`, for instance :
  `Parameter A : Prop`. By using a global declaration, the current environment is extended with this declaration. It is not local to a section.

  Instead of `Parameter` we can also use `Axiom`.

- Local definitions.

  A definition gives the value of an identifier, and hence its type. The syntax of a local definition is `Let v := fun x:A => x` or, giving the type explicitly, `Let v : A -> A := fun x:A => x`. This definition also can be written as `Let v (x:A) : A := x`. The scope of a local definition is the section where it occurs.

- Global definitions.

  The syntax of a global definition is `Definition`. For example:
  `Definition v := fun x:A => x`, or, giving the type explicitly,
  `Definition v : A -> A := fun x:A => x`. It also can be written as
  `Definition v (x:A) : A := x`.

Chapter 3

Inductive types

This chapter is concerned with inductive definitions in Coq. A lot of data types can be defined in λ-calculus. This yields however representations that are not always very efficient. More efficient representations can be obtained using inductive types.

3.1 Expressivity

Untyped lambda calculus. In the course *Introduction to theoretical computer science*, we have seen representations of the data types natural numbers, booleans, and pairs with some elementary operations. Moreover, in untyped λ-calculus a fixed point operator can be defined. A *fixpoint* of a function $f : X \to X$ is an $x \in X$ such that $f(x) = x$. For untyped λ-calculus there is the following important result:

Theorem 3.1.1. *For every λ-term F there is a term M such that*

$$FM =_\beta M.$$

The proof can be given by using the fixed point combinator

$$\mathbf{Y} = \lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx))$$

The fixed point operator can be used to defined recursive functions, like for instance

$$\begin{array}{rcl} 0! & = & 1, \\ (n+1)! & = & (n+1)\cdot n!. \end{array}$$

in λ-calculus. Here a function is defined in terms of itself. The important thing is that on the right-hand side, it is applied to an argument that is essentially simpler than the argument on the left-hand side.

The class of recursive functions is the smallest class of functions $\mathbb{N} \to \mathbb{N}$ that contains the *initial functions*

1. *projections*:
   $U_i^m(n_1, \ldots, n_m) = n_i$ for all $i \in \{1, \ldots, m\}$,
2. *successor*:
   $Suc(n) = n + 1$,
3. *zero*:
   $Zero(n) = 0$,

and that is moreover closed under the following:

1. *composition*:
   if $g : \mathbb{N}^k \to \mathbb{N}$ and $h_1, \ldots, h_k : \mathbb{N}^m \to \mathbb{N}$ are recursive, then $f : \mathbb{N}^m \to \mathbb{N}$ defined by
   $$f(n_1, \ldots, n_m) = g(h_1(n_1, \ldots, n_m), \ldots, h_k(n_1, \ldots, n_m))$$
   is also recursive,
2. *primitive recursion*:
   if $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ are recursive, then $f : \mathbb{N}^{k+1} \to \mathbb{N}$ defined by
   $$\begin{array}{rcl} f(0, n_1, \ldots, n_k) & = & g(n_1, \ldots, n_k) \\ f(n+1, n_1, \ldots, n_k) & = & h(f(n, n_1, \ldots, n_k), n, n_1, \ldots, n_k) \end{array}$$
   is also recursive,
3. *minimalization*:
   if $g : \mathbb{N}^{k+1} \to \mathbb{N}$ is recursive, and for all $n_1, \ldots, n_k$ there exists an n such that $g(n, n_1, \ldots, n_k) = 0$, then $f : \mathbb{N}^k \to \mathbb{N}$ defined by
   $$f(n_1, \ldots, n_k) = \mu n.g(n, n_1, \ldots, n_k)$$
   is also recursive.

In the last clause, $\mu n.g(n, n_1, \ldots, n_k)$ denotes the smallest natural number m such that $g(m, n_1, \ldots, n_k) = 0$.

Kleene has shown the following:

Theorem 3.1.2. *All recursive functions are λ-definable.*

The proof is given using the following lemma:

Lemma 3.1.3.

1. *The initial functions are λ-definable.*
2. *The λ-definable functions are closed under composition, primitive recursion, and minimalization.*

Simply typed lambda calculus. The fixpoint combinator $\mathbf{Y}$ is not typable in the simply typed λ-calculus. As a matter of fact, the simply typed λ-calculus has essentially less expressive power than the untyped λ-calculus. A result by Schwichtenberg, given below, states that the functions that are definable in simply typed λ-calculus are exactly the extended polynomials, which is a smaller class than the one of all recursive functions.

The class of *extended polynomials* is the smallest class of functions $\mathbb{N} \to \mathbb{N}$ that contains

1. *projections*:
   $U_i^m(n_1, \ldots, n_m) = n_i$ for all $i \in \{1, \ldots, m\}$,
2. *constant function*:
   $k(n) = k$,
3. *signum function*:
   $sig(0) = 0$,
   $sig(n+1) = 1$,

and that is moreover closed under the following:

1. *addition*:
   if $f : \mathbb{N}^i \to \mathbb{N}$ and $g : \mathbb{N}^j \to \mathbb{N}$ are extended polynomials, then $(f+g) : \mathbb{N}^{i+j} \to \mathbb{N}$ defined by

   $$(f+g)(n_1, \ldots, n_i, m_1, \ldots, m_j) = f(n_1, \ldots, n_i) + g(m_1, \ldots, m_j)$$

   is also an extended polynomial,
2. *multiplication*:
   if $f : \mathbb{N}^i \to \mathbb{N}$ and $g : \mathbb{N}^j \to \mathbb{N}$ are extended polynomials, then $(f \cdot g) : \mathbb{N}^{i+j} \to \mathbb{N}$ defined by

   $$(f \cdot g)(n_1, \ldots, n_i, m_1, \ldots, m_j) = f(n_1, \ldots, n_i) \cdot g(m_1, \ldots, m_j)$$

   is also an extended polynomial.

Theorem 3.1.4. *The functions definable in simply typed λ-calculus are exactly the extended polynomials.*

An example of a function that is definable in λ-calculus but not in simply typed λ-calculus is *Ackermann's Function*, denoted by A, that is defined as follows:

$$\begin{array}{rcl} \mathsf{A}(0,n) & = & n+1, \\ \mathsf{A}(m+1,0) & = & \mathsf{A}(m,1), \\ \mathsf{A}(m+1,n+1) & = & \mathsf{A}(m,\mathsf{A}(m+1,n)). \end{array}$$

A representation of the natural numbers. Let A be some type. The natural numbers can be represented as countably infinitely many different closed inhabitants in normal form of type $(A \to A) \to (A \to A)$: represent $k \in \mathbb{N}$ as $\lambda s{:}A \to A.\, \lambda A{:}n.\, s^k(n)$ where we use the notation ^{k} defined as follows:

$$\begin{array}{lcl} F^0(P) & = & P, \\ F^{n+1}(P) & = & F\,(F^n(P)). \end{array}$$

A few natural numbers in their encoding as Church numeral:

- The representation of 0 is $\lambda s{:}A \to A.\, \lambda n{:}A.\, n$. As a proof:

$$\cfrac{\cfrac{[A^n]}{A \to A}\; I[n]\to}{(A \to A) \to (A \to A)}\; I[s]\to$$

- The representation of 1 is $\lambda x{:}A \to A.\, \lambda n{:}A.\, (s\, n)$. As a proof:

$$\cfrac{\cfrac{\cfrac{[A \to A^s] \qquad [A^n]}{A}\; E\to}{A \to A}\; I[n]\to}{(A \to A) \to (A \to A)}\; I[s]\to$$

- The representation of 2 is $\lambda x{:}A \to A.\, \lambda n{:}A.\, (s\,(s\, n))$. As a proof:

$$\cfrac{\cfrac{\cfrac{[A \to A^s] \qquad \cfrac{[A \to A^s] \qquad [A^n]}{A}\; E\to}{A}\; E\to}{A \to A}\; I[n]\to}{(A \to A) \to (A \to A)}\; I[s]\to$$

Below we will see an easier encoding of natural numbers as an inductive type.

3.2 Universes of Coq

Every expression in Coq is typed. The type of a type is also called a sort. The three sorts in Coq are the following:

- `Prop` is the universe of logical propositions. If `A:Prop` then `A` denotes the class of terms representing proofs of the proposition `A`.
- `Set` is the universe of program types or specifications. Besides programs also well-knows sets as the booleans and the natural numbers are in `Set`.
- `Type` is the universe of `Prop` and `Set`. In fact there is in infinite hierarchy of sorts `Type`(i) with `Type`(i) : `Type`$(i+1)$. From the user point of view we have `Prop:Type` and `Set:Type`, and further `Type:Type`. This can be seen using `Check`.

3.3 Inductive types

Inductive types: introduction. In order to explain the notion of inductive type we consider as an example the inductive type representing the natural numbers. Its definition is as follows:

```
  Inductive nat : Set :=  O : nat | S : nat->nat.
```

This definition is already present in Coq, check this with `Check nat.`. Let us comment on the definition. A new element of `Set` is declared with name `nat`. The constructors of `nat` are `O` and `S`. Because this is an inductive definition, the set `nat` is the smallest set that contains `O` and is closed under application of `S`, so `O` and `S` determine all elements of `nat`. The constructors of an inductive definition correspond to introduction rules. In this case, the introduction rules for `nat` can be seen as follows:

$$\frac{}{\mathtt{O} : \mathtt{nat}} \qquad \frac{\mathtt{p} : \mathtt{nat}}{(\mathtt{S}\,\mathtt{p}) : \mathtt{nat}}$$

Recursive definitions. In Coq recursive functions can be defined using the `Fixpoint` construction. For non-recursive functions the `Definition` construction is used. In case a recursive function is defined where the input is of an inductive type, in the definition all constructors of the inductive type are considered in turn. This is done using the `match` construction. As an example we consider the definition of the function `plus: nat -> nat -> nat`:

```
Fixpoint plus (n m : nat) {struct n} : nat :=
match n with
  | O => m
  | S p => S (plus p m)
end.
```

Here the recursion is in the first argument of `plus`. In the recursive call, the argument `p` is strictly smaller than the original argument `n` which is `(S p)`.

The first line of the definition reads as follows: a recursive function `plus` is defined. It takes as input one argument of type `nat` which is bound to the parameter `n`. Then the output is a function of type `nat -> nat`. The output is the function that is on the second till the last line: it takes as input an argument of type `nat` which is bound to the argument `m`. Then the behaviour is described distinguishing cases: either `n` is `O` or `n` is of the form `(S p)`.

Inductive types: elimination. Elimination for `nat` corresponds to reasoning by cases: for an element `n` of `nat` there are two possibilities: either `n = O` or `n  = (S p)` for some `p` in `nat`. With the definition of an inductive type, automatically three terms that implement reasoning by cases are defined. In the case of `nat`, we have

```
nat_ind
nat_rec
nat_rect
```

Here we consider only the first one. Using the command `Check`, we find:

```
nat_ind : forall P : nat -> Prop,
          P 0 ->
          (forall n : nat, P n -> P (S n)) ->
          forall n : nat, P n
```

This type can be read as follows. For every property `P` of the natural numbers, if we have that

- the property `P` holds for `O`, *and*
- the property `P` holds for `n` implies that the property `P` holds for the successor of `n`,

then the property `P` holds for every natural number.

The tactic `elim` tries to apply `nat_ind`. That is, if the current goal `G` concerns an element `n` of `nat`, then `elim` tries to find an abstraction `P` of the current goal such that `Pn` = `G`. Then it applies `nat_ind P`. See also the use of `elim` in the following example.

Example. This example concerns the proof of an elementary property of the function `plus`. On the left is the 'normal' proof and on the right are the corresponding steps in Coq.

Lemma. $\forall n \in \mathbb{N} : n = \mathsf{plus}\ n\ 0$	`Lemma plus_n_O :`
	`forall n : nat, n = plus n O.`
Let $n \in \mathbb{N}$.	`intro n.`
To Prove: $n = \mathsf{plus}\ n\ 0$.	*Goal:* `n = plus n O.`
induction on n.	`elim n.`
Basisstep	
To Prove: $0 = \mathsf{plus}\ 0\ 0$.	*Goal 1:* `O = plus O O.`
Definition of plus yields	`simpl.`
$\mathsf{plus}\ 0\ 0 = 0$.	
To Prove: $0 = 0$.	*Goal 1:* `O = O.`
	`reflexivity.`
Inductionstep	
To Prove:	*Goal:*
$\forall m \in \mathbb{N}$:	`forall m :  nat,`
$m = \mathsf{plus}\ m\ 0 \rightarrow$	`m = plus m O ->`
$Sm = \mathsf{plus}\ (Sm)\ 0$.	`S m = plus (S m) O).`
Let $m \in \mathbb{N}$.	`intro m.`
Suppose that $m = \mathsf{plus}\ m\ 0$.	`intro IH.`
To Prove: $Sm = \mathsf{plus}\ (Sm)\ 0$	*Goal:* `S m = plus (S m) O.`
Definition of plus yields	`simpl.`
$\mathsf{plus}\ (Sm)\ 0 = S(\mathsf{plus}\ m\ 0)$.	
To Prove: $Sm = S(\mathsf{plus}\ m\ 0)$.	*Goal:* `S m = S(plus m O).`
By the induction hypothesis	`rewrite <- IH.`
$\mathsf{plus}\ m\ 0 = m$.	
To Prove: $Sm = Sm$.	*Goal:* `S m = S m.`
	`reflexivity.`

Prop and bool. `Prop` is the universe of the propositions in Coq. We have `True : Prop` and `False : Prop`. The proposition `True` denotes the class of terms representing proofs of `True`. It has one inhabitant. The proposition `False` denotes the class of terms representing proofs of `False`. It has no inhabitants since we do not have a proof of `False`. Both `True` and `False` are defined inductively:

```
Inductive True : Prop :=  I : True .
Inductive False : Prop :=  .
```

Further, `bool` is the inductive type of the booleans:

```
Inductive bool : Set :=  true : bool | false : bool .
```

We have `bool:Set`. Operations on elements of Prop, like for instance `not`, cannot be applied to elements of `bool`. If in doubt, use `Check`.

3.4 Coq

- `Inductive`

  This is not the complete syntax! The construct `Inductive` is used to give an inductive definition of a set, as in the example of natural numbers. Use `Print nat.` and `Print bool.` to see examples of inductive types.

- `match ... with ... end.`

  This construct is used to distinguish cases according to the definition of an inductive type. For instance, if `m` is of type `nat`, we can make a case distinction as follows:

  ```
  match m with
    | O   => t1
    | S p => t2
  end.
  ```

  Here the case distinction is: `m` is either `O` or of the form `(S p)`. The `|` is used to separate the cases, and `t1` and `t2` are terms.

- `Fixpoint`

  This construct is used to give a recursive definition. (For a non-recursive definition, use `Definition`.) To illustrate its use we consider as an example an incomplete definition of `plus:nat->nat->nat`.

  ```
  Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
    | O => m
    | S p => S (plus p m)
  end.
  ```

  Here a recursive definition of the function `plus` taking as input two natural numbers is given; the recursion is in the *first* argument (indicated by the parameter n). An alternative definition is:

  ```
  Fixpoint plus (n m : nat) {struct m} : nat :=
    match m with
    | O => n
    | S p => S (plus n p)
    end.
  ```

  Here the recursion is on the *second* argument (indicated by the parameter m).

- `elim`

  So far we have seen the use of `elim x` if `x` is the label of a hypothesis of the form $A \wedge B$ or $A \vee B$.

  In fact then `elim x` applies either `and_ind` or `or_ind`.

  In case that `x` is of some inductive type, say `t`, then `elim x` tries to apply `t_ind`. The result is that induction on `x` is started.

- `induction` *id* `.`

  This tactic does `intros until` *id* `; elim` *id*. In the example, instead of the two first steps

  ```
  intro n .
  elim .
  ```

  we can use

  ```
  induction n .
  ```

- `Eval compute in`

  This is an abbreviation for `Eval cbv iota beta zeta delta in`. We will see more about this later on; for now: it can be used to compute the normal form of a term.

- `simpl .`

  This tactic can be applied to any goal. The goal is simplified by performing $\beta\iota$-reduction. This means that β-redexes are contracted, and also redexes for definitions like for instance the function `plus` are reduced.

- `rewrite` *id* `.` Let *id* be of type `forall (x1:A1)...(xn:An), t1 = t2`, so some universal quantifications followed by an equality.

  An application of `rewrite` *id* or equivalently `rewrite ->` *id* yields that in the goal all occurrences of `t1` are replaced by `t2`.

  An application of `rewrite <-` *id* yields that in the goal all occurrences of `t2` are replaced by `t1`.

- `reflexivity .`

  This tactic can be applied to a goal of the form `t=u`. It checks whether `t` and `u` are convertible (using $\beta\iota$-reduction) and if this is the case solves the goal.

3.5 Inversion

The tactic `inversion` is, very roughly speaking, used to put a hypothesis in a form that is more useful. It can be used with argument *label* if *label* is the label of a hypothesis that is built from an inductive predicate. We consider two examples of the use of `inversion`.

In the first example we use the predicate `le` and a declared variable `P`:

```
Inductive le (n:nat) : nat -> Prop :=
  le_n : le n n
| le_S : forall m : nat, le n m -> le n (S m).
Variable P : nat -> nat -> Prop .
```

Suppose we want to prove an implication of the following form:

```
Lemma een : forall n m : nat, le (S n) m -> P n m .
```

We start with the necessary introductions. This yields:

```
1 subgoal

  n : nat
  m : nat
  H : le (S n) m
  ============================
   P n m
```

Now `inversion H .` generates two new goals, one for each of the ways in which `H` could have been derived from the definition of `le`:

```
2 subgoals

  n : nat
  m : nat
  H : le (S n) m
  H0 : S n = m
  ============================
   P n (S n)

subgoal 2 is:

  n : nat
  m : nat
  H : le (S n) m
  m0 : nat
  H0 : le (S n) m0
  H1 : S m0 = m
  ============================
   P n (S m0)
```

In the second example, we again use `P` and in addition the predicate `leq`:

```
Inductive leq : nat -> nat -> Prop :=
  leq_0 : forall m : nat, leq 0 m |
  leq_s : forall n m : nat, leq n m -> leq (S n) (S m) .
```

Suppose we wish to prove the following implication:

```
Lemma twee : forall n m : nat, leq (S n) m -> P n m .
```

We start with the necessary introductions. This yields:

```
1 subgoal

  n : nat
  m : nat
  H : leq (S n) m
  ============================
   P n m
```

Now `inversion H .` yields one new goal, since there was only one possible way in which `H` could have been concluded from the definition of `leq`:

```
1 subgoal

  n : nat
  m : nat
  H : leq (S n) m
  n0 : nat
  m0 : nat
  H1 : leq n m0
  H0 : n0 = n
  H2 : S m0 = m
  ============================
   P n (S m0)
```

Chapter 4

1st-order predicate logic

This chapter is concerned with first-order predicate logic. This is an extension of first-order propositional logic obtained by adding universal quantification ($\forall$) and existential quantification ($\exists$) over terms. There is no quantification over formulas or predicates. We consider a proof system for natural deduction, and first-order predicate logic in Coq. In order to study the theory of Coq later on, we distinguish minimal, intuitionistic, and classical first-order predicate logic. Partly we recapitulate what was already introduced in the previous chapter.

4.1 Terms and formulas

Symbols. To start with, we assume:

- a countably infinite set of *variables*, written as $x, y, z, \ldots$.

Further, we have a *signature* consisting of:

- a set of *function symbols*, written as $f, g, \ldots$,
- a set of *predicate symbols*, written as $R, S, \ldots$.

Every function symbol and every predicate symbol has a fixed *arity*, which is a natural number that prescribes the number of arguments it is supposed to have. Finally, we make use of the following logical connectives:

- $\rightarrow$ for implication,
- $\wedge$ for conjunction,
- $\vee$ for disjunction,
- $\perp$ for falsum,
- $\top$ for truth,

- $\forall$ for universal quantification,
- $\exists$ for existential quantification.

Only the last two are new compared to first-order propositional logic.

Algebraic terms. Let Σ be a signature consisting of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$. *Algebraic terms* over Σ are expressions built from variables and function symbols where the arity of the function symbols is respected. The inductive definition of the set of algebraic terms is as follows:

1. a variable x is an algebraic term over Σ,
2. if f is a function symbol in $\mathcal{F}$ of arity n, and $M_1, \ldots, M_n$ are algebraic terms over Σ, then $f(M_1, \ldots, M_n)$ is an algebraic term over Σ.

The terminology *algebraic* terms is used to distinguish between these terms and the λ-terms that are considered later on. The shorter terminology term is used instead if no confusion is expected. Note that for algebraic terms the functional notation (with parentheses and commas) is used.

Formulas. Let Σ be a signature consisting of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$. The set of *first-order formulas* over Σ is inductively defined as follows:

1. if R is a predicate symbol in $\mathcal{P}$ of arity n, and $M_1, \ldots, M_n$ are algebraic terms over Σ, then $R(M_1, \ldots, M_n)$ is a formula,
2. the constant $\bot$ is a formula,
3. the constant $\top$ is a formula,
4. if A and B are formulas, then $(A \to B)$ is a formula,
5. if A and B are formulas, then $(A \wedge B)$ is a formula,
6. if A and B are formulas, then $(A \vee B)$ is a formula,
7. if A is a formula and x is a variable, then $(\forall x.\, A)$ is a formula,
8. if A is a formula and x is a variable, then $(\exists x.\, A)$ is a formula.

A formula of the form $R(M_1, \ldots, M_n)$ is called an *atomic* formula. The atomic formulas come in the place of the propositional variables in first-order propositional logic. Further, only the formulas built using universal and existential quantification are new here.

Parentheses. For the connectives $\to$, $\wedge$, and $\vee$, the same conventions concerning parentheses are assumed as for first-order propositional logic. Also, outermost parentheses are omitted. For the universal quantification, there is no general consensus about how to minimize the numbers of parentheses. Here we adopt the convention used by Coq: the quantifier scope extends to the right as much as possible. So we write for instance $\forall x.\, A{\to}B$ instead of $\forall x.\, (A{\to}B)$. For existential quantification we do the same.

Free and bound variables. The universal quantifier $\forall$ and the existential quantifier $\exists$ *bind* variables. An occurrence of a variable is *bound* if it is in the scope of a $\forall$ or of a $\exists$. For instance, in the formula $\forall x.\, P(\underline{x})$ the underlined occurrence of x is bound. An occurrence of a variable is *free* if it is not bound. For instance, in the formula $\forall y.\, P(\underline{x})$ the underlined occurrence of x is free. In the following, we identify formulas that are equal up to a renaming of bound variables (or α-conversion). For instance, if R is a unary predicate symbol, then the formulas $\forall x.R(x)$ and $\forall y.R(y)$ are identified.

Substitution. We need the notion of substitution of a term N for a variable x in a term M, and of substitution of a term N for a variable x in a formula A. Both forms of substitution are denoted by $[x := N]$. We assume that bound variables are renamed in order to avoid unintended capturing of free variables.

The inductive definition of the substitution of a term N for a variable x in a *term* M, notation $M[x := N]$, is as follows:

1. $x[x := N] = N$,
2. $y[x := N] = y$,
3. $f(M_1, \ldots, M_n)[x := N] = f(M_1[x := N], \ldots, M_n[x := N])$.

The inductive definition of the substitution of a term N for a variable x in a *formula* A, notation $A[x := N]$, is as follows:

1. $R(M_1, \ldots, M_n)[x := N] = R(M_1[x := N], \ldots, M_n[x := N])$,
2. $\bot[x := N] = \bot$,
3. $\top[x := N] = \top$,
4. $(A_1 \to A_2)[x := N] = (A_1[x := N]) \to (A_2[x := N])$,
5. $(A_1 \wedge A_2)[x := N] = (A_1[x := N]) \wedge (A_2[x := N])$,
6. $(A_1 \vee A_2)[x := N] = (A_1[x := N]) \vee (A_2[x := N])$,
7. $(\forall y.A_1)[x := N] = \forall y.(A_1[x := N])$,
8. $(\exists y.A_1)[x := N] = \exists y.(A_1[x := N])$.

Note that in the last two clauses we assume the name y to be chosen such that the term N does not contain free occurrences of y.

4.2 Natural deduction

4.2.1 Intuitionistic logic

The rules. The natural deduction proof system for first-order predicate logic is an extension of the one for first-order propositional logic. There are two new connectives: ∀ and ∃. So there are four new rules: the introduction and elimination rule for universal quantification, and the introduction and elimination rule for existential quantification. For completeness we give here all the rules.

1. The *assumption rule.*

   A labeled formula A^x is a proof.

$$A^x$$

   Such a part of a proof is called an *assumption.*

2. The *implication introduction rule*:

$$\dfrac{\begin{matrix}\vdots\\ B\end{matrix}}{A \to B}\; I[x]{\to}$$

3. The *implication elimination rule*:

$$\dfrac{\begin{matrix}\vdots\\ A \to B\end{matrix} \qquad \begin{matrix}\vdots\\ A\end{matrix}}{B}\; E{\to}$$

4. The *conjunction introduction rule*:

$$\dfrac{\begin{matrix}\vdots\\ A\end{matrix} \qquad \begin{matrix}\vdots\\ B\end{matrix}}{A \wedge B}\; I{\wedge}$$

5. The *conjunction elimination rules*:

$$\cfrac{\begin{matrix}\vdots\\ A \wedge B\end{matrix}}{A}\,El\wedge \qquad \cfrac{\begin{matrix}\vdots\\ A \wedge B\end{matrix}}{B}\,Er\wedge$$

6. The *disjunction introduction rules*:

$$\cfrac{\begin{matrix}\vdots\\ A\end{matrix}}{A \vee B}\,Il\vee \qquad \cfrac{\begin{matrix}\vdots\\ B\end{matrix}}{A \vee B}\,Ir\vee$$

7. The *disjunction elimination rule*:

$$\cfrac{A \vee B \qquad \begin{matrix}\vdots\\ A \to C\end{matrix} \qquad \begin{matrix}\vdots\\ B \to C\end{matrix}}{C}$$

8. The *falsum rule*:

$$\cfrac{\begin{matrix}\vdots\\ \bot\end{matrix}}{A}$$

9. The *universal quantification introduction rule*:

$$\frac{A}{\forall x.A}\ I\forall$$

with x not in uncanceled assumptions.

10. The *universal quantification elimination rule*:

$$\frac{\forall x.A}{A[x := M]}\ E\forall$$

Here M is an algebraic term.

11. The *existential quantification introduction rule*:

$$\frac{A[x := M]}{\exists x.\, A}\; I\exists$$

12. The *existential quantification elimination rule*:

$$\frac{\exists x.\, A \qquad \forall x.\, (A \to B)}{B}\; E\exists$$

with x not free in B.

Comments.

- In the rule $I{\to}$ zero, one, or more assumptions A are canceled. Those are exactly the assumptions that are labeled with x.
- In the rule $I\forall$ the side-condition is that the variable x does not occur in any uncanceled assumption. Intuitively, the derivation does not depend on x.
- In the rule $E\forall$ there is the usual assumption concerning substitution: no free variables in M are captured by the substitution.
- In the $I\exists$ rule, there is the usual assumption concerning substitution: no variables in M are captured by the substitution.
- In the $E\exists$ rule, there is the following side-condition: x is not free in B. Intuitively, nothing special about x is assumed.

Tautologies. As in the case of first-order propositional logic, a formula is said to be a *tautology* if there exists a proof of it with no uncanceled assumptions.

Examples. We assume unary predicates P and Q, and a nullary predicate A. In addition we assume a term N.

1.

$$\cfrac{\cfrac{[\forall x.P(x)^w]}{P(N)}\; E\forall}{(\forall x.P(x)) \to P(N)}\; I[w]{\to}$$

2.

$$\cfrac{\cfrac{[A^w]}{\forall x.A}\; I\forall}{A \to \forall x.A}\; I[w]{\to}$$

3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[(\forall x.P(x)\to Q(x))^w]}{P(y)\to Q(y)}\,E\forall \qquad \dfrac{[\forall x.P(x)^v]}{P(y)}\,E\forall}{Q(y)}\,E{\to}}{\forall y.Q(y)}\,I\forall}{(\forall x.P(x))\to\forall y.Q(y)}\,I[v]{\to}}{(\forall x.P(x)\to Q(x))\to(\forall x.P(x))\to\forall y.Q(y)}\,I[w]{\to}$$

4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[(A\to\forall x.P(x))^v] \qquad [A^w]}{\forall x.P(x)}\,E{\to}}{P(y)}\,E\forall}{A\to P(y)}\,I[w]{\to}}{\forall y.A\to P(y)}\,I\forall}{(A\to\forall x.P(x))\to\forall y.A\to P(y)}\,I[v]{\to}$$

5.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall y.\,P(x)\to A]}{P(x)\to A} \qquad [P(x)]}{A}}{(\forall y.\,P(y)\to A)\to A}}{P(x)\to(\forall y.\,P(y)\to A)\to A}}{\forall x.\,(P(x)\to(\forall y.\,P(y)\to A)\to A)}$$

6. The following proof is *not* correct:

$$\dfrac{\dfrac{[P(x)^w]}{\forall x.\,P(x)}\,I\forall}{P(x)\to\forall x.\,P(x)}\,I[w]{\to}$$

7. The following proof is *not* correct:

$$\dfrac{\dfrac{\dfrac{[(\exists x.\,P(x))^v] \qquad \dfrac{\dfrac{[P(x)^w]}{P(x)\to Px}\,I[w]\to}{\forall x.\,P(x)\to P(x)}\,I\forall}{P(x)}\,E\exists}{\forall z.\,P(z)}\,I\forall}{(\exists x.\,P(x))\to\forall z.\,P(z)}\,I[v]{\to}$$

8. We abbreviate $\exists x.\, P(x) \vee \exists x.\, Q(x)$ as A.

$$\dfrac{\dfrac{(\exists x\, P(x) \vee Q(x)) \qquad \dfrac{\dfrac{\dfrac{P(y)\vee Q(y) \qquad P(y)\to A \qquad Q(y)\to A}{\exists x.\,P(x)\vee\exists x.\,Q(x)}}{(P(y)\vee Q(y))\to(\exists x.\,P(x))\vee(\exists x.\,Q(x))}}{\forall x.\,(P(x)\vee Q(x))\to((\exists x.\,P(x))\vee(\exists x.\,Q(x)))}}{(\exists x.\,P(x))\vee(\exists x.\,Q(x))}}{(\exists x.\,P(x)\vee Q(x))\to(\exists x.\,P(x))\vee(\exists x.\,Q(x))}$$

What remains to be shown is $P(y) \to A$ and $Q(y) \to A$. We give a proof of the first:

$$\dfrac{\dfrac{\dfrac{P(y)}{\exists x.\, P(x)}}{(\exists x.\, P(x)) \vee (\exists x.\, Q(x))}}{P(y) \to ((\exists x.\, P(x)) \vee (\exists x.Q(x)))}$$

Intuitionism. The proof system presented above is intuitionistic. For instance the formula $A \vee \neg A$ is not a tautology.

Interpretation. The interpretation due to Brouwer, Heyting and Kolmogorov given in Chapter 1 is extended to the case of first-order predicate logic:

- A proof of $A \to B$ is a method that transforms a proof of A into a proof of B.
- A proof of $A \wedge B$ consists of a proof of A and a proof of B.
- A proof of $A \vee B$ consists of first, either a proof of A or a proof of B, and second, something indicating whether it is A or B that is proved.
- (There is no proof of $\perp$.)
- A proof of $\forall x.A$ is a method that transforms a term M into a proof of $A[x := M]$.
- A proof of $\exists x.A$ consists of first a term M, and second a proof of $A[x := M]$. The term M is called a witness.

4.2.2 Minimal logic

Minimal first-order predicate logic is the fragment where we only use the connectives $\to$ and $\forall$. So a formula is either of the form $R(M_1, \ldots, M_n)$, or of the form $A \to B$, or of the form $\forall x.\, A$. We mention this fragment separately because of its correspondence with dependently typed λ-calculus, which is important for the study of the foundations of Coq (see Chapter 6). The notions of detour and detour elimination are studied here only for minimal logic, not for full intuitionistic logic.

Detours. A *detour* is an introduction rule for a connective, immediately followed by an elimination rule for the same connective. Since here we consider two connectives, namely $\to$ and $\forall$, there are two kinds of detours:

- An application of the $I{\to}$ rule immediately followed by an application of the $E{\to}$ rule. The schematic form of such a detour is:

$$\dfrac{\dfrac{B}{A \to B}\,I[x]{\to} \qquad A}{B}\,E{\to}$$

- An application of the $I\forall$ rule immediately followed by an application of the $E\forall$ rule. The schematic form of such a detour is:

$$\dfrac{\dfrac{A}{\forall x.A}\,I\forall}{A[x := M]}\,E\forall$$

Detour elimination. *Detour elimination*, or *proof normalization*, consists of elimination of detours. This is done according to the following two rules:

$$\dfrac{\dfrac{B}{A \to B}\,I[x]{\to} \quad \begin{array}{c}\vdots\\ A\end{array}}{B}\,E{\to} \qquad \to \qquad \text{B}$$

$$\dfrac{\dfrac{A}{\forall x.A}\,I\forall}{A[x := M]}\,E\forall \qquad \to \qquad \text{A}$$

In the first rule, all assumptions A in the proof of B that are labeled with x are replaced by the proof of A indicated by the dots. In the second rule, all occurrences of the variable x in A are replaced by M.

Examples. Two examples of proof normalization steps:

1.

$$\dfrac{\dfrac{\dfrac{[(A \to A)^y]}{B \to (A \to A)}\,I[z]{\to}}{(A \to A) \to B \to (A \to A)}\,I[y]{\to} \qquad \dfrac{[A^x]}{A \to A}\,I[x]{\to}}{B \to (A \to A)}\,E{\to} \qquad \to \qquad \dfrac{\dfrac{[A^x]}{A \to A}\,I[x]{\to}}{B \to (A \to A)}\,I[z]{\to}$$

2.

$$\dfrac{\dfrac{\dfrac{[P(x)^w]}{P(x) \to P(x)}\ I[w]{\to}}{\forall x.P(x) \to P(x)}\ I\forall}{P(f(a)) \to P(f(a))}\ E\forall \qquad \to \qquad \dfrac{[P(f(a))^w]}{P(f(a)) \to P(f(a))}\ I[w]{\to}$$

4.3 Coq

Algebraic Terms in Coq.

- The set of algebraic terms is represented by a variable `Terms` that is assumed, with declaration `Terms:Set`.
- A variable x is represented in Coq as a variable in `Terms`, so with declaration `x : Terms`.
- A n-ary function symbol f is represented in Coq as a variable of type `Terms -> ... -> Terms -> Terms` with $n+1$ times terms, so with declaration `f : Terms -> ... -> Terms -> Terms`.

  For instance, a binary symbol f is represented as a variable with declaration `f : Terms -> Terms -> Terms`.
- In this way, all algebraic terms are represented in Coq as a term of type `Terms`.

Formulas in Coq.

- An n-ary predicate P is represented in Coq as a variable of type `Terms -> ... -> Terms -> Prop` with n times `Terms`, so with declaration `P : Terms -> ... -> Terms -> Prop`.

  For instance, a binary predicate P is represented as a variable with declaration `P : Terms -> Terms -> Prop`.
- The formula $\perp$ is represented in Coq as `False`.
- A formula of the form $A \to B$ is represented in Coq as `A -> B`.

  Note that A and B may be composed formulas; the definition of the representation is in fact an inductive definition. Here with `A` the representation of A in Coq is meant, and with `B` the representation of B in Coq.

  Note that we may write `forall x:A, B` instead of `A -> B`. The reason is that universal quantification over a variable that does notx occur in the body and an implication are represented in Coq by the same type.
- A formula of the form $A \wedge B$ is represented in Coq as `A /\ B`.

- A formula of the form $A \vee B$ is represented in Coq as `A \/ B`.
- A formula of the form $\forall x.A$ is represented in Coq as `forall x:Terms, A`.

  Note that in Coq we need to give the domain of the bound variable x explicitly.

  Note further that both in predicate logic and in Coq the convention is that the scope of the universal quantification is as far as possible to the right. So for instance in predicate logic $\forall x.\, A \to B$ is $\forall x.\, (A \to B)$, and analogously in Coq `forall x:Terms, A -> B` is `forall x:Terms, (A -> B)`.
- A formula of the form $\exists x.A$ can be written in Coq in two different ways.

  The first possibility is to write `exists x:Terms, A`. This is an expression of type `Prop`.

  The second possibility is to write `{ x:Terms | A }`. This is an expression of type `Set`.

  These two ways of expressing the existential quantifier behave differently with respect to program extraction, as we will see later. Here we will use the first representation.
- Of course quantification may occur over another domain than the one of `Terms`: we have in general `forall x:D, A` and `exists x:D, A` and `{ x:D | A }` for some `D` in `Set`.
- In this way, all formulas of predicate logic are represented in Coq as a term in `Prop`.

Tactics.

- `intro`

  This tactic can be used both for introduction of the implication and for introduction of the universal quantification. That is:

  If the current goal is of the form `A -> B`, then `intro x.` transforms the goal into the new goal `B`, and a new hypothesis `x : A` is added to the list of hypotheses.

  If the current goal is of the form `forall x:A, B`, then `intro x.` transforms the goal into the new goal `B`, and a new hypothesis `x : A` is added to the list of hypotheses.

  NB: if the current goal is of the form `forall x:A, B`, then `intro y.` transforms the goal into the new goal `B'`, where `B'` is obtained from `B` by replacing `x` by `y`, and a new hypothesis `y : A` is added to the list of hypotheses.
- `apply`

  This tactic can be used both for elimination of the implication and for elimination of the universal quantification.

If the current goal is of the form `B` and in the list of hypotheses there is a hypothesis `x : A1 -> ... -> An -> B`, then `apply x` transforms the goal into n new subgoals, namely `A1`, ... , `An`.

If the current goal is of the form `B` and in the list of hypotheses there is a hypothesis `x : forall (y1:A1) ... (yn:An), C`, then `apply x` tries to match `B` with `C`, that is, it tries to find the right instance of the universal quantification `forall (y1:A1) ... (yn:An), C`.

Sometimes you need to give explicitly the form in which you want to use a hypothesis of the form `x : forall (y1:A1) ... (yn:An), C`. This can be done by stating `apply (x a1 ... an)`: here the hypothesis is used with instance $C[y_1 := a_1, \ldots, y_n := a_n]$. An alternative for `apply (x a1 ... an)` is `apply x with a1 ... an`, and yet another alternative is `apply x with (x1 := a1) ... (xn := an)`.

If `apply H` gives an error in Coq of the following form: 'Error: Cannot refine to conclusions with meta-variables' then try to use `apply` with the arguments of `H` given explicitly.

- `exists`

  This tactic can be used for the introduction of the existential quantification.

  If the current goal is of the form `exists x:D, A` then `exists d` transforms the goal into `A` where free occurrences of `x` are replaced by `d`. Here we must have `d : D` in the current context. The term `d` is said to be a witness for the existential proof.

- `elim`

  This tactic can be applied to any goal.

  If the current goal is `G` and there is a hypothesis `H : exists x:D, A` then `elim H` transforms the goal into `forall x:D, A -> G`.

  If *label* is the label of a hypothesis of the form `A /\ B` and the current goal is `G`, then `elim` *label* transforms the goal into `A -> B -> G`.

  If *label* is the label of a hypothesis of the form `A \/ B` and the current goal is `G`, then `elim` *label* transforms the goal into two goals: `A -> G` and `B -> G`.

Chapter 5

Program extraction

Constructive functional programming is done in three steps, the last one being program extraction:

1. program specification
2. proof of existence
3. program extraction

In the practical work we consider the specification of a sorting algorithm from which we extract a program for insertion sort.

5.1 Program specification

Here we discuss the general form of a specification. If we are concerned with program abstraction we often consider formulas of the following form:

$$\forall a : A.\, P(a) \to \exists b : B.\, Q(a, b)$$

with P a predicate on A and Q a predicate on $A \times B$, and consider such formula as a specification. According to the interpretation discussed above, a program that verifies this specification should do the following:

> It maps an input $a : A$ to an object $b : B$ together with a program that transforms a program that verifies $P(a)$ into a program that verifies $Q(a, b)$.

The aim of program extraction is to obtain from a proof of a formula of the form $\forall a : A.\, P(a) \to \exists b : B.\, Q(a, b)$ a function or program

$$f : A \to B$$

with the following property:

$$\forall a : A.\, P(a) \to Q(a, f(a)).$$

This is not possible for all formulas. Coq contains a module, called `Extraction`, that permits one to extract in some cases a program from a proof.

5.2 Proof of existence

Most of the work is in this part. The style and contents of the program that is extracted in the third step originate from the style and contents of the proof of existence in the second step.

5.3 Program extraction

This part is automated. Roughly, the part of the proof which is about proving is erased. This is the part in `Prop`. The part of the proof which is about calculating remains. This is the part in `Set`. The extracted program is CAML or Haskell. Recently a new extraction module was written for Coq, for more information see the homepage of the Coq project.

5.4 Insertion sort

We consider a formula that can be seen as the specification of a sorting algorithm. We assume the type `natlist` for the finite lists of natural numbers, and in addition the following predicates:

- The predicate `Sorted` on `natlist` that expresses that a list is sorted (that is, the elements form an increasing list of natural numbers).
- The predicate `Permutation` taking two elements of `natlist` as input, that expresses that those two lists are permutations of each other (that is, they contain exactly the same elements but possibly in a different order).
- The predicate `Inserted` is used in the definition of `Permutation`. It takes as input a natural number, a natlist, and another natlist. It expresses that the second natlist is obtained by inserting the natural number in some position in the first natlist.

The formula or specification we are interested in is then the following:

```
Theorem Sort : forall l : natlist,
  {l' : natlist | Permutation l l' /\ Sorted l'}.
```

Note the use of the existential quantification in `Set` (not in `Prop`): this is essential for the use of `Extraction`. The aim of the practical work of this week is to prove this formula correct, and extract from the proof a mapping

$$f : \texttt{natlist} \rightarrow \texttt{natlist}$$

in the form of a ML program, that verifies the following:

$$\forall l : \texttt{natlist}.\, \texttt{Sorted}(f(l)) \wedge \texttt{Permutation}(l, f(l)).$$

Proof of Sort. The proof of sort proceeds by induction on the natlist l. The base case (for the empty list), follows from the definition of Sorted and the definition of Permutation. In the induction step, the predicates Insert and Permutation are used, in a somewhat more difficult way.

5.5 Coq

Existential quantification in Coq. There are two different ways to write an existential quantification $\exists x : A. P$ in Coq:

```
exists x:A, P
```

with `exists x:A, P : Prop`, and

```
{x:A | P}
```

with `{x:A | P} : Set`. The existential quantification `exists x:A, P` in `Prop` is considered as a logical proposition without computational information. It is used only in logical reasoning. The existential quantification `{x:A | P}` in `Set` is considered as the type of the terms in `A` that verify the property `P`. It is used to extract a program from a proof.

Chapter 6

λ-calculus with dependent types

This chapter contains first an informal introduction to lambda calculus with dependent types (λP). This is an extension of simply typed λ-calculus with the property that types may depend on terms. It is explained how predicate logic can be coded in λP. Then we give a formal presentation of λP.

6.1 Dependent types: introduction

We consider some examples from the programming point of view. One of the reasons to consider typed programming languages is that type checking can reveal programming errors at compile time. The more informative the typing system is, the more errors can be detected. Of course it is a trade-off: type checking becomes harder for more complex typing systems.

Mapping from natural numbers to lists. We consider a mapping zeroes with the following behaviour:

$$\begin{array}{rcl} \mathsf{zeroes}\,0 & = & \mathsf{nil} \\ \mathsf{zeroes}\,1 & = & (\mathsf{cons}\,0\,\mathsf{nil}) \\ \mathsf{zeroes}\,2 & = & (\mathsf{cons}\,0\,(\mathsf{cons}\,0\,\mathsf{nil})) \\ & \vdots & \end{array}$$

The mapping zeroes takes as input a natural number and yields as output a finite list of natural numbers, so a possible type for zeroes is $\mathsf{nat} \to \mathsf{natlist}$. However, a careful inspection of the specification of F reveals that more can be said: the function F maps a natural number n to a list of natural numbers of length exactly n. From a programming point of view, it may be important to express this information in the typing. This is because the more information a type provides, the more errors can be detected by type checking. In the example, if the typing provides the information that the term zeroes 2 is a list of natural numbers of length 2, then it is clear by only inspecting the types that taking the 3rd element of zeroes 2 yields an error. In order to detect the error you don't need to compute zeroes 2.

So what we need here are types

- zeronatlist representing the lists of natural numbers of length 0,
- onenatlist representing the lists of natural numbers of length 1,
- twonatlist representing the lists of natural numbers of length 2,
- ...

Then we need a typing system to derive

- zeroes 0 : zeronatlist,
- zeroes 1 : onenatlist,
- zeroes 2 : twonatlist,
- ...

We will see below how this is done in a systematic way.

Append. The function append takes as input two finite lists of natural numbers and returns the concatenation of these lists. In Coq:

```
Fixpoint append (k l : natlist) {struct k} : natlist :=
  match k with
    nil => l
  | cons h t => cons h (append t l)
  end.
```

Here the type of append is natlist $\to$ natlist $\to$ natlist. If the length of the list is relevant, then a more informative type for append is a type expressing that if append takes as input a list of length n and a list of length m, then it yields as output a list of length $n + m$.

Reverse. The function reverse reverses the contents of a list. In Coq:

```
Fixpoint reverse (k : natlist) : natlist :=
  match k with
    nil => nil
  | cons h t => append (reverse t) (cons h nil)
  end.
```

The type of reverse is here natlist $\to$ natlist. The length of a list is invariant under application of reverse. So a more informative type for reverse should express that if reverse takes as input a list of length n, then it yields as output a list of length n.

Constructors. In order to express the type representing the lists of length n in a systematic way, we use a *constructor* $\mathsf{natlist\_dep}$ that takes as input a natural number n and yields as output the type representing the lists of length n. What is the type of this constructor? Recall that data types like nat and $\mathsf{natlist}$ live in Set. The type of $\mathsf{natlist\_dep}$ is then: $\mathsf{nat} \to \mathsf{Set}$. Now the desired typings for the first example above are:

- $\mathsf{zeroes}\,0 : \mathsf{natlist\_dep}\,0$,
- $\mathsf{zeroes}\,1 : \mathsf{natlist\_dep}\,1$,
- $\mathsf{zeroes}\,2 : \mathsf{natlist\_dep}\,2$
- ...

A first observation is that in this way types and terms are no longer separate entities. For instance $\mathsf{natlist\_dep}\,0$ is the type of $\mathsf{zeroes}\,0$ and contains application and a sub-term of type nat. So it also seems to be a term. Indeed in λP all expressions are called terms, although we still also use the terminology 'type': if $M : A$ then we say that A is a type of M.

A second observation is that with dependent types, types may be reducible. For instance, intuitively we have besides $\mathsf{zeroes}\,0 : \mathsf{natlist\_dep}\,0$ also

$$\mathsf{zeroes}\,0 : \mathsf{natlist\_dep}\,((\lambda x{:}\mathsf{nat}.\,x)\,0)$$

Indeed in λP there is the *conversion rule* in the typing system, which roughly speaking expresses that if we have $M : A$ and $A = A'$, then we also have $M : A'$. The question arises what the $=$ means. In pure λP, this is β-equality. In Coq, besides β we also use for instance δ reduction.

A third issue is the following. In the simply typed λ-calculus, the statement 'nat is a type' is done on a meta-level. That is, there is no formal system to derive this. In λP, we derive $\mathsf{nat} : \mathsf{Set}$ in a formal system. (This formal system is given later on in the course notes.) We have for instance

$$\begin{array}{lclclcl}
0 & : & \mathsf{nat} & : & \mathsf{Set} & : & \mathsf{Type} \\
\mathsf{S} & : & \mathsf{nat} \to \mathsf{nat} & : & \mathsf{Set} & : & \mathsf{Type} \\
\mathsf{nil} & : & \mathsf{natlist} & : & \mathsf{Set} & : & \mathsf{Type} \\
\mathsf{cons} & : & \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist} & : & \mathsf{Set} & : & \mathsf{Type}
\end{array}$$

And also

$$\begin{array}{rclclcl}
 & & \mathsf{natlist\_dep} & : & \mathsf{nat} \to \mathsf{Set} & : & \mathsf{Type} \\
\mathsf{nil} & : & \mathsf{natlist\_dep}\,0 & : & \mathsf{Set} & : & \mathsf{Type}
\end{array}$$

If $K : \mathsf{Type}$, then K is called a *kind*. If $C : K$ for some kind K, then C is called a *constructor*.

Application. We now know the type of natlist_dep and of for instance zeroes 0, but we still have to discuss the type zeroes, and the way the type of zeroes 0 is obtained using the application rule.

First concerning the type of zeroes: the intuition is: *for all n in* nat*,* zeroes *is of type* natlist_dep n. This is denoted as follows:

$$\mathsf{zeroes} : \Pi n{:}\mathsf{nat}.\ \mathsf{natlist\_dep\ n}$$

Such a type is called a product type.

Concerning application, we first recall that the application rule in simply typed λ-calculus can be instantiated as follows:

$$\frac{\mathsf{S} : \mathsf{nat} \to \mathsf{nat} \qquad \mathsf{0} : \mathsf{nat}}{\mathsf{S\,0} : \mathsf{nat}}$$

The application rule in λP generalizes the one for simply typed λ-calculus. So in λP we still have the previous application.

The application rule in λP expresses that a term of type $\Pi x{:}A.\,B$ can be applied to a term of type A. For example:

$$\frac{\mathsf{zeroes} : \Pi n{:}\mathsf{nat}.\ \mathsf{natlist\_dep}\, n \qquad \mathsf{0} : \mathsf{nat}}{\mathsf{zeroes\,0} : \mathsf{natlist\_dep\,0}}$$

and

$$\frac{\mathsf{zeroes} : \Pi n{:}\mathsf{nat}.\ \mathsf{natlist\_dep}\, n \qquad \mathsf{S\,0} : \mathsf{nat}}{\mathsf{zeroes\,(S\,0)} : \mathsf{natlist\_dep\,(S\,0)}}$$

It illustrates the use of dependent types. In general, if a term F has a type of the form $\Pi x{:}A.\,B$, then F can be applied to a term M of type A, and this yields the term $F\,M$ of type B *where x is replaced by M*. Still informally, application with dependent types is according to the following rule:

$$\frac{F : \Pi x{:}A.\,B \qquad M : A}{F\,M : B[x := M]}$$

So it seems that at this point we have two application rules:

$$\frac{F : \Pi x{:}A.\,B \qquad M : A}{F\,M : B[x := M]} \qquad \frac{F : A \to B \qquad M : A}{F\,M : B}$$

The first rule is for application with dependent types, and the second one for application with function types. The second rule can however be seen as an instance of the first one: observe that B is the same as $B[x := M]$ if x does not occur in B, and write a function type $A \to B$ as a dependent type $\Pi x{:}A.\,B$. This makes the presentation of λP more uniform. We will still use the notation $A \to B$ instead of $\Pi x{:}A.\,B$ in cases that x does not occur in B.

The types of the functions in the examples are as follows:

zeroes	:	$\Pi n{:}\mathsf{nat}.\ \mathsf{natlist\_dep}\, n$
append	:	$\Pi n, m{:}\mathsf{nat}.\ \mathsf{natlist\_dep}\, n \to \mathsf{natlist\_dep}\, m \to \mathsf{natlist\_dep}\,(\mathsf{plus}\, n\, m)$
reverse	:	$\Pi n{:}\mathsf{nat}.\ \mathsf{natlist\_dep}\, n \to \mathsf{natlist\_dep}\, n$

6.2 λP

In this section we give a formal presentation of λP. First we build a set of *pseudo-terms*. Then the *terms* are selected from the pseudo-terms using a type system.

Symbols. We assume the following:

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

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. The set of *pseudo-terms* of λP is defined by induction according to the following grammar:

$$\mathsf{P} ::= \mathsf{Var} \mid * \mid \Box \mid \Pi\mathsf{Var}{:}\mathsf{P}.\,\mathsf{P} \mid \lambda\mathsf{Var}{:}\mathsf{P}.\,\mathsf{P} \mid (\mathsf{P}\,\mathsf{P}).$$

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 as Prop. In λP both of these are identified. This means that λP does not make the distinction between Set and Prop that Coq makes.

- $\Box$ represents the set of kinds.

  In Coq $\Box$ is written as Type.

- 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 doesn't occur in B, then we also write $A \to B$ 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.

Examples of pseudo-terms are:

- x (a variable),
- $*$,
- $\lambda x{:}\mathsf{nat}.\, x$,
- $(*\,\Box)$,
- $(\lambda x{:}\mathsf{nat}.\, x\,x)\,(\lambda x{:}\mathsf{nat}.\, x\,x)$,
- $\lambda x{:}*.\, x$,
- $\Pi x{:}\mathsf{nat}.\,(\mathsf{natlist\_dep}\,x)$.

We will see below that not all these pseudo-terms are terms.

Environments. 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 $\Gamma, \Delta, \ldots$.

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

1. The *start rule.*

$$\frac{}{\vdash * : \Box}$$

2. The *variable rule.*

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

3. The *weakening rule.*

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

4. The *product rule.*

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

5. The *abstraction rule.*

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

6. The *application rule.*

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

7. The *conversion rule.*

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

Some remarks concerning the typing system:

- A rule that contains an s is in fact an abbreviation of two rules, one in which s is $*$ and one in which s is $\Box$. For instance, there actually are two variable rules:

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

  and

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

  The first is used for variables of which the type is a data-type or a proposition. The second is used for variables that themselves are types, like nat or natlist_dep.

- The start rule is the only rule without a premise. A rule without premise is sometimes called an *axiom.* All derivations in the typing system of λP start (at the top) with a start rule.

- The weakening rule only changes the environment. It is necessary because if two derivations are combined (this happens besides in the weakening rule also in the abstraction rule, the application rule, and the conversion rule), then the two environments of these derivations must be equal.

  In logic, the weakening rule is one of the structural rules. In most logics one may add an assumption and then still the same statements are valid. (This is not anymore the case in linear logic.)

- There are two product rules.

  If $B : *$ where B represents a data-type, then the product rule is used to build products that are also data-types. An example of such a product is $\Pi x{:}\mathsf{nat}.\, \mathsf{nat}$. (It represents the data-type of functions from nat to nat.)

  If $B : *$ where B represent a proposition, then the product rule is used to build products that are also propositions. An example of such a product is $\Pi x{:}\mathsf{nat}.\, A$ with $A : *$. (It represents the proposition $\forall x : \mathsf{nat}.\, A$.)

  If $B : \Box$, then the product rule is used to build products that are kinds. An example of such a product is $\Pi x{:}\mathsf{nat}.\, *$. (It represents in λP the type of natlist_dep.)

- The conversion rule permits one to use types that are the same up to β-conversion (in Coq: $\beta\iota\zeta\delta$-conversion) in the same way.

  For example, if we have $l : (\mathsf{natlist\_dep}\,((\lambda x{:}\mathsf{nat}.\,x)\,0))$, then we can also derive $l : (\mathsf{natlist\_dep}\,0)$.

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.

Examples.

- We have $\vdash * : \Box$. Hence $*$ and $\Box$ are terms.
- Note that in the typing system given here we cannot derive $\vdash \Box : \Box$. As a matter of fact, we cannot derive $\Gamma \vdash \Box : A$ for any Γ and A. In Coq however, we have from the user's point of view $\Box : \Box$. In fact there is an infinite hierarchy of types with $\mathsf{Type}_i : \mathsf{Type}_{i+1}$. As a consequence, the pseudo-term $(*\,\Box)$ is not a term.
- An application of the variable rule:

$$\frac{\dfrac{\vdash * : \Box}{X : * \vdash X : *}}{X : *, x : X \vdash x : X}$$

- A similar application of the variable rule:

$$\frac{\dfrac{\vdash * : \Box}{A : * \vdash A : *}}{A : *, p : A \vdash p : A}$$

- Note that we cannot declare $U : \Box$. We can do that in Coq.
- We have seen a derivation of $X : *, x : X \vdash x : X$. We can continue using the weakening rules:

$$\frac{X : *, x : X \vdash x : X \qquad \dfrac{\dfrac{\vdash * : \Box \qquad \vdash * : \Box}{X : * \vdash * : \Box} \qquad \dfrac{\vdash * : \Box}{X : * \vdash X : *}}{X : *, x : X \vdash * : \Box}}{X : *, x : X, Y : * \vdash x : X}$$

- We have that $\Pi x{:}\mathsf{nat}.\,\mathsf{nat}$ (also written as $\mathsf{nat} \to \mathsf{nat}$) is a term:

$$\frac{\dfrac{\vdash * : \Box}{\mathsf{nat} : * \vdash \mathsf{nat} : *} \qquad \dfrac{\dfrac{\vdash * : \Box}{\mathsf{nat} : * \vdash \mathsf{nat} : *} \qquad \dfrac{\vdash * : \Box}{\mathsf{nat} : * \vdash \mathsf{nat} : *}}{\mathsf{nat} : *, x : \mathsf{nat} \vdash \mathsf{nat} : *}}{\mathsf{nat} : * \vdash \Pi x{:}\mathsf{nat}.\,\mathsf{nat} : *}$$

So here we derive in the formal system that nat $\to$ nat is in $*$ (intuition: is a data-type) whereas in the previous presentation of simply typed λ-calculus the statement 'nat $\to$ nat is a type' is stated and proved on a meta-level.

- We have a similar derivation showing that $\Pi x{:}A.\,A$ (also written as $A \to A$) is a proposition:

$$\dfrac{\dfrac{\vdash * : \square}{A : * \vdash A : *} \qquad \dfrac{\dfrac{\vdash * : \square}{A : * \vdash A : *} \quad \dfrac{\vdash * : \square}{A : * \vdash A : *}}{A : *, x : A \vdash A : *}}{A : * \vdash \Pi x{:}A.\,A : *}$$

- The pseudo-term $\Pi x{:}\mathsf{nat}.\,*$ is a term:

$$\dfrac{\dfrac{\vdash * : \square}{\mathsf{nat} : * \vdash \mathsf{nat} : *} \qquad \dfrac{\dfrac{\vdash * : \square \quad \vdash * : \square}{\mathsf{nat} : * \vdash * : \square} \quad \dfrac{\vdash * : \square}{\mathsf{nat} : * \vdash \mathsf{nat} : *}}{\mathsf{nat} : *, x : \mathsf{nat} \vdash * : \square}}{\mathsf{nat} : * \vdash \Pi x{:}\mathsf{nat}.\,* : \square}$$

  This derivation shows that if nat is declared as a data-type, so nat : $*$, then nat$\to*$ lives in $\square$, so nat$rightarrow*$ is a kind.

- We have seen a derivation of $\mathsf{nat} : * \vdash \Pi x{:}\mathsf{nat}.\,\mathsf{nat} : *$. Call that derivation Δ and use it to derive that $\mathsf{nat} : * \vdash \lambda x{:}\mathsf{nat}.\,x : \Pi x{:}\mathsf{nat}.\,\mathsf{nat}$:

$$\dfrac{\dfrac{\dfrac{\vdash * : \square}{\mathsf{nat} : * \vdash \mathsf{nat} : *}}{\mathsf{nat} : *, x : \mathsf{nat} \vdash x : \mathsf{nat}} \qquad \Delta}{\mathsf{nat} : * \vdash \lambda x{:}\mathsf{nat}.\,x : \Pi x{:}\mathsf{nat}.\,\mathsf{nat}}$$

Simply typed λ-calculus. λP is an extension of simply typed λ-calculus. The presentation of simply typed λ-calculus that we have seen before is different from the presentation of λP here. How is simply typed λ-calculus a subsystem of λP?

It is obtained by omitting the product rule with $B : \square$. Then we have only types like nat $\to$ nat or $A \to B$ (with A and B in $*$), or nat $\to B$. Then we don't have nat $\to *$, so there is no type for natlist_dep.

6.3 Predicate logic in λP

Lambda calculi with dependent type have been introduced by De Bruijn in order to represent mathematical theorems and their proofs. The 'P' in λP stands for 'predicate'. We now discuss the Curry-Howard-De Bruijn isomorphism between a fragment of λP and predicate logic with only $\to$ and $\forall$.

We assume a signature Σ consisting of a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$. We define here a fragment of λP that depends on Σ.

- $\star$ represents the universe of propositions.
- There is a distinguished type variable Terms that intuitively represents the set of algebraic terms.
- All kinds are of the form $\mathsf{Terms} \to \ldots \mathsf{Terms} \to \star$ with 0 or more times Terms. So we have for instance the kinds $\star$ and $\mathsf{Terms} \to \mathsf{Terms} \to \star$.
- For every n-ary function symbol f in $\mathcal{F}$ there is a distinguished variable f of type $\mathsf{Terms} \to \ldots \to \mathsf{Terms} \to \mathsf{Terms}$ with $n+1$ times Terms.
- For every n-ary relation symbol R in $\mathcal{P}$ there is a distinguished variable R of type $\mathsf{Terms} \to \ldots \to \mathsf{Terms} \to \star$ with n times Terms.

Algebraic terms are translated into λP as follows:

- The algebraic term x in predicate logic is translated into the λ-term x with $x : \mathsf{Terms}$ in λP.
- The algebraic term $f(M_1, \ldots, M_n)$ in predicate logic is translated into the λ-term $f\, M_1^* \ldots M_n^*$ with $M_1^*, \ldots, M_n^*$ the translations of $M_1, \ldots, M_n$. Note that $f : \mathsf{Terms} \to \ldots \to \mathsf{Terms} \to \mathsf{Terms}$ with $n+1$ times Terms.

Formulas are translated into λP as follows:

- The atomic formula $R(M_1, \ldots, M_n)$ in predicate logic is translated into the λ-term $R\, M_1^* \ldots M_n^*$ with $M_1^*, \ldots, M_n^*$ the translations of $M_1, \ldots, M_n$. Note that $R : \mathsf{Terms} \to \ldots \to \mathsf{Terms} \to \star$ with n times Terms.
- The formula $A \to B$ in predicate logic is translated into the λ-term $\Pi x{:}A^*.\, B^*$, also written as $A^* \to B^*$.
- The formula $\forall x.A$ in predicate logic is translated into the λ-term $\Pi x{:}\mathsf{Terms}.\, A^*$ with A^* the translation of A.

Note that we have the following:

- The translation of an algebraic term is a λ-term in Terms because we have
$$x : \mathsf{Terms}$$
and
$$f\, M_1^* \ldots M_n^* : \mathsf{Terms}$$
- The translation of a formula is a λ-term in $\star$ because we have
$$R\, M_1^* \ldots M_n^* : \star$$
and, for $A : \star$ and $B : \star$
$$\Pi x{:}A^*.\, B^* : \star$$
and, for $A : \mathsf{Terms}$ and $B : \star$,
$$\Pi x{:}A^*.\, B^* : \star$$

The Isomorphism. Using the translation, we find the following correspondence between formulas of predicate logic and types of λP is as follows:

$R(M_1,\ldots,M_n)$	$\sim$	$R\,M_1^* \ldots M_n^*$
$A{\rightarrow}B$	$\sim$	$\Pi x{:}A^*.\,B^*$
$\forall x.\,A$	$\sim$	$\Pi x{:}\mathsf{Terms}.\,A^*$

Further, we find the following correspondence between the rules of predicate logic and the rules of λP (we come back to this point later):

assumption	$\sim$	term variable
$\rightarrow$ introduction	$\sim$	abstraction
$\forall$ introduction	$\sim$	abstraction
$\rightarrow$ elimination	$\sim$	application
$\forall$ elimination	$\sim$	application

Examples. We consider examples corresponding to the examples in Section 7.2.

1. The formula $(\forall x.\,P(x)) \rightarrow P(N)$ in predicate logic corresponds to the λP-term $(\Pi x : \mathsf{Terms}.\,P\,x) \rightarrow (P\,N)$. An inhabitant of this term is:
$$\lambda w : (\Pi x : \mathsf{Terms}.\,P\,x).\,(w\,N)$$
2. The formula $A \rightarrow \forall x.A$ in predicate logic corresponds to the λP-term $A \rightarrow \Pi x : \mathsf{Terms}.\,A$. An inhabitant of this term is:
$$\lambda w : A.\,\lambda x : \mathsf{Terms}.\,w$$
3. The formula $(\forall x.P(x) \rightarrow Q(x)) \rightarrow (\forall x.P(x)) \rightarrow \forall y.Q(y)$ in predicate logic corresponds to the λP-term $(\Pi x : \mathsf{Terms}.\,P\,x \rightarrow Q\,x) \rightarrow (\Pi x : \mathsf{Terms}.\,P\,x) \rightarrow (\Pi y : \mathsf{Terms}.\,Q\,y)$. An inhabitant of this term is:
$$\lambda w : (\Pi x : \mathsf{Terms}.\,P\,x \rightarrow Q\,x).\,\lambda v : (\Pi x : \mathsf{Terms}.\,P\,x).\,\lambda y : \mathsf{Terms}.\,w\,y\,(v\,y)$$
4. The formula $(A \rightarrow \forall x.P(x)) \rightarrow \forall y.A \rightarrow P(y)$ in predicate logic corresponds to the λP-term $(A \rightarrow (\Pi x : \mathsf{Terms}.\,P\,x)) \rightarrow \Pi y : \mathsf{Terms}.\,(A \rightarrow P\,y)$. An inhabitant of this term is:
$$\lambda v : (A \rightarrow \Pi x : \mathsf{Terms}.\,P\,x).\,\lambda y : \mathsf{Terms}.\,\lambda w : A.\,v\,w\,y$$
5. The formula $\forall x.\,(P(x) \rightarrow (\forall y.\,P(y) \rightarrow A) \rightarrow A)$ in predicate logic corresponds to the λP-term $\Pi x : \mathsf{Terms}.\,(P\,x \rightarrow (\Pi y : \mathsf{Terms}.\,P\,y \rightarrow A) \rightarrow A)$. An inhabitant of this term is:
$$\lambda x : \mathsf{Terms}.\,\lambda u : P\,x.\,\lambda v : (\Pi y : P\,x \rightarrow A).\,v\,x\,u$$

Chapter 7

Second-order propositional logic

In this chapter first-order propositional logic is extended with universal and existential quantification over propositional variables. In this way second-order propositional logic is obtained. We consider a natural deduction proof system.

7.1 Formulas

The *formulas* of second-order propositional logic are built from the propositional variables and the connectives mentioned above inductively:

1. a propositional variable a is a formula,
2. the constant $\perp$ is a formula,
3. if A and B are formulas, then $(A \to B)$ is a formula,
4. if A and B are formulas, then $(A \wedge B)$ is a formula,
5. if A and B are formulas, then $(A \vee B)$ is a formula,
6. if A is a formula, then $(\forall a.\, A)$ is a formula,
7. if A is a formula, then $(\exists a.\, A)$ is a formula.

The difference with first-order propositional logic is that now we allow quantification over propositional variables. On the one hand, second-order propositional logic is more restricted than first-order predicate logic, because all predicate symbols have arity 0 (and are also called propositional letters). Hence *propositional* instead of *predicate*. On the other hand, second-order propositional logic is more general than first-order predicate logic, because quantification over propositions is allowed. Hence *second-order* instead of *first-order*.

Order. In the description of systems or problems we often use the terminology 'nth order'. The n is the order of the variables over which quantification takes place.

First-order predicate logic is said to be *first-order* because the (term) variables over which quantification can be done denote individuals in the domain. So quantification is over first-order variables. Now we can consider also variables over functions and predicates, which both take first-order objects as input. Variables over functions and predicates are second-order. If we also quantify over those variables, we obtain second-order predicate logic. Then it is for instance possible to write $\forall P.\, \forall x.\, P(x) \to P(x)$.

Now let's consider propositional logic. A propositional letter is like a predicate letter with arity 0 (so it expects no arguments). A propositional variable is a variable over propositions, so a variable of order 2. If we admit quantification over propositional variables, we obtain 2nd order propositional logic. Then it is for instance possible to write $\forall p.\, p \to p$.

Parentheses. We assume the conventions concerning parentheses as for first-order propositional logic. For instance, we write $A \to B \to C$ instead of $(A \to (B \to C))$. For the scope of the quantifiers we follow the Coq convention: the quantifier scope extends to the right as much as possible. So for instance we may write $\forall a.\, a \to b$ instead of $\forall a.\, (a \to b)$.

It seems there is no generally agreed on convention concerning quantification, so we use parentheses to avoid confusion.

Free and bound variables. The quantifiers $\forall$ and $\exists$ bind propositional variables. The conventions concerning free and bound variables are the same as before. For instance, we identify the formulas $\forall a.\, (a \to a)$ and $\forall b.\, (b \to b)$.

Substitution. We need the notion of substitution of a formula A for a propositional variable a, notation $[a := A]$. It is assumed that bound propositional variables are renamed in order to avoid unintended capturing of free propositional variables. The inductive definition of substitution is as follows:

1. $a[a := A] = A$,
2. $b[a := A] = b$,
3. $\bot[a := A] = \bot$,
4. $(B \to B')[a := A] = B[a := A] \to B'[a := A]$,
5. $(B \wedge B')[a := A] = (B[a := A]) \wedge (B'[a := A])$,
6. $(B \vee B')[a := A] = (B[a := A]) \vee (B'[a := A])$,
7. $(\forall b.\, B)[a := A] = \forall b.\, (B[a := A])$,
8. $(\exists b.\, B)[a := A] = \exists b.\, B[a := A]$.

Impredicativety. The meaning of a formula of the form $\forall p.A$ depends on the meaning of all formulas $A[p := B]$. So in A, all p's can be replaced by some formula B. The formula B can be simpler than the formula A, but it can also be A itself, or a formula that is more complex than A. That is, the meaning of $\forall p.\, A$ depends on the meaning of formulas that are possibly more complex than A. This is called *impredicativety.* It makes that many proof methods, for instance based on induction on the structure of a formula, fail.

7.2 Intuitionistic logic

The natural deduction proof system for second-order propositional logic is an extension of the one for first-order propositional logic. The new rules are the introduction and elimination rules for universal and existential quantification. These are similar to the rules for quantification in first-order predicate logic; note however that the domain of quantification is different. For completeness we give below all the rules for second-order propositional logic.

The introduction and elimination rules.

1. The *assumption rule.*

   A labeled formula A^x is a proof.

   $$A^x$$

   Such a part of a proof is called an *assumption.*

2. The *implication introduction rule*:

   $$\dfrac{\begin{matrix}\vdots\\ B\end{matrix}}{A \to B}\; I[x] \to$$

3. The *implication elimination rule*:

   $$\dfrac{\begin{matrix}\vdots\\ A \to B\end{matrix} \qquad \begin{matrix}\vdots\\ A\end{matrix}}{B}\; E \to$$

4. The *conjunction introduction rule*:

$$\dfrac{\begin{matrix}\vdots & & \vdots \\ A & & B\end{matrix}}{A \wedge B}\ I\wedge$$

5. The *conjunction elimination rules*:

$$\dfrac{\begin{matrix}\vdots \\ A \wedge B\end{matrix}}{A} El\wedge \qquad\qquad \dfrac{\begin{matrix}\vdots \\ A \wedge B\end{matrix}}{B} Er\wedge$$

6. The *disjunction introduction rules*:

$$\dfrac{\begin{matrix}\vdots \\ A\end{matrix}}{A \vee B} Il\vee \qquad\qquad \dfrac{\begin{matrix}\vdots \\ B\end{matrix}}{A \vee B} Ir\vee$$

7. The *disjunction elimination rule*:

$$\dfrac{\begin{matrix} & \vdots & \vdots \\ A \vee B & A \to C & B \to C\end{matrix}}{C}$$

8. The *falsum rule*:

$$\dfrac{\begin{matrix}\vdots \\ \bot\end{matrix}}{A}$$

9. The *universal quantification introduction rule.*

$$\frac{A}{\forall a.\, A}\ I\forall$$

with a not in uncancelled assumptions.

10. The *universal quantification elimination rule.*

$$\frac{\forall a.\, B}{B[a := A]}\; E\forall$$

with A some formula.

11. The *existential quantification introduction rule.*

$$\frac{B[a := A]}{\exists a.\, B}\; I\exists$$

with A some formula.

12. The *existential quantification elimination rule.*

$$\frac{\exists a.\, A \qquad \forall a.\, A \to B}{B}\; E\exists$$

with a not free in B.

Examples. The main difference between tautologies in *second-order* propositional logic and the ones that can be derived in (*first-order*) propositional logic is that in the first case the quantification over formulas can be done in a formal way, whereas in the second case the quantification over formulas is informal (on a meta-level). For instance, in propositional logic the formula $A \to A$ is a tautology for all formulas A. In second-order propositional logic this quantification can be made formal: the formula $\forall a.\, a \to a$ is a tautology.

1.

$$\cfrac{\cfrac{[a^x]}{a \to a}\; I[x]\to}{\forall a.\, a \to a}\; I\forall$$

2.

$$\cfrac{\cfrac{\cfrac{\cfrac{[a^x]}{b \to a}\; I[y]}{a \to b \to a}\; I[x]\to}{\forall b. a \to b \to a}\; I\forall}{\forall a.\, \forall b.\, a \to b \to a}\; I\forall$$

3. The following proof is not correct:

$$\dfrac{a}{\forall a.\, a}\; I\forall$$

4. The following proof is not correct:

$$\dfrac{\exists a.\, a \to b \qquad \dfrac{\dfrac{[a \to b^x]}{(a \to b) \to (a \to b)}\; I[x] \to}{\forall a.\, (a \to b) \to (a \to b)}\; I\forall}{a \to b}\; E\exists$$

Interpretation. The interpretation of second-order propositional logic due to Brouwer, Heyting and Kolmogorov is as follows:

- A proof of $A \to B$ is a method that transforms a proof of A into a proof of B.
- A proof of $A \wedge B$ consists of a proof of A and a proof of B.
- A proof of $A \vee B$ consists of first, either a proof of A or a proof of B, and second, something indicating whether it is A or B that is proved.
- (There is no proof of $\perp$.)
- A proof of $\forall p.\, A$ is a method that transforms every proof of any formula B into a proof of $A[p := B]$.
- A proof of $\exists p.\, A$ consists of a proposition B with a proof of B, and a proof of $A[p := B]$. The proposition B is called a *witness*.

7.3 Minimal logic

Minimal second-order propositional logic is the fragment of intuitionistic logic where we have only the connectives $\to$ and $\forall$. Again, the notions of detour and detour elimination are studied here only for the *minimal* fragment.

Detours. In second-order propositional logic there are two kinds of detours:

- An application of the $I \to$ rule immediately followed by an application of the $E \to$ rule. The schematic form of such a detour is as follows:

$$\dfrac{\dfrac{[B^x]}{A \to B}\; I[x] \to \qquad A}{B}\; E \to$$

  This kind of detour is also present in minimal first-order propositional logic and in minimal first-order predicate logic.

- An application of the $I\forall$ rule immediately followed by an application of the $E\forall$ rule. The schematic form of such a detour is as follows:

$$\dfrac{\dfrac{B}{\forall a.\, B}\, I\forall}{B[a := A]}\, E\forall$$

This kind of detour is similar to a detour present in minimal first-order predicate logic, but there the universal quantification is different (namely over term variables, and here over propositional variables).

Detour elimination. *Proof normalization* or *detour elimination* in second-order propositional logic consists of elimination of the detours as given above. This is done according to the following two rules:

$$\dfrac{\dfrac{[B^x]}{A \to B}\, I[x]\to \quad \begin{array}{c}\vdots\\ A\end{array}}{B}\, E\to \qquad \to \qquad \mathrm{B}$$

$$\dfrac{\dfrac{B}{\forall a.\, B}\, I\forall}{B[a := A]}\, E\forall \qquad \to \qquad \mathrm{B}$$

The first rule is also present in minimal first-order propositional logic and in minimal first-order predicate logic. Here all assumptions A in the proof of B that are labelled with x are replaced by the proof of A indicated by the dots.

The second rule is typical for minimal second-order propositional logic. In minimal first-order predicate logic a similar rule is present, but for a different kind of universal quantification. Here all occurrences of the propositional variable a in the proof of B are replaced by the formula (or proposition) A.

In Chapter 8 we will study the connection between proofs in minimal second-order propositional logic and terms in polymorphic λ-calculus, and between detour elimination in minimal second-order propositional logic and β-reduction in polymorphic λ-calculus.

An example of proof normalization (labels are omitted):

$$\dfrac{\dfrac{\dfrac{\dfrac{[A \to A^z] \quad [A^x]}{A}}{(A \to A)}}{(A \to A) \to (A \to A)} \qquad \dfrac{[A^y]}{(A \to A)}}{(A \to A)} \qquad \to \qquad \dfrac{\dfrac{\dfrac{[A^y]}{(A \to A)} \quad [A^x]}{A}}{A \to A}$$

$$\rightarrow \qquad \frac{[A^x]}{A \rightarrow A}$$

And another example:

$$\dfrac{\dfrac{\dfrac{[a]}{a \rightarrow a}}{\forall a.\, a \rightarrow a}}{(A \rightarrow B) \rightarrow (A \rightarrow B)} \qquad \rightarrow \qquad \frac{A \rightarrow B}{(A \rightarrow B) \rightarrow (A \rightarrow B)}$$

7.4 Classical logic

Classical logic is obtained from intuitionistic logic in the usual way, by adding the law of excluded middle, or Pierce's Law, or the ex falso sequitur quodlibet rule.

Intuitively, the intended meaning of $\forall p.\, A$ (where p may occur in A) is that A holds for all possible meanings of p. In classical logic, the meaning of p can be true or false. If true is denoted by $\top$, and false by $\bot$, then the formula $\forall p.\, A$ is classically equivalent to the formula $A[p := \top] \wedge A[p := \bot]$.

Similarly, intuitively the intended meaning of $\exists p.\, A$ (where p may occur in A) is that there is a meaning of p for which A holds. The formula $\exists p.\, A$ is equivalent to the formula $A[p := \top] \vee A[p := \bot]$.

This shows that every property that can be expressed in second-order propositional logic can also be expressed in first-order propositional logic (in the classical case). Indeed, for first-order propositional logic there is the result of *functional completeness* stating that every boolean function can be expressed by means of a first-order proposition. So it is not possible to increase the expressive power of propositional logic in this respect. In particular, we cannot express more when we can use quantification over propositional variables, as in second-order propositional logic.

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 $\Pi a{:}{*}.\, a \to a$. The intuitive meaning of this type is '$a \to a$ 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

$$\lambda x{:}\mathsf{nat}.\, x : \mathsf{nat}{\to}\mathsf{nat}$$

and the identity on booleans

$$\lambda x{:}\mathsf{bool}.\, x : \mathsf{bool} \to \mathsf{bool}$$

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:

$$\lambda x{:}a.\, x : a{\to}a.$$

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:

$$\lambda a{:}*\,.\,\lambda x{:}a.\, x : \Pi a{:}*\,.\, a{\to}a.$$

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 $\Pi a{:}\,*\,.\,a{\to}a$. This type can intuitively be seen as 'for all types a: $a{\to}a$'. 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 $\Pi a{:}\,*\,.\,a{\to}a$ 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:

$$\frac{\lambda a{:}\,*\,.\,\lambda x{:}a.\,x : \Pi a{:}\,*\,.\,a{\to}a \qquad \mathsf{nat} : *}{(\lambda a{:}\,*\,.\,\lambda x{:}a.\,x)\;\mathsf{nat} : \mathsf{nat}{\to}\mathsf{nat}}$$

and

$$\frac{\lambda a{:}\,*\,.\,\lambda x{:}a.\,x : \Pi a{:}\,*\,.\,a{\to}a \qquad \mathsf{bool} : *}{(\lambda a{:}\,*\,.\,\lambda x{:}a.\,x)\;\mathsf{bool} : \mathsf{bool}{\to}\mathsf{bool}}$$

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:

$$\mathsf{NLength} : \mathsf{natlist}{\to}\mathsf{nat}$$

and

$$\mathsf{BLength} : \mathsf{boollist}{\to}\mathsf{nat}.$$

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 ($\mathsf{polylist}\,a$) the type of finite lists of terms of type a, for every a. Then we can define the polymorphic length function

$$\mathsf{polylength} : \Pi a{:}\,*\,.\,\mathsf{polylist}\,a{\to}\mathsf{nat}$$

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

$$\frac{\mathsf{polylength} : \Pi a{:}\,*\,.\,(\mathsf{polylist}\,a){\to}\mathsf{nat} \qquad \mathsf{nat} : *}{\mathsf{polylength}\;\mathsf{nat} : \mathsf{polylist}\;\mathsf{nat}{\to}\mathsf{nat}}$$

and

$$\frac{\mathsf{polylength} : \Pi a{:}\,*\,.\,(\mathsf{polylist}\,a){\to}\mathsf{nat} \qquad \mathsf{bool} : *}{\mathsf{polylength}\;\mathsf{bool} : \mathsf{polylist}\;\mathsf{bool}{\to}\mathsf{nat}}$$

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 $x, y, z, \ldots$,
- a symbol $*$,
- a symbol $\Box$.

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:

$$\mathsf{P} ::= \mathsf{Var} \mid * \mid \Box \mid \Pi\mathsf{Var}{:}\mathsf{P}.\,\mathsf{P} \mid \lambda\mathsf{Var}{:}\mathsf{P}.\,\mathsf{P} \mid (\mathsf{P}\,\mathsf{P})$$

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.
- $\Box$ 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 $A \to B$ 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 $\Gamma, \Delta, \ldots$.

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. $\Box[x := P] = \Box$,
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. $(M\, N)[x := P] = (M[x := P])\, (N[x := P])$.

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 $\Box$.**

1. The *start rule.*

$$\frac{}{\vdash * : \Box}$$

2. The *variable rule.*

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

3. The *weakening rule.*

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

4. The *product rule.*

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

5. The *abstraction rule.*

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

6. The *application rule.*

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

7. The *conversion rule.*

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

Some remarks concerning the typing system:

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

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

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

  This rule is present neither in $\lambda\!\to$ nor in $\lambda 2$.
- The typical products that are present in $\lambda 2$ but not in $\lambda\!\to$ are the ones using the rule

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

  This rule is used to build for instance $\Pi a{:}*.\, a \to a$.

  This rule is present neither in $\lambda\!\to$ nor in λP.
- Instantiation of a polymorphic function is done by means of application. For instance:

  $$\frac{\Gamma \vdash \mathsf{polyid} : \Pi a{:}*.\, a \to a \qquad \Gamma \vdash \mathsf{nat} : *}{\Gamma \vdash (\mathsf{polyid}\,\mathsf{nat}) : \mathsf{nat} \to \mathsf{nat}}$$

- 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:

$$\begin{array}{lcl} \mathsf{new\_false} & = & \Pi a{:}{*}.\, a \\ (\mathsf{new\_and}\, A\, B) & = & \Pi c{:}{*}.\, (A \to B \to c) \to c \\ (\mathsf{new\_or}\, A\, B) & = & \Pi c{:}{*}.\, (A \to c) \to (A \to c) \to c \end{array}$$

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:

$$\frac{\Gamma \vdash P : \mathsf{new\_false} \qquad \Gamma \vdash A : *}{\Gamma \vdash (P\, A) : A}$$

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 $\Gamma \vdash P : (\mathsf{new\_and}\, A\, B)$ 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:

$$\frac{\dfrac{\Gamma \vdash P : \Pi a{:}{*}.\, (A \to B \to a) \to a \qquad \Gamma \vdash A : *}{\Gamma \vdash (P\, A) : (A \to B \to A) \to A} \qquad \Gamma \vdash L : A \to B \to A}{\Gamma \vdash (P\, A\, L) : A}$$

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;

$$\mathsf{N} \quad = \quad \Pi a{:}{*}.\, a \to (a \to a) \to a$$

The natural numbers are defined as follows:

$$\begin{array}{lcl} \mathsf{0} & = & \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, o \\ \mathsf{1} & = & \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, (s\, o) \\ \mathsf{2} & = & \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, (s\, (s\, o)) \\ \vdots & & \end{array}$$

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

$$\mathsf{n} \quad = \quad \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, s^n o$$

with the abbreviation ^{n} defined as:

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

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:

$$\mathsf{S} \quad = \quad \lambda n{:}\mathsf{N}.\, \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, s\, (n\, a\, o\, s)$$

We have for instance the following:

$$\begin{array}{ll} \mathsf{S}\, \mathsf{0} & = \\ (\lambda n{:}\mathsf{N}.\, \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, s\, (n\, a\, o\, s))\, \mathsf{0} & \to_\beta \\ \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, s\, (\mathsf{0}\, a\, o\, s) & = \\ \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, s\, ((\lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, o)\, a\, o\, s) & \to_\beta^* \\ \lambda a{:}{*}.\, \lambda o{:}a.\, \lambda s{:}a \to a.\, s\, o & = \\ \mathsf{1}. & \end{array}$$

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

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

$$\begin{array}{lcl} \mathsf{B} & = & \Pi a{:}{*}.\, a \to a \to a \\ \mathsf{T} & = & \lambda a{:}{*}.\, \lambda x{:}a.\, \lambda y{:}a.\, x \\ \mathsf{F} & = & \lambda a{:}{*}.\, \lambda x{:}a.\, \lambda y{:}a.\, y \end{array}$$

Note that we have

$$\begin{array}{lcl} \mathsf{T} & : & \mathsf{B} \\ \mathsf{F} & : & \mathsf{B} \end{array}$$

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

$$\mathsf{C} \;=\; \lambda a{:}{*}.\, \lambda b{:}\mathsf{B}.\, \lambda x{:}a.\, \lambda y{:}a.\, b\, a\, x\, y$$

with $\mathsf{C} : \Pi a{:}{*}.\, \mathsf{B} \to a \to a \to a$. The operator C is similar to the conditional 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:

$$\begin{array}{ll} \mathsf{C}\, A\, \mathsf{T}\, M\, N & = \\ (\lambda a{:}{*}.\, \lambda b{:}\mathsf{B}.\, \lambda x{:}a.\, \lambda y{:}a.\, b\, a\, x\, y)\, A\, \mathsf{T}\, M\, N & \to_\beta^* \\ \mathsf{T}\, A\, M\, N & = \\ (\lambda a{:}{*}.\, \lambda x{:}a.\, \lambda y{:}a.\, x)\, A\, M\, N & \to_\beta^* \\ M & \end{array}$$

and

$$\begin{array}{ll} \mathsf{C}\, A\, \mathsf{F}\, M\, N & = \\ (\lambda a{:}{*}.\, \lambda b{:}\mathsf{B}.\, \lambda x{:}a.\, \lambda y{:}a.\, b\, a\, x\, y)\, A\, \mathsf{F}\, M\, N & \to_\beta^* \\ \mathsf{F}\, A\, M\, N & = \\ (\lambda a{:}{*}.\, \lambda x{:}a.\, \lambda y{:}a.\, y)\, A\, M\, N & \to_\beta^* \\ N. & \end{array}$$

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

$$\begin{array}{lcl} \mathsf{not} & = & \lambda b{:}\mathsf{B}.\, \lambda a{:}*.\, \lambda x{:}a.\, \lambda y{:}a.\, b\, a\, y\, x \\ \mathsf{and} & = & \lambda b{:}\mathsf{B}.\, \lambda b'{:}\mathsf{B}.\, \lambda a{:}*.\, \lambda x{:}a.\, \lambda y{:}a.\, b\, a\, (b'\, a\, x\, y)\, y \\ \mathsf{or} & = & \lambda b{:}\mathsf{B}.\, \lambda b'{:}\mathsf{B}.\, \lambda a{:}*.\, \lambda x{:}a.\, \lambda y{:}a.\, b\, a\, x\, (b'\, a\, x\, y) \end{array}$$

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 $A{\to}B$ corresponds to a type $\Pi x{:}A.\, B$, also written as $A{\to}B$.
- 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 $\to$-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 $(\forall b.\, b) \to a$ in prop2:

$$\dfrac{\dfrac{[(\forall b.\, b)^x]}{a}\, E\forall}{(\forall b.\, b) \to a}\, I[x] \to$$

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

$$\lambda x : (\Pi b : \star.\, b).\, (x\, a)$$

Chapter 9

On inhabitation

This week we study a decision procedure for the inhabitation problem in simply typed λ-calculus. This problem corresponds to the provability problem in minimal logic via the Curry-Howard-De Bruijn isomorphism. Further we consider inductive predicates.

9.1 More lambda calculus

Shape of the β-normal forms. We recall from Chapter veranderen !! the definition of the set NF which is inductively defined by the following clauses:

1. if $x : A_1 \to \ldots \to A_n \to B$, and $M_1, \ldots, M_n \in \mathsf{NF}$ with $M_1 : A_1, \ldots, M_n : A_n$, then $x\, M_1 \ldots M_n \in \mathsf{NF}$,
2. if $M \in \mathsf{NF}$, then $\lambda x{:}A.\, M \in \mathsf{NF}$.

It can be shown that the set NF contains exactly all λ-terms in β-normal form.

Long Normal Forms. A *long normal form* or $\beta\overline{\eta}$-normal form is a term that is in β-normal form, and that moreover satisfies the property that every sub-term has the maximum number of arguments according to its type. Sub-terms are then said to be 'fully applied'. We give an inductive definition of the set of long normal forms.

The set LNF of long normal forms is inductively defined as follows:

1. if $x : A_1 \to \ldots \to A_n \to b$ with b a type variable, and $M_1, \ldots, M_n \in \mathsf{LNF}$ with $M_1 : A_1, \ldots, M_n : A_n$, then $x\, M_1 \ldots M_n \in \mathsf{LNF}$,
2. if $M \in \mathsf{LNF}$, then $\lambda x{:}A.\, M \in \mathsf{LNF}$.

Note that the difference between this definition and the previous one is in the first clause.

Let a, b, and c be type variables. Some examples:

- Let $x : a \to b$. Then $x \in \mathsf{NF}$ but $x \notin \mathsf{LNF}$.
- Let $x : a \to b$ and $y : a$. Then $x\,y \in \mathsf{NF}$ and $x\,y \in \mathsf{LNF}$.
- Let $x : a \to b \to c$ and let $y : a$. Then $x\,y \in \mathsf{NF}$ but $x\,y \notin \mathsf{LNF}$.
- Let $x : a \to b \to c$, let $y : a$, and let $z : b$. Then $\lambda x{:}a \to b \to c.\, x\,y\,z \in \mathsf{NF}$ and $\lambda x{:}a \to b \to c.\, x\,y\,z \in \mathsf{LNF}$.
- Let $x : (a \to b) \to c$ and $y : a \to b$. Then $x\,y \in \mathsf{NF}$ but $x\,y \notin \mathsf{LNF}$.
- Let $x : (a \to b) \to c$, and $y : a \to b$, and $z : a$. Then $x\,(\lambda z{:}a.\, y\,z) \in \mathsf{NF}$ and $x\,(\lambda z{:}a.\, y\,z) \in \mathsf{LNF}$.

We will make use of long normal forms in the decision procedure for the inhabitation problem.

Eta. So far we have considered λ-calculus with β-reduction, where the β-reduction rule is obtained by orienting the β-axiom $(\lambda x{:}A.\, M)\,N =_\beta M[x := N]$. from left to right.

Another important axiom in the λ-calculus, which we only add if explicitly mentioned, is the η-axiom. It is as follows:

$$\lambda x{:}A.\, (M\,x) =_\eta M$$

where x doesn't occur free in M. For the decision procedure of the inhabitation problem we make use of the η-axiom.

The *η-reduction* rule is obtained by orienting the η-axiom from left to right (the side condition that x doesn't occur free in M is then a side condition to the rule). The η-reduction relation is denoted by $\to_\eta$. The *η-expansion* rule is obtained by orienting the η-axiom from right to left. The η-expansion rule is subject to the side condition concerning the bound variable, and also to conditions that ensure that η-expansion doesn't create β-redexes.

We make use of the following two properties:

1. Types are preserved under η-reduction. That is, if $\Gamma \vdash M : A$ and $M \to_\eta M'$, then $\Gamma \vdash M' : A$. This property is called subject reduction for η-reduction.
2. If $\Gamma \vdash M : A$ for a term M in β-normal form, then there exists a term P in long normal form with $\Gamma \vdash P : A$ and $P \twoheadrightarrow_\eta M$.

Now it is easy to see that if a type is inhabited, then it is inhabited by a long normal form: Suppose that the type A is inhabited, that is $\vdash M : A$ for some M. Because of strong normalization for β-reduction M has a normal form. Because of confluence for β-reduction this normal form is unique. Let N be the β-normal form of M. Because of subject reduction for β-reduction, we have $\vdash N : A$. Because of the second property mentioned above, there exists a long normal form P with $\vdash P : A$ and $P \twoheadrightarrow_\eta N$.

9.2 Inhabitation

The provability question in logic corresponds via the Curry-Howard-De Bruijn isomorphism to the inhabitation question in λ-calculus. The two questions are stated as follows:

- The provability question.

  This is the question whether, given a formula A, there is a proof showing that A is a tautology.

  Sometimes this problem is considered in the situation where we may assume some formulas as hypotheses.

- The type inhabitation problem.

  This is the question whether, given a type A, there is a closed inhabitant of A. Sometimes this is abbreviated as $\vdash ? : A$.

  Moreover, also this problem may be considered in a certain given environment where additional type declarations are given.

Provability in minimal logic is decidable. (Recall from for instance the course *introduction to logic* that classical propositional logic is decidable.) Here we consider the inhabitation problem and explain that it is decidable.

The Decision Procedure. We informally describe a procedure to decide whether a type A is inhabited by a closed lambda term. Observe that every simple type A is of the form $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow b$ with b a type variable, for some $n \geq 0$. We consider the following question:

? Does a term M exist such that $M : A$ and such that all free variables of M are among $x_1 : A_1, \ldots, x_n : A_n$?

We distinguish two cases: the case that A is a type variable, so of the form a, and the case that A is an arrow type, so of the form $D \rightarrow D'$ for some types D and D'.

1. Suppose that $A = a$, so a type variable.

   If we have that $A_i = a$ for some $i \in \{1, \ldots, n\}$, then we take $M = x_i$ and the procedure succeeds.

   If we have that $A_i = C_1 \rightarrow \ldots \rightarrow C_k \rightarrow a$ for some $i \in \{1, \ldots, n\}$, then we obtain k new questions:

   ? Does a term M_1 exist such that $M_1 : C_1$ and such that all free variables of M_1 are among $x_1 : A_1, \ldots, x_n : A_n$?

   $\vdots$

   ? Does a term M_k exist such that $M_k : C_k$ and such that all free variables of M_k are among $x_1 : A_1, \ldots, x_n : A_n$?

Otherwise, if there is no A_i such that $A_i = C_1 \to \ldots \to C_k \to a$, then the procedure fails.

2. Suppose that $A = D \to D'$, so A is an arrow type. Since we work with long normal forms, we have that M is an abstraction, so of the form $\lambda y{:}D.\, M'$ with $M' : D'$. We then answer the following new question:

   ? Does a term M' exist such that $M' : D'$ and such that all free variables of M' are among $x_1 : A_1, \ldots, x_n : A_n, y : D$?

The procedure can be slightly optimized by identifying variables that have the same type.