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 $\bot$, 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 $\bot$ 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:

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

   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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

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

   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:

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

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:

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

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:

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

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 $\mathsf{EM} \to \mathsf{PL}$ (Lemma one), $\mathsf{PL} \to \mathsf{DN}$ (Lemma two), $\mathsf{DN} \to \mathsf{EM}$ (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 $(A \to \bot) \to \bot$.

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

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:

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

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 $\rightarrow$. 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 $A \rightarrow B$, 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 $A \rightarrow B$ 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 $\rightarrow$, then we look in the list of assumptions whether there is an assumption of the form $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$, 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 $A \rightarrow A$ 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`

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

Goal: `C->B`

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

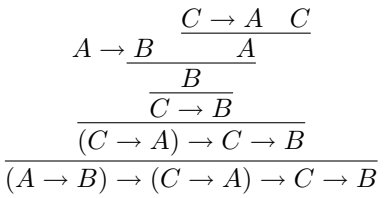
$$\frac{\dfrac{\dfrac{B}{C \to B}}{(C \to A) \to C \to B}}{(A \to B) \to (C \to A) \to C \to B}$$

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

Goal: `C`

`exact z.`

done! *Proof completed.*

Notation.

- $\bot$ 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 $A_1 \to \ldots \to A_n \to B$ 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 $A_1 \to \ldots \to A_n \to B$, 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 $\rightarrow$ is assumed to be associative to the right.

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

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

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 $A{\rightarrow}B$. For instance, the identity function on natural numbers is written as $\lambda x{:}\mathsf{nat}.\, x$.

If F is a function of type $A{\rightarrow}B$, 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 $A \rightarrow B$.

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

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 $\Lambda^{\to}_{A}$, 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.

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

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 $\to_\beta^*$ for the reflexive-transitive closure of $\to_\beta$. 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 $x : A_1 \to \ldots \to A_n \to B$, 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 $\lambda x{:}A.\, M \in \mathsf{NF}$.

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 $M \to_\beta^* N$ and $M \to_\beta^* N'$, there exists a term P such that $N \to_\beta^* P$ and $N' \to_\beta^* P$. 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: $\to$ 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

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

Goal: `A`

`apply y.`
Assumptions do not change

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

Goal: `C`

`exact z.`

done!

Subtree proved!

The corresponding type derivation is as follows:

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

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:

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

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

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

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:

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

  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

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

  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

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

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

   $$\begin{aligned} f(0, n_1, \ldots, n_k) &= g(n_1, \ldots, n_k) \\ f(n+1, n_1, \ldots, n_k) &= h(f(n, n_1, \ldots, n_k), n, n_1, \ldots, n_k) \end{aligned}$$

   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:

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

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:

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

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

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

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

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

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 `0`, *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_O : forall m : nat, leq O 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:

- $\to$ for implication,
- $\wedge$ for conjunction,
- $\vee$ for disjunction,
- $\bot$ 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 $\rightarrow$, $\wedge$, 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 $\forall x.\, A \rightarrow B$ instead of $\forall x.\, (A \rightarrow B)$. 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. $(A_1 \rightarrow A_2)[x := N] = (A_1[x := N]) \rightarrow (A_2[x := N])$,
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: $\forall$ and $\exists$. 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*:

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

3. The *implication elimination rule*:

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

4. The *conjunction introduction rule*:

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

5. The *conjunction elimination rules*:

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

6. The *disjunction introduction rules*:

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

7. The *disjunction elimination rule*:

$$\frac{A \vee B \qquad \begin{array}{c}\vdots\\ A \to C\end{array} \qquad \begin{array}{c}\vdots\\ B \to C\end{array}}{C}$$

8. The *falsum rule*:

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

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.

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

2.

$$\frac{\dfrac{[A^w]}{\forall x.A}\ I\forall}{A \to \forall x.A}\ I[w]\to$$

3.

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

4.

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

5.

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

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.

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

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 $\bot$.)
- 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 $\rightarrow$ and $\forall$, there are two kinds of detours:

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

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

- 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:

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

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

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.

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

2.

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

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 $\bot$ 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 $\forall x.\, A \rightarrow B$ is $\forall x.\, (A \rightarrow B)$, 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

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

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:

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

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 → natlist → 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 → 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* 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 natlist live in Set. The type of natlist_dep is then: nat $\rightarrow$ Set. Now the desired typings for the first example above are:

- zeroes 0 : natlist_dep 0,
- zeroes 1 : natlist_dep 1,
- zeroes 2 : natlist_dep 2
- ...

A first observation is that in this way types and terms are no longer separate entities. For instance natlist_dep 0 is the type of 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 zeroes 0 : 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 nat : Set in a formal system. (This formal system is given later on in the course notes.) We have for instance

$$\begin{array}{lcllcl}
0 & : & \mathsf{nat} & : & \mathsf{Set} & : \quad \mathsf{Type} \\
\mathsf{S} & : & \mathsf{nat} \rightarrow \mathsf{nat} & : & \mathsf{Set} & : \quad \mathsf{Type} \\
\mathsf{nil} & : & \mathsf{natlist} & : & \mathsf{Set} & : \quad \mathsf{Type} \\
\mathsf{cons} & : & \mathsf{nat} \rightarrow \mathsf{natlist} \rightarrow \mathsf{natlist} & : & \mathsf{Set} & : \quad \mathsf{Type}
\end{array}$$

And also

$$\begin{array}{rcllcl}
 & & \mathsf{natlist_dep} & : & \mathsf{nat} \rightarrow \mathsf{Set} & : \quad \mathsf{Type} \\
\mathsf{nil} & : & \mathsf{natlist_dep}\,0 & : & \mathsf{Set} & : \quad \mathsf{Type}
\end{array}$$

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:

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

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:

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

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:

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

and

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

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:

$$\begin{array}{lcl}
\mathsf{zeroes} & : & \Pi n{:}\mathsf{nat}.\ \mathsf{natlist_dep}\ n \\
\mathsf{append} & : & \Pi n, m{:}\mathsf{nat}.\ \mathsf{natlist_dep}\ n \to \mathsf{natlist_dep}\ m \to \mathsf{natlist_dep}\,(\mathsf{plus}\,n\,m) \\
\mathsf{reverse} & : & \Pi n{:}\mathsf{nat}.\ \mathsf{natlist_dep}\ n \to \mathsf{natlist_dep}\ n
\end{array}$$

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:

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

- A similar application of the variable rule:

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

- 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:

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

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

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

So here we derive in the formal system that $\mathsf{nat} \to \mathsf{nat}$ is in $*$ (intuition: is a data-type) whereas in the previous presentation of simply typed λ-calculus the statement '$\mathsf{nat} \to \mathsf{nat}$ 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:

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

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

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

This derivation shows that if nat is declared as a data-type, so $\mathsf{nat} : *$, then $\mathsf{nat} \to *$ lives in $\Box$, so $\mathsf{nat} \mathit{rightarrow} *$ 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 $\mathsf{nat} : * \vdash \lambda x{:}\mathsf{nat}.\, x : \Pi x{:}\mathsf{nat}.\, \mathsf{nat}$:

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

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 $B : \Box$. Then we have only types like $\mathsf{nat} \to \mathsf{nat}$ or $A \to B$ (with A and B in $*$), or $\mathsf{nat} \to B$. Then we don't have $\mathsf{nat} \to *$, so there is no type for $\mathsf{natlist\_dep}$.

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:

$$\begin{array}{lcl} R(M_1,\ldots,M_n) & \sim & R\,M_1^*\,\ldots M_n^* \\ A{\to}B & \sim & \Pi x{:}A^*.\,B^* \\ \forall x.\,A & \sim & \Pi x{:}\mathsf{Terms}.\,A^* \end{array}$$

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
$\to$ introduction	$\sim$	abstraction
$\forall$ introduction	$\sim$	abstraction
$\to$ elimination	$\sim$	application
$\forall$ elimination	$\sim$	application

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

1. The formula $(\forall x.\,P(x)) \to P(N)$ in predicate logic corresponds to the λP-term $(\Pi x : \mathsf{Terms}.\,P\,x) \to (P\,N)$. An inhabitant of this term is:

$$\lambda w : (\Pi x : \mathsf{Terms}.\,P\,x).\,(w\,N)$$

2. The formula $A \to \forall x.A$ in predicate logic corresponds to the λP-term $A \to \Pi x : \mathsf{Terms}.\,A$. An inhabitant of this term is:

$$\lambda w : A.\,\lambda x : \mathsf{Terms}.\,w$$

3. The formula $(\forall x.P(x) \to Q(x)) \to (\forall x.P(x)) \to \forall y.Q(y)$ in predicate logic corresponds to the λP-term $(\Pi x : \mathsf{Terms}.\,P\,x \to Q\,x) \to (\Pi x : \mathsf{Terms}.\,P\,x) \to (\Pi y : \mathsf{Terms}.\,Q\,y)$. An inhabitant of this term is:

$$\lambda w : (\Pi x : \mathsf{Terms}.\,P\,x \to Q\,x).\,\lambda v : (\Pi x : \mathsf{Terms}.\,P\,x).\,\lambda y : \mathsf{Terms}.\,w\,y\,(v\,y)$$

4. The formula $(A \to \forall x.P(x)) \to \forall y.A \to P(y)$ in predicate logic corresponds to the λP-term $(A \to (\Pi x : \mathsf{Terms}.\,P\,x)) \to \Pi y : \mathsf{Terms}.\,(A \to P\,y)$. An inhabitant of this term is:

$$\lambda v : (A \to \Pi x : \mathsf{Terms}.\,P\,x).\,\lambda y : \mathsf{Terms}.\,\lambda w : A.\,v\,w\,y$$

5. The formula $\forall x.\,(P(x) \to (\forall y.\,P(y) \to A) \to A)$ in predicate logic corresponds to the λP-term $\Pi x : \mathsf{Terms}.\,(P\,x \to (\Pi y : \mathsf{Terms}.\,P\,y \to A) \to A)$. An inhabitant of this term is:

$$\lambda x : \mathsf{Terms}.\,\lambda u : P\,x.\,\lambda v : (\Pi y : P\,x \to A).\,v\,x\,u$$

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 $\bot$ 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*:

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

3. The *implication elimination rule*:

   $$\frac{\begin{array}{c}\vdots\\ A \to B\end{array} \qquad \begin{array}{c}\vdots\\ A\end{array}}{B}\ E \to$$

4. The *conjunction introduction rule*:

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

5. The *conjunction elimination rules*:

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

6. The *disjunction introduction rules*:

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

7. The *disjunction elimination rule*:

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

8. The *falsum rule*:

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

9. The *universal quantification introduction rule.*

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

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.

$$\frac{\dfrac{[a^x]}{a \to a}\ I[x]\to}{\forall a.\, a \to a}\ I\forall$$

2.

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

3. The following proof is not correct:

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

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 $\bot$.)
- 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:

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

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

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):

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

$$A \to A$$

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

And another example:

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

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 $\mathsf{natlist}$ of finite lists of natural numbers, and a type $\mathsf{boollist}$ 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 $\mathsf{polylist}$ 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 $\lambda{\to}$ (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

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

  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

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

  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 $M \to_\beta M'$, 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:

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

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:

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

with $\mathsf{C} : \Pi a{:}*.\, \mathsf{B} \to a \to a \to a$. The operator C is similar to the conditional c of system **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 $\rightarrow$-detour corresponds to a β-redex in $\lambda 2$.
- A $\forall$-detour corresponds to a β-redex in $\lambda 2$.
- A proof normalization step in second-order propositional logic corresponds to a β-reduction step in $\lambda 2$.

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

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

The type $(\Pi b{:}*\,.\, b) \rightarrow a$ corresponds to the formula $(\forall b.\, b) \rightarrow a$. We assume $a : \star$. Then an inhabitant of $(\Pi b{:}*\,.\, b) \rightarrow a$ 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 subterm 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 $A_1 \to \ldots \to A_n \to b$ 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 $D \to D'$ 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 $A_i = C_1 \to \ldots \to C_k \to a$ 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.

Introduction to Type Theory

Herman Geuvers

Radboud University Nijmegen, The Netherlands
Technical University Eindhoven, The Netherlands

1 Overview

These notes comprise the lecture "Introduction to Type Theory" that I gave at the Alpha Lernet Summer School in Piriapolis, Uruguay in February 2008. The lecture was meant as an *introduction* to typed λ-calculus for PhD. students that have some (but possibly not much) familiarity with logic or functional programming. The lecture consisted of 5 hours of lecturing, using a beamer presentation, the slides of which can be found at my homepage[1]. I also handed out exercises, which are now integrated into these lecture notes.

In the lecture, I attempted to give an introductory overview of type theory. The problem is: there are so many type systems and so many ways of defining them. Type systems are used in programming (languages) for various purposes: to be able to find simple mistakes (e.g. caused by typing mismatches) at compile time; to generate information about data to be used at runtime, ... But type systems are also used in theorem proving, in studying the the foundations of mathematics, in proof theory and in language theory.

In the lecture I have focussed on the use of type theory for compile-time checking of functional programs and on the use of types in proof assistants (theorem provers). The latter combines the use of types in the foundations of mathematics and proof theory. These topics may seem remote, but as a matter of fact they are not, because they join in the central theme of these lectures:

Curry-Howard isomorphism of formulas-as-types
(and proofs-as-terms)

This isomorphism amounts to two readings of typing judgments

$$M : A$$

- M is a term (program, expression) of the data type A
- M is a proof (derivation) of the formula A

The first reading is very much a "programmers" view and the second a "proof theory" view. They join in the implementation of proof assistants using type systems, where a term (proof) of a type (formula) is sought for interactively between the user and the system, and where terms (programs) of a type can also be used to define and compute with functions (as algorithms).

[1] `url: http:://www.cs.ru.nl/H.Geuvers/Uruguay2008SummerSchool.html/`

For an extensive introduction into the Curry-Howard isomorphism, we refer to [39].

The contents of these notes is as follows.

1. Introduction: what are types and why are they not sets?
2. Simply typed λ-calculus (Simple Type Theory) and the Curry Howard isomorphism
3. Simple Type Theory: "Curry" type assignment, principle type algorithm and normalization
4. Polymorphic type theory: full polymorphism and ML style polymorphism
5. Dependent type theory: logical framework and type checking algorithm

In the course, I have also (briefly) treated higher order logic, the λ-cube, Pure Type Systems and inductive types, but I will not do that here. This is partly because of space restrictions, but mainly because these notes should be of a very introductory nature, so I have chosen to treat lesser things in more detail.

2 Introduction

2.1 Types and sets

Types are not sets. Types are a bit like sets, but types give syntactic information, e.g.

$$3 + (7 * 8)^5 : \mathsf{nat}$$

whereas sets give semantic information, e.g.

$$3 \in \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}$$

Of course, the distinction between syntactical and semantical information can't always be drawn that clearly, but the example should be clear: $3 + (7 * 8)^5$ is of type nat simply because 3, 7 and 8 are natural numbers and $*$ and $+$ are operations on natural numbers. On the other hand, $3 \in \{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ (x^n + y^n \neq z^n)\}$, because there are no positive x, y, z such that $x^n + y^n = z^n$. This is an instance of 'Fermat's last Theorem', proved by Wiles. To establish that 3 is an element of that set, we need a *proof*, we can't just read it off from the components of the statement. To establish that $3 + (7 * 8)^5 : \mathsf{nat}$ we don't need a proof but a *computation*: our "reading the type of the term" is done by a simple computation.

One can argue about what can be "just read off", and what not; a simple criterion may be whether there is an algorithm that establishes the fact. So then we draw the line between "is of type" $(:)$ and "is an element of" $(\in)$ as to whether the relation is decidable or not. A further refinement may be given by arguing that a type checking algorithm should be of low complexity (or compositional or syntax directed).

There are very many different type theories and also mathematicians who base their work on set theory use types as a *high level ordering mechanism*, usually in an informal way. As an example consider the notion of *monoid*, which is

defined as a tuple $\langle A, \cdot, e\rangle$, where A is a set, $\cdot$ a binary operation on A and e an element of A, satisfying the monoidic laws. In set theory, such an ordered pair $\langle a, b\rangle$ is typically defined as $\{\{a\}, \{a, b\}\}$, and with that we can define ordered triples, but one usually doesn't get into those details, as they are irrelevant *representation issues*: the only thing that is relevant for an ordered pair is that one has *pairing* and *projection* operators, to create a pair and to take it apart. This is exactly how an ordered pair would be defined in type theory: if A and B are types, then $A \times B$ is a type; if $a : A$ and $b : B$, then $\langle a, b\rangle : A \times B$; if $p : A \times B$, then $\pi_1\, p : A$, $\pi_2\, t : B$ and moreover $\pi_1\langle a, b\rangle = a$ and $\pi_2\langle a, b\rangle = b$. So mathematicians use a kind of high level typed language, to avoid irrelevant representation issues, even if they may use set theory as their foundation. However, this high level language plays a more important role than just a language, as can be seen from the problems that mathematicians study: whether $\sqrt{2}$ is an element of the set π is not considered a relevant question. A mathematician would probably not even considered this as a meaningful question, because *the types don't match*: π isn't a set but a number. (But in set theory, everything is a set.) Whether $\sqrt{2} \in \pi$ depends on the actual representation of the real numbers as sets, which is quite arbitrary, so the question is considered irrelevant.

We now list a number of issues and set side by side how set theory and type theory deal with them.

Collections Sets are "collections of things", where the things themselves are again sets. There are all kinds of ways for putting things together in a set: basically (ignoring some obvious consistency conditions here) one can just put all the elements that satisfy a property together in a set. Types are collections of objects of the same intrinsic nature or the same structure. There are specific ways of forming new types out of existing ones.

Existence Set theory talks about what things *exist*. The infinity axiom states that an infinite set exists and the power set axiom states that the set of subsets of a set exists. This gives set theory a clear *foundational* aspect, apart from its informal use. It also raises issues whether a "choice set" exists (as stated by the axiom of choice) and whether *inaccessible cardinals* exist. (A set X such that for all sets Y with $|Y| < |X|$, $|2^Y| < |X|$.) Type theory talks about how things can be *constructed* (syntax, expressions). Type theory defines a formal *language*. This puts type theory somewhere in between the research fields of software technology and proof theory, but there is more: being a system describing what things can be *constructed*, type theory also has something to say about the *foundations of mathematics*, as it also – just like set theory – describes what exists (can be constructed) and what not.

Extensionality versus intensionality Sets are extensional: Two sets are equal if they contain the same elements. For example $\{n \in \mathbb{N} \mid \exists x, y, z \in \mathbb{N}^+(x^n + y^n = z^n)\} = \{0, 1, 2\}$. So set equality is undecidable. In general it requires a proof to

establish the equality of two sets. Types are intensional[2]. Two types are equal if they have the same *representation*, something that can be verified by simple syntactic considerations. So, $\{n \mid \exists x, y, z : \mathsf{nat}^+(x^n + y^n \neq z^n)\} \neq \{n \mid n = 0 \vee n = 1 \vee n = 2\}$ because these two types don't have the same representation. Of course, one may wonder what types these are exactly, or put differently, what an object of such a type is. We'll come to that below.

Decidability of $:$*, undecidability of* $\in$ Membership is undecidable in set theory, as it requires a proof to establish $a \in A$. Typing (and type checking) is decidable[3]. Verifying whether M is of type A requires purely syntactic methods, which can be cast into a *typing algorithm*. As indicated before, types are about syntax: $3 + (7 * 8)^5 : \mathsf{nat}$, because 3, 7, 8 are of type nat and the operations take objects of type nat to nat. Similarly, $\frac{1}{2}\Sigma_{n=0}^{\infty} 2^{-n} : \mathbb{N}$ is not a typing judgment, because one needs additional information to know that the sum is divisible by 2.

The distinction between syntax and semantics is not always as sharp as it seems. The more we know about semantics (a model), the more we can formalize it and "turn it into syntax". For example, we can turn

$$\{n \in \mathbb{N} \mid \exists x, y, z \in \mathbb{N}^+(x^n + y^n = z^n)\}$$

into a (syntactic) type , with decidable type checking , if we take as its terms pairs

$$\langle n, p\rangle : \{n : \mathsf{nat} \mid \exists x, y, z : \mathsf{nat}^+(x^n + y^n = z^n)\}$$

where p is a proof of $\exists x, y, z \in \mathsf{nat}^+(x^n + y^n = z^n)$. If we have decidable proof checking, then it is decidable whether a given pair $\langle n, p\rangle$ is typable with the above type or not.

In these notes, we will study the formulas-as-types and proof-as-terms embedding, which gives syntactic representation of proofs that can be *type checked* to see whether they are correct and what formula (their type) they prove. So with such a representation, proof checking is certainly decidable. We can therefore summarize the difference between set theory and type theory as the difference between *proof checking* (required to check a typing judgment), which is decidable and *proof finding* (which is required to check an element-of judgment) which is not decidable.

2.2 A hierarchy of type theories

In this paper we describe a numbers of type theories. These could be described in one framework of the λ-cube or Pure Type Systems, but we will not do that here. For the general framework we refer to [5, 4]. This paper should be seen

[2] But the first version of Martin-Löf's type theory is extensional – and hence has undecidable type checking. This type theory is the basis of the proof assistant Nuprl[10].

[3] But there are type systems with undecidable type checking, for example the Curry variant of system F (see Section 5.2). And there are more exceptions to this rule.

(and used) as a very introductory paper, that can be used as study material for researchers interested in type theory, but relatively new to the field.

Historically, untyped λ-calculus was studied in much more depth and detail (see [3]) before the whole proliferation of types and related research took off. Therefore, in overview papers, one still tends to first introduce untyped λ-calculus and then the typed variant. However, if one knows nothing about either subjects, typed λ-calculus is more natural then the untyped system, which – at first sight – may seem like a pointless token game with unclear semantics. So, we start off from the simply typed λ- calculus.

The following diagrams give an overview of the lectures I have given at the Alfa Lernet Summer School. The first diagram describes simple type theory and polymorphic type theory, which comes in two flavors: à la Church and à la Curry. Apart from that, I have treated a weakened version of $\lambda 2$, corresponding to polymorphism in functional languages like ML. This will also be discussed in this paper. A main line of thought is the formulas-as-types embedding, so the corresponding logics are indicated in the left column.

The second diagram deals with the extension with dependent types. In the lectures I have treated all systems in the diagram, but in these notes, only the first row will be discussed: first order dependent type theory $\lambda\mathsf{P}$ and two ways of interpreting logic into it: a *direct encoding* of minimal predicate logic and a *logical framework* encoding of many different logics. The first follows the Curry-Howard version of the formulas-as-types embedding, which we also follow for $\lambda\!\!\rightarrow$ and $\lambda 2$. The second follows De Bruijn's version of the formulas-as-types embedding, where we encode a logic in a context using dependent types. The rest of the diagram is not treated here, but it is treated on the slides[4].

Logic		TT a la Church	Also known as	TT a la Curry
PROP	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda\!\!\rightarrow$	STT	$\lambda\!\!\rightarrow$
PROP2	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda 2$	system F	$\lambda 2$

					Remarks
PRED	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda\mathsf{P}$	LF	$\overset{\text{f-as-t}}{\longleftarrow}$	Many logics
HOL	$\overset{\text{f-as-t}}{\longrightarrow}$	$\lambda\mathsf{HOL}$			language of HOL is STT
HOL	$\overset{\text{f-as-t}}{\longrightarrow}$	CC	Calc. of Constr.		
		PTS			different PTSs for HOL

3 Simple type theory $\boldsymbol{\lambda\!\!\rightarrow}$

In our presentation of the simple type theory, we have just arrow types. This is the same as the original system of [9], except for the fact that we allow type variables, whereas Church starts form two base types ι and o. A very natural

[4] `url: http:://www.cs.ru.nl/H.Geuvers/Uruguay2008SummerSchool.html/`

extension is the one with product types and possibly other type constructions (like sum types, a unit type, ...). A good reference for the simple type theory extended with product types is [27].

Definition 1. *The types of* $\lambda\!\!\rightarrow$ *are*

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ}\!\rightarrow\!\mathsf{Typ})$$

where TVar *denotes the countable set of type variables.*

Convention 2
- *Type variables will be denoted by* $\alpha, \beta, \gamma, \ldots$. *Types will be denoted by* $\sigma, \tau, \ldots$.
- *In types we let* brackets associate to the right *and we* omit outside brackets*:* $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!(\beta\!\rightarrow\!\gamma)\!\rightarrow\!\alpha\!\rightarrow\!\gamma$ *denotes* $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!((\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma))$

Example 1. The following are types: $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!\alpha$, $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!((\beta\!\rightarrow\!\gamma)\!\rightarrow\!(\alpha\!\rightarrow\!\gamma))$. Note the higher order structure of types: we read $(\alpha\!\rightarrow\!\beta)\!\rightarrow\!\alpha$ as the *type of functions that take functions from* α *to* β *to values of type* α.

Definition 3. *The* terms *of* $\lambda\!\!\rightarrow$ *are defined as follows*

- *There are countably many typed variables* $x_1^\sigma, x_2^\sigma, \ldots$, *for every* σ.
- *Application: if* $M : \sigma\!\rightarrow\!\tau$ *and* $N : \sigma$, *then* $(M\,N) : \tau$
- *Abstraction: if* $P : \tau$, *then* $(\lambda x^\sigma.P) : \sigma\!\rightarrow\!\tau$

So the binary application operation is not written. One could write $M \cdot N$, but that is not done in λ-calculus. The λ is meant to *bind* the variable in the *body*: in $\lambda x^\sigma.M$, x^σ is bound in M. We come to that later.

The idea is that $\lambda x^\sigma.M$ is the function $x \mapsto M$ that takes an input argument P and produces the output $M[x := P]$, M *with* P *substituted for* x. This will be made precise by the *β-reduction rule*, which is the computation rule to deal with λ-terms. We come to this in Definition 6.

Convention 4
- *Term variables will be denoted by* $x, y, z, \ldots$. *Terms will be denoted by* $M, N, P, \ldots$.
- *Type annotations on variables will only be written at the λ-abstraction: we write* $\lambda x^\sigma.x$ *instead of* $\lambda x^\sigma.x^\sigma$.
- *In term applications we let* brackets associate to the left *and we* omit outside brackets *and brackets around iterations of abstractions:* $M\,N\,P$ *denotes* $((M\,N)\,P)$ *and* $\lambda x^{\alpha\rightarrow\beta}.\lambda y^{\beta\rightarrow\gamma}.\lambda z^\alpha.xz(yz)$ *denotes* $(\lambda x^{\alpha\rightarrow\beta}.(\lambda y^{\beta\rightarrow\gamma}.(\lambda z^\alpha.((xz)(yz)))))$

Examples 2. For every type σ we have the term $\mathbf{I}_\sigma := \lambda x^\sigma.x$ which is of type $\sigma\!\rightarrow\!\sigma$. This is the *identity combinator* on σ.
For types σ and τ we have the term $\mathbf{K}_{\sigma\tau} := \lambda x^\sigma.\lambda y^\tau.x$ of type $\sigma\!\rightarrow\!\tau\!\rightarrow\!\sigma$. This term, called the K *combinator* takes two inputs and returns the first.

Here are some more interesting examples of typable terms:
$\lambda x^{\alpha\rightarrow\beta}.\lambda y^{\beta\rightarrow\gamma}.\lambda z^\alpha.y(xz) : (\alpha\!\rightarrow\!\beta)\!\rightarrow\!(\beta\!\rightarrow\!\gamma)\!\rightarrow\!\alpha\!\rightarrow\!\gamma$,
$\lambda x^\alpha.\lambda y^{(\beta\rightarrow\alpha)\rightarrow\alpha}.y(\lambda z^\beta.x) : \alpha\!\rightarrow\!((\beta\!\rightarrow\!\alpha)\!\rightarrow\!\alpha)\!\rightarrow\!\alpha$.

To show that a term is of a certain type, we have to "build it up" using the inductive definition of terms (Definition 3). For $\lambda x^{\alpha\to\beta}.\lambda y^{\beta\to\gamma}.\lambda z^{\alpha}.y(xz)$, we find the type as follows:

- If $x : \alpha\to\beta$, $y : \beta\to\gamma$ and $z : \alpha$, then $xz : \beta$,
- so $y(xz) : \gamma$,
- so $\lambda z^{\alpha}.y(xz) : \alpha\to\gamma$,
- so $\lambda y^{\beta\to\gamma}.\lambda z^{\alpha}.y(xz) : (\beta\to\gamma)\to\alpha\to\gamma$,
- so $\lambda x^{\alpha\to\beta}.\lambda y^{\beta\to\gamma}.\lambda z^{\alpha}.y(xz) : (\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma$

In λ-calculus (and type theory) we often take a number of λ-abstractions together, writing $\lambda x^{\sigma}y^{\tau}.x$ for $\lambda x^{\sigma}.\lambda y^{\tau}.x$. The conventions about types and applications fit together nicely. If $F : \sigma\to\tau\to\rho$, $M : \sigma$ and $P : \tau$, then

$$F\,M : \tau\to\rho \quad \text{and} \quad F\,M\,P : \rho$$

Given the bracket convention for types, every type of $\lambda\to$ can be written as

$$\sigma_1\to\sigma_2\to\ldots\to\alpha$$

with α a type variable.

The lack of product types is largely circumvented by dealing with functions of multiple arguments by *Currying*: We don't have $F : \sigma\times\tau\to\rho$ but instead we can use $F : \sigma\to\tau\to\rho$, because the latter F is a function that takes two arguments, of types σ and τ, and produces a term of type ρ.

3.1 Computation, free and bound variables, substitution

A λ-term of the form $(\lambda x^{\sigma}.M)P$ is a β- redex (*red*ucible *ex*pression). A redex can be *contracted*:

$$(\lambda x^{\sigma}.M)P \longrightarrow_{\beta} M[x := P]$$

where $M[x := P]$ denotes M with P substituted for x.

As an example, we have $(\lambda x^{\sigma}.\lambda y^{\tau}.x)P \longrightarrow_{\beta} \lambda y^{\tau}.P$. But what if $P = y$? then $(\lambda x^{\sigma}.\lambda y^{\tau}.x)y \longrightarrow_{\beta} \lambda y^{\tau}.y$, which is clearly not what we want, because the *free* y has become *bound* after reduction. The λ is a binder and we have to make sure that free variables don't get bound by a substitution. The solution is to *rename* bound variables before substitution.

Definition 5. *We define the notions of free and bound variables of a term, FV and BV.*

$$\begin{array}{rlrl} FV(x) &= \{x\} & BV(x) &= \emptyset \\ FV(MN) &= FV(M)\cup FV(N) & BV(MN) &= BV(M)\cup BV(N) \\ FV(\lambda x^{\sigma}.M) &= FV(M)\setminus\{x\} & BV(\lambda x^{\sigma}.M) &= BV(M)\cup\{x\} \end{array}$$

$M \equiv N$ *or* $M =_{\alpha} N$ *if* M *is equal to* N *modulo* renaming of bound variables. *A* closed *term is a term without free variables; closed terms are sometimes also called* combinators.

The renaming of bound variable x is done by taking a "fresh" variable (i.e. one that does not yet occur in the term, either free or bound), say y and replace all bound occurrences of x by y and λx by λy.

Examples 3.
- $\lambda x^\sigma.\lambda y^\tau.x \equiv \lambda x^\sigma.\lambda z^\tau.x$
- $\lambda x^\sigma.\lambda y^\tau.x \equiv \lambda y^\sigma.\lambda x^\tau.y$. This equality can be obtained by first renaming y to z, then x to y and then z to y.
- NB we also have $\lambda x^\sigma.\lambda y^\tau.y \equiv \lambda x^\sigma.\lambda x^\tau.x$. This equality is obtained by renaming the second x to y in the second term.

In the last example, we observe that our description of renaming above is slightly too informal. It is not symmetric, as we cannot rename y in the first term to x, and we may at some point not wish to rename with a completely fresh variable, but just with one that is not "in scope". We leave it at this and will not give a completely formal definition, as we think that the reader will be capable of performing α-conversion in the proper way. Fully spelled out definitions can be found in [11, 24, 3].

The general idea of (typed) λ-calculus is that we don't distinguish between terms that are α convertible: we consider terms *modulo α-equality* and we don't distinguish between $\lambda x^\sigma.\lambda y^\tau.x$ and $\lambda x^\sigma.\lambda z^\tau.x$. This implies that all our operations and predicates should be defined on α-equivalence classes, a property that we don't verify for every operation we define, but that we should be aware of.

When reasoning about λ-terms we use concrete terms (and not α-equivalence classes). We will avoid terms like $\lambda x^\sigma.\lambda x^\tau.x$, because they can be confusing. In examples we always rename bound variables such that no clashes can arise.
This is known as the *Barendregt convention*: when talking about a set of λ-terms, we may always assume that all free variables are different from the bound ones and that all bound variables are distinct.

Before reduction or substitution, we rename (if necessary):

$$(\lambda x^\sigma.\lambda y^\tau.x)y \equiv (\lambda x^\sigma.\lambda z^\tau.x)y \longrightarrow_\beta \lambda z^\tau.y$$

Definition 6. *The notions of one-step β-reduction, $\longrightarrow_\beta$, multiple-step β-reduction, $\longrightarrow\!\!\!\!\rightarrow_\beta$, and β-equality, $=_\beta$ are defined as follows.*

$$\begin{aligned}
(\lambda x^\sigma.M)N &\longrightarrow_\beta M[x := N] \\
M \longrightarrow_\beta N &\Rightarrow M\,P \longrightarrow_\beta N\,P \\
M \longrightarrow_\beta N &\Rightarrow P\,M \longrightarrow_\beta P\,N \\
M \longrightarrow_\beta N &\Rightarrow \lambda x^\sigma.M \longrightarrow_\beta \lambda x^\sigma.N
\end{aligned}$$

$\longrightarrow\!\!\!\!\rightarrow_\beta$ is the transitive reflexive closure of $\longrightarrow_\beta$. $=_\beta$ is the transitive reflexive symmetric closure of $\longrightarrow_\beta$.

The type $(\sigma\to\sigma)\to\sigma\to\sigma$ is called the type of *numerals over σ*, nat_σ. The way to encode natural numbers as closed terms of type nat_σ is as follows.

$$c_n := \lambda f^{\sigma\to\sigma}.\lambda x^\sigma.f^n(x)$$

where

$$f^n(x) \text{ denotes } \underbrace{f(\ldots f(f}_{n \text{ times } f} x))$$

So $c_2 := \lambda f^{\sigma\to\sigma}.\lambda x^{\sigma}.f(f\,x)$. These are also known as the *Church numerals*. (For readability we don't denote the dependency of c_n on the type σ, but leave it implicit.) A Church numeral c_n denotes the n-times iteration: it is a higher order function that takes a function $f : \sigma\to\sigma$ and returns the n-times iteration of f.

Example 4. We show a computation with the Church numeral c_2: we apply it to the identity $\mathbf{I}_\sigma$.

$$\begin{aligned}
\lambda z^{\sigma}.c_2\,\mathbf{I}_\sigma\,z &\equiv \lambda z^{\sigma}.(\lambda f^{\sigma\to\sigma}.\lambda x^{\sigma}.f(f\,x))\mathbf{I}_\sigma\,z\\
&\longrightarrow_\beta \lambda z^{\sigma}.(\lambda x^{\sigma}.\mathbf{I}_\sigma(\mathbf{I}_\sigma\,x))z\\
&\longrightarrow_\beta \lambda z^{\sigma}.\mathbf{I}_\sigma(\mathbf{I}_\sigma\,z)\\
&\longrightarrow_\beta \lambda z^{\sigma}.\mathbf{I}_\sigma\,z\\
&\longrightarrow_\beta \lambda z^{\sigma}.z \equiv \mathbf{I}_\sigma
\end{aligned}$$

In the above example, we see that at a certain point there are several ways to reduce: we can contract the inner or the outer redex with $\mathbf{I}_\sigma$. In this case the result is exactly the same. In general there are many redexes within a term that can be reduced, and they may all yield a different result. Often we want to fix a certain method for reducing terms, or we only want to contract redexes of a certain shape. This can be observed in the following example.

Examples 5. Define the $\mathbf{S}$ combinator as follows.

$$\mathbf{S} := \lambda x^{\sigma\to\sigma\to\sigma}.\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.x\,z(y\,z) \;:\; (\sigma\to\sigma\to\sigma)\to(\sigma\to\sigma)\to\sigma\to\sigma$$

Then $\mathbf{S}\,\mathbf{K}_{\sigma\sigma}\,\mathbf{I}_\sigma : \sigma\to\sigma$ and

$$\mathbf{S}\,\mathbf{K}_{\sigma\sigma}\,\mathbf{I}_\sigma \longrightarrow_\beta (\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(y\,z))\mathbf{I}_\sigma$$

There are several ways of reducing this term further:

$$\begin{aligned}
(\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(y\,z))\mathbf{I}_\sigma &\quad \text{is a redex}\\
\mathbf{K}_{\sigma\sigma}\,z &\quad \text{is a redex}
\end{aligned}$$

$$\begin{aligned}
(\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(y\,z))\mathbf{I}_\sigma &\longrightarrow_\beta \lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\,z(\mathbf{I}_\sigma\,z)\\
&\equiv \lambda z^{\sigma}.(\lambda p^{\sigma}\,q^{\sigma}.p)\,z(\mathbf{I}_\sigma\,z)\\
&\longrightarrow_\beta \lambda z^{\sigma}.(\lambda q^{\sigma}.z)\,(\mathbf{I}_\sigma\,z)\\
\textit{Call by Value}\qquad &\longrightarrow_\beta \lambda z^{\sigma}.(\lambda q^{\sigma}.z)z\\
&\longrightarrow_\beta \lambda z^{\sigma}.z
\end{aligned}$$

But also

$$\begin{array}{rll}
(\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.\mathbf{K}_{\sigma\sigma}\, z(y\, z))\mathbf{I}_{\sigma} & \equiv & (\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.(\lambda p^{\sigma}\, q^{\sigma}.p)\, z(y\, z))\mathbf{I}_{\sigma} \\
& \longrightarrow_{\beta} & (\lambda y^{\sigma\to\sigma}.\lambda z^{\sigma}.(\lambda q^{\sigma}.z)(y\, z))\mathbf{I}_{\sigma} \\
& \longrightarrow_{\beta} & \lambda z^{\sigma}.(\lambda q^{\sigma}.z)\,(\mathbf{I}_{\sigma}\, z) \\
\textit{Call by Name} & \longrightarrow_{\beta} & \lambda z^{\sigma}.z
\end{array}$$

In the previous example we have seen that the term $\lambda z^{\sigma}.(\lambda q^{\sigma}.z)\,(\mathbf{I}_{\sigma}\, z)$ can be reduced in several ways. *Call-by-name* is the ordinary β-reduction, where one can contract any β-redex. In *call-by-value*, one is only allowed to reduce $(\lambda x.M)N$ if N is a *value*, where a value is an abstraction term or a variable ([34]). So to reduce a term of the form $(\lambda x.M)((\lambda y.N)P)$ "call-by-value", we first have to contract $(\lambda y.N)P$. Call-by-value restricts the number of redexes that is allowed to be contracted, but it does not prescribe which is the next redex to contract. More restrictive variations of β-reduction are obtained by defining a *reduction strategy* which is a recipe that describes for every term which redex to contract. Well-known reduction strategies are *left-most outermost* or *right-most innermost*. To understand these notions it should be observed that redexes can be *contained in another*, e.g. in $(\lambda x.M)((\lambda y.N)P)$ or in $(\lambda x.(\lambda y.N)P)Q$, but they can also be *disjoint*, in which case there's always one to the left of the other. Other reduction strategies select a set of redexes and contract these simultaneously (a notion that should be defined first of course). For example, it is possible to define the simultaneous contraction of all redexes in a term, which is usually called a *complete development*. We don't go into the theory of reduction strategies or developments here, but refer to the literature [3]. Reduction in simple type theory enjoys some important properties that we list here. We don't give any proofs, as they can be found in the standard literature [5].

Theorem 1. *The simple type theory enjoys the following important properties.*

- *Subject Reduction*
  If $M : \sigma$ and $M \longrightarrow_{\beta} P$, then $P : \sigma$.
- *Church-Rosser*
  If M is a well-typed term in $\lambda\!\to$ and $M \longrightarrow\!\!\!\!\rightarrow_{\beta} P$ and $M \longrightarrow\!\!\!\!\rightarrow_{\beta} N$, then there is a (well-typed) term Q such that $P \longrightarrow\!\!\!\!\rightarrow_{\beta} Q$ and $N \longrightarrow\!\!\!\!\rightarrow_{\beta} Q$.
- *Strong Normalization*
  If M is well-typed in $\lambda\!\to$, then there is no infinite β-reduction path starting from M.

Subject reduction states – looking at it from a programmers point of view – that well-typed programs don't go wrong: evaluating a program $M : \sigma$ to a value indeed returns a value of type σ. Church-Rosser states that it doesn't make any difference for the final value *how* we reduce: we always get the same value. Strong Normalization states that no matter how one evaluates, one always obtains a value: there are no infinite computations possible.

3.2 Simple type theory presented with derivation rules

Our definition of $\lambda\!\!\rightarrow$ terms (Definition 3) is given via a standard inductive definition of the terms. This is very close to Church' [9] original definition. A different presentation can be given by presenting the inductive definition of the terms in rule form:

$$\frac{}{x^\sigma : \sigma} \qquad \frac{M : \sigma\!\rightarrow\!\tau \;\; N : \sigma}{MN : \tau} \qquad \frac{P : \tau}{\lambda x^\sigma.P : \sigma\!\rightarrow\!\tau}$$

The advantage is that now we also have a *derivation tree*, a proof of the fact that the term has that type. We can reason over these derivations.

In the above presentations, the set of free variables of a term is a global notion, that can be computed by the function FV. This is sometimes felt as being a bit imprecise and then a presentation is given with *contexts* to explicitly declare the free variables of a term.

$$x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n$$

is a context, if all the x_i are distinct and the σ_i are all $\lambda\!\!\rightarrow$-types. Contexts are usually denoted by Γ and we write $x \in \Gamma$ if x is one of the variables declared in Γ.

Definition 7. *The* derivation rules *of $\lambda\!\!\rightarrow$ à la Church are as follows.*

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma\!\rightarrow\!\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma\!\rightarrow\!\tau}$$

We write $\Gamma \vdash_{\lambda\rightarrow} M : \sigma$ if there is a derivation using these rules with conclusion $\Gamma \vdash M : \sigma$.

So note that – apart from the context – we now also write the type as a declaration in the λ-abstraction (and not as a superscript): $\lambda x : \sigma.x$ instead of $\lambda x^\sigma.x$. This presents us with a slightly different view on the base syntax: we don't see the variables as being typed (x^σ), but we take the view of a countably infinite collection of untyped variables that we assign a type to in the context (the free variables) or in the λ-abstraction (the bound variables).

To relate this Definition with the one of 3, we state – without proof – the following fact, where we ignore the obvious isomorphism that "lifts" the types in the λ-abstraction to a superscript.

Fact 1. *If $\Gamma \vdash M : \sigma$, then $M : \sigma$ (Definition 3) and $FV(M) \subseteq \Gamma$.*
If $M : \sigma$ (Definition 3), then $\Gamma \vdash M : \sigma$, where Γ consists exactly of declarations of all the $x \in FV(M)$ to their corresponding types.

As an example, we give a complete derivation of $\vdash \mathbf{K}_{\sigma\tau} : \sigma \rightarrow \tau \rightarrow \sigma$.

$$\dfrac{\dfrac{x : \sigma, y : \tau \vdash x : \sigma}{x : \sigma \vdash \lambda y{:}\tau.x : \tau \rightarrow \sigma}}{\vdash \lambda x{:}\sigma.\lambda y{:}\tau.x : \sigma \rightarrow \tau \rightarrow \sigma}$$

Derivations of typing judgments tend to get quite broad, because we are constructing a derivation tree. Moreover, this tree may contain quite a lot of duplications. So, when we are looking for a term of a certain type, the tree format may not be the most efficient and easy-to-use. We therefore introduce a *Fitch style representation* of typing derivations, named after the logician Fitch, who has developed a natural deduction system for logic in this format, also called *flag deduction* style [16]. We don't show that these two derivation styles derive the same set of typable terms, because it should be fairly obvious. (And a precise proof involves quite some additional notions and notation.)

Definition 8. *The Fitch style presentation of the rules of $\lambda\!\!\rightarrow$ is as follows.*

1	$x : \sigma$	
2	...	
3	...	
4	$M : \tau$	
5	$\lambda x{:}\sigma.M : \sigma \to \tau$	*abs,* 1, 4

abs-rule

1	...	
2	...	
3	$M : \sigma \to \tau$	
4	...	
5	...	
6	$N : \sigma$	
7	...	
8	$M\,N : \tau$	*app,* 3, 6

app-rule

In a Fitch deduction, a hypothesis is introduced by "raising a flag", e.g. the $x : \sigma$ in the left rule. A hypothesis is discharged when we "withdraw the flag", which happens at line 5. In a Fitch deduction one usually numbers the lines and refers to them in the *motivation* of the lines: the "abs,$1, 4$" and the "app, $3, 6$" at the end of lines 5 and 7.

Some remarks apply.

- It should be understood that one can raise a flag under an already open flag (one can nest flags), but the variable x in the newly raised flag should be *fresh*: it should not be declared in any of the open flags.
- In the app-rule, the order of the M and N can of course be different. Also the terms can be in a "smaller scope", that is: $M : \sigma$ may be higher in the deduction under less flags. Basically the M and N should just be "in scope", where a flag-ending ends the scope of all terms that are under that flag.

We say that a Fitch deduction derives the judgement $\Gamma \vdash M : \sigma$ if $M : \sigma$ is on the last line of the deduction the raised flags together form the context Γ.

Example 6. We show an example of a derivation of a term of type

$$(\alpha{\to}\beta{\to}\gamma){\to}(\alpha{\to}\beta){\to}\alpha{\to}\gamma$$

We show the derivation in two stages, to indicate how one can use the Fitch deduction rules to incrementally construct a term of a given type.

$$
\begin{array}{ll}
1 & \left|\begin{array}{l} x : \alpha{\to}\beta{\to}\gamma \\ \hline \left|\begin{array}{l} y : \alpha{\to}\beta \\ \hline \left|\begin{array}{l} z : \alpha \\ \hline \left|\begin{array}{l} ?? \\ ?? \\ ? : \gamma \end{array}\right. \end{array}\right. \\ \lambda z{:}\alpha.? : \alpha{\to}\gamma \end{array}\right. \\
8 & \left|\begin{array}{l} \lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.? : (\alpha{\to}\beta){\to}\alpha{\to}\gamma \end{array}\right. \\
9 & \lambda x{:}\alpha{\to}\beta{\to}\gamma.\lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.? : (\alpha{\to}\beta{\to}\gamma){\to}(\alpha{\to}\beta){\to}\alpha{\to}\gamma
\end{array}
$$

$$
\begin{array}{ll}
1 & \left|\begin{array}{l} x : \alpha{\to}\beta{\to}\gamma \\ \hline \left|\begin{array}{l} y : \alpha{\to}\beta \\ \hline \left|\begin{array}{l} z : \alpha \\ \hline \left|\begin{array}{l} x\,z : \beta{\to}\gamma \\ y\,z : \beta \\ x\,z(y\,z) : \gamma \end{array}\right. \end{array}\right. \\ \lambda z{:}\alpha.x\,z(y\,z) : \alpha{\to}\gamma \end{array}\right. \\
8 & \left|\begin{array}{l} \lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.x\,z(y\,z) : (\alpha{\to}\beta){\to}\alpha{\to}\gamma \end{array}\right. \\
9 & \lambda x{:}\alpha{\to}\beta{\to}\gamma.\lambda y{:}\alpha{\to}\beta.\lambda z{:}\alpha.x\,z(y\,z) : (\alpha{\to}\beta{\to}\gamma){\to}(\alpha{\to}\beta){\to}\alpha{\to}\gamma
\end{array}
$$

Exercises 1. 1. Construct a term of type $(\delta{\to}\delta{\to}\alpha){\to}(\alpha{\to}\beta{\to}\gamma){\to}(\delta{\to}\beta){\to}\delta{\to}\gamma$
2. Construct two terms of type $(\delta{\to}\delta{\to}\alpha){\to}(\gamma{\to}\alpha){\to}(\alpha{\to}\beta){\to}\delta{\to}\gamma{\to}\beta$
3. Construct a term of type $((\alpha{\to}\beta){\to}\alpha){\to}(\alpha{\to}\alpha{\to}\beta){\to}\alpha$
4. Construct a term of type $((\alpha{\to}\beta){\to}\alpha){\to}(\alpha{\to}\alpha{\to}\beta){\to}\beta$ (Hint: use the previous exercise.)

3.3 The Curry-Howard formulas-as-types correspondence

Using the presentation of $\lambda{\to}$ with derivation rules, it is easier to make the *Curry-Howard formulas-as-types* correspondence precise. The idea is that there are two readings of a judgement $M : \sigma$:

1. term as algorithm/program, type as specification :
   M is a function of type σ
2. type as a proposition, term as its proof :
   M is a proof of the proposition σ

More precisely, the Curry-Howard formulas-as-types correspondence states that there is a natural one-to-one correspondence between typable terms in $\lambda{\to}$

and derivations in *minimal proposition logic*. Looking at it from the logical point of view: the judgement $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash M : \sigma$ can be read as M *is a proof of* σ *from the assumptions* $\tau_1, \tau_2, \ldots, \tau_n$.

Definition 9. *The system of minimal proposition logic* PROP *consists of*

– *implicational propositions, generated by the following abstract syntax:*

$$\mathsf{prop} ::= \mathsf{PropVar} | (\mathsf{prop} \to \mathsf{prop})$$

– *derivation rules (Δ is a set of propositions, σ and τ are propositions)*

$$\frac{\sigma\to\tau \quad \sigma}{\tau}\to\text{-}E \qquad \frac{\begin{array}{c}[\sigma]^j\\ \vdots\\ \tau\end{array}}{\sigma\to\tau}\,[j]\to\text{-}I$$

We write $\Delta \vdash_{\mathrm{PROP}} \sigma$ if there is a derivation using these rules with conclusion σ and non-discharged assumptions *in Δ.*

Note the difference between a context, which is basically a *list*, and a *set* of assumptions. Logic (certainly in natural deduction) is usually presented using a *set* of assumptions, but there is no special reason for not letting the assumptions be a list, or a multi-set.

We now give a precise definition of the formulas-as-types correspondence. For this we take the presentation of PROP with lists (so Δ is a list in the following definition). As a matter of fact the *formulas-as-types* part of the definition is trivial: a proposition in PROP is just a type in $\lambda\!\to$, but the most interesting part of the correspondence is the *proofs-as-terms* embedding, maybe best called the *deductions-as-term* embedding. For PROP, this part is also quite straightforward, but we describe it in detail nevertheless.)

Definition 10. *The* deductions-as-terms *embedding from derivation of* PROP *to term of $\lambda\!\to$ is defined inductively as follows. We associate to a list of propositions Δ a context Γ in the obvious way by replacing σ_i in Δ with $x_i : \sigma_i \in \Gamma$. On the left we give the inductive clause for the derivation and on the right we describe on top of the line the terms we have (by induction) and below the line the term that the derivation gets mapped to.*

$$\frac{}{\Delta \vdash \sigma}\,\sigma \in \Delta \;\rightsquigarrow\; \frac{}{\Gamma \vdash x : \sigma}\,x : \sigma \in \Gamma$$

$$\frac{\sigma\to\tau \quad \sigma}{\tau}\to\text{-}E \rightsquigarrow \frac{\Gamma_1 \vdash M : \sigma\to\tau \quad \Gamma_2 \vdash N : \sigma}{\Gamma_1 \cup \Gamma_2 \vdash M\,N : \tau}$$

$$\frac{\begin{array}{c}[\sigma]^j\\ \vdots\\ \tau\end{array}}{\sigma\to\tau}\,[j]\to\text{-}I \quad\rightsquigarrow\quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma\to\tau}$$

We denote this embedding by $\overline{-}$*, so if* $\mathcal{D}$ *is a derivation from* PROP*,* $\mathcal{D}$ *is a* $\lambda\!\to$*-term.*

For a good understanding we give a detailed example.

Example 7. Consider the following natural deduction derivation PROP and the term in $\lambda\!\to$ it gets mapped to.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\alpha\to\beta\to\gamma]^3 \;\; [\alpha]^1}{\beta\to\gamma} \qquad \dfrac{[\alpha\to\beta]^2 \;\; [\alpha]^1}{\beta}}{\gamma}\,1}{\alpha\to\gamma}\,2}{(\alpha\to\beta)\to\alpha\to\gamma}\,3}{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma} \;\mapsto\; \begin{array}{l}\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz)\\ :(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma\end{array}$$

To create the term on the right, it is best to decorate the deduction tree with terms, starting from the leaves (decorated by variables) and working downwards, finally creating a term for the root node that is the λ-term that corresponds to the whole deduction.

$$\dfrac{\dfrac{\dfrac{\dfrac{[x:\alpha\to\beta\to\gamma]^3 \;\; [z:\alpha]^1}{xz:\beta\to\gamma} \qquad \dfrac{[y:\alpha\to\beta]^2 \;\; [z:\alpha]^1}{yz:\beta}}{xz(yz):\gamma}\,1}{\lambda z{:}\alpha.xz(yz):\alpha\to\gamma}\,2}{\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz):(\alpha\to\beta)\to\alpha\to\gamma}\,3}{\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz):(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}$$

Theorem 2 (Soundness, Completeness of formulas-as-types).

1. *If* $\mathcal{D}$ *is a natural deduction in* PROP *with conclusion* σ *and non-discharged assumption in* Δ*, then*

$$\boldsymbol{x} : \boldsymbol{\Delta} \vdash \overline{\mathcal{D}} : \sigma \text{ in } \lambda\!\to.$$

2. *If* $\Gamma \vdash M : \sigma$ *in* $\lambda\!\to$*, then there is a derivation of* σ *from the* Δ *in* PROP*, where* Δ *is* Γ *without the variable-assignments.*

We don't give the proofs, as they are basically by a straightforward induction. The second part of the Theorem can be strengthened a bit: we can construct a derivation $\mathcal{D}$ out of the term M, as an inverse to the mapping $\overline{-}$. So, the formulas-as-types correspondence constitutes an *isomorphism* between derivations in PROP and well-typed terms in $\lambda\!\to$.

Exercise 2. Add types to the λ-abstractions and give the derivation that corresponds to the term $\lambda x.\lambda y.y(\lambda z.y\,x) : (\gamma\to\varepsilon)\to((\gamma\to\varepsilon)\to\varepsilon)\to\varepsilon$.

In the λ-calculus we have a notion of computation, given by β-reduction: $(\lambda x{:}\sigma.M)P \longrightarrow_\beta M[x := P]$. Apart from that, there is also a notion of η-reduction: $\lambda x{:}\sigma.M\,x \longrightarrow_\eta M$ if $x \notin \mathrm{FV}(M)$. The idea of considering these terms as equal is quite natural, because they behave exactly the same as functions: $(\lambda x{:}\sigma.M\,x)P \longrightarrow_\beta M\,P$. In a typed setting, it is very natural to consider the rule in the opposite direction, because then one can make sure that every term of a function type has a normal form that is a λ-abstraction. Of course this requires a proviso to prevent an infinite η-reduction of the form $x \longrightarrow_\eta \lambda y{:}\sigma.x\,y \longrightarrow_\eta \lambda y{:}\sigma.(\lambda z{:}\sigma.x\,z)y \ldots$

In natural deduction we also have a notion of computation: *cut-elimination* or *detour-elimination*. If one introduces a connective and then immediately eliminates it again, this is called a cut or a detour. A *cut* is actually a *rule* in the sequent calculus representation of logic and the *cut-elimination* theorem in sequent calculus states that the cut-rule is derivable and thus superfluous. In natural deduction, we can eliminate detours, which are often also called cuts, a terminology that we will also use here.

Definition 11. *A* cut *in a deduction in minimal propositional logic is a place where an $\rightarrow$-I is immediately followed by an elimination of that same $\rightarrow$. Graphically ($\mathcal{D}_1$ and $\mathcal{D}_2$ denote deductions):*

$$\dfrac{\dfrac{\begin{array}{c}[\sigma]^1\\ \mathcal{D}_1\\ \tau\end{array}}{\sigma\rightarrow\tau}\,1 \qquad \dfrac{\mathcal{D}_2}{\sigma}}{\tau}$$

Cut-elimination *is defined by replacing the cut in a deduction in the way given below.*

$$\dfrac{\dfrac{\begin{array}{c}[\sigma]^1\\ \mathcal{D}_1\\ \tau\end{array}}{\sigma\rightarrow\tau}\,1 \qquad \dfrac{\mathcal{D}_2}{\sigma}}{\tau} \quad\longrightarrow\quad \begin{array}{c}\mathcal{D}_2\\ \sigma\\ \mathcal{D}_1\\ \tau\end{array}$$

So every occurrence of the discharged assumption $[\sigma]^1$ in $\mathcal{D}_1$ is replaced by the deduction $\mathcal{D}_2$.

It is not hard to prove that eliminating a cut yields a well-formed natural deduction again, with the same conclusion. The set of non-discharged assumptions remains the same or shrinks. (In case there is no occurrence of $[\sigma]^1$ at all; then $\mathcal{D}_2$ is removed and also its assumptions.) That this process terminates is not obvious: if $[\sigma]^1$ occurs several times, $\mathcal{D}_2$ gets copied, resulting in a larger deduction.

Lemma 2. *Cut-elimination in* PROP *corresponds to β-reduction in $\lambda\!\!\rightarrow$: if $\mathcal{D}_1 \longrightarrow_{cut} \mathcal{D}_2$, then $\overline{\mathcal{D}_1} \longrightarrow_\beta \overline{\mathcal{D}_2}$*

The proof of this Lemma is indicated in the following diagram.

$$\dfrac{\dfrac{\begin{array}{c}[x:\sigma]^1\\ \mathcal{D}_1\\ M:\tau\end{array}}{\lambda x{:}\sigma.M:\sigma\!\rightarrow\!\tau}\,1 \qquad \dfrac{\mathcal{D}_2}{P:\sigma}}{(\lambda x{:}\sigma.M)P:\tau} \quad \longrightarrow_\beta \quad \begin{array}{c}\mathcal{D}_2\\ P:\sigma\\ \mathcal{D}_1\\ M[x:=P]:\tau\end{array}$$

To get a better understanding of the relation between cut-elimination and β-reduction, we now study an example.

Example 8. Consider the following proof of $A\!\rightarrow\!A\!\rightarrow\!B, (A\!\rightarrow\!B)\!\rightarrow\!A \vdash B$.

$$\dfrac{\dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B} \qquad \dfrac{(A\!\rightarrow\!B)\!\rightarrow\!A \qquad \dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B}}{A}}{B}$$

It contains a cut: a $\rightarrow$-I directly followed by an $\rightarrow$-E. We now present the same proof after reduction

$$\dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B}{A\!\rightarrow\!B} \qquad \dfrac{(A\!\rightarrow\!B)\!\rightarrow\!A \qquad \dfrac{\dfrac{A\!\rightarrow\!A\!\rightarrow\!B\ \ [A]^1}{A\!\rightarrow\!B}\ [A]^1}{B}}{A}}{B}$$

We now present the same derivations of $A\!\rightarrow\!A\!\rightarrow\!B, (A\!\rightarrow\!B)\!\rightarrow\!A \vdash B$, now with term information

$$\dfrac{\dfrac{\dfrac{p : A{\to}A{\to}B \;\; [x : A]^1}{p\,x : A{\to}B} \qquad [x : A]^1}{p\,x\,x : B}}{\lambda x{:}A.p\,x\,x : A{\to}B} \qquad \dfrac{q : (A{\to}B){\to}A \qquad \dfrac{\dfrac{\dfrac{p : A{\to}A{\to}B \;\; [x : A]^1}{p\,x : A{\to}B} \qquad [x : A]^1}{p\,x\,x : B}}{\lambda x{:}A.p\,x\,x : A{\to}B}}{q(\lambda x{:}A.p\,x\,x) : A}$$

$$(\lambda x{:}A.p\,x\,x)(q(\lambda x{:}A.p\,x\,x)) : B$$

The term contains a β-redex: $(\lambda x{:}A.p\,x\,x)\,(q(\lambda x{:}A.p\,x\,x))$ We now present the reduced proof of $A{\to}A{\to}B, (A{\to}B){\to}A \vdash B$ with term info. For reasons of page size we summarize the derivation of $q(\lambda x{:}A.p\,x\,x) : A$ as $\mathcal{D}$. So

$$\dfrac{\boxed{\mathcal{D}}}{q(\lambda x{:}A.p\,x\,x) : A} \;:=\; \dfrac{q : (A{\to}B){\to}A \qquad \dfrac{\dfrac{\dfrac{p : A{\to}A{\to}B \;\; [x{:}A]^1}{p\,x : A{\to}B} \qquad [x{:}A]^1}{p\,x\,x : B}}{\lambda x{:}A.p\,x\,x : A{\to}B}}{q(\lambda x{:}A.p\,x\,x) : A}$$

This is the sub-derivation that gets copied under cut-elimination (β-reduction).

$$\dfrac{p : A{\to}A{\to}B \qquad \dfrac{\boxed{\mathcal{D}}}{q(\lambda x{:}A.p\,x\,x) : A}}{p(q(\lambda x{:}A.p\,x\,x)) : A{\to}B} \qquad \dfrac{\boxed{\mathcal{D}}}{q(\lambda x{:}A.p\,x\,x) : A}$$

$$p(q(\lambda x{:}A.p\,x\,x))(q(\lambda x{:}A.p\,x\,x)) : B$$

4 Type assignment versus typed terms

4.1 Untyped $\boldsymbol{\lambda}$-calculus

Simple Type Theory is not very expressive: one can only represent a limited number of functions over the nat_σ (see the paragraph after Definition 6) data types in $\lambda{\to}$. We can allow more functions to be definable by relaxing the type constraints. The most flexible system is to have no types at all.

Definition 12. *The terms of the untyped λ-calculus, Λ, are defined as follows.*

$$\Lambda ::= \mathit{Var} \mid (\Lambda\,\Lambda) \mid (\lambda\,\mathit{Var}.\Lambda)$$

Examples are the well-known combinators that we have already seen in a typed fashion: $\mathbf{K} := \lambda x\, y.x$, $\mathbf{S} := \lambda x\, y\, z.x\, z(y\, z)$. But we can now do more: here are some well-known untyped λ-terms $\omega := \lambda x.x\, x$, $\Omega := \omega\, \omega$. The notions of β-reduction and β-equality generalize from the simple typed case, so we don't repeat it here. An interesting aspect is that we can now have *infinite reductions*. (Which is impossible in $\lambda\!\!\rightarrow$, as that system is Strongly Normalizing.) The simplest infinite reduction is the following *loop*:

$$\Omega \longrightarrow_\beta \Omega$$

A term that doesn't loop but whose reduction path contains infinitely many different terms is obtained by putting $\omega_3 := \lambda x.x\, x\, x$, $\Omega_3 := \omega_3\omega_3$. Then:

$$\Omega_3 \longrightarrow_\beta \omega_3\, \omega_3\, \omega_3 \longrightarrow_\beta \omega_3\, \omega_3\, \omega_3\, \omega_3 \longrightarrow_\beta \ldots$$

The untyped λ-calculus was defined by Church [9] and proposed as a system to capture the notion of *mechanic computation*, for which Turing proposed the notion of Turing machine. An important property of the untyped λ-calculus is that it is *Turing complete*, which was proved by Turing in 1936, see [13]. The power of Λ lies in the fact that you can *solve recursive equations*.

A recursive equation is a question of the following kind:

– Is there a term M such that

$$M\, x =_\beta x\, M\, x?$$

– Is there a term M such that

$$M\, x =_\beta \mathsf{if}\ (\mathsf{Zero}\, x)\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{Mult}\, x\, (M\, (\mathsf{Pred}\, x))?$$

So, we have two expressions on either side of the $=_\beta$ sign, both containing an unknown M and we want to know whether a solution for M exists.

The answer is: **yes**, if we can rewrite the equation to one of the form

$$M =_\beta \boxed{\ldots M \ldots} \tag{1}$$

Note that this is possible for the equations written above. For example the first equation is solved by a term M that satisfies $M =_\beta \lambda x.x\, M\, x$.

That we can solve equation of the form (1) is because every term in the λ-calculus has a *fixed point*. Even more: we have a *fixed point combinator*.

Definition 13. – *The term M is a* fixed point *of the term P if $P\, M =_\beta M$.*
– *The term Y is a* fixed point combinator *if for every term P, $Y\, P$ is a fixed point of P, that is if*

$$P\, (Y\, P) =_\beta Y\, P.$$

In the λ-calculus we have various fixed point combinators, of which the Y-combinator is the most well-known one: $Y := \lambda f.(\lambda x.f(x\, x))(\lambda x.f(x\, x))$.

Exercise 3. Verify that the Y as defined above is a fixed point combinator: $P\,(Y\,P) =_\beta Y\,P$ for every λ-term P.
Verify that $\Theta := (\lambda x\,y.y(x\,x\,y))(\lambda x\,y.y(x\,x\,y))$ is also a fixed point combinator that is even *reducing*: $\Theta\,P \longrightarrow\!\!\!\!\rightarrow_\beta P\,(\Theta\,P)$ for every λ-term P.

The existence of fixed-points is the key to the power of the λ-calculus. But we also need natural numbers and booleans to be able to write programs. In Section 3 we have already seen the *Church numerals*:

$$c_n := \lambda f.\lambda x.f^n(x)$$

where

$$f^n(x) \text{ denotes } \underbrace{f(\ldots f(f}_{n \text{ times } f}\ x))$$

The successor is easy to define for these numerals: $\mathsf{Suc} := \lambda n.\lambda f\,x.f(n\,f\,x)$. Addition can also be defined quite easily, but if we are lazy we can also use the fixed-point combinator. We want to solve

$$\mathsf{Add}\,n\,m := \mathsf{if}\,(\mathsf{Zero}\,n)\,\mathsf{then}\,m\,\mathsf{else}\,\mathsf{Add}\,(\mathsf{Pred}\,n)\,m)$$

where Pred is the predecessor function, Zero is a test for zero and $\mathsf{if}\ldots\mathsf{then}\ldots\mathsf{else}$ is a case distinction on booleans. The booleans can be defined by

$$\begin{aligned}\mathsf{true} &:= \lambda x\,y.x\\ \mathsf{false} &:= \lambda x\,y.y\\ \mathsf{if}\,b\,\mathsf{then}\,P\,\mathsf{else}\,Q &:= b\,P\,Q.\end{aligned}$$

Exercise 4.
1. Verify that the booleans behave as expected:
   $\mathsf{if\,true\,then}\,P\,\mathsf{else}\,Q =_\beta P$ and $\mathsf{if\,false\,then}\,P\,\mathsf{else}\,Q =_\beta Q$.
2. Define a test-for-zero Zero on the Church numerals :$\mathsf{Zero}\,c_0 =_\beta \mathsf{true}$ and $\mathsf{Zero}\,c_{n+1} =_\beta \mathsf{false}$. (Defining the predecessor is remarkably tricky!)

Apart from the natural numbers and booleans, it is not difficult to find encodings of other data, like lists and trees. Given the expressive power of the untyped λ-calculus and the limited expressive power of $\lambda{\to}$, one may wonder why we want types. There are various good reasons for that, most of which apply to the the "typed versus untyped programming languages" issue in general.

Types give a (partial) specification. Types tell the programmer – and a person reading the program — what a program (λ-term) does, to a certain extent. Types only give a very partial specification, like $f : \mathbb{N} \to \mathbb{N}$, but depending on the type system, this information can be enriched, for example: $f : \Pi n : \mathbb{N}.\exists m : \mathbb{N}.m > n$, stating that f is a program that takes a number n and returns an m that is larger than n. In the Chapter by Bove and Dybjer, one can find examples of that type and we will also come back to this theme in this Chapter in Section 6.

"Well-typed programs never go wrong" (Milner). The *Subject Reduction property* guarantees that a term of type σ remains to be of type σ under evaluation. So, if $M : \mathtt{nat}$ evaluates to a value v, we can be sure that v is a natural number.

The type checking algorithm detects (simple) mistakes. Types can be checked at compile time (*statically*) and this is a simple but very useful method to detect simple mistakes like typos and applying functions to the wrong arguments. Of course, in a more refined type system, type checking can also detect more subtle mistakes.

Typed terms always terminate(?) In typed λ-calculi used for representing proofs (following the Curry-Howard isomorphism), the terms are always terminating, and this is seen as an advantage as it helps in proving consistency of logical theories expressed in these calculi. In general, termination very much depends on the typing system. In this paper, all type systems only type terminating (strongly normalizing) λ-terms, which also implies that these systems are not Turing complete. Type systems for programming languages will obviously allow also non-terminating calculations. A simple way to turn $\lambda\!\!\rightarrow$ into a Turing complete language is by adding fixed point combinators (for every type σ) $Y_\sigma : (\sigma\!\rightarrow\!\sigma)\!\rightarrow\!\sigma$ with the reduction rule $Y\,f \rightarrow f(Y\,f)$. This is basically the system PCF, first defined and studied by Plotkin [35].

Given that we want types, the situation with a system like $\lambda\!\!\rightarrow$ as presented in Section 3, is still unsatisfactory from a programmers point of view. Why would the programmer have to write all those types? The compiler should compute the type information for us!

For M an *untyped* term, we want the type system to *assign* a type σ to M (or say that M is not typable). Such a type system is called a *type assignment system*, or also *typing à la Curry* (as opposed to the *typing à la Church* that we have seen up to now).

4.2 Simple type theory à la Church and à la Curry

We now set the two systems side-by-side: $\lambda\!\!\rightarrow$ à la Church and à la Curry.

Definition 14. *In $\lambda\!\!\rightarrow$ à la Curry, the terms are*

$$\Lambda ::= Var \mid (\Lambda\,\Lambda) \mid (\lambda\, Var.\Lambda)$$

In $\lambda\!\!\rightarrow$ à la Church, the terms are

$$\Lambda_{Ch} ::= Var \mid (\Lambda_{Ch}\,\Lambda_{Ch}) \mid (\lambda\, Var{:}\sigma.\Lambda_{Ch})$$

where σ ranges over the simple types, as defined in Definition 1.

These sets of terms are just the *preterms*. The typing rules will select the *well-typed* terms from each of these sets.

Definition 15. *The typing rules of $\lambda\!\!\rightarrow$ à la Church and $\lambda\!\!\rightarrow$ à la Curry are as follows. (The ones for the Church system are the same as the ones in Definition 7.)*
$\lambda\!\!\rightarrow$ (à la Church):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \quad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma{\rightarrow}\tau}$$

$\lambda\!\!\rightarrow$ (à la Curry):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \quad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma{\rightarrow}\tau}$$

The rules à la Curry can of course also be given in the Fitch style, which is the style we use when giving derivations of typings.

Exercise 5. Give a full derivation of

$$\vdash \lambda x.\lambda y.y(\lambda z.y\,x) : (\gamma{\rightarrow}\varepsilon){\rightarrow}((\gamma{\rightarrow}\varepsilon){\rightarrow}\varepsilon){\rightarrow}\varepsilon$$

in Curry style $\lambda\!\!\rightarrow$

We can summarize the differences between *Typed Terms* and *Type Assignment* as follows:

- With typed terms (typing à la Church), we have terms with type information in the λ-abstraction: $\lambda x{:}\alpha.x : \alpha{\rightarrow}\alpha$. As a consequence:
  - Terms have unique types,
  - The type is directly computed from the type info in the variables.
- With type assignment (typing à la Curry), we assign types to untyped λ-terms: $\lambda x.x : \alpha{\rightarrow}\alpha$. As a consequence:
  - Terms do not have unique types,
  - A *principal type* can be computed (using unification).

Examples 9.
- Typed Terms:

  $$\lambda x{:}\alpha.\lambda y{:}(\beta{\rightarrow}\alpha){\rightarrow}\alpha.y(\lambda z{:}\beta.x)$$

  has only the type $\alpha{\rightarrow}((\beta{\rightarrow}\alpha){\rightarrow}\alpha){\rightarrow}\alpha$
- Type Assignment: $\lambda x.\lambda y.y(\lambda z.x)$ can be assigned the types
  - $\alpha{\rightarrow}((\beta{\rightarrow}\alpha){\rightarrow}\alpha){\rightarrow}\alpha$
  - $(\alpha{\rightarrow}\alpha){\rightarrow}((\beta{\rightarrow}\alpha{\rightarrow}\alpha){\rightarrow}\gamma){\rightarrow}\gamma$
  - ...

  with $\alpha{\rightarrow}((\beta{\rightarrow}\alpha){\rightarrow}\gamma){\rightarrow}\gamma$ being the *principal type*, a notion to be defined and discussed later.

There is an obvious connection between Church and Curry typed $\lambda\!\!\rightarrow$, given by the *erasure map*.

Definition 16. *The* erasure *map* $|-|$ *from* $\lambda\!\!\rightarrow$ *à la Church to* $\lambda\!\!\rightarrow$ *à la Curry is defined by erasing all type information:*

$$\begin{aligned} |x| &:= x \\ |M\,N| &:= |M|\,|N| \\ |\lambda x : \sigma.M| &:= \lambda x.|M| \end{aligned}$$

So, e.g. $|\lambda x{:}\alpha.\lambda y{:}(\beta{\rightarrow}\alpha){\rightarrow}\alpha.y(\lambda z{:}\beta.x))| = \lambda x.\lambda y.y(\lambda z.x))$.

Theorem 3. *If* $M : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Church, then* $|M| : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Curry. If* $P : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Curry, then there is an* M *such that* $|M| \equiv P$ *and* $M : \sigma$ *in* $\lambda\!\!\rightarrow$ *à la Church.*

The proof is by an easy induction on the derivation.

4.3 Principal types

We now discuss the notion of a principal type in $\lambda\!\!\rightarrow$ à la Curry. We will describe an algorithm, the *principal type algorithm*, that, given a closed untyped term M, computes a type σ if M is typable with type σ in $\lambda\!\!\rightarrow$, and "reject" if M is not typable. Moreover, the computed type σ is "minimal" in the sense that all possible types for M are substitution instances of σ.

Computing a principal type for M in $\lambda\!\!\rightarrow$ à la Curry proceeds as follows:

1. Assign a type variable to every variable x in M.
2. Assign a type variable to every *applicative sub-term* of M.
3. Generate a (finite) set of equations E between types that need to hold in order to ensure that M is typable.
4. Compute a "minimal substitution" S, substituting types for type variables, that makes all equations in E hold. (This is a *most general unifier* for E.)
5. With S compute the type of M.

The algorithm described above can fail only if their is no unifying substitution for E. In that case we return "reject" and conclude that M is not typable. An *applicative sub-term* is a term that is not a variable and does not start with a λ. (So it is a sub-term of the form $P\,Q$). One could label all sub-terms with a type variable, but that just adds superfluous overhead. We show how the algorithm works by elaborating an example.

Example 10. We want to compute the principal type of $\lambda x.\lambda y.y(\lambda z.yx)$.

1. Assign type variables to all term variables: $x : \alpha, y : \beta, z : \gamma$.
2. Assign type variables to all applicative sub-terms: $y\,x : \delta$, $y(\lambda z.y\,x) : \varepsilon$. These two steps yield the following situation, where we indicate the types of the variables and applicative sub-terms by super- and subscripts.

$$\lambda x^{\alpha}.\lambda y^{\beta}.\underbrace{y^{\beta}(\lambda z^{\gamma}.\overbrace{y^{\beta}x^{\alpha}}^{\delta})}_{\varepsilon}$$

3. Generate equations between types, necessary for the term to be typable:

$$E = \{\beta = \alpha{\to}\delta, \beta = (\gamma{\to}\delta){\to}\varepsilon\}$$

The equation $\beta = \alpha{\to}\delta$ arises from the sub-term $\overbrace{y^{\beta}x^{\alpha}}^{\delta}$, which is of type δ if β is a function type with domain α and range δ. The equation $\beta = (\gamma{\to}\delta){\to}\varepsilon$ arises from the sub-term $\underbrace{y^{\beta}(\lambda z^{\gamma}.\overbrace{y\,x}^{\delta})}_{\varepsilon}$, which is of type ε if β is a function type with domain $\gamma{\to}\delta$ and range ε.

4. Find a most general substitution (a *most general unifier*) for the type variables that solves the equations:

$$S := \{\alpha := \gamma{\to}\delta,\ \ \beta := (\gamma{\to}\delta){\to}\varepsilon,\ \ \delta := \varepsilon\}$$

5. The *principal type* of $\lambda x.\lambda y.y(\lambda z.yx)$ is now

$$(\gamma{\to}\varepsilon){\to}((\gamma{\to}\varepsilon){\to}\varepsilon){\to}\varepsilon$$

Exercise 6. 1. Compute the principal type for $\mathbf{S} := \lambda x.\lambda y.\lambda z.x\,z(y\,z)$

2. Which of the following terms is typable? If it is, determine the *principal type*; if it isn't, show that the typing algorithm rejects the term.
   (a) $\lambda z\,x.z(x(\lambda y.y\,x))$
   (b) $\lambda z\,x.z(x(\lambda y.y\,z))$
3. Compute the principal type for $M := \lambda x.\lambda y.x(y(\lambda z.x\,z\,z))(y(\lambda z.x\,z\,z))$.

We now introduce the notions required for the principal types algorithm.

Definition 17. – *A* type substitution *(or just substitution) is a map S from type variables to types. As a function, we write it* after the type*, so σS denotes the result of carrying out substitution S on σ.*

– *Most substitutions we encounter are the identity on all but a finite number of type variables, so we often denote a substitution as $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$. We view a type substitution as a function that is carried out* in parallel *so $[\alpha := \beta{\to}\beta, \beta := \alpha{\to}\gamma]$ applied to $\alpha{\to}\beta$ results in $(\beta{\to}\beta){\to}\alpha{\to}\gamma$.*

– *We can compose substitutions in the obvious way: $S;T$ is obtained by first performing S and then T.*

– *A* unifier *of the types σ and τ is a substitution that "makes σ and τ equal", i.e. an S such that $\sigma S = \tau S$.*

– *A* most general unifier *(or mgu) of the types σ and τ is the "simplest substitution" that makes σ and τ equal, i.e. an S such that*
  - *$\sigma S = \tau S$*
  - *for all substitutions T such that $\sigma T = \tau T$ there is a substitution R such that $T = S;R$.*

All these notions generalize to lists instead of pairs σ, τ. We say that S *unifies* the list of equations $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$ if $\sigma_1 S = \tau_1 S, \ldots, \sigma_n S = \tau_n S$, that is: S makes all equations true.

The crucial aspect in the principal type algorithm is the computability of a *most general unifier* for a set of type equations. The rest of the algorithm should be clear from the example and we don't describe it in detail here.

Definition 18. *We define the algorithm U that, when given a list of type equations $E = \langle \sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n \rangle$ outputs a substitution S or "reject" as follows. U looks at the first type equation $\sigma_1 = \tau_1$ and depending on its form it outputs:*

- $U(\langle \alpha = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \sigma_2 = \tau_2, \ldots, \sigma_n = \tau_n \rangle)$.
- $U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) :=$ *"reject" if* $\alpha \in FV(\tau_1)$, $\tau_1 \neq \alpha$.
- $U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) :=$
  $[\alpha := V(\tau_1), U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \ldots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle)]$, *if* $\alpha \notin FV(\tau_1)$, *where V abbreviates* $U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \ldots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle)$.
- $U(\langle \sigma_1 = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \alpha = \sigma_1, \ldots, \sigma_n = \tau_n \rangle)$
- $U(\langle \mu \to \nu = \rho \to \xi, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \mu = \rho, \nu = \xi, \ldots, \sigma_n = \tau_n \rangle)$

Theorem 4. *The function U computes the* most general unifier *of a set of equations E. That is,*

- *If $U(E) =$ "reject", then there is no substitution S that unifies E.*
- *If $U(E) = S$, then S unifies E and for all substitutions T that unify E, there is a substitution R such that $T = S; R$ (S is most general).*

Definition 19. *The type σ is a* principal type *for the closed untyped λ-term M if*

- *$M : \sigma$ in $\lambda\!\to$ à la Curry*
- *for all types τ, if $M : \tau$, then $\tau = \sigma S$ for some substitution S.*

Theorem 5 (Principal Types). *There is an algorithm* PT *that, when given a closed (untyped) λ-term M, outputs*

- *A* principal type *σ such that $M : \sigma$ in $\lambda\!\to$ à la Curry.*
- *"reject" if M is not typable in $\lambda\!\to$ à la Curry.*

The algorithm is the one we have described before. We don't give it in formal detail, nor the proof of its correctness, but refer to [5] and [40]. This algorithm goes back to the type inference algorithm for simply typed lambda calculus of Hindley [23], which was independently developed by Milner [29] and extended to the weakly polymorphic case (see Section 5.1). Damas [12] has proved it correct and therefore this algorithm is often referred to as the Hindley-Milner or Damas-Milner algorithm.

If one wants to type an *open term* M, i.e. one that contains free variables, one is actually looking for what is known as a *principal pair*, consisting of a context Γ and a type σ such that $\Gamma \vdash M : \sigma$ and if $\Gamma' \vdash M : \tau$, then there

is a substitution S such that $\tau = \sigma S$ and $\Gamma' = \Gamma S$. (A substitution extends straightforwardly to contexts.) However, there is a simpler way of attacking this problem: just apply the PT algorithm for closed terms to $\lambda x_1 \ldots \lambda x_n.M$ where $x_1, \ldots, x_n$ is the list of free variables in M.

The following describes a list of typical decidability problems one would like to have an algorithm for in a type theory.

Definition 20.

$\vdash M : \sigma$?	Type Checking Problem	*TCP*
$\vdash M : ?$	Type Synthesis *or* Type Assginment Problem	*TSP, TAP*
$\vdash ? : \sigma$	Type Inhabitation Problem	*TIP*

Theorem 6. *For $\lambda\!\!\rightarrow$, all problems defined in Definition 20 are decidable, both for the Curry style and for the Church style versions of the system.*

For Church style, TCP and TSP are trivial, because we can just "read off" the type from the term that has the variables in the λ-abstractions decorated with types. For Curry style, TSP is solved by the PT algorithm. This also gives a way to solve TCP: to verify if $M : \sigma$, we just compute the principal type of M, say τ, and verify if σ is a substitution instance of τ (which is decidable).

In general, one may think that TCP is easier than TSP, but they are (usually) equivalent: Suppose we need to solve the TCP $M\,N : \sigma$. The only thing we can do is to solve the TSP $N :?$ and if this gives answer τ, solve the TCP $M : \tau\!\rightarrow\!\sigma$. So we see that these problems are tightly linked.

For Curry systems, TCP and TSP soon become undecidable if we go beyond $\lambda\!\!\rightarrow$. In the next section we will present the polymorphic λ-calculus, whose Curry style variant has an undecidable TCP.

TIP is decidable for $\lambda\!\!\rightarrow$, as it corresponds to *provability* in PROP, which is known to be decidable. This applies to both the Church and Curry variants, because they have the same inhabited types (as a consequence of Theorem 3). TIP is undecidable for most extensions of $\lambda\!\!\rightarrow$, because TIP corresponds to provability in some logic and provability gets easily undecidable (e.g. already in very weak systems of predicate logic).

As a final remark: if we add a context to the problems in Definition 20, the decidability issues remain the same. For TIP, the problem is totally equivalent since

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash ? : \sigma \iff \vdash ? : \sigma_1\!\rightarrow \ldots \rightarrow\!\sigma_n\!\rightarrow\!\sigma$$

For the Church system, TSP is also totally equivalent:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M :? \iff \vdash \lambda x_1{:}\sigma_1 \ldots . \lambda x_n{:}\sigma_n.M :?$$

and similarly for TCP.

For the Curry system, the situation is slightly different, because in the TSP $\Gamma \vdash M :?$ the free variables are "forced" to be of specific types, which they are not in $\vdash \lambda \boldsymbol{x}.M :?$. Nevertheless, also if we add a context, TSP and TCP remain decidable and the principal type technique that we have described still works.

4.4 Properties of λ→; Normalization

We now list the most important meta-theoretic properties of $\lambda\!\!\rightarrow$.

Theorem 7. – *For $\lambda\!\!\rightarrow$ à la Church:* Uniqueness of types
If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
– Subject Reduction
If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : \sigma$.
– Strong Normalization
If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

These are proved using the following more basic properties of $\lambda\!\!\rightarrow$.

Proposition 1. – Substitution property
If $\Gamma, x : \tau, \Delta \vdash M : \sigma$, $\Gamma \vdash P : \tau$, then $\Gamma, \Delta \vdash M[x := P] : \sigma$.
– Thinning
If $\Gamma \vdash M : \sigma$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash M : \sigma$.

The proof of these properties proceeds by induction on the typing derivation – where we sometimes first have to prove some auxiliary Lemmas that we haven't listed here – except for the proof of Strong Normalization, which was first proved by Tait [38]. As Strong Normalization is such an interesting property and it has an interesting proof, we devote the rest of this section to it. We first study the problem of *Weak Normalization*, stating that every term has (a reduction path to) a normal form.

Definition 21. – *A λ-term M is* weakly normalizing *or WN if there is a reduction sequence starting from M that terminates.*
– *A λ-term M is* strongly normalizing *or SN if all reduction sequences starting from M terminate.*
A type system is WN if all well-typed terms are WN, and it is SN if all well-typed terms are SN.

What is the problem with normalization?

– Terms may get larger under reduction
$(\lambda f.\lambda x.f(fx))P \longrightarrow_\beta \lambda x.P(Px)$, which blows up if P is large.
– Redexes may get multiplied under reduction.
$(\lambda f.\lambda x.f(fx))((\lambda y.M)Q) \longrightarrow_\beta \lambda x.((\lambda y.M)Q)(((\lambda y.M)Q)x)$
– New redexes may be created under reduction.
$(\lambda f.\lambda x.f(fx))(\lambda y.N) \longrightarrow_\beta \lambda x.(\lambda y.N)((\lambda y.N)x)$

To prove WN, we would like to have a reduction strategy that does not create new redexes, or that makes the term shorter in every step. However, this idea is too naive and impossible to achieve. We can define a more intricate notion of "size" of a term and a special reduction strategy that decreases the size of a term at every step, but to do that we have to analyze more carefully what can happen during a reduction. We give the following Lemma about "redex creation", the proof of which is just a syntactic case analysis.

Lemma 3. *There are four ways in which "new" β-redexes can be created in a β-reduction step.*

- Creation
$$(\lambda x. \ldots (x\,P) \ldots)(\lambda y.Q) \longrightarrow_\beta \ldots (\lambda y.Q)P \ldots$$
Here we really create a new redex, by substituting a λ-abstraction for a variable that is in function position.
- Multiplication
$$(\lambda x. \ldots x \ldots x \ldots)((\lambda y.Q)R) \longrightarrow_\beta \ldots (\lambda y.Q)R \ldots (\lambda y.Q)R \ldots$$
Here we copy (possibly many times) an existing redex, thereby creating new ones.
- Hidden redex
$$(\lambda x.\lambda y.Q)R\,P \longrightarrow_\beta (\lambda y.Q[x := R])P$$
Here the redex $(\lambda y.Q)P$ was already present in a hidden form, being "shaded" by the λx; it is revealed by contracting the outer redex.
- Identity
$$(\lambda x.x)(\lambda y.Q)R \longrightarrow_\beta (\lambda y.Q)R$$
This is a different very special case of a "hidden redex": by contracting the identity, the redex $(\lambda y.Q)R$ is revealed.

We now define an appropriate size and an appropriate reduction strategy that proves weak normalization. The proof is originally due to Turing and was first written up by Gandy [17].

Definition 22. *The* height *(or order) of a type $h(\sigma)$ is defined by*

- $h(\alpha) := 0$
- $h(\sigma_1 \to \ldots \to \sigma_n \to \alpha) := max(h(\sigma_1), \ldots, h(\sigma_n)) + 1$.

The idea is that the height of a type σ is at least 1 higher than of any of the domains types occurring in σ. In the definition, we use the fact that we can write types in a "standard form" $\sigma_1 \to \ldots \to \sigma_n \to \alpha$. But it is equivalent to define h directly by induction over the definition of types, as is stated in the following exercise.

Exercise 7. Prove that the definition of h above is equivalent to defining

- $h(\alpha) := 0$
- $h(\sigma \to \tau) := \max(h(\sigma) + 1, h(\tau))$.

Definition 23. *The* height *of a redex $(\lambda x{:}\sigma.P)Q$ is the height of the type of $\lambda x{:}\sigma.P$.*

As an example, we look at the "identity" redex creation case of lemma 3. Note that the height of the redex in $(\lambda x{:}\sigma.x)(\lambda y{:}\tau.Q)R$ is $h(\sigma)+1$ and that the height of the redex in its reduct, $(\lambda y{:}\tau.Q)R$, is $h(\sigma)$. (Note that the type of $\lambda y{:}\tau.Q$) is just σ.) So the created redex has lesser height.

This will be the key idea to our reduction strategy: we will select a redex whose reduction only creates redexes of lesser height.

Definition 24. *We assign a* measure m *to the terms by defining*

$$m(N) := (h_r(N), \#N)$$

where

- $h_r(N) =$ *the maximum height of a redex in* N*,*
- $\#N =$ *the number of redexes of maximum height* $h_r(N)$ *in* N*.*

The measures of terms are ordered in the obvious *lexicographical* way:

$$(h_1, x) <_l (h_2, y) \text{ iff } h_1 < h_2 \text{ or } (h_1 = h_2 \text{ and } x < y).$$

Theorem 8 (Weak Normalization). *If* P *is a typable term in* $\lambda{\to}$*, then there is a terminating reduction starting from* P*.*

Proof. Pick a redex of maximum height $h_r(P)$ inside P that does not contain any other redex of height $hr(P)$. Note that this is always possible: If R_1 and R_2 are redexes, R_1 is contained in R_2 or the other way around. Say we have picked $(\lambda x{:}\sigma.M)N$.

Reduce this redex, to obtain $M[x := N]$. We claim that *this does not create a new redex of height* $h_r(P)$ $(\star)$. This is the important step and the proof is by analyzing the four possibilities of redex creation as they are given in Lemma 3. We leave this as an exercise.

If we write Q for the reduct of P, then, as a consequence of $(\star)$, we find that $m(Q) <_l m(P)$. As there are no infinitely decreasing $<_l$ sequences, this process must terminate and then we have arrived at a normal form.

Exercise 8. Check claim $(\star)$ in the proof of Theorem 8. (Hint: Use Lemma 3.)

Strong Normalization for $\lambda{\to}$ is proved by constructing a *model* of $\lambda{\to}$. We give the proof for $\lambda{\to}$ à la Curry. The proof is originally due to Tait [38], who proposed the interpretation of the $\to$ types as given below. Recently, combinatorial proofs have been found, that give a "measure" to a typed λ-term and then prove that this measure decreases *for all* reduction steps that are possible. See [26].

Definition 25. *The interpretation of* $\lambda{\to}$*-types is defined as follows.*

- $[\![\alpha]\!] := \mathsf{SN}$ *(the set of strongly normalizing* λ*-terms).*
- $[\![\sigma{\to}\tau]\!] := \{M \mid \forall N \in [\![\sigma]\!](MN \in [\![\tau]\!])\}$.

Note that the interpretation of a function type is countable: it is not (isomorphic to) the full function space, but it contains only the functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$ that can be λ-*defined*, i.e. are representable by a λ-term. This set is obviously countable. We have the following closure properties for $[\![\sigma]\!]$.

Lemma 4. *1.* $[\![\sigma]\!] \subseteq \mathsf{SN}$
2. $xN_1 \dots N_k \in [\![\sigma]\!]$ *for all* x, σ *and* $N_1, \dots, N_k \in \mathsf{SN}$.
3. If $M[x := N]\boldsymbol{P} \in [\![\sigma]\!]$, $N \in \mathsf{SN}$, *then* $(\lambda x.M)N\boldsymbol{P} \in [\![\sigma]\!]$.

Proof. All three parts are by induction on the structure of σ. The first two are proved simultaneously. (NB. In the case of $\sigma = \rho{\to}\tau$ for the proof of (1), we need that $[\![\rho]\!]$ is non-empty, which is guaranteed by the induction hypothesis for (2).) For (1) also use the fact that, if $M\,N \in \mathsf{SN}$, then also $M \in \mathsf{SN}$.

Exercise 9. Do the details of the proof of Lemma 4.

Proposition 2.

$$\left.\begin{array}{l} x_1{:}\tau_1, \dots, x_n{:}\tau_n \vdash M : \sigma \\ N_1 \in [\![\tau_1]\!], \dots, N_n \in [\![\tau_n]\!] \end{array}\right\} \Rightarrow M[x_1 := N_1, \dots, x_n := N_n] \in [\![\sigma]\!]$$

Proof. By induction on the derivation of $\Gamma \vdash M : \sigma$, using (3) of the Lemma 4

Corollary 1 (Strong Normalization for $\lambda{\to}$). $\lambda{\to}$ *is SN*

Proof. By taking $N_i := x_i$ in Proposition 2. (Note that $x_i \in [\![\tau_i]\!]$ by Lemma 4.) Then $M \in [\![\sigma]\!] \subseteq \mathsf{SN}$.

Exercise 10. Verify the details of the Strong Normalization proof. That is, prove Proposition 2 in detail by checking the inductive cases.

In the Strong Normalization proof, we have constructed a model that has the special nature that the interpretation of the function space is countable. If one thinks about semantics in general, one of course can also take the full set-theoretic function space as interpretation of $\sigma{\to}\tau$. We elaborate a little bit on this point, mainly as a reference for a short discussion in the Section on polymorphic λ-calculus.

We say that $\lambda{\to}$ *has a simple set-theoretic model*. Given sets $[\![\alpha]\!]$ for type variables α, define

$$[\![\sigma{\to}\tau]\!] := [\![\tau]\!]^{[\![\sigma]\!]} \ \text{(set theoretic function space } [\![\sigma]\!] \to [\![\tau]\!])$$

Now, if any of the base sets $[\![\alpha]\!]$ is infinite, then there are higher and higher infinite cardinalities among the $[\![\sigma]\!]$, because the cardinality of $[\![\sigma{\to}\tau]\!]$ is always strictly larger than that of $[\![\sigma]\!]$.

There are smaller models, e.g.

$$[\![\sigma{\to}\tau]\!] := \{f \in [\![\sigma]\!] \to [\![\tau]\!] | f \text{ is definable}\}$$

where *definability* means that it can be constructed in some formal system. This restricts the collection to a *countable* set. As an example we have seen in the SN proof the following interpretation

$$[\![\sigma{\to}\tau]\!] := \{f \in [\![\sigma]\!] \to [\![\tau]\!] | f \text{ is } \lambda\text{-definable}\}$$

The most important thing we want to note for now is that in $\lambda{\to}$ we have a lot of freedom in choosing the interpretation of the $\to$-types. In the polymorphic λ-calculus, this is no longer the case.

5 Polymorphic Type Theory

Simple type theory $\lambda{\to}$ is not very expressive: we can only define generalized polynomials as functions [37] and we don't have a clear notion of data types. Also, in simple type theory, we cannot 'reuse' a function. For example, $\lambda x{:}\alpha.x : \alpha{\to}\alpha$ and $\lambda x{:}\beta.x : \beta{\to}\beta$, which is twice the identity in slightly different syntactic form. Of course, in the Curry version we have $\lambda x.x : \alpha{\to}\alpha$ and $\lambda x.x : \beta{\to}\beta$, but then still we can't have *the same term* $\lambda x.x$ being of type $\alpha{\to}\alpha$ and of type $\beta{\to}\beta$ *at the same time*. To see what we mean with that, consider the following term that we can type

$$(\lambda y.y)(\lambda x.x)$$

In the Church version, this would read, e.g.

$$(\lambda y{:}\sigma{\to}\sigma.y)(\lambda x{:}\sigma.x)$$

which shows that we can type this term with type $\sigma{\to}\sigma$. To type the two identities with the same type at the same time, we would want to type the following (of which the term above is a one-step reduct):

$$(\lambda f.f\ f)(\lambda x.x).$$

But this term is not typable: f should be of type $\sigma{\to}\sigma$ and of type $(\sigma{\to}\sigma){\to}\sigma{\to}\sigma$ at the same time, which we can't achieve in $\lambda{\to}$.

We want to define functions that can treat types *polymorphically*. We add types of the form $\forall\alpha.\sigma$.

Examples 11.
- $\forall\alpha.\alpha{\to}\alpha$
  If $M : \forall\alpha.\alpha{\to}\alpha$, then M can map any type to itself.
- $\forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha$
  If $M : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha$, then M can take two inputs (of arbitrary types) and return a value of the first input type.

There is a weak and a strong version of polymorphism. The first is present in most functional programming languages, therefore also called *ML style polymorphism*. The second allows more types and is more immediate if one takes a logical view on types. We first treat the weak version.

5.1 Typed λ-calculus with weakly polymorphic types

Definition 26. *In* weak $\lambda 2$ *(the system with* weak polymorphism*) we have the following additional types*

$$\mathsf{Typ}_w := \forall\alpha.\mathsf{Typ}_w | \mathsf{Typ}$$

where Typ *is the collection of* $\lambda\!\!\rightarrow$*-types as defined in Definition 1.*

So, the weak polymorphic types are obtained by adding $\forall\alpha_1.\ldots\forall\alpha_n.\sigma$ for σ a $\lambda\!\!\rightarrow$-type.

We can formulate polymorphic λ-calculus in Church and in Curry style. As for $\lambda\!\!\rightarrow$, the two systems are different in the type information that occurs in the terms, but now the difference is larger: in polymorphic λ-calculus we also have abstractions over types.

Definition 27. *The terms of* weak $\lambda 2$ *à la Church are defined by*

$$\Lambda_2^{ch} ::= Var \mid (\Lambda_2^{ch}\ \Lambda_2^{ch}) \mid (\lambda\, Var{:}\mathsf{Typ}.\Lambda_2^{ch}) \mid (\lambda\mathsf{TVar}.\Lambda_2^{ch}) \mid \Lambda_2^{ch}\ \mathsf{Typ}$$

The terms of the Curry version of the calculus are of course just Λ. This means that in the Curry version we will not record the abstractions over type variables. This is made precise in the following rules.

Definition 28. *The* Derivation rules *for* weak $\lambda 2$ *(ML-style polymorphism) in Church style are as follows*

$$\frac{x:\sigma\in\Gamma}{\Gamma\vdash x:\sigma}\ \frac{\Gamma,x:\sigma\vdash M:\tau}{\Gamma\vdash\lambda x{:}\sigma.M:\sigma\!\rightarrow\!\tau}\ \textit{if}\ \sigma,\tau\in\mathsf{Typ}\ \frac{\Gamma\vdash M:\sigma\!\rightarrow\!\tau\quad\Gamma\vdash N:\sigma}{\Gamma\vdash M\,N:\tau}$$

$$\frac{\Gamma\vdash M:\sigma}{\Gamma\vdash\lambda\alpha.M:\forall\alpha.\sigma}\ \alpha\notin FV(\Gamma)\quad\frac{\Gamma\vdash M:\forall\alpha.\sigma}{\Gamma\vdash M\tau:\sigma[\alpha:=\tau]}\ \textit{if}\ \tau\in\mathsf{Typ}$$

Definition 29. *The* derivation rules *for* weak $\lambda 2$ *(ML-style polymorphism) in Curry style are as follows.*

$$\frac{x:\sigma\in\Gamma}{\Gamma\vdash x:\sigma}\ \frac{\Gamma,x:\sigma\vdash M:\tau}{\Gamma\vdash\lambda x.M:\sigma\!\rightarrow\!\tau}\ \textit{if}\ \sigma,\tau\in\mathsf{Typ}\ \frac{\Gamma\vdash M:\sigma\!\rightarrow\!\tau\quad\Gamma\vdash N:\sigma}{\Gamma\vdash M\,N:\tau}$$

$$\frac{\Gamma\vdash M:\sigma}{\Gamma\vdash M:\forall\alpha.\sigma}\ \alpha\notin FV(\Gamma)\quad\frac{\Gamma\vdash M:\forall\alpha.\sigma}{\Gamma\vdash M:\sigma[\alpha:=\tau]}\ \textit{if}\ \tau\in\mathsf{Typ}$$

Examples 12. 1. In $\lambda 2$ à la Curry: $\lambda x.\lambda y.x:\forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$.
2. In $\lambda 2$ à la Church we have the following, which is the same term as in the previous case, but now with type information added: $\lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$.
3. In $\lambda 2$ à la Curry: $z:\forall\alpha.\alpha\!\rightarrow\!\alpha\vdash z\,z:\forall\alpha.\alpha\!\rightarrow\!\alpha$.

4. In $\lambda 2$ à la Church, we can annotate this term with type information to obtain: $z : \forall\alpha.\alpha{\to}\alpha \vdash \lambda\alpha.z\,(\alpha{\to}\alpha)\,(z\,\alpha) : \forall\alpha.\alpha{\to}\alpha$.
5. We do *not* have $\vdash \lambda z.z\,z : \ldots$

We can make flag deduction rules for this system as follows.

$$\begin{array}{l|l|l} 1 & \left|\begin{array}{l} x:\sigma \\ \hline \ldots \end{array}\right. & 1 \mid \ldots \\ 2 & & 2 \mid \ldots \\ 3 & \left|\begin{array}{l} \ldots \end{array}\right. & 3 \mid M : \sigma \to \tau \\ 4 & \left|\begin{array}{l} M:\tau \end{array}\right. & 4 \mid \ldots \\ 5 & \lambda x{:}\sigma.M : \sigma \to \tau & 5 \mid \ldots \\ & & 6 \mid N : \sigma \\ & & 7 \mid \ldots \\ & & 8 \mid M\,N : \tau \end{array}$$

if $\sigma, \tau \in \mathsf{Typ}$

$$\begin{array}{l|l|l} 1 & \left|\begin{array}{l} \alpha \\ \hline \ldots \end{array}\right. & 1 \mid \ldots \\ 2 & & 2 \mid \ldots \\ 3 & \left|\begin{array}{l} \ldots \end{array}\right. & 3 \mid M : \forall\alpha.\sigma \\ 4 & \left|\begin{array}{l} M:\sigma \end{array}\right. & 4 \mid \ldots \\ 5 & \lambda\alpha.M : \forall\alpha.\sigma & 5 \mid \ldots \\ & & 6 \mid M\,\tau : \sigma[\alpha := \tau] \end{array}$$

if α fresh

The "freshness" condition in the rule means that α should *not occur free above the flag*. In terms of contexts, this means precisely that the type variable that we intend to abstract over does not occur in the context Γ. The freshness condition excludes the following wrong flag deduction.

$$\text{WRONG}\quad \begin{array}{l|l} 1 & \left|\begin{array}{l} f : \alpha{\to}\beta \\ \hline \ldots \end{array}\right. \\ 2 & \left|\begin{array}{l} \alpha \\ \hline \ldots \end{array}\right. \\ 3 & \left|\begin{array}{l} x:\alpha \\ \hline \ldots \end{array}\right. \\ 4 & \left|\begin{array}{l} f\,x : \beta \end{array}\right. \\ 5 & \lambda x.f\,x : \alpha{\to}\beta \\ 6 & \lambda x.f\,x : \forall\alpha.\alpha{\to}\beta \end{array}$$

Of course, we should not be able to derive

$$f : \alpha{\to}\beta \vdash \lambda x.f\,x : \forall\alpha.\alpha{\to}\beta$$

Examples 13. Here are the first four examples of 12, now with flag deductions. In the first row we find the derivations in the Church systems, in the second row

the ones in the Curry system.

$$
\begin{array}{l|l}
1 & \alpha \\
\hline
2 & \quad \beta \\
\hline
3 & \quad\quad x : \alpha \\
\hline
4 & \quad\quad\quad y : \beta \\
\hline
5 & \quad\quad \lambda y{:}\beta.x : \beta{\to}\alpha \\
6 & \quad \lambda x{:}\alpha.\lambda y{:}\beta.x : \alpha{\to}\beta{\to}\alpha \\
7 & \quad \lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\beta.\alpha{\to}\beta{\to}\alpha \\
8 & \lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha
\end{array}
\qquad
\begin{array}{l|l}
1 & z : \forall\alpha.\alpha{\to}\alpha \\
\hline
2 & \quad \alpha \\
\hline
3 & \quad z\,\alpha : \alpha{\to}\alpha \\
4 & \quad z\,(\alpha{\to}\alpha) : (\alpha{\to}\alpha){\to}\alpha{\to}\alpha \\
5 & \quad z\,(\alpha{\to}\alpha)(z\,\alpha) : \alpha{\to}\alpha \\
6 & \lambda\alpha.z\,(\alpha{\to}\alpha)(z\,\alpha) : \forall\alpha.\alpha{\to}\alpha
\end{array}
$$

$$
\begin{array}{l|l}
1 & \alpha \\
\hline
2 & \quad \beta \\
\hline
3 & \quad\quad x : \alpha \\
\hline
4 & \quad\quad\quad y : \beta \\
\hline
5 & \quad\quad \lambda y.x : \beta{\to}\alpha \\
6 & \quad \lambda x.\lambda y.x : \alpha{\to}\beta{\to}\alpha \\
7 & \quad \lambda x.\lambda y.x : \forall\beta.\alpha{\to}\beta{\to}\alpha \\
8 & \lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha
\end{array}
\qquad
\begin{array}{l|l}
1 & z : \forall\alpha.\alpha{\to}\alpha \\
\hline
2 & \quad \alpha \\
\hline
3 & \quad z : \alpha{\to}\alpha \\
4 & \quad z : (\alpha{\to}\alpha){\to}\alpha{\to}\alpha \\
5 & \quad z\,z : \alpha{\to}\alpha \\
6 & z\,z : \forall\alpha.\alpha{\to}\alpha
\end{array}
$$

In the types, $\forall$ only occurs on the outside. Therefore, in programming languages (that usually follow a Curry style typing discipline) it is usually left out and all type variables are *implicitly universally quantified over*. This means that one doesn't write the $\forall$, so the types are restricted to Typ and hence we don't have the last two rules. That any type variable can be instantiated with a type, is then made formal by changing the variable rule into

$$\frac{}{\Gamma, x{:}\forall\boldsymbol{\alpha}.\sigma \vdash x : \sigma[\boldsymbol{\alpha} := \boldsymbol{\tau}]} \text{ if } \boldsymbol{\tau} \subseteq \mathsf{Typ}$$

This should be read as that σ doesn't contain any $\forall$ anymore. So we have universal types only in the context (as types of declared variables) and only at this place we can instantiate the $\forall\alpha$ with types, which must be simple types. It can be proven that this system is equivalent to $\lambda 2$ à la Curry in the following sense. Denote by $\vdash_v$ the variant of the weak $\lambda 2$ rules à la Curry where we don't have rules for $\forall$ and the adapted rule for variables just described. Then

$$\Gamma \vdash_v M : \sigma \Rightarrow \Gamma \vdash M : \sigma$$
$$\Gamma \vdash M : \forall\boldsymbol{\alpha}.\sigma \Rightarrow \Gamma \vdash_v M : \sigma[\boldsymbol{\alpha} := \boldsymbol{\tau}]$$

This is proved by induction on the derivation, using a Substitution Lemma that holds for all type systems that we have seen so far and that we therefore state here in general.

Lemma 5 (Substitution for types). *If $\Gamma \vdash M : \sigma$, then $\Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$, for all systems à la Curry defined so far.*
For the Church systems we have types in the terms. Then the Substitution Lemma states: If $\Gamma \vdash M : \sigma$, then $\Gamma[\alpha := \tau] \vdash M[\alpha := \tau] : \sigma[\alpha := \tau]$.

For all systems, this Lemma is proved by a straightforward induction over the derivation.

With weak polymorphism, type checking is still decidable: the principal types algorithm can be extended to incorporate *type schemes*: types of the form $\forall \boldsymbol{\alpha}.\sigma$. We have observed that weak polymorphism allows terms to have many (polymorphic) types, but we cannot abstract over variables of these types. This is allowed with *full polymorphism*, also called *system F style polymorphism.*

5.2 Typed λ-calculus with full polymorphism

Definition 30. *The types of* $\lambda 2$ *with* full (system F-style) polymorphism *are*

$$\mathsf{Typ}_2 := \mathsf{TVar} \mid (\mathsf{Typ}_2 \to \mathsf{Typ}_2) \mid \forall \alpha.\mathsf{Typ}_2$$

1. *The derivation rules for* $\lambda 2$ *with* full (system F-style) polymorphism *in Curry style are as follows. (Note that σ and τ range over Typ_2.)*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha.\sigma}\ \alpha \notin FV(\Gamma) \quad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]}$$

2. *The derivation rules for* $\lambda 2$ *with* full (system F-style) polymorphism *in Church style are as follows. (Again, note that σ and τ range over Typ_2.)*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau} \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda \alpha.M : \forall \alpha.\sigma}\ \alpha \notin FV(\Gamma) \quad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]}$$

So now, $\forall$ can also occur deeper in a type. We can write flag deduction rules for the full $\lambda 2$ in the obvious way, for both the Curry and the Church variant of the system. We now give some examples that are only valid with full polymorphism.

Examples 14. – $\lambda 2$ à la Curry: $\lambda x.\lambda y.x : (\forall \alpha.\alpha) \to \sigma \to \tau$.
– $\lambda 2$ à la Church: $\lambda x{:}(\forall \alpha.\alpha).\lambda y{:}\sigma.x\tau : (\forall \alpha.\alpha) \to \sigma \to \tau$.

Here are the flag deductions that prove the two typings in the Examples.

1	$x : \forall\alpha.\alpha$
2	$\quad y : \sigma$
3	$\quad x\,\tau : \tau$
4	$\lambda y{:}\sigma.x\,\tau : \sigma\to\tau$
5	$\lambda x{:}\forall\alpha.\alpha.\lambda y{:}\sigma.x\,\tau : (\forall\alpha.\alpha)\to\sigma\to\tau$

1	$x : \forall\alpha.\alpha$
2	$\quad y : \sigma$
3	$\quad x : \tau$
4	$\lambda y.x : \sigma\to\tau$
5	$\lambda x.\lambda y.x : (\forall\alpha.\alpha)\to\sigma\to\tau$

In $\lambda 2$ we use the following abbreviations for types: $\bot := \forall\alpha.\alpha$, $\top := \forall\alpha.\alpha\to\alpha$. The names are derived from the behavior of the types. From a term of type $\bot$, we can create a term of any type: $\lambda x{:}\bot.x\sigma : \bot\to\sigma$ for any type σ. So $\bot$ is in some sense the "smallest type". Also, $\bot$ is empty: there is no closed term of type $\bot$. On the other hand, $\top$ is the type with one canonical closed element: $\lambda\alpha.\lambda x{:}\alpha.x : \top$. We can now also type a term like $\lambda x.x\,x$.

Examples 15.
- In Curry $\lambda 2$: $\lambda x.xx : \bot\to\bot$, $\lambda x.xx : \top\to\top$
- In Church $\lambda 2$: $\lambda x{:}\bot.x(\bot\to\bot)x : \bot\to\bot$, $\lambda x{:}\top.x\,\top\,x : \top\to\top$.
- In Church $\lambda 2$: $\lambda x{:}\bot.\lambda\alpha.x(\alpha\to\alpha)(x\alpha) : \bot\to\bot$, $\lambda x{:}\top.\lambda\alpha.x(\alpha\to\alpha)(x\alpha) : \top\to\top$.

We show two typings in the previous example by a flag deduction.

1	$x : \bot$
2	$\quad \alpha$
3	$\quad x : \alpha\to\alpha$
4	$\quad x : \alpha$
5	$\quad x\,x : \alpha$
6	$x\,x : \forall\alpha.\alpha$
7	$\lambda x{:}\bot.x\,x : \bot\to\bot$

1	$x : \top$
2	$\quad x\,\top : \top\to\top$
3	$\quad x\,\top\,x : \top$
4	$\lambda x{:}\top.x\,\top\,x : \top\to\top$

Exercises 11.
1. Verify using a flag deduction that in Church $\lambda 2$: $\lambda x{:}\top.\lambda\alpha.x(\alpha\to\alpha)(x\alpha) : \top\to\top$.
2. Verify using a flag deduction that in Curry $\lambda 2$: $\lambda x.xx : \top\to\top$
3. Find a type in Curry $\lambda 2$ for $\lambda x.x\,x\,x$
4. Find a type in Curry $\lambda 2$ for $\lambda x.x\,x\,(x\,x)$

With full polymorphism, type checking becomes undecidable [41] for the Curry version of the system. For the Church version it is clearly still decidable, as we have all necessary type information in the term.

Definition 31. *We define the* erasure map *from $\lambda 2$ à la Church to $\lambda 2$ à la Curry as follows.*

$$\begin{array}{llll} |x| & := x & & \\ |\lambda x{:}\sigma.M| & := |\lambda x.M| & |\lambda\alpha.M| & := |M| \\ |MN| & := |M|\,|N| & |M\sigma| & := |M| \end{array}$$

We have the following proposition about this erasure map, that relates the Curry and Church systems to each other.

Proposition 3. *If $\Gamma \vdash M : \sigma$ in $\lambda 2$ à la Church, then $\Gamma \vdash |M| : \sigma$ in $\lambda 2$ à la Curry.*
If $\Gamma \vdash P : \sigma$ in $\lambda 2$ à la Curry, then there is an M such that $|M| \equiv P$ and $\Gamma \vdash M : \sigma$ in $\lambda 2$ à la Church.

The proof is by straightforward induction on the derivations. We don't give the details. In the Examples of 12, 14 and 15, we can see the Proposition at work: an erasure of the Church style derivation gives a Curry style derivation. The other way around: any Curry style derivation can be "dressed up" with type information to obtain a Church style derivation. The undecidability of type checking in $\lambda 2$ à la Curry can thus be rephrased as: we cannot algorithmically reconstruct the missing type information in an untyped term. In Example 15 we have seen two completely different ways to "dress up" the term $\lambda x.x\,x$ to make it typable in $\lambda 2$-Church. This is a general pattern: there are many possible ways to add typing information to an non-typable term to make it typable in $\lambda 2$-Church.

We have opposed "weak polymorphism" to "full polymorphism" and we referred to the first also as ML-style polymorphism. It is the case that polymorphism in most functional programming languages is weaker than full polymorphism, and if we restrict ourselves to the pure λ-calculus, it is just the weak polymorphic system that we have described. However, to regain some of the "full polymorphism", ML has additional constructs, like *let polymorphism*.

$$\frac{\Gamma \vdash M : \sigma \;\; \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \mathsf{let}\, x = M \,\mathsf{in}\, N : \tau} \text{ for } \tau \text{ a } \lambda{\to}\text{-type}, \sigma \text{ a } \lambda 2\text{-type}$$

We can see a term $\mathsf{let}\, x = M \,\mathsf{in}\, N$ as a β-redex $(\lambda x{:}\sigma.N)M$. So the let-rule allows the formation of a β-redex $(\lambda x{:}\sigma.N)M$, for σ a polymorphic type, while we cannot form the abstraction term $\lambda x{:}\sigma.N : \sigma{\to}\tau$. So it is still impossible to have a universal quantifier deeper inside a type, though we can have a polymorphic term as a subterm of a well-typed term. For the extension of the principal type algorithm to include weak polymorphism with lets we refer to [30].

Exercise 12. 1. Type the term $(\lambda f.f\,f)(\lambda x.x)$ in (full) $\lambda 2$-Curry.
2. Type the term $\mathsf{let}\, f = \lambda x.x \,\mathsf{in}\, f\,f$ in weak $\lambda 2$-Curry with the let-rule as given above.

5.3 Meta-theoretic Properties

We now recall the decidability issues of Definition 20 and look into them for $\lambda 2$.

Theorem 9. *Decidability properties of the (weak and full) polymorphic λ-calculus*

- *TIP is decidable for the weak polymorphic λ-calculus, undecidable for the full polymorphic λ-calculus.*
- *TCP and TSP are equivalent.*

– *TCP*	*à la Church*	*à la Curry*
ML-style	*decidable*	*decidable*
System F-style	*decidable*	*undecidable*

With full polymorphism (system F), untyped terms contain too little information to compute the type [41]. TIP is equivalent to provability in logic. For full $\lambda 2$, this is *second order intuitionistic proposition logic* (to be discussed in the next Section), which is known to be undecidable. For weak $\lambda 2$, the logic is just a very weak extension of PROP which is decidable.

5.4 Formulas-as-types for full $\lambda 2$

There is a formulas-as-types isomorphism between full system-F style $\lambda 2$ and second order proposition logic, PROP2.

Definition 32. *Derivation rules of* PROP2*:*

$$\frac{\begin{array}{c}\tau_1 \dots \tau_n \\ \vdots \\ \sigma\end{array}}{\forall\alpha.\sigma}\ \forall\text{-}I,\ \textit{if}\ \alpha \notin FV(\tau_1,\dots,\tau_n) \qquad \frac{\forall\alpha.\sigma}{\sigma[\alpha := \tau]}$$

NB This is constructive second order proposition logic: *Peirce's law*

$$\forall\alpha.\forall\beta.((\alpha{\to}\beta){\to}\alpha){\to}\alpha$$

is not derivable (so there is no closed term of this type). The logic only has implication and universal quantification, but the other connectives are now definable.

Definition 33. *Definability of the other intuitionistic connectives.*

$$\begin{aligned}\bot &:= \forall\alpha.\alpha\\ \sigma{\wedge}\tau &:= \forall\alpha.(\sigma{\to}\tau{\to}\alpha){\to}\alpha\\ \sigma{\vee}\tau &:= \forall\alpha.(\sigma{\to}\alpha){\to}(\tau{\to}\alpha){\to}\alpha\\ \exists\alpha.\sigma &:= \forall\beta.(\forall\alpha.\sigma{\to}\beta){\to}\beta\end{aligned}$$

Proposition 4. *All the standard constructive deduction rules (elimination and introduction rules) are derivable using the definitions in 33*

Example 16. We show the derivability of the $\wedge$-elimination rule by showing how to derive σ from $\sigma \wedge \tau$:

$$\frac{\dfrac{\forall\alpha.(\sigma{\to}\tau{\to}\alpha){\to}\alpha}{(\sigma{\to}\tau{\to}\sigma){\to}\sigma} \quad \dfrac{\dfrac{[\sigma]^1}{\tau{\to}\sigma}}{\sigma{\to}\tau{\to}\sigma}\,1}{\sigma}$$

There is a formulas-as-types embedding of PROP2 into $\lambda 2$ that maps deductions to terms. It can be defined inductively, as we have done for PROP and $\lambda\!\!\rightarrow$ in Definition 10. We don't give the definitions, but illustrate it by an example.

Example 17. The definable $\wedge$-elimination rule in PROP2 under the deductions-as-terms embedding yields a λ-terms that "witnesses" the construction of an object of type σ out of an object of type $\sigma \wedge \tau$.

$$\dfrac{\dfrac{M : \forall\alpha.(\sigma\!\rightarrow\!\tau\!\rightarrow\!\alpha)\!\rightarrow\!\alpha}{M\sigma : (\sigma\!\rightarrow\!\tau\!\rightarrow\!\sigma)\!\rightarrow\!\sigma} \quad \dfrac{\dfrac{[x:\sigma]^1}{\lambda y{:}\tau.x : \tau\!\rightarrow\!\sigma}}{\lambda x{:}\sigma.\lambda y{:}\tau.x : \sigma\!\rightarrow\!\tau\!\rightarrow\!\sigma}\,1}{M\sigma(\lambda x{:}\sigma.\lambda y{:}\tau.x) : \sigma}$$

So the following term is a "witness" for the $\wedge$-elimination.

$$\lambda z{:}\sigma\!\wedge\!\tau.z\,\sigma\,(\lambda x{:}\sigma.\lambda y{:}\tau.x) : (\sigma \wedge \tau)\!\rightarrow\!\sigma$$

Exercise 13. Prove the derivability of some of the other logical rules:

1. Define $\mathsf{inl} : \sigma \rightarrow \sigma \vee \tau$
2. Define pairing : $\langle -, - \rangle : \sigma \rightarrow \tau \rightarrow \sigma \times \tau$
3. Given $f : \sigma\!\rightarrow\!\rho$ and $g : \tau\!\rightarrow\!\rho$, construct a term $\mathsf{case}\ f\ g : \sigma \vee \tau\!\rightarrow\!\rho$

5.5 Data types in $\lambda 2$

In $\lambda\!\!\rightarrow$ we can define a type of "natural numbers over a type σ": $(\sigma\!\rightarrow\!\sigma)\!\rightarrow\!\sigma\!\rightarrow\!\sigma$. In $\lambda 2$, we can define this type polymorphically.

$$\mathsf{Nat} := \forall\alpha.(\alpha\!\rightarrow\!\alpha)\!\rightarrow\!\alpha\!\rightarrow\!\alpha$$

This type uses the encoding of natural numbers as Church numerals

$$n \mapsto c_n := \lambda f.\lambda x.f(\ldots(fx)) \quad n\text{-times } f$$

- $0 := \lambda\alpha.\lambda f{:}\alpha\!\rightarrow\!\alpha.\lambda x{:}\alpha.x$
- $S := \lambda n{:}\mathsf{Nat}.\lambda\alpha.\lambda f{:}\alpha\!\rightarrow\!\alpha.\lambda x{:}\alpha.f(n\,\alpha\,x\,f)$

Proposition 5. *Over the type* Nat*, functions can be defined by* iteration*: if $c : \sigma$ and $g : \sigma\!\rightarrow\!\sigma$, then there is a function*

$$\mathsf{It}\,c\,g : \mathsf{Nat}\!\rightarrow\!\sigma$$

satisfying

$$\mathsf{It}\,c\,g\,0 = c$$
$$\mathsf{It}\,c\,g\,(S\,x) = g(\mathsf{It}\,c\,g\,x)$$

Proof. $\mathsf{It}\,c\,g$: $\mathsf{Nat}\!\rightarrow\!\sigma$ is defined as $\lambda n{:}\mathsf{Nat}.n\,\sigma\,g\,c$. It is left as a (quite ease) exercise to see that this satisfies the equations.

The function It acts as an iterator: it takes a "begin value" c and a map g and n times iterates g on c, where n is the natural number input. So $\mathsf{It}\,c\,g\,n = g(\ldots(g\,c))$, with n times g.

Examples 18. 1. Addition

$$\mathsf{Plus} := \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.\mathsf{It}\,m\,S\,n$$

or if we unfold the definition of It: $\mathsf{Plus} := \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.n\,\mathsf{Nat}\,S\,m$, which says: iterate the $+1$ function n times on begin value m.

2. Multiplication

$$\mathsf{Mult} := \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.\mathsf{It}\,0\,(\lambda x{:}\mathsf{Nat}.\mathsf{Plus}\,m\,x)\,n$$

The predecessor is notably difficult to define! The easiest way to define it is by first defining *primitive recursion* on the natural numbers. This means that if we have $c : \sigma$ and $f : \mathsf{Nat}{\to}\sigma{\to}\sigma$, we want to define a term $\mathsf{Rec}\,c\,f : \mathsf{Nat}{\to}\sigma$ satisfying

$$\begin{aligned}\mathsf{Rec}\,c\,f\,0 &= c\\ \mathsf{Rec}\,c\,f\,(S\,x) &= f\,x\,(\mathsf{Rec}\,c\,f\,x)\end{aligned}$$

(Note that if we can define functions by primitive recursion, the predecessor is just $P := \mathsf{Rec}\,0\,(\lambda x\,y : \mathsf{Nat}.x)$.)

It is known that primitive recursion can be encoded in terms of iteration, so therefore we can define the predecessor in $\lambda 2$. However, the complexity (in terms of the number of reduction steps) of this encoding is very bad. As a consequence:

$$\mathsf{Pred}(n+1) \longrightarrow\!\!\!\!\!\rightarrow_\beta n$$

in a number of steps of $O(n)$.

Exercise 14. 1. Complete the details in the proof of Proposition 5

2. Verify in detail that addition and multiplication as defined in Example 18 behave as expected.
3. Define the data type $\mathsf{Three} := \forall\alpha{:}*.\alpha{\to}\alpha{\to}\alpha{\to}\alpha$.
   (a) Give three different closed inhabitants of the type Three in $\lambda 2$ à la Church: $\mathsf{one}, \mathsf{two}, \mathsf{three} : \mathsf{Three}$.
   (b) Define a function $\mathsf{Shift} : \mathsf{Three} \to \mathsf{Three}$ that does the following

$$\begin{aligned}\mathsf{Shift}\,\mathsf{one} &=_\beta \mathsf{two}\\ \mathsf{Shift}\,\mathsf{two} &=_\beta \mathsf{three}\\ \mathsf{Shift}\,\mathsf{three} &=_\beta \mathsf{one}\end{aligned}$$

Apart from the natural numbers, many other algebraic data types are definable in $\lambda 2$. Here is the example of lists over a base type A.

$$\mathsf{List}_A := \forall\alpha.\alpha{\to}(A{\to}\alpha{\to}\alpha){\to}\alpha$$

The representation of lists over A as terms of this type uses the following encoding

$$[a_1, a_2, \ldots, a_n] \mapsto \lambda x.\lambda f. f a_1 (f a_2 (\ldots (f a_n x))) \quad n\text{-times } f$$

We can now define the constructors Nil (empty list) and Cons (to "cons" an element to a list) as follows.

- $\mathsf{Nil} := \lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha.x$
- $\mathsf{Cons} := \lambda a{:}A.\lambda l{:}\mathsf{List}_A.\lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha. f\, a (l\, \alpha\, x\, f)$

Note that the definition of Cons conforms with the representation of lists given above.

Proposition 6. *Over the type* List_A *we can define functions by* iteration*: if* $c : \sigma$ *and* $g : A{\to}\sigma{\to}\sigma$*, then there is a function*

$$\mathsf{It}\, c\, g : \mathsf{List}_A{\to}\sigma$$

satisfying

$$\mathsf{It}\, c\, g\, \mathsf{Nil} = c$$
$$\mathsf{It}\, c\, g\, (\mathsf{Cons}\, a\, l) = g\, a\, (\mathsf{It}\, c\, g\, l)$$

Proof. $\mathsf{It}\, c\, g : \mathsf{List}_A{\to}\sigma$ is defined as $\lambda l{:}\mathsf{List}_A. l\, \sigma\, c\, g$. This satisfies the equations in the proposition. Basically, we have, for $l = [a_1, \ldots, a_n]$, $\mathsf{It}\, c\, g\, l = g\, a_1 (\ldots (g\, a_n\, c))$ (n times g).

Example 19. A standard function one wants to define over lists is the "map" function, which given a function on the carrier type, extends it to a function on the lists over this carrier type. Given $f : \sigma{\to}\tau$, $\mathsf{Map}\, f : \mathsf{List}_\sigma{\to}\mathsf{List}_\tau$ should apply f to all elements in a list. It is defined by

$$\mathsf{Map} := \lambda f{:}\sigma{\to}\tau.\mathsf{It}\, \mathsf{Nil} (\lambda x{:}\sigma.\lambda l{:}\mathsf{List}_\tau.\mathsf{Cons}(f\, x) l).$$

Then

$$\begin{aligned} \mathsf{Map}\, f\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{Map}\, f\, (\mathsf{Cons}\, a\, k) &= \mathsf{It}\, \mathsf{Nil} (\lambda x{:}\sigma.\lambda l{:}\mathsf{List}_\tau.\mathsf{Cons}(f\, x) l)\, (\mathsf{Cons}\, a\, k) \\ &= (\lambda x{:}\sigma.\lambda l{:}\mathsf{List}_\tau.\mathsf{Cons}(f\, x) l) a (\mathsf{Map}\, f\, k) \\ &= \mathsf{Cons}(f\, a)(\mathsf{Map}\, f\, k) \end{aligned}$$

This is exactly the recursion equation for Map that we would expect.

Many more data-types can be defined in $\lambda 2$. The product of two data-types is defined in the same way as the (logical) conjunction: $\sigma{\times}\tau := \forall\alpha.(\sigma{\to}\tau{\to}\alpha){\to}\alpha$. we have already seen how to define projections and pairing (the $\wedge$-elimination

and $\wedge$-introduction rules). The disjoint union (or sum) of two data-types is defined in the same way as the logical disjunction: $\sigma{+}\tau := \forall\alpha.(\sigma{\to}\alpha){\to}(\tau{\to}\alpha){\to}\alpha$. We can define inl, inr and case. The type of binary trees with nodes in A and leaves in B can be defined as follows.

$$\mathsf{Tree}_{A,B} := \forall\alpha.(B{\to}\alpha){\to}(A{\to}\alpha{\to}\alpha{\to}\alpha){\to}\alpha$$

and we define $\mathsf{leaf} : B{\to}\mathsf{Tree}_{A,B}$ by $\mathsf{leaf} := \lambda b{:}B.\lambda\alpha.\lambda l{:}B{\to}\alpha.\lambda j{:}A{\to}\alpha{\to}\alpha{\to}\alpha.\, l\, b$.

Exercises 15.
- Define a function of type $\mathsf{List}_A{\to}\mathsf{Nat}$ that computes the length of a list.
- Define $\mathsf{join} : \mathsf{Tree}_{A,B} \to \mathsf{Tree}_{A,B} \to A \to \mathsf{Tree}_{A,B}$ that takes two trees and a node label and builds a tree.
- Define a function $\mathsf{TreeSum} : \mathsf{Tree}_{\mathsf{Nat},\mathsf{Nat}}{\to}\mathsf{Nat}$ that computes the sum of all leaves and nodes in a binary tree.
- Give the *iteration scheme* over binary trees and show that it is definable in $\lambda 2$.

5.6 Meta-theory of $\lambda 2$; Strong Normalization

Theorem 10.
- *For $\lambda 2$ à la Church: Uniqueness of types*
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
- *Subject Reduction*
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.
- *Strong Normalization*
  If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

The third property is remarkably complicated and we address it in detail below. The first two are proved by induction on the derivation, using some other meta-theoretic properties, that one also proves by induction over the derivation (the Substitution Property we have already seen for types; this is the same property for terms):

Proposition 7.
- Substitution property
  If $\Gamma, x : \tau, \Delta \vdash M : \sigma$, $\Gamma \vdash P : \tau$, then $\Gamma, \Delta \vdash M[x := P] : \sigma$.
- Thinning
  If $\Gamma \vdash M : \sigma$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash M : \sigma$.

We now elaborate on the proof of Strong Normalization of β-reduction for $\lambda 2$. The proof is an extension of the Tait proof of SN for $\lambda{\to}$, but it needs some crucial extra ingredients that have been developed by Girard [21]. We motivate these additional notions below.

In $\lambda 2$ à la Church there are two kinds of β-reductions:

- Kind 1, term-applied-to-term: $(\lambda x{:}\sigma.M)P \longrightarrow_\beta M[x := P]$
- Kind 2, term-applied-to-type: $(\lambda\alpha.M)\tau \longrightarrow_\beta M[\alpha := \tau]$

The second kind of reductions does no harm: (i) there are no infinite β-reduction paths with solely reductions of kind 2; (ii) if $M \longrightarrow_\beta N$ with a reduction of kind 2, then $|M| \equiv |N|$. So, if there is an infinite β-reduction in $\lambda 2$-Church, then there is one in $\lambda 2$-Curry and we are done if we prove SN for $\lambda 2$ à la Curry.

Recall the model construction in the proof for $\lambda\!\!\rightarrow$:

- $[\![\alpha]\!] := \mathsf{SN}$.
- $[\![\sigma\!\rightarrow\!\tau]\!] := \{M \mid \forall N \in [\![\sigma]\!](MN \in [\![\tau]\!])\}$.

So, now the question is: How to define $[\![\forall\alpha.\sigma]\!]$? A natural guess would be to define $[\![\forall\alpha.\sigma]\!] := \mathit{\Pi}_{X\in U}[\![\sigma]\!]_{\alpha:=X}$, where U is some set capturing all possible interpretations of types. This $\mathit{\Pi}$-set is a set of functions that take an element X of U (an interpretation of a type) and yield an element of the interpretation of σ where we assign X to α.

But now the problem is that $\mathit{\Pi}_{X\in U}[\![\sigma]\!]_{\alpha:=X}$ gets too big: if there is any type with more than one element (something we would require for a model), then $\mathrm{card}(\mathit{\Pi}_{X\in U}[\![\sigma]\!]_{\alpha:=X}) > \mathrm{card}(U)$. The cardinality of the interpretation of $\forall\alpha.\sigma$ would be larger than the set it is a member of and that's impossible. So we cannot interpret the $\forall$ as a $\mathit{\Pi}$-set (or a union for the same reason).

Girard has given the solution to this problem: $[\![\forall\alpha.\sigma]\!]$ should be very small:

$$[\![\forall\alpha.\sigma]\!] := \bigcap_{X\in U} [\![\sigma]\!]_{\alpha:=X}$$

This conforms with the idea that $\forall\alpha.\sigma$ is the type of terms that act *parametrically* on a type. A lot of literature has been devoted to the nature of polymorphism, for which the terminology *parametricity* has been introduced, intuitively saying that a function operates on types *without looking into them*. So a parametric function cannot act differently on Nat and Bool. This implies, e.g. that there is only one parametric function from α to α: the identity. See [1] for more on parametricity.

The second important novelty of Girard is the actual definition of U, the collection of all *possible interpretations* of types. U will be defined as SAT, the collection of *saturated sets* of (untyped) λ-terms.

Definition 34. *$X \subset \mathit{\Lambda}$ is* saturated *if*

- $xP_1 \dots P_n \in X$ *(for all $x \in \mathsf{Var}$, $P_1, \dots, P_n \in \mathsf{SN}$)*
- $X \subseteq \mathsf{SN}$
- *If $M[x := N]\boldsymbol{P} \in X$ and $N \in \mathsf{SN}$, then $(\lambda x.M)N\boldsymbol{P} \in X$.*

The definition of saturated sets basically arises by taking the closure properties that we *proved* for the interpretation of $\lambda\!\!\rightarrow$-types as a *definition* of the collection of possible interpretation of $\lambda 2$-types.

Definition 35. *Let $\rho : \mathsf{TVar} \to SAT$ be a* valuation *of type variables. Define $[\![\sigma]\!]_\rho$ by:*

- $[\![\alpha]\!]_\rho := \rho(\alpha)$
- $[\![\sigma{\to}\tau]\!]_\rho := \{M | \forall N \in [\![\sigma]\!]_\rho (MN \in [\![\tau]\!]_\rho)\}$
- $[\![\forall\alpha.\sigma]\!]_\rho := \cap_{X \in SAT} [\![\sigma]\!]_{\rho,\alpha:=X}$

Proposition 8.

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \sigma \Rightarrow M[x_1 := P_1, \ldots, x_n : P_n] \in [\![\sigma]\!]_\rho$$

for all valuations ρ *and* $P_1 \in [\![\tau_1]\!]_\rho, \ldots, P_n \in [\![\tau_n]\!]_\rho$

The proof is by induction on the derivation of $\Gamma \vdash M : \sigma$.

Corollary 2. $\lambda 2$ *is SN*

Proof. Take P_1 to be x_1, ..., P_n to be x_n.

Exercise 16. Verify the details of the proof of the Proposition.

We end this section with some remarks on semantics. In the section on $\lambda{\to}$ we have seen that the SN proof consists of the construction of a model, where the interpretation of the function type is "small" (not the full function space). But for $\lambda{\to}$, there are also models where the function type is the full set-theoretic function space. These are often referred to as *set-theoretical models* of type theory.

Theorem 11 (Reynolds[36]). $\lambda 2$ *does not have a non-trivial set-theoretic model.*

This is a remarkable theorem, also because there are no requirements for the interpretation of the other constructs, only that the model is sound. The proof proceeds by showing that if $[\![\sigma{\to}\tau]\!] := [\![\tau]\!]^{[\![\sigma]\!]}$ (the set theoretic function space), then $[\![\sigma]\!]$ is a singleton set for every σ. We call such a model *trivial*, as all types are interpreted as the empty set or the singleton set. This is a sound interpretation, corresponding with the interpretation of the formulas of PROP2 in a classical way (suing a truth table semantics). It is also called the *proof irrelevance semantics* as all proofs of propositions are identified. As said, it is a sound model, but not a very interesting one, certainly from a programmers point of view, because all natural numbers are identified.

So we can rephrase Reynolds result as: in an interesting $\lambda 2$-model, $[\![\sigma{\to}\tau]\!]$ must be small.

6 Dependent Type Theory

In the paper by Bove and Dybjer, we can see "Dependent Types at Work" and that paper also gives some of the history and intuitions behind it. In this Section I will present the rules. The problem with dependent types is that "everything depends on everything", so we can't first define the types and then the terms. We will have two "universes": **type** and **kind**. Dybjer and Bove used "Set" for

what I call **type**: the universe of types. (We can't have **type** : **type**, so to type **type** we need to have another universe: **type** : **kind**.)

we define first order dependent type theory, $\lambda\mathsf{P}$. This system is also known as LF (Logical Framework, [22]) The judgements of the system are of the form

$$\Gamma \vdash M : B$$

where

- Γ is a context
- M and B are terms taken from the set of *pseudo-terms*.

Definition 36. *The set of pseudo-terms is defined by*

$$\mathsf{T} ::= \mathsf{Var} \mid \mathbf{type} \mid \mathbf{kind} \mid (\mathsf{TT}) \mid (\lambda x{:}\mathsf{T}.\mathsf{T}) \mid \Pi x{:}\mathsf{T}.\mathsf{T},$$

Furthermore, there is an *auxiliary* judgement

$$\Gamma \vdash$$

to denote that Γ is a *correct context*.

Definition 37. *The* derivation rules *of* $\lambda\mathsf{P}$ *are (*$\mathbf{s}$ *ranges over* $\{\mathbf{type}, \mathbf{kind}\}$*):*

$$(\mathit{base})\ \emptyset \vdash \quad (\mathit{ctxt})\ \frac{\Gamma \vdash A : \mathbf{s}}{\Gamma, x{:}A \vdash}\ \text{if } x \text{ not in } \Gamma \quad (\mathit{ax})\ \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{type} : \mathbf{kind}}$$

$$(\mathit{proj})\ \frac{\Gamma \vdash}{\Gamma \vdash x : A}\ \text{if } x{:}A \in \Gamma \quad (\Pi)\ \frac{\Gamma, x{:}A \vdash B : \mathbf{s} \ \ \Gamma \vdash A : \mathbf{type}}{\Gamma \vdash \Pi x{:}A.B : \mathbf{s}}$$

$$(\lambda)\ \frac{\Gamma, x{:}A \vdash M : B \ \ \Gamma \vdash \Pi x{:}A.B : \mathbf{s}}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\ (\mathit{app})\ \frac{\Gamma \vdash M : \Pi x{:}A.B \ \ \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$(\mathit{conv})\ \frac{\Gamma \vdash M : B \ \ \Gamma \vdash A : \mathbf{s}}{\Gamma \vdash M : A} A =_{\beta\eta} B$$

In this type theory, we have a new phenomenon, which is the Π*-type*:

$$\Pi x{:}A.B(x) \simeq \text{the type of functions } f \text{ such that}$$
$$f\,a : B(a) \text{ for all } a{:}A$$

The Π-type is a generalization of the well-known function type: if $x \notin \mathrm{FV}(B)$, then $\Pi x{:}A.B$ is just $A{\to}B$. So, we will use the arrow notation $A{\to}B$ as an abbreviation for $\Pi x{:}A.B$ in case $x \notin \mathrm{FV}(B)$. The Π rule allows to form two forms of function types in $\lambda\mathsf{P}$

$$(\Pi)\ \frac{\Gamma \vdash A : \mathbf{type} \ \ \Gamma, x{:}A \vdash B : \mathbf{s}}{\Gamma \vdash \Pi x{:}A.B : \mathbf{s}}$$

- With $\mathbf{s} = \mathbf{kind}$, we can form $A{\to}A{\to}\mathbf{type}$ and $A{\to}\mathbf{type}$.
- With $\mathbf{s} = \mathbf{type}$ and $P : A{\to}\mathbf{type}$, we can form $A{\to}A$ and $\Pi x{:}A.P\,x{\to}P\,x$.

6.1 Formulas-as-types: minimal predicate logic into λP

Following the methods for $\lambda\!\!\rightarrow$ and $\lambda 2$, we can embed logic into first order dependent type theory following the Curry-Howard formulas-as-types embedding. For $\lambda\mathsf{P}$, we can embed *minimal first order predicate logic*, the predicate logic with just implication and universal quantification and the intuitionistic rules for these connectives. The idea is to represent *both the domains and the formulas* of the logic as types. In predicate logic we need to interpret a *signature* for the logic, that tells us which constants, functions and relations there are (and, in case of many-sortedness, which are the domains). This is not difficult but it involves some overhead when giving a precise formal definition. So, we don't give a completely formal definition but just an example.

Example 20. Minimal first order predicate logic over one domain with one constant, one unary function and two unary and one binary relation is embedded into $\lambda\mathsf{P}$ by considering the context

$$\Gamma := A : \mathbf{type}, a : A, f : A{\rightarrow}A, P : A{\rightarrow}\mathbf{type}, Q : A{\rightarrow}\mathbf{type}, R : A{\rightarrow}A{\rightarrow}\mathbf{type}.$$

Implication is represented as $\rightarrow$ and $\forall$ is represented as Π:

$$\forall x{:}A.P\,x \mapsto \Pi x{:}A.P\,x$$
$$\forall x{:}A.R\,x\,x{\rightarrow}P\,x \mapsto \Pi x{:}A.R\,x\,x{\rightarrow}P\,x$$

the intro and elim rules are just λ-abstraction and application, both for implication and universal quantification.

The terms of type A act as the first order terms of the language: a, $f\,a$, $f(f\,a)$ etc. The formulas are encoded as terms of type **type**: $P\,a$, $R\,a\,a$ are the closed atomic formulas and with $\rightarrow$, Π and variables we build the first order formulas from that.

In $\lambda\mathsf{P}$, we can give a precise derivation that the context Γ is correct: $\Gamma \vdash$. These derivations are quite lengthy, because in a derivation tree the same judgment is derived several times in different branches of the tree. Therefore such derivations are best given in *flag style*. It should be clear by now how we turn a set of derivations rules into a flag format. We will usually omit derivations of the correctness of a context, but for completeness we here give one example, in flag format. We give a precise derivation of the judgment

$$A : \mathbf{type}, P : A{\rightarrow}\mathbf{type}, a : A \vdash P\,a{\rightarrow}\mathbf{type} : \mathbf{kind}$$

NB. we use the $\rightarrow$-formation rule as a degenerate case of the Π-formation rule (if $x \notin \mathrm{FV}(B)$).

$$\frac{\Gamma \vdash A : \mathbf{type}\ \ \Gamma \vdash B : \mathbf{s}}{\Gamma \vdash A \rightarrow B : \mathbf{s}}\ {\rightarrow}\text{-form}$$

1	$\mathbf{type} : \mathbf{kind}$	
2	$A : \mathbf{type}$	ctxt-proj, 1
3	$A \to \mathbf{type} : \mathbf{kind}$	$\to$-form, 2, 1
4	$P : A \to \mathbf{type}$	ctxt-proj, 3
5	$a : A$	ctxt-proj, 2
6	$P\,a : \mathbf{type}$	app, 4, 5
7	$P\,a \to \mathbf{type} : \mathbf{kind}$	$\to$-form, 6, 1

Example 21. We illustrate the use of application and abstraction to encode elimination and introduction rules of the logic. take Γ to be the context of Example 20.

$$\Gamma \vdash \lambda z{:}A.\lambda h{:}(\Pi x, y{:}A.R\,x\,y).h\,z\,z : \Pi z{:}A.(\Pi x, y{:}A.R\,x\,y){\to}R\,z\,z$$

This term is a proof of $\forall z{:}A.(\forall x, y{:}A.R(x, y)){\to}R(z, z)$. The first λ encodes a $\forall$-introduction, the second λ an implication-introduction.

Example 22. We now show how to construct a term of type $(\Pi x{:}A.P\,x{\to}Q\,x){\to}(\Pi x{:}A.P\,x){\to}\Pi x{:}A.Q\,x$ in the context Γ. We do this by giving a derivation in "flag style", where we omit derivations of the well-formedness of types and contexts. We write σ for $(\Pi x{:}A.P\,x{\to}Q\,x){\to}(\Pi x{:}A.P\,x){\to}\Pi x{:}A.Q\,x$.

1	$A : \mathbf{type}$	
2	$P : A \to \mathbf{type}$	
3	$Q : A \to \mathbf{type}$	
4	$h : \Pi x{:}A.P\,x{\to}Q\,x$	
5	$g : \Pi x{:}A.P\,x$	
6	$x : A$	
7	$h\,x : P\,x \to Q\,x$	app, 4, 6
8	$g\,x : P\,x$	app, 5, 6
9	$h\,x(g\,x) : Q\,x$	app, 7, 8
10	$\lambda x{:}A.h\,x(g\,x) : \Pi x{:}A.Q\,x$	λ-rule, 6, 9
11	$\lambda g{:}\Pi x{:}A.P\,x.\lambda x{:}A.h\,x(g\,x) : (\Pi x{:}A.P\,x) \to \Pi x{:}A.Q\,x$	λ-rule, 5, 10
12	$\lambda h{:}\Pi x{:}A.P\,x{\to}Q\,x.\lambda g{:}\Pi x{:}A.P\,x.\lambda x{:}A.h\,x(g\,x) : \sigma$	λ-rule, 4, 11

So:

$$\Gamma \vdash \lambda h{:}\Pi x{:}A.P\,x{\to}Q\,x.\lambda g{:}\Pi x{:}A.P\,x.\lambda x{:}A.h\,x(g\,x) : \sigma$$

Exercise 17. 1. Find terms of the following types (NB $\to$ binds strongest)

$$(\Pi x{:}A.P\,x{\to}Q\,x){\to}(\Pi x{:}A.P\,x){\to}\Pi x{:}A.Q\,x$$

and

$$(\Pi x{:}A.P\,x{\to}\Pi z.R\,z\,z){\to}(\Pi x{:}A.P\,x){\to}\Pi z{:}A.R\,z\,z).$$

2. Find a term of the following type and write down the context in which this term is typed.

$$(\Pi x{:}A.P\,x{\to}Q){\to}(\Pi x{:}A.P\,x){\to}Q$$

What is special about your context? (It should somehow explicitly state that the type A is not empty.)

The representation that we have just described is called the *direct encoding* of logic in type theory. This is the formulas-as-types embedding originally due to Curry and Howard and described first in formal detail in [25]. Apart from this, there is the *LF encoding* of logic in type theory. This is the formulas-as-types embedding as it was invented by De Bruijn in his Automath project [31]. We describe it now.

6.2 LF embedding of logic in type theory

For $\lambda{\to}$, $\lambda 2$ and $\lambda\mathsf{P}$ we have seen *direct* representations of logic in type theory. Characteristics of such an encoding are:

- Connectives each have a counterpart in the type theory:

implication	$\sim$ $\to$-type
universal quantification	$\sim$ $\forall$-type

- Logical rules have their direct counterpart in type theory:

$\to$-introduction	$\sim$ λ-abstraction
$\to$- elimination	$\sim$ application
$\forall$-introduction	$\sim$ λ-abstraction
$\forall$-elimination	$\sim$ application

- the context declares a *signature*, *local variables* and *assumptions*.

There is another way of interpreting logic in type theory, due to De Bruijn, which we call the *logical framework* representation of logic in type theory. The idea is to use type theory as a framework in which various logics can be encoded by choosing an appropriate context. Characteristics of the LF encoding are:

- Type theory is used as a *meta system* for encoding ones own logic.
- The context is used as a *signature for the logic*: one chooses an appropriate context Γ_L in which the logic L (including its proof rules) is declared.
- The type system is a meta-calculus for dealing with *substitution* and *binding*.

We can put these two embeddings side by side by looking at the trivial proof of A implies A.

	proof	formula
direct embedding	$\lambda x{:}A.x$	$A{\to}A$
LF embedding	$\mathsf{imp\_intr}\,A\,A\,\lambda x{:}T\,A.x$	$T(A \Rightarrow A)$

For the LF embedding of minimal proposition logic into $\lambda\mathsf{P}$, we need the following context.

$$\begin{aligned} \Rightarrow &: \mathsf{prop}\to\mathsf{prop}\to\mathsf{prop} \\ \mathsf{T} &: \mathsf{prop}\to\mathbf{type} \\ \mathsf{imp\_intr} &: (A,B:\mathsf{prop})(\mathsf{T}\,A\to\mathsf{T}\,B)\to\mathsf{T}(A\Rightarrow B) \\ \mathsf{imp\_el} &: (A,B:\mathsf{prop})\mathsf{T}(A\Rightarrow B)\to\mathsf{T}\,A\to\mathsf{T}\,B. \end{aligned}$$

The idea is that prop is the type of names of propositions and that T "lifts" a name φ to the type of its proofs $T\,\varphi$. The terms $\mathsf{imp\_intr}$ and $\mathsf{imp\_el}$ encode the introduction and elimination for implication.

Exercise 18. Verify that $\mathsf{imp\_intr}\,A\,A\,\lambda x{:}T\,A.x : T(A\Rightarrow A)$ in the context just described.

In the following table we summarize the difference between the two encodings

Direct embedding	LF embedding
One type system : One logic	One type system : Many logics
Logical rules $\sim$ type theoretic rules	Logical rules $\sim$ context declarations

Apart from this, a direct embedding aims at describing a formulas–as-types *isomorphism* between the logic and the type theory, whereas the LF idea is to provide a system for enabling a formulas–as–types embedding for many different logics. For $\lambda\!\to$ and $\lambda 2$ there is indeed a one-one correspondence between deductions in logic and typable terms in the type theory. For the case of $\lambda\mathsf{P}$ and minimal predicate logic, this is not so obvious, as we have identified the domains and the formulas completely: they are all of type **type**. This gives rise to types of the form $\Pi x{:}A.P\,x\to A$ and allows to form predicates over formulas, like $B : (\Pi x{:}A.R\,x\,x)\to\mathbf{type}$, that don't have a correspondence in the logic. It can nevertheless be shown that the direct embedding of PRED into $\lambda\mathsf{P}$ is complete, but that requires some effort. See [18] for details.

Now, we show some examples of logics in the logical framework LF – which is just $\lambda\mathsf{P}$. Then we exhibit the properties of LF that make this work.

Minimal propositional logic in $\lambda\mathsf{P}$ Fix the signature (context) of minimal propositional logic.

$$\begin{aligned} \mathsf{prop} &: \mathbf{type} \\ \mathsf{imp} &: \mathsf{prop}\to\mathsf{prop}\to\mathsf{prop} \end{aligned}$$

As a notation we introduce

$$A\Rightarrow B \text{ for } \mathsf{imp}\,A\,B$$

The type prop is the type of 'names' of propositions. A term of type prop can not be inhabited (proved), as it is not a type. We 'lift' a name $p : \mathsf{prop}$ to the type of its proofs by introducing the following map:

$$\mathsf{T} : \mathsf{prop}\to\mathbf{type}.$$

The intended meaning of $\mathsf{T}p$ is 'the type of proofs of p'. We interpret 'p is valid' by '$\mathsf{T}p$ is inhabited'. To derive $\mathsf{T}p$ we also encode the logical derivation rules by adding to the context

$$\begin{aligned}\mathsf{imp\_intr} &: \Pi p, q : \mathsf{prop}.(\mathsf{T}p \to \mathsf{T}q) \to \mathsf{T}(p \Rightarrow q),\\ \mathsf{imp\_el} &: \Pi p, q : \mathsf{prop}.\mathsf{T}(p \Rightarrow q) \to \mathsf{T}p \to \mathsf{T}q.\end{aligned}$$

$\mathsf{imp\_intr}$ takes two (names of) propositions p and q and a term $f : \mathsf{T}\,p \to \mathsf{T}\,q$ and returns a term of type $\mathsf{T}(p \Rightarrow q)$ Indeed $A \Rightarrow A$ is now valid: $\mathsf{imp\_intr}A\,A(\lambda x{:}\mathsf{T}\,A.x) : \mathsf{T}(A \Rightarrow A)$

Exercise 19. Construct a term of type $\mathsf{T}(A \Rightarrow (B \Rightarrow A))$ in the context with $A, B : \mathsf{prop}$.

Definition 38. *Define Σ_{PROP} to be the signature for minimal proposition logic, PROP, as just constructed.*

Now, why would this be a "good" encoding? Are all derivations represented as terms in $\lambda\mathsf{P}$? And if a type is inhabited, is the associated formula then provable? We have the following desired properties of the encoding.

Definition 39. – Soundness *of the encoding states that*

$$\vdash_{PROP} A \Rightarrow \Sigma_{PROP}, a_1{:}\mathsf{prop}, \ldots, a_n{:}\mathsf{prop} \vdash p : \mathsf{T}\,A \text{ for some } p.$$

where $\{a_, \ldots, a_n\}$ is the set of proposition variables in A.
– Adequacy *(or completeness) states the converse:*

$$\Sigma_{PROP}, a_1{:}\mathsf{prop}, \ldots, a_n{:}\mathsf{prop} \vdash p : \mathsf{T}\,A \Rightarrow \vdash_{PROP} A$$

Proposition 9. *The LF encoding of* PROP *in $\lambda\mathsf{P}$ is sound and adequate.*

The proof of soundness is by induction on the derivation of $\vdash_{\mathrm{PROP}} A$. Adequacy also holds, but it is more involved to prove. One needs to define a canonical form of terms of type $\mathsf{T}\,A$ (the so called *long $\beta\eta$-normal-form*) and show that these are in one-one correspondence with proofs. See [22] for details.

Minimal predicate logic over one domain A in $\lambda\mathsf{P}$ Signature:

$$\begin{aligned}\mathsf{prop} &: \mathbf{type},\\ A &: \mathbf{type},\\ \mathsf{T} &: \mathsf{prop} \to \mathbf{type}\\ \mathsf{f} &: A \to A,\\ \mathsf{R} &: A \to A \to \mathsf{prop},\\ \Rightarrow &: \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop},\\ \mathsf{imp\_intr} &: \Pi p, q : \mathsf{prop}.(\mathsf{T}p \to \mathsf{T}q) \to \mathsf{T}(p \Rightarrow q),\\ \mathsf{imp\_el} &: \Pi p, q : \mathsf{prop}.\mathsf{T}(p \Rightarrow q) \to \mathsf{T}p \to \mathsf{T}q.\end{aligned}$$

Now we encode $\forall$ by observing that $\forall$ takes a $P : A{\to}\mathsf{prop}$ and returns a proposition, so:

$$\forall : (A{\to}\mathsf{prop}){\to}\mathsf{prop}$$

Universal quantification $\forall x{:}A.(Px)$ is then translated by $\forall(\lambda x{:}A.(Px))$

Definition 40. *The signature: Σ_{PRED} is defined by adding to the above the following intro and elim rules for $\forall$.*

$$\begin{aligned}\forall &: (A{\to}\mathsf{prop}){\to}\mathsf{prop},\\ \forall\_\mathsf{intr} &: \Pi P{:}A{\to}\mathsf{prop}.(\Pi x{:}A.\mathsf{T}(Px)){\to}\mathsf{T}(\forall P),\\ \forall\_\mathsf{elim} &: \Pi P{:}A{\to}\mathsf{prop}.\mathsf{T}(\forall P){\to}\Pi x{:}A.\mathsf{T}(Px).\end{aligned}$$

The proof of

$$\forall z{:}A(\forall x, y{:}A.Rxy) \Rightarrow Rzz$$

is now mirrored by the proof-term

$$\begin{aligned}\forall\_\mathsf{intr}[\_](\ &\lambda z{:}A.\mathsf{imp\_intr}[\_][\_](\lambda h{:}\mathsf{T}(\forall x, y{:}A.Rxy).\\ &\forall\_\mathsf{elim}[\_](\forall\_\mathsf{elim}[\_]hz)z) \ )\end{aligned}$$

For readability, we have replaced the instantiations of the Π-type by $[\_]$.
This term is of type

$$\mathsf{T}(\forall(\lambda z{:}A.\mathsf{imp}(\forall(\lambda x{:}A.(\forall(\lambda y{:}A.\mathsf{R}xy))))(\mathsf{R}zz)))$$

Exercise 20. Construct a proof-term that mirrors the (obvious) proof of $\forall x(P\,x \Rightarrow Q\,x) \Rightarrow \forall x.P\,x \Rightarrow \forall x.Q\,x$

Proposition 10. *We have soundness and adequacy for minimal predicate logic:*

$$\vdash_{\mathrm{PRED}} \varphi \Rightarrow \Sigma_{\mathrm{PRED}}, x_1{:}A, \ldots, x_n{:}A \vdash p : \mathsf{T}\varphi, \text{ for some } p,$$

where $\{x_1, \ldots, x_n\}$ is the set of free variables in φ.

$$\Sigma_{\mathrm{PRED}}, x_1{:}A, \ldots, x_n{:}A \vdash p : \mathsf{T}\varphi \Rightarrow \vdash_{\mathrm{PRED}} \varphi$$

6.3 Meta-theory of λP

Proposition 11.
- *Uniqueness of types*
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma{=_\beta}\tau$.
- *Subject Reduction*
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : \sigma$.
- *Strong Normalization*
  If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

The proofs are by induction on the derivation, by first proving auxiliary lemmas like Substitution and Thinning. SN can be proved by defining a reduction preserving map from $\lambda\mathsf{P}$ to $\lambda{\to}$. Then, an infinite reduction path in $\lambda\mathsf{P}$ would give rise to an infinite reduction path in $\lambda{\to}$, so SN for $\lambda\mathsf{P}$ follows from SN for $\lambda{\to}$. See [22] for details. We now come back to the decidability questions of Definition 20.

Proposition 12. *For* $\lambda\mathsf{P}$*:*

- *TIP is undecidable*
- *TCP/TSP is decidable*

The undecidability of TIP follows from the fact that provability in minimal predicate logic is undecidable. A more straightforward proof is given in [7], by interpreting the halting problem for register machines as a typing problem (in a specific context) in $\lambda\mathsf{P}$.

We will expand on the decidability of TCP below. It is shown by defining two algorithms simultaneously: one that does type synthesis $\Gamma \vdash M :?$ and one that does *context checking*: $\Gamma \vdash ?$. This is to mirror the two forms of judgment in $\lambda\mathsf{P}$.

Remark 1. One can also introduce a *Curry variant* of $\lambda\mathsf{P}$. This is done in [2]. A related issue is whether one can type an *untyped* λ-term in $\lambda\mathsf{P}$. So, given an M, is there a context Γ, a type A and a term P such that $\Gamma \vdash P : A$ and $|P| \equiv M$. Here, $|-|$ is the erasure map defined by

$$\begin{array}{ll} |x| & := x \\ |\lambda x{:}\sigma.M| := |\lambda x.M| & |MN| := |M|\,|N| \end{array}$$

The answer to this question is yes, because an untyped term is $\lambda\mathsf{P}$-typable iff it is typable in $\lambda{\to}$. But there is a little snag: if we *fix* the context Γ, the problem becomes undecidable, as was shown in [14], where Dowek gives a context Γ and a term M such that $\exists P, A(\Gamma \vdash P : A \wedge |P| = M)$ is equivalent to the Post correspondence problem.

6.4 Type Checking for $\lambda\mathsf{P}$

We define algorithms Ok$(-)$ and Type_$(-)$ simultaneously:

- Ok$(-)$ takes a context and returns 'accept' or 'reject'
- Type_$(-)$ takes a context and a term and returns a term or 'reject'.

Definition 41. *The type synthesis algorithm* Type_$(-)$ *is* sound *if*

$$\mathrm{Type}_\Gamma(M) = A \;\Rightarrow\; \Gamma \vdash M : A \textit{ (for all } \Gamma \textit{ and } M\textit{)}$$

The type synthesis algorithm Type_$(-)$ *is* complete *if*

$$\Gamma \vdash M : A \;\Rightarrow\; \mathrm{Type}_\Gamma(M) =_\beta A \textit{ (for all } \Gamma \textit{ and } M\textit{)}$$

Completeness only makes sense if we have uniqueness of types: only then it makes sense to check if the type that is given to us is convertible to the type computed by Type. In case we don't have uniqueness of types, one would let Type_$(-)$ compute a set of possible types, one for each β-equivalence class.

Definition 42.

$$
\begin{aligned}
\mathrm{Ok}(<>) &= \textit{`accept'}\\
\mathrm{Ok}(\Gamma, x{:}A) &= \mathsf{if}\ \mathrm{Type}_\Gamma(A) \in \{\mathbf{type}, \mathbf{kind}\}\ \mathsf{then}\ \mathrm{Type}_\Gamma(A)\ \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(x) &= \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{and}\ x{:}A \in \Gamma\ \mathsf{then}\ A\ \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(\mathbf{type}) &= \mathsf{if}\ \mathrm{Ok}(\Gamma)\ \mathsf{then}\ \mathbf{kind}\ \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(MN) &= \mathsf{if}\ \mathrm{Type}_\Gamma(M) = C\ \mathsf{and}\ \mathrm{Type}_\Gamma(N) = D\\
&\quad \mathsf{then}\ \ \mathsf{if}\ C \twoheadrightarrow_\beta \Pi x{:}A.B\ \mathsf{and}\ A =_\beta D\\
&\qquad \mathsf{then}\ B[x := N]\ \mathsf{else}\ \textit{`reject'}\\
&\quad \mathsf{else}\ \ \textit{`reject'},\\
\mathrm{Type}_\Gamma(\lambda x{:}A.M) &= \mathsf{if}\ \mathrm{Type}_{\Gamma, x:A}(M) = B\\
&\quad \mathsf{then}\qquad \mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathbf{type}, \mathbf{kind}\}\\
&\qquad\qquad \mathsf{then}\ \Pi x{:}A.B\ \mathsf{else}\ \textit{`reject'}\\
&\quad \mathsf{else}\ \textit{`reject'},\\
\mathrm{Type}_\Gamma(\Pi x{:}A.B) &= \mathsf{if}\ \mathrm{Type}_\Gamma(A) = \mathbf{type}\ \mathsf{and}\ \mathrm{Type}_{\Gamma, x:A}(B) = s\\
&\quad \mathsf{then}\ s\ \mathsf{else}\ \textit{`reject'}
\end{aligned}
$$

Proposition 13. *The type checking algorithm is sound:*

$$
\begin{aligned}
\mathrm{Type}_\Gamma(M) = A &\Rightarrow \Gamma \vdash M : A\\
\mathrm{Ok}(\Gamma) = \textit{`accept'} &\Rightarrow \Gamma \vdash
\end{aligned}
$$

The proof is by simultaneous induction on the computation of Type and Ok.

For completeness, we need to prove the following simultaneously, which we would prove by induction on the derivation.

$$
\begin{aligned}
\Gamma \vdash A : \mathbf{s} &\Rightarrow \mathrm{Type}_\Gamma(A) = \mathbf{s}\\
\Gamma \vdash M : A &\Rightarrow \mathrm{Type}_\Gamma(M) =_\beta A\\
\Gamma \vdash &\Rightarrow \mathrm{Ok}(\Gamma) = \text{`accept'}
\end{aligned}
$$

The first slight strengthening of completeness is not a problem: in case the type of A is **type** or **kind**, $\mathrm{Type}_\Gamma(A)$ returns exactly **type** or **kind** (and not a term $=_\beta$-equal to it). The problem is the λ-rule, where $\mathrm{Type}_{\Gamma, x:A}(M) = C$ and $C =_\beta B$ and we know that $\mathrm{Type}_\Gamma(\Pi x{:}A.B) = \mathbf{s}$, but we need to know that $\mathrm{Type}_\Gamma(\Pi x{:}A.C) = \mathbf{s}$, because that is the side condition in the Type algorithm for the $\lambda x{:}A.M$ case.

The solution is to change the definition of Type a little bit. This is motivated by the following Lemma, which is specific to the type theory $\lambda\mathsf{P}$.

Lemma 6. *The derivable judgements of $\lambda\mathsf{P}$ remain exactly the same if we replace the λ-rule by*

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

The proof is by induction on the derivation. Now we can in the λ-case of the definition of Type replace the side condition

$$\mathsf{if}\ \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathbf{type}, \mathbf{kind}\}$$

by

$$\mathsf{if}\ \mathrm{Type}_\Gamma(A) \in \{\mathbf{type}\}$$

Definition 43. *We adapt the definition of* Type *in Definition 42 by replacing the λ-abstraction case by*

$$\begin{array}{rl} \mathrm{Type}_\Gamma(\lambda x{:}A.M) = & \mathsf{if}\ \mathrm{Type}_{\Gamma,x:A}(M) = B \\ & \quad \mathsf{then} \quad \mathsf{if}\ \mathrm{Type}_\Gamma(A) = \mathbf{type} \\ & \qquad\qquad \mathsf{then}\ \Pi x{:}A.B\ \mathsf{else}\ \textit{`reject'} \\ & \quad \mathsf{else}\ \textit{`reject'}, \end{array}$$

Then soundness still holds and we have the following.

Proposition 14.

$$\begin{aligned} \Gamma \vdash A : \mathbf{s} &\Rightarrow \mathrm{Type}_\Gamma(A) = \mathbf{s} \\ \Gamma \vdash M : A &\Rightarrow \mathrm{Type}_\Gamma(M) =_\beta A \\ \Gamma \vdash &\Rightarrow \mathrm{Ok}(\Gamma) = \textit{`accept'} \end{aligned}$$

As a consequence of soundness and completeness we find that

$$\mathrm{Type}_\Gamma(M) = \text{`reject'} \quad \Rightarrow \quad M \text{ is not typable in } \Gamma$$

Completeness implies that Type terminates correctly on *all well-typed terms*. But we want that Type terminates on *all pseudo terms*: we want to assure that on a non-typable term, Type returns 'reject', which is not guaranteed by Soundness and Completeness.

To prove that Type_(−) terminates on all inputs, we need to make sure that $\mathrm{Type}_\Gamma(M)$ and $\mathrm{Ok}(\Gamma)$ are called on arguments of lesser size. As "size" we take the sum of the lengths of the context Γ and the term M, and then all cases are decreasing, apart from λ-abstraction and application. In the λ-abstraction case, Type is called on a pseudo-term $\Pi x{:}A.B$ that is not necessarily smaller. But our replacement of the side condition in Type for the λ-abstraction case in Definition 43 solves this problem.

In the case of application, the function Type is called on smaller inputs, but the algorithms requires β-equality and β-reduction checking:

$$\begin{array}{rl} \mathrm{Type}_\Gamma(MN) = & \mathsf{if}\ \mathrm{Type}_\Gamma(M) = C\ \mathsf{and}\ \mathrm{Type}_\Gamma(N) = D \\ & \quad \mathsf{then}\ \ \mathsf{if}\ C \longrightarrow\!\!\!\!\!\rightarrow_\beta \Pi x{:}A.B\ \mathsf{and}\ A =_\beta D \\ & \qquad\quad \mathsf{then}\ B[x := N]\ \mathsf{else}\ \text{`reject'} \\ & \quad \mathsf{else}\ \ \text{`reject'}, \end{array}$$

So, we need to decide β-reduction and β-equality, which, for pseudo-terms is undecidable. The solution is that Type will only check β-equality (and reduction) for *well-typed* terms, and we know that $\lambda\mathsf{P}$ is SN and CR, so this is decidable.

Proposition 15. *Termination of* Type *and* Ok*: For all pseudo-terms M and pseudo-contexts Γ,* $\mathrm{Type}_\Gamma(M)$ *terminates (in either a pseudo-term A or 'reject') and* $\mathrm{Ok}(\Gamma)$ *terminates (in either 'accept' or 'reject').*

The proof is by induction on the size of the inputs, using the Soundness (Proposition 13) and decidability of β-equality for well-typed terms in the application case.

7 Conclusion and further reading

In this paper we have introduced various type theories by focussing on the Curry-Howard formulas-as-types embedding and by highlighting some of the programming aspects related to type theory. As stated in the Introduction, we could have presented the type systems à la Church in a unified framework of the "λ-cube" or "Pure Type Systems", but for expository reasons we have refrained from doing so. In the PTS framework, one can study the systems $F\omega$, a higher order extension of system F, $\lambda\mathsf{HOL}$, a type systems for higher order (predicate) logic, and the Calculus of Constructions, a higher order extension of $\lambda\mathsf{P}$. Also one can generalize these systems to the logically inconsistent systems λU and $\lambda\star$ where **type** is itself a type. (These systems are computationally not inconsistent so still interesting to study.) We refer to [4, 5] and the references in those papers for further reading.

In another direction, there are various typing constructs that can be added to make the theories more powerful. The most prominent one is the addition of a scheme for inductive types, which also adds an induction and a well-founded recursion principle for every inductive type. Examples are the Calculus of Inductive Constructions (CIC) and Martin-Löf type theory. The latter is introduced in the paper by Bove and Dybjer and a good reference is [32]. CIC is the type theory implemented in the proof assistant Coq and a good reference for both the theory and the tool is [6].

In the direction of programming, there is a world of literature and a good starting point is [33]. We have already mentioned PCF [35], which is the simplest way to turn $\lambda\!\!\rightarrow$ into a Turing complete language. The next thing to add are algebraic data types, which are the programmer's analogue to inductive types. Using these one can define and compute with data types of lists, trees etc. as a primitive in the language (as opposed to coding them in $\lambda 2$ as we have done in Section 5.5). Other important concepts to add, especially to the type systems à la Curry, are overloading and subtyping. A good reference for further reading is [33].

References

1. E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott, Functorial polymorphism. *Theoretical Computer Science* 70, 35-64, 1990.

2. S. van Bakel, L. Liquori, S. Ronchi Della Rocca and P. Urzyczyn, Comparing Cubes of Typed and Type Assignment Systems. *Ann. Pure Appl. Logic* 86(3): 267-303, 1997.
3. H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, vol. 103 of Studies in Logic and the Foundations of Mathematics, North Holland, 1981.
4. H. Barendregt and H. Geuvers, Proof Assistants using Dependent Type Systems, Chapter 18 of the *Handbook of Automated Reasoning* (Vol 2), eds. A. Robinson and A. Voronkov, Elsevier, pp. 1149 – 1238, 2001.
5. H. Barendregt, Lambda Calculi with Types, *Handbook of Logic in Computer Science*, Volume 1, Abramsky, Gabbay, Maibaum (Eds.), Clarendon, 1992.
6. Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art : the Calculus of Inductive Constructions*, EATCS Series: Texts in Theoretical Computer Science, Springer 2004, 469 pp.
7. M. Bezem and J. Springintveld, A Simple Proof of the Undecidability of Inhabitation in λP. *J. Funct. Program.* 6(5): 757-761, 1996.
8. C. Böhm and A. Berarducci, Automatic synthesis of typed λ-programs on term algebras, *Theoretical Computer Science* 39, pp. 135154, 1985.
9. A. Church, A formulation of the simple theory of types, *Journal of Symbolic Logic* 5, pp. 5668, 1940.
10. R.L Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki and S.F. Smith, *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, NJ, 1986.
11. H.B. Curry, R. Feys, and W. Craig, *Combinatory Logic*, Vol. 1. North–Holland, Amsterdam, 1958.
12. L. Damas and R. Milner, Principal type-schemes for functional programs, *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207-212, ACM, 1982.
13. M. Davis, editor, *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems And Computable Functions*, Raven Press, New York, 1965.
14. G. Dowek, The Undecidability of Typability in the Lambda-Pi-Calculus *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, LNCS 664, pp. 139 - 145, 1993.
15. G. Dowek, Collections, sets and types, *Mathematical Structures in Computer Science* 9, pp. 1-15, 1999.
16. F. Fitch, *Symbolic Logic, An Introduction*, The Ronald Press Company, 1952.
17. R.O. Gandy, An early proof of normalization by A.M. Turing. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 453455. Academic Press, 1980.
18. H. Geuvers and E. Barendsen, Some logical and syntactical observations concerning the first order dependent type system λP, *Math. Struc. in Comp. Sci.* vol. 9-4, 1999, pp. 335 – 360.
19. J.-Y. Girard, P. Taylor, and Y. Lafont, *Proofs and types*. Cambridge tracts in theoretical computer science, no. 7. Cambridge University Press.
20. J.-Y. Girard, The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159192, 1986.
21. J.-Y. Girard, Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types, *Proceedings of the Second Scandinavian Logic Symposium*, pp. 63–92. Studies in Logic and the Foundations of Mathematics, Vol. 63, North-Holland, Amsterdam, 1971.

22. R. Harper, F. Honsell and G. Plotkin, A Framework for Defining Logics, *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science*, LICS'87, Ithaca, NY, USA, 22–25 June 1987
23. J.R. Hindley, The principal type-scheme of an object in combinatory logic, *Transactions of the American Mathematical Society* 146 (1969), 29–60.
24. J.R. Hindley and J.P. Seldin, *Introduction to combinators and lambda-calculus*. London Mathematical Society Student Texts. Cambridge University Press, 1986.
25. W. Howard, The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.
26. F. Joachimski, R. Matthes, Short proofs of normalization for the simply- typed lambda-calculus, permutative conversions and Gödel's T. *Arch. Math. Log.* 42(1): 59-87, 2003.
27. J. Lambek and P. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.
28. P. Martin-Löf, *Intuitionistic type theory*, Bibliopolis, 1984.
29. R. Milner, Robin, A Theory of Type Polymorphism in Programming, *JCSS* 17: 348-375, 1978.
30. A. Mycroft, Polymorphic type schemes and recursive definitions, *International Symposium on Programming*, LNCS 167, pp. 217-228, Springer 1984.
31. R. Nederpelt, H. Geuvers and R. de Vrijer (eds.) *Selected Papers on Automath*, Studies in Logic, Vol. 133, North-Holland, 1994.
32. B. Nordström, K. Petersson and J. Smith, *Programming in Martin-Löf's Type Theory, An Introduction*, Oxford University Press, 1990.
33. Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
34. G. Plotkin, Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comp. Sci.* 1(2): 125-159, 1975.
35. G. Plotkin, LCF considered as a programming language, *Theor. Comp. Sci.* 5, pp. 223255, 1977.
36. J.C. Reynolds, Polymorphism is not Set-Theoretic. In G. Kahn, D.B. MacQueen, G. Plotkin (Eds.): *Semantics of Data Types*, International Symposium, Sophia-Antipolis, France, June 27-29, LNCS 173, Springer 1984, pp. 145-156.
37. H. Schwichtenberg, Definierbare Funktionen im λ-Kalkül mit Typen, *Archiv für Mathematische Logik und Grundlagenforschung* 17, pp. 113114, 1976.
38. W.W. Tait, Intensional interpretation of functionals of finite type. *J. Symbol. Logic*, v. 32, n. 2, pages 187–199, 1967.
39. P. Urzyczyn and M. Sørensen, *Lectures on the Curry-Howard Isomorphism*, Volume 149 of Studies in Logic and the Foundations of Mathematics, Elsevier, 2006.
40. M. Wand, A simple algorithm and proof for type inference,*Fundamenta Informaticae* X, pp. 115122, 1987.
41. J.B Wells, Typability and type-checking in the second-order λ-calculus are equivalent and undecidable, *Proceedings of the 9th Annual Symposium on Logic in Computer Science*, Paris, France, IEEE Computer Society Press, pp. 176185, 1994.