Lecture Notes for the Course Semantics (WS 09/10)

Gert Smolka and Jan Schwinghammer
Saarland University

February 2, 2010

These notes cover part of the material presented in the lectures. They are mainly based on the notes from WS 07/08. You are advised to take your own notes during lectures. Additional material and pointers to the literature are given on the web site of the course (www.ps.uni-sb.de/courses/sem-ws09).

Contents

1 Functional Programming and Standard ML

For this course, we require familiarity with the basics of functional programming and the programming language Standard ML. In case functional programming is new to you, you should work through one of the texts given on the web pages of the course. You should have a Standard ML interpreter installed on your computer. If you have not worked with Standard ML before, we recommend that you use Alice, an interpreter for Standard ML that comes with an easy to use graphical user interface. Links to interpreters and to reading material on Standard ML can be found on the web pages of the course.

Standard ML was designed by leading researches in the field of programming languages. We use Standard ML in this course since it realizes key ideas of the theory of programming languages in a practical language. In the following, we review some features of Standard ML that are essential to this course. For now, we will mainly work with examples, deeper explanations will be given later. You can learn a lot about Standard ML by experimenting with the given examples on an interpreter.

1.1 Equations and Declarations

The heart and soul of functional programming are functions and equations. Computation is accomplished by applying equations from left to right.

Syntactically, a Standard ML program is a sequence of declarations. Here is an example:

```
val a = 2*7
val b = 2*a-8
fun abs x = if x<0 then ~x else x
fun add x y = x + y
val succ = add 1
val pred = add ~1
val c = add a b + succ b
```

Each line is a declaration. A declaration starts with either the keyword *val* or *fun* and continues with an equation. Declarations that start with *fun* define procedures. Procedures are algorithmic versions of functions. A program is executed

by executing the declarations in the order they are given. The execution of the first and second declaration produces the simplified equations

$$a = 14$$
$$b = 20$$

which are called **bindings**: The identifier a is bound to the number 14, and b to 20. The declarations of the procedures *abs*, *add*, *succ*, and *pred* cannot be further simplified. The procedures *succ* and *pred* are obtained from the two-argument procedure *add* by applying it to a single argument. The procedure *succ* computes the successor function for integers, and *pred* the predecessor function. The declaration of c is executed by applying the simplified equations (i.e., bindings) for a, b, *add*, and *succ* from left to right:

$$\begin{aligned}
c &= \mathit{add}\ a\ b + \mathit{succ}\ b \\
&= \mathit{add}\ 14\ b + \mathit{succ}\ b \\
&= \mathit{add}\ 14\ 20 + \mathit{succ}\ b \\
&= (14 + 20) + \mathit{succ}\ b \\
&= 34 + \mathit{succ}\ b \\
&= 34 + \mathit{succ}\ 20 \\
&= 34 + \mathit{add}\ 1\ 20 \\
&= 34 + (1 + 20) \\
&= 34 + 21 \\
&= 55
\end{aligned}$$

The execution of the declaration for c produces the binding $c = 55$. Note that every execution step amounts to a left-to-right application of an equation (operations like + come with built-in equations).

In the context of functional programming, execution is usually referred to as **evaluation**. For instance, one says that the expression $9 - 2$ evaluates to 7.

1.2 Recursion

Suppose we want to write a procedure that computes, given two numbers x and n, the power x^n (n must be a natural number). We can start from the equations

$$x^0 = 1$$
$$x^n = x \cdot x^{n-1} \qquad \text{if } n > 0$$

These equations yield an algorithm for computing powers, as can be seen from the following example:

$$3^2 = 3 \cdot 3^1 = 3 \cdot (3 \cdot 3^0) = 3 \cdot (3 \cdot 1) = 3 \cdot 3 = 9$$

Note that the equations are applied from left to right. The equations are **exhaustive**, that is, to each power at least one of the equations is applicable. Moreover, the equations are **terminating**, that is, the equations cannot be applied infinitely often to a given power. In Standard ML, the two equations can be formulated as a recursive procedure:

```
fun power x n = if n<1 then 1 else x*power x (n-1)
```
power : int → int → int

Note that the procedure is formulated with a single equation, which combines the original equations by means of a conditional.

1.3 Types

Standard ML is a statically typed language. Given a program, an interpreter will derive types for all identifiers of the program and check that the program is well-typed. Execution is only attempted if the program is well-typed. Standard ML is designed such that the derived types are uniquely determined. Here are the types of the identifiers declared by our example program:

$$\begin{aligned} a &: int \\ b &: int \\ abs &: int \to int \\ add &: int \to int \to int \\ succ &: int \to int \\ pred &: int \to int \\ c &: int \end{aligned}$$

The type of a procedure starts with the types of the arguments and ends with the type of the result, where the types are separated by the symbol "→". The type *int* → *int* → *int* of *add* suggests that we get a procedure of type *int* → *int* when we apply *plus* to a single argument of type *int*. Here are two notational conventions for types and applications that free us from writing parentheses:

$$\begin{aligned} X \to Y \to Z &:= X \to (Y \to Z) \\ fxy &:= (fx)y \end{aligned}$$

In set theory, we have the isomorphism equivalence

$$X \times Y \to Z \quad \cong \quad X \to (Y \to Z)$$

Thus there are two possibilities for the representation of binary operations as functions. The usual **cartesian representation** appearing on the left combines the two arguments into a pair. The **cascaded representation** appearing on the right gets along without pairing by taking the arguments one after the other. The cascaded representation was discovered by Moses Schönfinkel around 1920. Its often called **curried representation**, after Haskell Curry who promoted its use.

Exercise 1.3.1 Write two procedures

$$cas : (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$
$$car : (\alpha \to \beta \to \gamma) \to \alpha * \beta \to \gamma$$

such that $car(cas\ f) = f$ and $cas(car\ g) = g$.

1.4 Polymorphic Higher-Order Procedures

A **higher-order procedure** is a procedure that takes a procedure as argument. As example, we consider a function *iter* that satisfies the equation

$$iter\ n\ x\ f = f^n x$$

The mathematical notation f^n stands for the function obtained by repeating n times the application of the function f:

$$f^0 x = x$$
$$f^n x = f^{n-1}(f x) \qquad \text{if } n > 0$$

If we rewrite the equations with *iter*, we obtain

$$iter\ 0\ x\ f = x$$
$$iter\ n\ x\ f = iter\ (n-1)\ (f x)\ f \qquad \text{if } n > 0$$

We say that *iter* **iterates** the **step function** f on the **start value** x. Based on the equations, we declare a procedure that computes the function *iter*:

```
fun iter n x f = if n<1 then x else iter (n-1) (f x) f
```
iter : int → α → (α → α) → α

Since there is no unique type for x and f, Standard ML types *iter* **polymorphically** using a type variable α. This means that *iter* can be used for every type α.

Now let's look again at the computation of powers. Since x^n can be obtained by multiplying 1 n times with x, we have

$$x^n = \mathit{iter}\ n\ 1\ (\lambda a.\, a \cdot x)$$

The notation $\lambda a.\, a \cdot x$ stands for the function that multiplies its argument with x. Based on this equation, we can declare a procedure *power* that computes x^n as follows:

```
fun power x n = iter n 1 (fn a => a*x)
```
power : int → int → int

Note that *power* is not a recursive procedure. The recursion needed for computing powers is obtained from the higher-order procedure *iter*. We now see that *iter* provides a recursion scheme that can be used to write procedures without explicit recursion.

Recall the definition of the factorials:

$$0! = 1$$
$$n! = n \cdot (n-1)! \qquad \text{if } n > 0$$

We can compute factorials with *iter* if we start from the pair $(1, 0!)$. This can be seen from the equations

$$(n+1, n!) = f(n,\ (n-1)!) \qquad \text{if } n > 0$$
$$f(k, a) = (k+1,\ k \cdot a)$$

From these equations we obtain the procedure

```
fun fac n = #2(iter n (1,1) (fn (k,a) => (k+1,k*a)))
```
fac : int → int

The projection operator #2 yields the second component of a tuple.

Exercise 1.4.1 (Fibonacci Numbers) Use *iter* to write a procedure *fib* : *int* → *int* that satisfies the equation *fib* n = if $n < 2$ then n else *fib*$(n-1)$ + *fib*$(n-2)$.

Exercise 1.4.2 (Loops) Consider the procedure

```
fun loop x p f = if p x then loop (f x) p f else x
```

Write a procedure *gcd* : *int* → *int* → *int* that yields the greatest common divisor of two positive numbers. Use *loop* to realize the necessary recursion. The procedure should satisfy the equation *gcd x y = if x = 0 then y else gcd* (*y mod x*) *x*.

1.5 A General Recursion Operator

We will now develop a higher-order procedure *fix* that is as powerful as explicit recursion. That is, every procedure with explicit recursion can be rewritten with *fix* to an equivalent procedure not using explicit recursion.

As starting point we take the recursive procedure

```
fun fac n = if n<2 then 1 else n*fac(n-1)
```
fac : int → int

We eliminate the recursion by taking the procedure needed for the recursion as argument:

```
fun fac' fac n = if n<2 then 1 else n*fac(n-1)
```
fac : (int → int) → int → int

We call fac' the **scheme** for *fac*. We now observe that the procedure $fac'\ fac$ behaves as the procedure *fac*, a fact that we can express with the equation

$$fac'\ fac\ =\ fac$$

When we apply the procedure *fix* to fac', we obtain a procedure that behaves like the procedure *fac*, a fact that we express with the equation

$$fix\ fac'\ =\ fac$$

We now have enough information to declare the procedure *fix*:

```
fun fix f x = f (fix f) x
```
fix : $((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$

Using *fix*, we can declare a procedure computing factorials as follows:

```
val fac = fix fac'
```
fac : int → int

The name *fix* for the recursion operator comes from the notion of a **fixed point**. In general, x is called a fixed point of a function f if $fx = x$. The equations above say that *fac* is a fixed point of fac', and that *fix* applied to fac' yields a fixed point of fac'.

Note that **iter** terminates if it is used with a terminating procedure. In contrast, *fix* may yield a non-terminating procedure if it is aplied to a terminating procedure (i.e., $fix(fn\ f \Rightarrow fn\ x \Rightarrow f\ x)$. This tells us that the expressivity of *fix* comes with the price that one has to worry about termination. In fact, recursion theory tells us that there is no programming language such that every procedure terminates and every total computable functions can be computed by at least one of its procedures.

Exercise 1.5.1 (Powers) Write a procedure $power : int \to int \to int$ such that $power\ x\ n = x^n$. Use *fix* to realize the necessary recursion.

Exercise 1.5.2 (Conditionals) Write a procedure

$$cond : bool \to (unit \to \alpha) \to (unit \to \alpha) \to \alpha$$

such that $cond\ t_0\ (fn\ () \Rightarrow t_1)\ (fn\ () \Rightarrow t_2)$ is semantically equivalent to the expression $if\ t_0\ then\ t_1\ else\ t_2$. The type unit has exactly one element, which is the empty tuple $()$. Explain why the type $bool \to \alpha \to \alpha \to \alpha$ does not suffice for *cond* (assume the execution of either t_1 or t_2 does not terminate). The technique of postponing the evaluation of an expression by packaging it into a procedure is known as **λ-lifting**.

1.6 Lists

In set theory, finite sequences can be represented as tuples $\langle x_1, \ldots, x_n \rangle$. There is an empty tuple $\langle\rangle$, and, for every x, a singleton tuple $\langle x \rangle$. We use X^* to denote the set of all tuples whose components are in the set X.

For programming, a recursive representation of finite sequences as lists is often preferable. Given a set X, the set $\mathcal{L}(X)$ of **lists over** X is defined as follows:

$$\mathcal{L}(X) \ := \ \{\langle\rangle\} \cup (X \times \mathcal{L}(X))$$

Thus a list over X is either the empty tuple (the **empty list**) or a pair $\langle x, xs \rangle$ of an $x \in X$ and a list xs over X. There is a bijection between X^* and $\mathcal{L}(X)$ that preserves the characteristic properties of sequences. We use the following notations:

$$\begin{aligned} nil \ &:= \ \langle\rangle \\ x :: xr \ &:= \ \langle x, xr \rangle && \text{read “}x\text{ cons }xr\text{”} \\ x_1 :: x_2 :: xr \ &:= \ x_1 :: (x_2 :: xr) \\ [x_1, \ldots, x_n] \ &:= \ x_1 :: \ldots x_n :: nil \end{aligned}$$

Standard ML provides lists based on these notations. For every type T, the type $T\ list$ has the lists over T as elements. Nil and cons are polymorphically typed:

$$nil : \alpha\ list$$
$$(::) : \alpha * \alpha\ list \to \alpha\ list$$

Syntactically, *nil* is a constant and :: is an infix operator (like + for numbers). Given a list $x :: xr$, we refer to x as the **head** and to xr as the **tail** of the list.

Length and **concatenation** of lists are defined as follows:

$$
\begin{array}{rcll}
|[x_1,\ldots,x_n]| & := & n & \text{length} \\
[x_1,\ldots,x_m]@[y_1,\ldots,y_n] & := & [x_1,\ldots,x_m,y_1,\ldots,y_n] & \text{concatenation}
\end{array}
$$

Length and concatenation can be computed recursively:

$$
\begin{aligned}
|nil| &= 0 \\
|x :: xr| &= 1 + |xr| \\
nil @ ys &= ys \\
(x :: xr) @ ys &= x :: (xr @ ys)
\end{aligned}
$$

In Standard ML, these equations can be realized with polymorphic procedures:

```
fun length nil = 0
  | length (x::xr) = 1 + length xr
```
length : α *list* → *int*

```
fun append nil ys = ys
  | append (x::xr) ys = x::append xr ys
```
append : α *list* * α *list* → α *list*

Note that the equations appear directly in the declarations of the procedures. Given the straightforward structure of lists, there is no need to combine the equations for nil and cons with a conditional. To combine the equations into one with a conditional, we need the following procedures:

```
fun null nil = true
  | null (x::xr) = false
```
null : α *list* → *bool*

```
fun hd nil = raise Empty
  | hd (x::xr) = x
```
hd : α *list* → α

```
fun tl nil = raise Empty
  | tl (x::xr) = xr
```
tl : α *list* → α *list*

The expression *raise Empty* will raise the exception *Empty*. Now we can write *append* with a conditional:

```
fun append xs ys = if null xs then ys
                   else hd xs :: append (tl xs) ys
```
append : α *list* * α *list* → α *list*

Here is procedure that yields the length of a list.

```
fun len xs = fix (fn f => (fn nil => 0 | x::xr => 1 + f xr)) xs
```

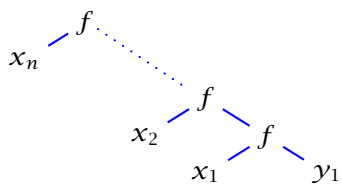
It realizes the necessary recursion with *fix*.

The canonical recursion scheme for lists is

```
fun foldl f y nil = y
  | foldl f y (x::xr) = foldl f (f(x,y)) xr
```

foldl : $(\alpha * \beta \to \beta) \to \beta \to \alpha$ *list* $\to \beta$

The evaluation of *foldl* $f\ y_1\ [x_1, \ldots, x_n]$ is best understood graphically:



The computation is bottom-up. First the step function f is applied to the first list element x_1 and the start value y_1, which yields an intermediate value y_2. Next f is applied to x_2 and y_2, which yields y_3. Finally, f is applied to x_n and y_n, which yields the final result. Here are procedures that compute the sum and the product of the elements of a list over integers:

```
val sum = foldl op+ 0
```

sum : int list → int

```
val product = foldl op* 1
```

sum : int list → int

The expression *op*+ and *op*∗ provide the procedures that correspond to the operations + and *.

Reversion of lists is defined as follows:

$$rev\ [x_1, \ldots, x_n]\ :=\ [x_n, \ldots, x_1]$$

Using *foldl*, we can declare a procedure that reverses lists as follows:

```
fun rev xs = foldl op:: nil xs
```

α *list* $\to \alpha$ *list*

Recall that Standard ML will type

```
val rev' = foldl op:: nil
```

monomorphically, that is, *rev′* can only be used with a single type, which is determined when *rev′* is used first. Polymorphic typing, if at all possible, can always be forced by declaring a procedure with *fun*.

We can also declare a procedure that concatenates lists with foldl:

```
fun append xs ys = foldl op:: ys (rev xs)
```

If we replace *rev* with its definition, we obtain

```
fun append xs ys = foldl op:: ys (foldl op:: nil xs)
```

Exercise 1.6.1 Write a procedure $mapr : (\alpha \to \beta) \to \alpha\ list \to \beta\ list$ that satisfies the equation $mapr\ f\ [x_1, \ldots, x_n] = [f\ x_n, \ldots, f\ x_1]$. Use *foldl* to realize the necessary recursion.

Exercise 1.6.2 Write a procedure $list : int \to (int \to \alpha) \to \alpha\ list$ that satisfies the equation $list\ n\ f = [f\ 1, \ldots f\ n]$. Use *iter* to realize the necessary recursion.

2 Specifying and Proving with Coq

We specify the types of Boolean values, natural numbers and lists and write procedures like addition, multiplication or list reversal. We prove facts like the commutativity of addition or the associativity of list concatenation. We learn how the Ackermann function can be implemented with higher-order primitive recursion and prove the correctness of the implementation. It becomes clear that a verification system like Coq has much in common with a functional programming system and that a theory of programming languages has much in common with logic.

3 PCF

We now change our mode of investigation. Rather than exploring a complex programming language such as Standard ML, we will define an idealized language PCF. At the example of PCF, we will illustrate basic mathematical techniques for the definition of programming languages. PCF was introduced around 1975 by Gordon Plotkin. We can see PCF as a fairly minimal language that, for every computable funtion, provides at least one procedure that computes it. In fact, the acronym PCF stands for partial computable functions.

PCF is an explicitly typed language where the argument variables of procedures must be introduced with a type. There is no polymorphism. Procedures are described with λ and recursion is obtained with *fix*. PCF has two base types, *bool* and *nat* (the natural numbers).

Required Reading: Pierce, Chapter 3. Introduces PCF without procedures. Explains inference rules and other basics.

3.1 Abstract Syntax

Figure 3.1 shows the abstract syntax of PCF. It defines sets of syntactic objects whose elements are called **types**, **variables**, **values** and **terms**. There are the inclusions $Var \subseteq Val \subseteq Ter$. The definition of values and terms is mutually recursive. There is only one variable binder (λ). A term is **open** if it contains a free occurrence of a variable, and **closed** otherwise. For technical convenience, variables are treated as values. Closed values represent the proper values, that is, the Boolean values, the natural numbers, and the procedures of PCF.

Syntactic objects are represented as pairs $\langle n, y\rangle$ where n is the **variant number** and y is the **constituent tuple**. For the variant numbers we number the syntactic variants introduced in Figure 3.1 starting from 1. Variables receive the variant number 4 and hence are represented as pairs $\langle 4, k\rangle$ where k is a natural number. An abstraction $\lambda x{:}T.\,t$ is represented as the pair $\langle 12, \langle x, T, t\rangle\rangle$, where 12 is the variant number and $\langle x, T, t\rangle$ is the constituent tuple. We choose the representation such that the inclusions $Var \subseteq Val \subseteq Ter$ hold.

We will implement the mathematical definitions for PCF in Standard ML. The abstract syntax can be implemented as follows:

$$
\begin{aligned}
T \in Ty &::= bool \mid nat \mid T \to T \\
x \in Var &::= \mathbb{N} \\
t \in Ter &::= false \mid true \mid if\ t\ then\ t\ else\ t \\
&\mid 0 \mid succ\ t \mid pred\ t \mid iszero\ t \\
&\mid x \mid \lambda x{:}T.t \mid t\ t \mid fix\ t \\
v \in Val &::= false \mid true \mid 0 \mid succ\ v \mid \lambda x{:}T.t \mid x
\end{aligned}
$$

Figure 3.1: Abstract syntax of PCF

```
datatype ty = Bool | Int | Proc of ty * ty
type var = string
datatype ter =
    False | True | If of ter*ter*ter
  | O | Succ of ter | Pred of ter |Iszero of ter
  | V of var | A of ter*ter | L of var*ty*ter | Fix of ter
```

Because Standard ML doesn't have subtyping, we don't introduce a type for values and define the type of terms directly. Moreover, variables are represented by the type string that is disjoint from the type *ter*. So there is a difference between a variable x and the term $V\ x$ that just consists of the variable x. Here is a procedure that tests whether a variable occurs free in a term:

```
fun free x (V y) = (x=y)
  | free x False = false
  | free x True = false
  | free x (If(t0,t1,t2)) = free x t0 orelse free x t1 orelse free x t2
  | free x O = false
  | free x (Succ t) = free x t
  | free x (Pred t) = free x t
  | free x (Iszero t) = free x t
  | free x (A(s,t)) = free x s orelse free x t
  | free x (L(y,ty,t)) = x<>y andalso free x t
  | free x (Fix t) = free x t
```

free : var → ter → bool

Exercise 3.1.1 (Values) Write a procedure *value* : *term* → *bool* that tests whether a term is a value.

Exercise 3.1.2 (Substitution) Write a procedure *subst* : *var* → *ter* → *ter* → *ter* such that *subst x s t* yields the term that is obtained from t by replacing every free occurrence of the variable x with the term s. Don't worry about capture.

$$\frac{}{\Gamma \vdash x : T}\ (x,T) \in \Gamma \qquad \frac{}{\Gamma \vdash \mathit{false} : \mathit{bool}} \qquad \frac{}{\Gamma \vdash \mathit{true} : \mathit{bool}}$$

$$\frac{\Gamma \vdash t_0 : \mathit{bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 : T} \qquad \frac{}{\Gamma \vdash 0 : \mathit{nat}}$$

$$\frac{\Gamma \vdash t : \mathit{nat}}{\Gamma \vdash \mathit{succ}\ t : \mathit{nat}} \qquad \frac{\Gamma \vdash t : \mathit{nat}}{\Gamma \vdash \mathit{pred}\ t : \mathit{nat}} \qquad \frac{\Gamma \vdash t : \mathit{nat}}{\Gamma \vdash \mathit{iszero}\ t : \mathit{bool}}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T} \qquad \frac{\Gamma[x{:=}T_1] \vdash t : T}{\Gamma \vdash (\lambda x : T_1 . t) : T_1 \to T}$$

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \mathit{fix}\ t : T}\ T \text{ functional}$$

Figure 3.2: Inference rules defining the typing relation

3.2 Typing Relation

We need a definition that says which terms are well-typed and what are the types of well-typed terms. To treat open terms, we provide types for free variables by means of a type environment. A **type environment** is a function $\Gamma \in \mathit{Var} \rightharpoonup \mathit{Ty}$. Hence $\Gamma \subseteq \mathit{Var} \times \mathit{Ty}$ for every type environment γ. The simplest type environment is $\emptyset$ (the empty set). We need the notation

$$\Gamma[x{:=}T] \ := \ \{(x,T)\} \cup \{\, (y,S) \in \Gamma \mid x \neq y \,\}$$

Note that $\Gamma[x{:=}T]$ is the type environment that behaves everywhere like Γ except on x, where it yields T.

The **typing relation** is a set $\vdash \subseteq (\mathit{Var} \rightharpoonup \mathit{Ty}) \times \mathit{Ter} \times \mathit{Ty}$. The typing relation will be defined such that $(\Gamma, t, T) \in \vdash$ holds if and only if t is well-typed and has type T with respect to Γ. We write $\Gamma \vdash t : T$ for $(\Gamma, t, T) \in \vdash$. The typing relation is defined recursively by the inference rules appearing in Figure 3.2.

There is exactly one inference rule for every syntactic form of terms, and the terms in the premises of the rules are always constituents of the term in the conclusion. Hence we have a recursion that reduces the size of terms. The rules are algorithmic in that they describe a recursive procedure that given Γ and t yields the unique type T such that $\Gamma \vdash t : T$ if it exists.

Proposition 3.2.1 (Unique Type) Given Γ and t, there is at most one T such that $\Gamma \vdash t : T$.

Proposition 3.2.2 If $\Gamma \vdash t : T$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash t : T$.

$$\frac{}{v \Downarrow v}\ v \in \{0, \mathit{false}, \mathit{true}\} \cup \mathit{Var} \ \text{ or } \ v = \lambda x{:}T.\,t$$

$$\frac{t_0 \Downarrow \mathit{true} \qquad t_1 \Downarrow v}{\mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 \Downarrow v} \qquad \frac{t_0 \Downarrow \mathit{false} \qquad t_2 \Downarrow v}{\mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 \Downarrow v}$$

$$\frac{t \Downarrow v}{\mathit{succ}\ t \Downarrow \mathit{succ}\ v} \qquad \frac{t \Downarrow 0}{\mathit{pred}\ t \Downarrow 0} \qquad \frac{t \Downarrow \mathit{succ}\ v}{\mathit{pred}\ t \Downarrow v}$$

$$\frac{t \Downarrow 0}{\mathit{iszero}\ t \Downarrow \mathit{true}} \qquad \frac{t \Downarrow \mathit{succ}\ v}{\mathit{iszero}\ t \Downarrow \mathit{false}}$$

$$\frac{t_1 \Downarrow \lambda x{:}T.\,t \qquad t_2 \Downarrow v_2 \qquad [x := v_2]t \Downarrow v}{t_1\ t_2 \Downarrow v}$$

$$\frac{t \Downarrow v_1 \qquad [x := \mathit{fix}\ v_1]t_1 \Downarrow v}{\mathit{fix}\ t \Downarrow v}\ v_1 = \lambda x{:}T.\,t_1$$

Figure 3.3: Inference rules defining the evaluation relation

Both propositions can be shown by inductive proofs that, for each rule, show that the conclusion satisfies the claim if the premises satisfy the claim. This form of induction is known as **rule induction**. Later, we will give a careful analysis of rule induction. In the literature, rule induction is often referred to as induction on the length of derivations.

Exercise 3.2.3 (Elaboration) Write a procedure $\mathit{elab} : (\mathit{var} \to \mathit{ty}) \to \mathit{ter} \to \mathit{ty}$ that yields the type of a term. Raise the exception *Error* if the term is not well-typed. The empty type environment and an update operation for environments can be declared as follows:

```
exception Error
fun empty x = raise Error
fun update f x t y = if y=x then t else f y
```

3.3 Big-Step Semantics

We will now define an **evaluation relation** $\Downarrow \subseteq \mathit{Ter} \times \mathit{Val}$ such that $t \Downarrow v$ holds if and only if the term t evaluates to v. The evaluation relation is defined recursively by the inference rules appearing in Figure 3.3. The approach that defines the evaluation relation of a language directly with inference rules is known as **big-step semantics**.

The rules for applications and *fix* make use of **term substitution**, which, in general, is a fairly complex operation. Term substitution appears with the notation $[x{:=}s]t$ which stands for the term that is obtained from t by capture-free replacement of ever free occurrence of x with s. The tricky point about substitution is the capture-freeness. If s is closed, capture-freeness is not a issue and we can simply replace ever free occurrence of x with s in the obvious way (see Exercise 3.1.2).

The rules defining the evaluation relation are algorithmic in that they yield a procedure that, given a term t, computes the value v of t (i.e., $t \Downarrow v$) if it exists.

Proposition 3.3.1 (Determinism) Given t, there is at most one v such that $t \Downarrow v$.

Proposition 3.3.2 (Type Preservation) If $\Gamma \vdash t : T$ and $t \Downarrow v$, then $\Gamma \vdash v : T$.

Exercise 3.3.3 (Evaluation) Write a procedure $eval : ter \to ter$ that yields the value of a closed term if it exists. Raise the exception *Error* if *eval* must quit because of a type inconsistency or a free variable occurrence.

Exercise 3.3.4 Given a well-typed closed term t for which there is no value v such that $t \Downarrow v$.

Exercise 3.3.5 The evaluation relation is defined for all terms, not just well-typed terms. Explain why the term $(\lambda x{:}int.xx)(\lambda x{:}int.xx)$ does not evaluate to a value.

3.4 Small-Step Semantics

If we evaluate a term t, there may be two reasons for not obtaining a value: either the evaluation does not terminate or the evaluation gets stuck because there is a type error. This distinction is not modelled by the evaluation relation, but it is implicitly contained in the inference rules defining the evaluation relation. To make the distinction explicit, we now define a **reduction relation** $\to \in Ter \times Ter$ that models single computation steps and relates to the evaluation relation in that $t \Downarrow v \iff t \to \cdots \to v$. Given the reduction relation, an non-terminating evaluation of t appears as an infinite reduction chain $t \to t_1 \to t_2 \to t_3 \to \cdots$ issuing from t, and an **evaluation error of t** appears as an finite reduction chain $t \to \cdots \to t'$ such that t' is neither a value nor reducible (i.e., there is no t'' such that $t \to t''$).

We define the reduction relation recursively with the inference rules appearing in Figure 3.4. The rules without premisses are called **proper reduction rules**,

$$\frac{}{\textit{if false then } t_1 \textit{ else } t_2 \to t_2} \qquad \frac{}{\textit{if true then } t_1 \textit{ else } t_2 \to t_1}$$

$$\frac{}{\textit{pred } 0 \to 0} \qquad \frac{}{\textit{pred}(\textit{succ } v) \to v}$$

$$\frac{}{\textit{iszero } 0 \to \textit{true}} \qquad \frac{}{\textit{iszero}(\textit{succ } v) \to \textit{false}}$$

$$\frac{}{(\lambda x{:}T.t)v \to [x{:=}v]t} \qquad \frac{}{\textit{fix } v \to [x{:=}\textit{fix } v]\, t}\ v = \lambda x{:}T.t$$

$$\frac{t_0 \to t_0'}{\textit{if } t_0 \textit{ then } t_1 \textit{ else } t_2 \to \textit{if } t_0' \textit{ then } t_1 \textit{ else } t_2}$$

$$\frac{t \to t'}{\textit{succ } t \to \textit{succ } t'} \qquad \frac{t \to t'}{\textit{pred } t \to \textit{pred } t'} \qquad \frac{t \to t'}{\textit{iszero } t \to \textit{iszero } t'}$$

$$\frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2} \qquad \frac{t \to t'}{v\, t \to v\, t'} \qquad \frac{t \to t'}{\textit{fix } t \to \textit{fix } t'}$$

Figure 3.4: Inference rules defining the reduction relation

and the rules with premisses are called **descent rules**. The approach that defines the reduction relation of a language directly with inference rules is known as **small-step semantics**.

The proper reduction rules follow directly from the evaluation rules. The same is true for the descent rules, with the exception of

$$\frac{t \to t'}{v\, t \to v\, t'}$$

In contrast to the alternative descent rule

$$\frac{t_2 \to t_2'}{t_1\, t_2 \to t_1\, t_2'}$$

the descent rule appearing in Figure 3.4 forces that the first constituent of an application is reduced before the second constituent. This order is not present in the evaluation rule for applications:

$$\frac{t_1 \Downarrow \lambda x{:}T.\, t \qquad t_2 \Downarrow v_2 \qquad [x := v_2]t \Downarrow v}{t_1\, t_2 \Downarrow v}$$

The property $t \Downarrow v \iff t \to \cdots \to v$ will hold no matter which of the two descent rules we use. However, with the descent rule in Figure 3.4 we obtain a deterministic reduction relation.

Proposition 3.4.1 (Determinism) Given t, there is at most one term t' such that $t \to t'$.

Proposition 3.4.2 (Coincidence) For every term t and every value v, we have $t \Downarrow v$ if and only if $t \to \cdots \to v$.

A term t is called **reducible** if there is a term t' such that $t \to t'$.

Proposition 3.4.3 (Progress) Every closed and well-typed term is either a value or reducible.

Proposition 3.4.4 (Type Preservation) If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.

Together, progress and type preservation yield **type safety**:

Proposition 3.4.5 (Type Safety) If $\emptyset \vdash t : T$, then there is either a value v such that $t \to \cdots \to v$, or the reduction of t does not terminate.

3.5 System T

It is instructive to contrast PCF with System T, a rather minimal computation system suggested by the logician Kurt Gödel in 1958. In contrast to PCF, computation in T always terminates. This comes at the price that T cannot express all computable functions (there is a general result that says that no generally terminating computation system can express all computable functions).

T replaces PCF's recursion operator *fix* with an operator *natrec* that provides primitive recursion. In Coq *natrec* can be expressed as follows:

```
Fixpoint natrec X (n: nat) (x: X) (f: nat->X->X)  := match n with
| O => x
| S n' => f n' (natrec X n' x f)
end.
```

In Standard ML we can write

```
datatype nat = O | S of nat
fun natrec O x f = x
  | natrec (S n) x f = f n (natrec n x f)
```

An interesting fact about *natrec* is that it can express procedures for addition, multiplication, exponention, factorial, and Ackermann without using Boolean values or conditionals. For addition we prove the claim in Coq as follows:

```
Goal forall x y, x+y = natrec nat x y (fun _ => S).
induction x; intuition.
simpl. rewrite IHx. reflexivity. Qed.
```

$$
\begin{aligned}
T \in Ty &::= nat \mid T \to T \\
x \in Var &::= \mathbb{N} \\
t \in Ter &::= 0 \mid S\,t \mid natrec\ t\ t\ t \mid x \mid \lambda x{:}T.t \mid t\,t \\
v \in Val &::= 0 \mid S\,v \mid \lambda x{:}T.t \mid x
\end{aligned}
$$

Figure 3.5: Abstract syntax of T

Figure 3.5 shows the abstract syntax of T. The definition of the typing relation can be synthesized from the following type schemes for the operators of T:

$$
\begin{aligned}
O &: nat \\
S &: nat \to nat \\
natrec &: nat \to T \to (nat \to T \to T) \to T \qquad \text{for all types } T
\end{aligned}
$$

Note that the operator *natrec* is polymorphic. However, polymorphism is not available within T. If we write a procedure $\lambda f{:}T.\ natrec\ f$ in T, we need to commit to one specific type T. This is in contrast to Coq, where we can write

```
fun (T:Type) => fun (f: T->T) => natrec T f
```

We can define evaluation and reduction relations for T in the style we used for PCF. This is a straightforward exercise. All we need to know are three equations:

$$
\begin{aligned}
(\lambda x{:}T.s)t &= [x{:=}t]s \\
natrec\ O\ x\ f &= x \\
natrec\ (S\ n)\ x f &= f\ n\ (natrec\ n\ x\ f)
\end{aligned}
$$

Tait [1967] showed for T that applying the equations from left to right always terminates, a famous result know as *(strong) normalization*. It is also known that applying the equations is *confluent*, that is, no matter how we apply the equations to a given term, we always end up with the same term.

Exercise 3.5.1 Express procedures for addition, multiplication, exponention, factorial, and Ackermann in System T. Realize your procedures in Standard ML and Coq. Prove the correctness of you procedures in Coq. Do the same for predecessor and subtraction.

Exercise 3.5.2 Define a typing relation, an evaluation relation, and a reduction relation for T. Follow the example of PCF and make sure that the following properties hold: unique type, determinism, preservation, progress, and coincidence. For terms not containing *natrec* the relations for T should agree with the relations for PCF.

3.6 A Problem with Fix

We have defined *natrec* in Coq and proven some results for it. Since the general recursion operator *fix* does not always terminate, it cannot be defined in Coq. There is a good reason why Coq insists on termination. The argument goes as follows:

1. Suppose Coq admits the definition of *fix*. Then we have the equation $fix\ g\ x = g\,(fix\ g)\,x$ for all g and x. We have used this equation for the definition of fix in Standard ML.
2. We can now give a term $t : nat$ for which we can show $S\,t = t$.
3. However, by induction, we can show $\forall x{:}nat.\ Sx \neq x$.

The bad term is $fix(\lambda fx.S(fx))\,0$. Here is the proof in Coq.

```
Proposition Fix_Problem:
(exists nfix: ((nat->nat)->(nat->nat))->(nat->nat),
   forall g x, nfix g x = g (nfix g) x))
-> False.
Proof.
intro H. inversion H as [nfix Efix]. clear H.
set (t:=nfix(fun f x => S(f x))0).
assert(C: S t = t).
unfold t at 2.  rewrite Efix. fold t. reflexivity.
induction t.
  inversion C.
  inversion C.  auto.
Qed.
```

The problem comes from the fact that the evaluation of t does not terminate. Hence t does not denote a natural number. This contradicts the typing $t{:}nat$.

We can still have an equational theory for PCF. This theory extends the denotation of every type T with a special value $\perp$ that serves as value for terms whose evaluation does not terminate. The corresponding theory is known as denotational semantics. See Winskel's textbook for more information.

Exercise 3.6.1 The problem with *fix* persists if we weaken the equation for *fix* to $fix(\lambda f{:}T.g\,f)x = g(\lambda f{:}T.g\,f)x$ where $T = nat \to nat$. We can obtain this equation from the reduction rules for PCF. Prove in Coq that this equation still suffices for inconsistency.

3.7 PCF$^-$

PCF$^-$ is a minimalistic variant of PCF that does not have a type *bool* but can still express all computable functions. The distinctive feature of PCF$^-$ is an operator *natcase*, whose formulation in Coq is as follows:

$$
\begin{aligned}
T \in Ty &::= nat \mid T \to T \\
x \in Var &::= \mathbb{N} \\
t \in Ter &::= 0 \mid S\,t \mid natcase\,t\,t\,t \mid x \mid \lambda x{:}T.t \mid t\,t \mid fix\,t \\
v \in Val &::= 0 \mid S\,v \mid \lambda x{:}T.t \mid x
\end{aligned}
$$

Figure 3.6: Abstract syntax of PCF$^-$

```
Fixpoint natcase X (n: nat) (x: X) (f: nat->X)  := match n with
| O => x
| S n' => f n'
end.
```

In Standard ML we would write

```
fun natcase 0 x f = x
  | natrec (S n) x f = f n
```

Figure 3.6 shows the abstract syntax of PCF$^-$.

Exercise 3.7.1 Express procedures for addition, multiplication, predecessor, and *natrec* in PCF$^-$. Realize your procedures in Standard ML.

Exercise 3.7.2 Define a typing relation, an evaluation relation, and a reduction relation for PCF$^-$. Follow the example of PCF and make sure that the following properties hold: unique type, determinism, preservation, progress, and coincidence. For terms not containing *natcase* the relations for PCF$^-$ should agree with the relations for PCF.

3.8 Evaluation Contexts and Redexes

The descent rules in the definition of the reduction relation can be formulated more compactly with so-called evaluation contexts. An evaluation context is a term with a hole at a position where a proper reduction rule could apply. For PCF, we define **evaluation contexts** as follows:

$$C ::= [\,] \mid if\ C\ then\ t\ else\ t \mid succ\,C \mid pred\,C \mid iszero\,C \mid C\,t \mid v\,C \mid fix\,C$$

The **application of a context** C to a term t yields the term that is obtained by filling the hole of C with t. We denote this term with $C[t]$.

The **top-level reduction relation** $\rightarrow_0 \subseteq Ter \times Ter$ is the relation defined by the proper reduction rules. Note that $\rightarrow_0 \subseteq \rightarrow$.

A term t is called a **redex** if there is a term t' such that $t \rightarrow_0 t'$. An evaluation context C is called a **reduction context for a term** t if there exists a **redex** s such that $t = C[s]$. The word redex stands for reducible expression. The plural of redex is redexes.

Proposition 3.8.1 For every term t:

1. There exists at most one evaluation context C such that there exists a redex s such that $t = C[s]$. If it exists, we call C **the reduction context of** t.
2. There exists at most one redex s such that there exists an evaluation context C such that $t = C[s]$. If it exists, we call s **the reduction redex of** t.
3. t is reducible if and only if there exists a reduction context C for t and a redesx s such that $t = C[s]$.

Proposition 3.8.2 For all terms t, t':

$$t \rightarrow t' \iff \exists s, s', C\colon\ t = C[s] \ \wedge\ s \rightarrow_0 s' \ \wedge\ C[s'] = t'$$

Exercise 3.8.3 Define the evaluation contexts for T and PCF$^-$.

3.9 Closure Semantics

We will now consider a semantics that represents values semantically. In particular, the natural numbers are represented as natural numbers. Moreover, procedures are represented as so-called closures. In contrast to the big- and small-step semantics we have seen before, the inference rules defining the closure semantics of PCF do not make use of substitution. As it comes to the implementation of programming languages, closure semantics is a more realistic model than big- and small-step semantics.

We consider PCF$^-$ and start with the definition of **semantic values**:

SV	:= N of $\mathbb{N}$	natural number
	| C of $Var \times Ter \times (Var \rightharpoonup SV)$	closure
	| R of $Var \times Var \times Ter \times (Var \rightharpoonup SV)$	recursive closure

The functions $\phi \in Var \rightharpoonup SV$ are called **value environments**. The letter ϕ will always denote a value environment. Similiar to syntactic objects, semantic values are represented as pairs whose first component is a variant number. For instance, $\mathsf{N}\,n = (2, n)$.

$$\frac{}{\phi \vdash 0 \rhd \mathsf{N}0} \qquad \frac{\phi \vdash t \rhd \mathsf{N}n}{\phi \vdash S\ t \rhd \mathsf{N}(n+1)}$$

$$\frac{\phi \vdash t \rhd \mathsf{N}0 \qquad \phi \vdash t_0 \rhd w}{\phi \vdash \mathit{natcase}\ t\ t_0\ (\lambda x{:}T.t_1) \rhd w}$$

$$\frac{\phi \vdash t \rhd \mathsf{N}(n{+}1) \qquad \phi[x{:=}\mathsf{N}n] \vdash t_1 \rhd w}{\phi \vdash \mathit{natcase}\ t\ t_0\ (\lambda x{:}T.t_1) \rhd w}$$

$$\frac{}{\phi \vdash x \rhd w}\ (x,w) \in \phi \qquad \frac{}{\phi \vdash \lambda x{:}T.\,t \rhd \mathsf{C}(x,t,\phi)}$$

$$\frac{\begin{array}{l}\phi \vdash t_1 \rhd w_1 \\ \phi \vdash t_2 \rhd w_2\end{array} \qquad \phi'[x{:=}w_2] \vdash t \rhd w}{\phi \vdash t_1\ t_2 \rhd w}\ w_1 = \mathsf{C}(x,t,\phi')$$

$$\frac{}{\phi \vdash \mathit{fix}(\lambda f{:}T_1.\,\lambda x{:}T_2.\,t) \rhd \mathsf{R}(f,x,t,\phi)}$$

$$\frac{\begin{array}{l}\phi \vdash t_1 \rhd w_1 \\ \phi \vdash t_2 \rhd w_2\end{array} \qquad (\phi'[f{:=}w_1])[x{:=}w_2] \vdash t \rhd w}{\phi \vdash t_1\ t_2 \rhd w}\ w_1 = \mathsf{R}(f,x,t,\phi')$$

Figure 3.7: Closure-evaluation relation for a fragment of PCF$^-$

A closure $\mathsf{C}(x,t,\phi')$ represents a procedure with an argument variable x, a body t, and an environment ϕ'. The environment provides the values of the free variables of t that are different from x. When the closure is applied, there will be a value for x, which means that we can evaluate the body t in an environment that binds all of its free variables.

A recursive closure $\mathsf{R}(f,x,t,\phi)$ represents the scheme of a recursive procedure with a **recursion variable** f, an argument variable x, a body t, and an environment ϕ'. When the recursive closure is applied, there will be a value for x. The body t will then be evaluated in the environment obtained from ϕ' by binding x to the argument value and f to the recursive closure $\mathsf{R}(f,x,t,\phi)$.

The **closure-evaluation relation** is a subset of $(\mathit{Var} \rightharpoonup \mathit{SV}) \times \mathit{Ter} \times \mathit{SV}$. We write $\phi \vdash t \rhd w$ if the triple (ϕ,t,w) is an element of the closure-evaluation relation. We define the closure-evaluation relation such that $\phi \vdash t \rhd w$ if and only if the term t evaluates to the semantic value w in the environment ϕ. Figure 3.7 defines the closure-evaluation relation by means of inference rules.

There is an important syntactic restriction that comes with closures. As one can see from the respective inference rule in Figure 3.7, the third argument of

natcase must be an abstraction and the argument of *fix* must be a double abstraction. The syntactic restrictions provide for the natural formulation of the inference rules for *natcase* and *fix*.

The closure semantics agrees with the big step semantics. Since the two semantics use different representations of values, expressing this statement precisely (what exactly means agree?) is tedious. However, it's easy to state the following:

Proposition 3.9.1 For every closed and well-typed term t that applies *fix* only to double abstractions, we have the following: $(\exists v\colon\ t \Downarrow v) \iff (\exists w\colon\ \emptyset \vdash t \rhd w)$.

We will see in the next section that the seemingly weak agreement expressed by the proposition is in fact a rather strong agreement.

Exercise 3.9.2 Explain what goes wrong if the typing assumption $\emptyset \vdash t : T$ is dropped in the statement of Proposition 3.9.1.

3.10 Contextual Equivalence

When we program, we employ an intuitive notion of **semantic equivalence** for the expressions forming the program. Given two semantically equivalent expressions s and t, the idea is that replacing s with t does not affect the result of a computation.

Around 1975, Plotkin came up with a surprisingly simple formal definition of semantic equivalence. Since Plotkin's definition is based on contexts and there may be different notions of semantic equivalence, the resulting relation is known as **contextual equivalence**.

This time we will employ a general notion of context where contexts may have their hole at any subterm position:

$$\begin{aligned} C \ ::= \ & [\,] \mid \mathit{if}\ C\ \mathit{then}\ t\ \mathit{else}\ t \mid \mathit{if}\ t\ \mathit{then}\ C\ \mathit{else}\ t \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ C \\ & \mid \mathit{succ}\ C \mid \mathit{pred}\ C \mid \mathit{iszero}\ C \mid C\ t \mid t\ C \mid \lambda x.C \mid \mathit{fix}\ C \end{aligned}$$

The **application of a context** C to a term t yields the term that is obtained by filling the hole of C with t. We denote this term with $C[t]$.

A **congruence** on the set of terms is an equivalence relation $\sim$ on *Ter* such that $s \sim t \Longrightarrow C[s] \sim C[t]$ for all terms s, t and all contexts C. A congruences is an abstract notion of equality that supports replacing equals with equals. Clearly, semantic equivalence for PCF should be a congruence.

We call a congruence $\sim$ **compatible** if $\rightarrow_0 \subseteq \sim$. Compatibility means, that s and t are congruent if s can be obtained from t by a top level application of a proper reduction rule. Obviously, semantic equivalence for PCF should be a compatible congruence.

Proposition 3.10.1 If $\sim$ is a compatible congruence, then:

1. $s \to t \Longrightarrow s \sim t$
2. $s \Downarrow v \Longrightarrow s \sim v$

We say that a term t **converges** if there is a value v such that $t \Downarrow v$. A term **diverges** if it doesn't converge. Note that all values converge, and that applications of the form $x\,y$ diverge. **Contextual equivalence** of terms is defined as follows:

$$s \sim t \;:\Longleftrightarrow\; \forall C\colon\; C[s] \text{ converges} \iff C[t] \text{ converges}$$

Proposition 3.10.2 Contextual equivalence is a compatible congruence relation.

At first glance it seems that contextual equivalence makes too many terms equivalent, but this is not the case. To show that two term s, t are not equivalent, it suffices to give a context C such that $C[s]$ converges and $C[t]$ doesn't converge. We say that such a context C **separates** s and t.
Here are examples:

1. The terms 0 and *succ* 0 are not equivalent since they are separated by the context *if iszero* [] *then 0 else 0 0*.
2. Two distinct variables x and y are not equivalent since they are separated by the context $(\lambda x.[]0)(\lambda x.x)$. (We don't give argument types because they don't matter for the example.)
3. The terms 0 and *pred 0* are contextually equivalent since *pred 0* $\to_0$ *0*.

Proposition 3.10.3 $s \not\sim t$ if one of the following conditions holds:

1. There is a context separating s and t.
2. There are a context C and two values v_1, v_2 such that $C[s] \Downarrow v_1$, $C[t] \Downarrow v_2$ and $v_1 \not\sim v_2$.

Proposition 3.10.4 If v_1, v_2 are closed values such that both have eithe type *bool* or *nat*, then $v_1 \sim v_2$ if and only if $v_1 = v_2$.

There are different possibilities for the definition of contextual equivalence. We have chosen one that is technically simple but has the drawback of ignoring types. If you want to know more about contextual equivalence, we recommend the chapter by Andrew Pitts in B. Pierce, Advanced Topics in Types and Programming Languages.

Exercise 3.10.5 Find a context that separates

a) *false* and *true*

b) $succ(succ\ 0)$ and $succ(succ(succ\ 0))$

Exercise 3.10.6 Find two terms $s \sim t$ such that $\emptyset \vdash s : nat$ and $\emptyset \nvdash t : nat$.

Exercise 3.10.7 Let us write $t \Uparrow$ if there is no v such that $t \Downarrow v$. Then, for all terms t and evaluation contexts E, $t \Uparrow$ implies $E[t] \Uparrow$.

a) Does the converse hold, i.e., $E[t] \Uparrow \implies t \Uparrow$?

b) Does the implication hold for arbitrary contexts C, i.e., $t \Uparrow \implies C[t] \Uparrow$?

Exercise 3.10.8 Show that if $t \Uparrow$ then $t \sim \Omega$, where $\Omega = fix\ \lambda x{:}T.x$. **Hint:** Recall that if s converges, then $s \to \ldots \to v$ for some v. Hence, if either of $C[t]$ or $C[\Omega]$ converges, you can use induction on the length of this reduction sequence. To make the induction work, you must generalize to contexts with multiple holes.

Exercise 3.10.9 Suppose $t \sim t'$. Then, for all x and v, $[x := v]t \sim [x := v]t'$.

3.11 Delayed Evaluation

There are two possibilities for the evaluation of applications. In our version of PCF we haven chosen the **call by value** variant, which means that we employ the evaluation rule

$$\frac{t_1 \Downarrow \lambda x{:}T.\,t \qquad t_2 \Downarrow v_2 \qquad [x := v_2]t \Downarrow v}{t_1\, t_2 \Downarrow v}$$

Call by value refers to the fact that the second constituent of an application is evaluated before the procedure is applied. In the **call by name** variant of PCF, one would emply the evaluation rule

$$\frac{t_1 \Downarrow \lambda x{:}T.\,t \qquad [x := t_2]t \Downarrow v}{t_1\, t_2 \Downarrow v}$$

which does not evaluate the second constituent of an application. Algol 60 is an early programming language that comes with call by name procedure application. Most modern programming languages employ call by value procedure application. A noteworthy exception is the functional programming language Haskell.

To study the difference between the two regimes, we extend the values of PCF with so-called **thunks**, which take the syntactic form

$$v \ ::= \ \cdots \ \mid \ lazy\ t$$

A thunk is a term that is only evaluated if its value is really needed. A call by name application can be expressed in PCF as $t_1\ (lazy\ t_2)$. The idea behind thunks

$$\frac{t \Downarrow v}{t \dot{\Downarrow} v}\ v \text{ is not a thunk} \qquad \frac{t \Downarrow \mathit{lazy}\ t' \quad t' \dot{\Downarrow} v}{t \dot{\Downarrow} v}$$

$$\frac{}{v \Downarrow v}\ v \in \{0, \mathit{false}, \mathit{true}\} \cup \mathit{Var} \text{ or } v = \lambda x{:}T.\,t \text{ or } v = \mathit{lazy}\ t$$

$$\frac{t_0 \dot{\Downarrow} \mathit{true} \quad t_1 \Downarrow v}{\mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 \Downarrow v} \qquad \frac{t_0 \dot{\Downarrow} \mathit{false} \quad t_2 \Downarrow v}{\mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 \Downarrow v}$$

$$\frac{t \Downarrow v}{\mathit{succ}\ t \Downarrow \mathit{succ}\ v} \qquad \frac{t \dot{\Downarrow} 0}{\mathit{pred}\ t \Downarrow 0} \qquad \frac{t \dot{\Downarrow} \mathit{succ}\ v}{\mathit{pred}\ t \Downarrow v}$$

$$\frac{t \dot{\Downarrow} 0}{\mathit{iszero}\ t \Downarrow \mathit{true}} \qquad \frac{t \dot{\Downarrow} \mathit{succ}\ v}{\mathit{iszero}\ t \Downarrow \mathit{false}}$$

$$\frac{t_1 \dot{\Downarrow} \lambda x{:}T.\,t \quad t_2 \Downarrow v_2 \quad [x := v_2]t \Downarrow v}{t_1\ t_2 \Downarrow v}$$

$$\frac{t \dot{\Downarrow} v_1 \quad [x := \mathit{fix}\ v_1]t_1 \Downarrow v}{\mathit{fix}\ t \Downarrow v}\ v_1 = \lambda x{:}T.\,t_1$$

Figure 3.8: Lazy and eager evaluation

can be made precise with two evaluation relations $\Downarrow$ (**lazy evaluation**) and $\dot{\Downarrow}$ (**eager evaluation**), which are defined in Figure 3.8 by inference rules. Note that $\Downarrow$ and $\dot{\Downarrow}$ are defined by mutual recursion. Eager evaluation is also referred to as **strict evaluation**.

From the rules in Figure 3.8 we see that lazy evaluation is used for the constituent of *succ* and the second constituent of applications. Eager evaluation is employed for the first constituent of conditionals, *pred*, *iszero*, aplications, and *fix*. These are the constituent where a value is really needed.

The typing rule for thunks is straightforward:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathit{lazy}\ t : T}$$

Alice ML is an extension of Standard ML that has thunks. We use Alice ML to demonstrate a programming technique based on thunks.

A **stream** is a list $v_1 :: v_2 :: \cdots :: (\mathit{lazy}\ t)$. A stream may represent an infinite list. The stream $0 :: 1 :: 2 :: \ldots$ of all natural numbers can be obtained as follows:

```
fun gen n = n :: (lazy gen(n+1))
val nats = gen 0
val nats = 0::_lazy : int list
```

Note that *gen 0* doesn't terminate if the recursion isn't delayed with *lazy*. The procedure

```
fun take 0 xs = nil
  | take n nil = raise Empty
  | take n (x::xr) = x::take (n-1) xr
```

yields the list of the first n elements of a list or stream:

```
take 7 nats
[0, 1, 2, 3, 4, 5, 6] : int list
```

We say that *take k* **forces** the computation of the first k elements of a stream. A lazy map procedure can be defined as follows:

```
fun mapz f nil = nil
  | mapz f (x::xr) = f x :: (lazy mapz f xr)
```

With *mapz* we can obtain the stream of squares from the stream *nats*:

```
val squares = mapz (fn x => x*x) nats
val squares = 0::_lazy : int list

take 7 squares
[0, 1, 4, 9, 16, 25, 36] : int list
```

To obtain the stream of all multiples of 3 and 5 in ascending order, we may use the recursive declaration

```
val rec muls = 1 :: (lazy merge (mul 3 muls) (mul 5 muls))
```

where the procedure *mul* multiplies every element of a stream with a given constant and *merge* merges two ascending streams into one ascending stream:

```
fun mul n = mapz (fn x => n*x)
fun merge nil ys = ys
  | merge xs nil = xs
  | merge (x::xr) (y::yr) = case Int.compare(x,y) of
          LESS => x::(lazy merge xr (y::yr))
        | EQUAL => x::(lazy merge xr yr)
        | GREATER => y::(lazy merge (x::xr) yr)
```

Unfortunately, the declaration of *muls* will be rejected by the current version of Alice. The problem can be circumvented by using a recursion operator *fixz*:

```
val muls = fixz (fn muls => 1 :: (lazy merge (mul 3 muls) (mul 5 muls)))
```

The recursion operator can be declared with futures and promises, two advanced programming constructs available in Alice:

```
fun fixz f = let val p = Promise.promise()
                 val x = Promise.future p
             in Promise.fulfill(p, f x); x end
```

Exercise 3.11.1 (Contextual Equivalence)

a) Are 1 and *lazy 1* contextually equivalent?

b) Give a a term t such that t and *lazy* t are not contextually equivalent.

3.12 Procedural Representation of Thunks

To delay the evaluation of a term t, we can turn it into a thunk *lazy* t. In languages without thunks we can represent *lazy* t as a procedure $\lambda x.t$ where the argument variable x does not occur in t. In contrast to *lazy* t, the eager evaluation of $\lambda x.t$ must be forced explicitly by an application of $\lambda x.t$. Moreover, the type of $\lambda x.t$ is different from the type of t. As it turns out, the stream-based programming techniques demonstrated above work well with the procedural representation of thunks. We refer to the procedural representation of thunks as **λ-lifting**.

We demonstrate stream-based programming in Standard ML. We start by declaring a type constructor for streams:

```
datatype 'a stream = S of 'a * (unit -> 'a stream)
```

The declaration excludes finite streams and employs the procedural representation of thunks. Now the stream of natural numbers and a procedure take can be declared as follows:

```
fun gen n = S(n, fn () => gen(n+1))
val nats = gen 0
fun take n (S(x,p)) = if n<1 then nil else x::take (n-1) (p())
val xs = take 7 nats
val xs = [0, 1, 2, 3, 4, 5, 6] : int list
```

The stream of squares can be obtained as follows:

```
fun maps f (S(x,p)) = S(f x, fn () => maps f (p()))
val squares = maps (fn x => x*x) nats
```

Finally, the stream of all multiples of 3 and 5 in ascending order can be obtained as follows:

```
fun mul k = maps (fn x => k*x)
fun merge (S(x,p)) (S(y,q)) = case Int.compare(x,y) of
        LESS => S(x, fn () => merge (p()) (S(y,q)))
      | EQUAL => S(x, fn () => merge (p()) (q()))
      | GREATER => S(y, fn () => merge (S(x,p)) (q()))
fun gen () = S(1, fn () => let val s = gen() in merge (mul 3 s) (mul 5 s) end)
val muls = gen()
```

Exercise 3.12.1 Write declarations that represent the stream $x_0 :: x_1 :: x_2 :: \cdots$ where

$$\begin{aligned} x_0 &= 0 \\ x_1 &= 1 \\ x_{n+2} &= x_n + x_{n+1} + 1 \end{aligned}$$

a) with *lazy* in Alice ML;

b) with the procedural representation of thunks in Standard ML.

4 Normalization

In this part of the course we consider both simply typed lambda calculus and System T with call-by-value reduction, and prove that every (well-typed) term in these systems reduces to a value. This property is called **normalization**, and is important when viewing these systems as basis for logics. (Recall the problems caused by assuming the existence of a recursion operator in Coq in the previous chapter!)

The proof uses the technique of **logical relations** and largely follows the normalization proof given in Chapter 12 of Pierce's "Types and Programming Languages" book, which is recommended reading.

Although normalization clearly does not extend to real programming languages, the general logical relations proof technique has many further applications, for instance in the study of contextual equivalence.

5 Computational Effects: Exceptions and State

Most programming languages have "impure" features like exceptions, assignment to mutable variables, or concurrent threads. In the literature on programming language theory these features are usually referred to as **computational effects**.

In this part of the course we look at two computational effects in detail: **exceptions** and **state**. It is based on Chapters 13 and 14 of Benjamin Pierce's book. There, it is shown how to extend the simply typed lambda calculus (or PCF) with these new features, and how to adapt the statement of type safety (progress and preservation) accordingly.

6 Hoare Logic

In the preceding chapter we have looked at idealized programming languages, realized as typed lambda calculi. In this section, we will look at a small imperative language that is slightly simpler than lambda calculus with state: it has no procedures, no dynamic memory allocation and only allows the storage of integer values. In return for these simplifications, we will consider a framework – Hoare logic – that permits specifications of program behaviour that are much more precise than the simple type systems we have seen so far. Hoare logic was developed by Tony Hoare in the late 1960's, based on earlier ideas of Robert Floyd for the logical description of flowchart programs.

After presenting syntax and semantics of the programming language IMP, the chapter presents the ingredients of Hoare logic:

- *assertions*, which are logical descriptions of program states,
- *Hoare triples*, which provide specifications for programs through annotation with assertions, and
- *deduction rules*, which form a proof system to derive triples that are *valid*.

This chapter is based on Chapters 2 and 6 of Glynn Winskel's textbook *The Formal Semantics of Programming Languages*. While Winskel uses a different (so-called denotational) semantics of IMP which we have not treated in this course, his book contains many additional explanations and examples for Hoare logic which you may find helpful.

6.1 Syntax and Semantics of Imp

The language consists of three different syntactic categories, *AExp* for **arithmetic expressions**, *BExp* for **boolean expressions**, and *Com* for **commands**:

$$
\begin{aligned}
a \in AExp &:= n \mid X \mid a + a \mid a - a \mid a * a \\
b \in BExp &:= true \mid false \mid a = a \mid a \leq a \mid b \wedge b \mid \neg b \\
c \in Com &:= skip \mid c; c \mid if\ b\ then\ c\ else\ c \mid X := a \mid while\ b\ do\ c
\end{aligned}
$$

where X ranges over a countably infinite set *Loc* of locations.

To give the semantics of IMP programs, we define a set of states,

$$State = Loc \to \mathbb{Z} ,$$

ranged over by σ. The definition of *State* is similar to what we used for the lambda calculus with references, but we exploit the fact that we do not need to model dynamic allocation: instead of finite partial functions we use *total* functions from *Loc* to $\mathbb{Z}$. Intuitively, this means that all variables used in IMP programs are global variables.

Each arithmetic expression a determines a state-dependent integer, i.e., a function $State \to \mathbb{Z}$. This function is written $[\![a]\!]$ and defined by recursion on the structure of a, as follows.

$$
\begin{aligned}
[\![n]\!]\,\sigma &= n \\
[\![X]\!]\,\sigma &= \sigma(X) \\
[\![a_1 + a_2]\!]\,\sigma &= [\![a_1]\!]\,\sigma + [\![a_2]\!]\,\sigma \\
[\![a_1 - a_2]\!]\,\sigma &= [\![a_1]\!]\,\sigma - [\![a_2]\!]\,\sigma \\
[\![a_1 * a_2]\!]\,\sigma &= [\![a_1]\!]\,\sigma * [\![a_2]\!]\,\sigma
\end{aligned}
$$

Analogously, each boolean expression denotes a function $[\![b]\!] : State \to \mathbb{B}$, where $\mathbb{B}$ is the two element set $\{true, false\}$.

$$
\begin{aligned}
[\![true]\!]\,\sigma &= true \\
[\![false]\!]\,\sigma &= false \\
[\![a_1 = a_2]\!]\,\sigma &= true, \text{ if } ([\![a_1]\!]\,\sigma = [\![a_2]\!]\,\sigma), \text{ and } false \text{ otherwise} \\
[\![a_1 \leq a_2]\!]\,\sigma &= true, \text{ if } ([\![a_1]\!]\,\sigma \leq [\![a_2]\!]\,\sigma), \text{ and } false \text{ otherwise} \\
[\![b_1 \wedge b_2]\!]\,\sigma &= true, \text{ if } [\![b_1]\!]\,\sigma = true \text{ and } [\![b_2]\!]\,\sigma = true, \text{ and } false \text{ otherwise} \\
[\![\neg b]\!]\,\sigma &= true, \text{ if } [\![b]\!]\,\sigma = false, \text{ and } false \text{ otherwise}
\end{aligned}
$$

Note that an alternative formulation of these functions could be given as inductively defined evaluation relations, $a|\sigma \Downarrow n$ and $b|\sigma \Downarrow false$ and $b|\sigma \Downarrow true$, resp. The above definition of $[\![a]\!]$ and $[\![b]\!]$ is slightly more convenient to work with in the following, and provides us with yet another perspective on the semantics of these expressions. Also recall that we have used essentially this style of semantics before, when writing the evaluator for PCF in Standard ML.

The semantics of commands is given in Fig. 6.1 by an inductively defined evaluation relation, written $c|\sigma \Downarrow \sigma'$. Note that commands do not return a result but just a new state – commands are evaluated purely for their side-effects on the state.

6.2 Assertions and Hoare Triples

Assertions describe states. Rather than committing to any particular assertion language (such as first-order logic over a suitable signature) and then defining its

$$\frac{}{skip|\sigma \Downarrow \sigma}$$

$$\frac{}{X := a|\sigma \Downarrow \sigma[X := [\![a]\!]\,\sigma]}$$

$$\frac{c_1\sigma \Downarrow \sigma' \qquad c_2|\sigma' \Downarrow \sigma''}{c_1; c_2|\sigma \Downarrow \sigma''}$$

$$\frac{c_1|\sigma \Downarrow \sigma'}{if\ b\ then\ c_1\ else\ c_2|\sigma \Downarrow \sigma'}\ [\![b]\!]\,\sigma = true$$

$$\frac{c_2|\sigma \Downarrow \sigma'}{if\ b\ then\ c_1\ else\ c_2|\sigma \Downarrow \sigma'}\ [\![b]\!]\,\sigma = false$$

$$\frac{c|\sigma \Downarrow \sigma' \qquad while\ b\ do\ c|\sigma' \Downarrow \sigma''}{while\ b\ do\ c|\sigma \Downarrow \sigma''}\ [\![b]\!]\,\sigma = true$$

$$\frac{}{while\ b\ do\ c|\sigma \Downarrow \sigma}\ [\![b]\!]\,\sigma = false$$

Figure 6.1: Evaluation of commands

interpretation, we use the following approach: an assertion is just a set of states,

$$Assn = State \to Prop\ ,$$

and we will freely use mathematical notation to define assertions. We also write $p \subseteq q$ if $p\sigma$ implies $q\sigma$, for all $\sigma \in State$.

A **Hoare triple** $\{p\}c\{q\}$ specifies the behaviour of the program $c \in Com$ in terms of its effect on the state. In $\{p\}c\{q\}$, the assertions $p, q \in Asst$ describe the state before and after evaluation of c; accordingly, p is called a **precondition** and q a **postcondition**. The informal meaning of the triple $\{p\}c\{q\}$ is:

> If c is run in a state where p holds, then if c terminates it will do so in a state where q holds.

Formally, we define a notion of validity for Hoare triples as follows:

Definition 6.2.1 A Hoare triple $\{p\}c\{q\}$ is **valid**, written $\vDash \{p\}c\{q\}$, if

$$\forall \sigma, \sigma' :\ p\,\sigma\ \wedge\ c|\sigma \Downarrow \sigma'\ \Longrightarrow\ q\,\sigma'\ .$$

$$\frac{}{\{p\}\mathit{skip}\{p\}}$$

$$\frac{}{\{\lambda\sigma.\, p(\sigma[X := [\![a]\!]\,\sigma])\}X := a\{p\}}$$

$$\frac{\{p\}c_1\{r\} \qquad \{r\}c_2\{q\}}{\{p\}c_1; c_2\{q\}}$$

$$\frac{\{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{true}\}c_1\{q\} \qquad \{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{false}\}c_2\{q\}}{\{p\}\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2\{q\}}$$

$$\frac{\{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{true}\}c\{p\}}{\{p\}\mathit{while}\ b\ \mathit{do}\ c\{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{false}\}}$$

$$\frac{p \subseteq p' \qquad \{p'\}c\{q'\} \qquad q' \subseteq q}{\{p\}c\{q\}}$$

Figure 6.2: Proof rules of Hoare logic

6.3 Hoare Rules

Fig. 6.2 contains the proof rules of Hoare logic. There is one rule per syntactic construct of the programming language, and one further rule (the rule of **consequence**) that can be used on any Hoare triple $\{p'\}c\{q'\}$ and lets us strengthen its preconditions and weaken its postconditions.

Proposition 6.3.1 (Soundness) If $\{p\}c\{q\}$ is provable with the rules in Fig. 6.2 then $\vDash \{p\}c\{q\}$.

Proof (sketch) It suffices to show that each rule preserves validity. We consider this for two cases:

Assignment We must show $\vDash \{\lambda\sigma.\, p(\sigma[X := [\![a]\!]\,\sigma])\}X := a\{p\}$. Let $\sigma, \sigma' \in \mathit{State}$ such that $p(\sigma[X := [\![a]\!]\,\sigma])$ holds and such that $X := a|\sigma \Downarrow \sigma'$. Then we must prove that $p\,\sigma'$ holds. This immediately follows since σ' equals $\sigma[X := [\![a]\!]\,\sigma]$, by inversion on $\Downarrow$.

While Assume that the hypothesis of the proof rule is valid,

$$\vDash \{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{true}\}c\{p\}.$$

We must show $\vDash \{p\}\mathit{while}\ b\ \mathit{do}\ c\{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{false}\}$. To this end, let $\sigma, \sigma' \in \mathit{State}$ such that $p\,\sigma$ holds and $\mathit{while}\ b\ \mathit{do}\ c|\sigma \Downarrow \sigma'$. We must prove

that $p\,\sigma'$ and $[\![b]\!]\,\sigma' = \mathit{false}$, which is done by induction on $\mathit{while}\ b\ \mathit{do}\ c|\sigma \Downarrow \sigma'$. There are two cases:

- Case $[\![b]\!]\,\sigma = \mathit{false}$ and $\sigma' = \sigma$. Then, both $p\,\sigma'$ and $[\![b]\!]\,\sigma' = \mathit{false}$ follow trivially.
- Case $[\![b]\!]\,\sigma = \mathit{true}$, $c|\sigma \Downarrow \sigma''$ and $\mathit{while}\ b\ \mathit{do}\ c|\sigma'' \Downarrow \sigma'$ for some $\sigma'' \in \mathit{State}$. Since $\vDash \{\lambda\sigma.\, p\sigma \wedge [\![b]\!]\,\sigma = \mathit{true}\}c\{p\}$ we obtain $p\,\sigma''$. Hence $p\,\sigma'$ and $[\![b]\!]\,\sigma' = \mathit{false}$ hold by induction hypothesis.

The cases for the remaining proof rules are similarly straightforward. ∎

7 From Proof to Type Systems

In this chapter we explain some of the logic behind Coq. To do so, we take a historical perspective and look at prominent proof and type systems. It will turn out that proof systems can be obtained as type systems (propositions as types principle). We ignore inductive types.

7.1 Natural Deduction

The development of modern proof systems started in 1935 with Gentzen's [3] system of natural deduction. Gentzen spoke of natural deduction since his goal was a system where the basic structure of proofs would resemble the structure of mathematical proofs. This was not the case for the axiomatic proof systems studied at the time (axiomatic proof systems are often called Hilbert systems). Gentzen did his work for first-order logic. Prawitz [8] is an excellent presentation of natural deduction.

7.1.1 Gentzen's System

Our presentation of Gentzen's system employs **sequents**, which are pairs $\Gamma \Rightarrow s$ consisting of a **context** Γ and a claim s. The context is a finite set of formulas and the claim is a single formula. The context contains the assumption under which the claim is stated. We are familiar with this setup from Coq.

The proof rules of natural deduction derive sequents from sequents. So every proof starts from some sequents that are trivially provable. Figure 7.1 shows the basic rules of natural deduction. The first rule introduces trivial sequents. The notation Γ, s represents the context $\Gamma \cup \{s\}$. Then there are an **introduction rule** and an **elimination rule** for each of the logical constants $\rightarrow$, $\forall$, $\wedge$, $\vee$, and $\exists$. Finally, there is a single rule for the logical constant $\perp$ (False in Coq).

In Gentzen's system the typing rules for the formulas are completely separate from the deduction rules. There are only two base types *Prop* and *Ind* for propositions and for individuals. Quantification is only possible over variables of type *Ind*, hence there is no need to specify the type of a quantified variable. Every variable has a type of the form $Ind \rightarrow \cdots \rightarrow Ind \rightarrow Ind$ or $Ind \rightarrow \cdots \rightarrow Ind \rightarrow Prop$.

$$\frac{}{\Gamma, s \Rightarrow s}$$

$$\frac{\Gamma, s \Rightarrow t}{\Gamma \Rightarrow s \to t} \qquad \frac{\Gamma \Rightarrow s \to t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

$$\frac{\Gamma \Rightarrow s^x_y}{\Gamma \Rightarrow \forall x.\, s}\ y \text{ fresh} \qquad \frac{\Gamma \Rightarrow \forall x.\, s}{\Gamma \Rightarrow s^x_t}$$

$$\frac{\Gamma \Rightarrow s_1 \qquad \Gamma \Rightarrow s_2}{\Gamma \Rightarrow s_1 \wedge s_2} \qquad \frac{\Gamma \Rightarrow s_1 \wedge s_2}{\Gamma \Rightarrow s_i}$$

$$\frac{\Gamma \Rightarrow s_i}{\Gamma \Rightarrow s_1 \vee s_2} \qquad \frac{\Gamma \Rightarrow s_1 \vee s_2 \qquad \Gamma, s_1 \Rightarrow t \qquad \Gamma, s_2 \Rightarrow t}{\Gamma \Rightarrow t}$$

$$\frac{\Gamma \Rightarrow s^x_t}{\Gamma \Rightarrow \exists x.\, s} \qquad \frac{\Gamma \Rightarrow \exists x.\, s \qquad \Gamma, s^x_y \Rightarrow t}{\Gamma \Rightarrow t}\ y \text{ fresh}$$

$$\frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow s}$$

Figure 7.1: Basic rules of natural deduction

The rules for the quantifiers employ **substitution**. The notation s^x_t represents the term that is obtained from s by replacing all free occurrences of the variable x with the term t. The type of x and t must be *Ind*.

Gentzen employed the simple substitution operator that does not rename bound variables and hence may capture. For the rules using the notation s^x_t there is the implicit side condition that the rule can only be applied if the term t does not contain variables that are captured.

The introduction rule for $\forall$ and the elimination rule for $\exists$ do only apply if the auxiliary variable y is fresh. For the introduction rule this means that y does not occur free in $\Gamma \Rightarrow \forall x.\, s$, and for the elimination rule this means that y does not occur free in $\Gamma \Rightarrow t$ and $\exists x.\, s$. Gentzen calls y an **eigenvariable**.

Gentzen's system is modular in that the rules for a logical constant do not involve the other logical constants. If one is not interested in some constant, one simply omits the rules for this constant.

The system in Figure 7.1 is the **intuitionistic system**. The **classical system** is obtained by adding a rule for **proof by contradiction**:

$$\frac{\Gamma, s \to \bot \Rightarrow \bot}{\Gamma \Rightarrow s}$$

The classical system can prove more sequents than the intuitionistic system. By default Coq does not support proof by contradiction, but one can obtain this proof rule by assuming the axiom $\forall X{:}Prop,\ ((X \to \bot) \to \bot) \to X$. Assuming the axiom preserves consistency. A proof system is **consistent** if it cannot prove $\bot$.

Exercise 7.1.1 Prove in Coq that the **double negation law** is equivalent to **Peirce's law**. Note the correspondence between the double negation law and the rule for proof by contradiction.

```
Definition DN := forall p: Prop, ((p->False)->False)->p.
Definition Peirce := forall p q:  Prop, ((p->q)->p)->p.
Proposition DN_Peirce : DN <-> Peirce.
```

7.1.2 Alpha Renaming

One can simplify the quantifier rules if one identifies formulas up to α-renaming. Technically, this means that one has the additional rule

$$\frac{\Gamma \Rightarrow s}{\Gamma \Rightarrow s'}\ s \sim_\alpha s'$$

The quantifier rules with the freshness constraints can now be formulated without eigenvariables and substitution:

$$\frac{\Gamma \Rightarrow s}{\Gamma \Rightarrow \forall x.\, s}\ x \text{ fresh} \qquad \frac{\Gamma \Rightarrow \exists x.\, s \qquad \Gamma, s \Rightarrow t}{\Gamma \Rightarrow t}\ x \text{ fresh}$$

7.1.3 Notation for Proofs

Gentzen gave a notation for proofs in his system that work's well with paper and pencil.

Example 7.1.2 Here is a proof of the sequent $p, q \Rightarrow \forall x.\, (p \to q \to x) \to x$.

$$\dfrac{\dfrac{\dfrac{\dfrac{1 \quad p}{q \to x} \quad q}{x}}{(p \to q \to x) \to x}\ 1}{\forall x.\, (p \to q \to x) \to x}$$

$$
\begin{array}{rcll}
\bot & := & \forall z.\, z & \text{where } z : \mathit{Prop} \\
s \wedge t & := & \forall z.\, (s \to t \to z) \to z & \text{where } z \text{ not free in } s, t \\
s \vee t & := & \forall z.\, (s \to z) \to (t \to z) \to z & \text{where } z \text{ not free in } s, t \\
\exists x.\, s & := & \forall z.\, (\forall x.\, s \to z) \to z & \text{where } z \text{ not free in } s
\end{array}
$$

Figure 7.2: Prawitz's reduction

The trick is to keep the contexts implicit. If we want to refer refer to an assumption introduced by an introduction rule, we refer to it using a numeric tag. In the example the tag 1 is associated with the formula $p \to q \to x$.

Example 7.1.3 Here is a proof of the sequent $\emptyset \Rightarrow p \to q \to p \wedge q$.

$$
\dfrac{\dfrac{\dfrac{1 \qquad 2}{p \wedge q}}{q \to p \wedge q}\,2}{p \to q \to p \wedge q}\,1
$$

Note that the tag 1 is associated with p and tag 2 is associated with q. □

7.1.4 Prawitz's Reduction

In his 1965 dissertation Prawitz [8] continued Gentzen's study of natural deduction. Among other topics Prawitz studied a system for second-order logic. Prawitz noticed that if one extends the first-order system with the ability to quantify over propositions, then one can express the logical constants $\bot$, $\wedge$, $\vee$, and $\exists$ with just implication and universal quantification. Figure 7.2 shows Prawitz's codings of the redundant logical constants. In the extended system one can prove that the coding is equivalent to the original. In the reduced system with just implication and universal quantification one can derive the rules for the codings that correspond to the introduction and elimination rules for the original. Example 7.1.2 derives the introduction rule for the coding of conjunction.

With Prawitz's reduction we can define conjunction, disjunction, existential quantification, and falsity in Coq without using inductive propositions.

Exercise 7.1.4 Derive the introduction and elimination rules for the codings of falsity ($\bot$), conjunction, disjunction, and existential quantification. The derivation of the introduction rule for conjunction is shown in Example 7.1.2.

Exercise 7.1.5 We express Prawitz's reduction in Coq as follows.

```
Definition bot := forall Z: Prop, Z.
Definition con (X Y : Prop) := forall Z: Prop, (X->Y->Z) -> Z.
Definition dis (X Y : Prop) := forall Z: Prop, (X->Z) -> (Y->Z) -> Z.
Definition ex (X : Type) (p : X -> Prop) :=
   forall Z : Prop, (forall x: X, p x -> Z) -> Z.
```

a) Prove the following propositions:

   i) $bot \leftrightarrow False$

   ii) $\forall p\, q.\ con\ p\ q \leftrightarrow p \wedge q$

   iii) $\forall p\, q.\ dis\ p\ q \leftrightarrow p \vee q$

   iv) $\forall X\, (p : X \to Prop).\ ex\ X\ p \leftrightarrow \exists x.px$

b) Formulate and prove theorems that show that the introduction and elimination rules for the coded constants can be derived. The claims for disjunction are as follows.

```
Theorem dis_intro_left: forall X Y :Prop, X -> dis X Y.
Theorem dis_intro_right: forall X Y :Prop, Y -> dis X Y.
Theorem dis_elim: forall X Y Z : Prop, dis X Y -> (X->Z) -> (Y->Z) -> Z.
```

7.1.5 Leibniz Equality

Using quantification over predicates $p : Ind \to Prop$, one can define equality as follows:

$$x = y \ := \ \forall p.\, px \to py$$

The idea goes back to the philosopher and mathematician Leibniz who stated that two objects are equal if and only if they have the same properties. Hence one speaks of **Leibniz equality**. Here is a natural deduction proof that shows that Leibniz equality is symmetric.

$$\dfrac{\dfrac{\dfrac{\dfrac{1}{(px \to px) \to py \to px} \quad p := \lambda z.\, pz \to px \quad \dfrac{2}{px \to px}\ 2}{py \to px}}{\forall p.\, py \to px}}{(\forall p.\, px \to py) \to \forall p.\, py \to px}\ 1$$

The proof assumes that the elimination rule for the universal quantifier β-reduces the instance s^x_t.

Exercise 7.1.6 Prove with natural deduction that Leibniz equality is reflexive.

Exercise 7.1.7 Prove that Coq's predefined equality agrees with Leibniz equality.

```
Definition eq (X:Type) (x y : X) : Prop := forall p : X -> Prop, p x -> p y.
Lemma eq_agrees : forall (X : Type) (x y : X),  eq X x y <-> x=y.
```

$$\frac{}{\Gamma, x : T, \Gamma' \Rightarrow x : T} \qquad \frac{\Gamma \Rightarrow s : T}{\Gamma \Rightarrow s' : T}\ s \sim_\alpha s'$$

$$\frac{\Gamma, x : T \Rightarrow s : U}{\Gamma \Rightarrow \lambda x{:}T.\, s : T \to U} \qquad \frac{\Gamma \Rightarrow s : T \to U \qquad \Gamma \Rightarrow t : T}{\Gamma \Rightarrow st : U}$$

Figure 7.3: Typing rules for STL

7.2 Simply Typed Lambda Calculus

We have already seen a call-by-value variant of the simply typed lambda calculus as a subsystem of PCF and T. We now present a logical variant of the simply typed lambda calculus which we call **STL**. We will see that STL can be seen as a natural deduction system with proof terms where propositions appear as types.

We assume that **type variables** (X) and **term variables** (x) are given. **Types**, **contexts**, and **terms** are defined inductively:

$$\begin{aligned} T &::= X \mid T \to T \\ \Gamma &::= () \mid \Gamma, x : T \\ s &::= x \mid \lambda x{:}T.\, s \mid ss \end{aligned}$$

Note that contexts are now ordered sequences (i.e., lists) of pairs $x : T$. We impose the restriction that a context associates a variable at most once with a type. Hence $(), x : X, x : X$ is not a context. By abuse of notation we write $\Gamma, x : T, \Gamma'$ for the context obtained as the concatenation of the contexts $\Gamma, x : T$ and Γ'.

Figure 7.3 gives the typing rules for STL. Note that the typing rules refine the first three deduction rules in Figure 7.1 by adding variables and proof terms. This is establishes the correspondence between propositions and types. We can understand STL as a proof system for propositional logic with just implication. The advantage of this system is that proofs are represented as terms, that is, as explicit mathematical objects. Note that the term variables in the context take the role of the numeric tags in Gentzen's proof notation.

STL can be extended with product and sum types, which we have seen before in PCF. From the natural deduction perspective product types appear as conjunctions and sum types appear as disjunctions. Moreover, the type unit appears as the proposition true.

We write $s \to_\beta t$ if the term t can be obtained from the term s by replacing

a **β-redex** $(\lambda x{:}T.u)v$ with u^x_v. In contrast to PCF, it does not matter where a β-redex appears. Consequenty, **β-reduction** (i.e., the relation $\rightarrow_\beta$) is no longer deterministic. We define **β-equivalence** as the least equivalence relation $\sim_\beta$ that contains $\rightarrow_\beta$ and $\sim_\alpha$. A term is **β-normal** if it contains no β-redex.

Proposition 7.2.1 (Church-Rosser) Let s and t be terms. Then $s \sim_\beta t$ if and only if there are terms s' and t' such that $s \rightarrow^*_\beta s'$, $t \rightarrow^*_\beta t'$, and $s' \sim_\alpha t'$.

Corollary 7.2.2 Let s and t be β-normal terms. Then $s \sim_\beta t$ if and only if $s \sim_\alpha t$.

Corollary 7.2.3 (Confluence) Let $s \rightarrow^*_\beta s'$ and $t \rightarrow^*_\beta t'$. Then there exist terms s'' and t'' such that $s' \rightarrow^*_\beta s''$, $t' \rightarrow^*_\beta t''$, and $s'' \sim_\alpha t''$.

We write $\Gamma \vdash s : t$ if the sequent $\Gamma \Rightarrow s : t$ is derivable in STL.

Proposition 7.2.4 (Unique Type) If $\Gamma \vdash s : T$ and $\Gamma \vdash s : T'$, then $T = T'$.

Proposition 7.2.5 (Type Preservation) If $\Gamma \vdash s : T$ and $s \rightarrow_\beta t$, then $\Gamma \vdash t : T$.

Proposition 7.2.6 (Termination) If $\Gamma \vdash s : T$, then β-reduction terminates on s.

The properties establish β-reduction as a well-behaved form of computation. Note that Church-Rosser and termination give us a decision procedure that decides β-equivalence for well-typed terms (i.e., for terms s and t such that $\Gamma \vdash s : T$ and $\Gamma \vdash t : T$ for some context Γ and some type T).

In the literature, type preservation is also known as *subject reduction*, and termination is also known as *strong normalization*.

From the natural deduction perspective β-reduction appears as proof transformation. Type preservation says that reduction of a proof does not invalidate it (i.e., yields a proof for the same property as the original proof).

Exercise 7.2.7 Give a term s on which β-reduction does not terminate.

7.3 Calculus of Constructions

It is possible to extend the simply typed lambda calculus such that one obtains a proof system for higher-order logic that subsumes Gentzen' natural deduction system for first-order logic. To do so, one introduces universally quantified types to provide for universal quantification. We have already encountered universally quantified types in the form of polymorphic types in ML and in Coq. The extension of the simply typed lambda calculus to polymorphic types is due to Girard who investigates a **polymorphic lambda calculus** in his dissertation [4]

from 1972. The polymorphic lambda calculus inherits most properties of the simply typed lambda calculus, in particular unique types, type preservation and termination.

In the early 1980's the polymorphic lambda calculus was generalized by Coquand and Huet to the **calculus of constructions** [2], which provided the logical basis for the first version of Coq. In the late 1980's the calculus of constructions was extended with a cumulative hierarchy of universes $\mathbf{U}_0 : \mathbf{U}_1 : \mathbf{U}_2 : \ldots$. The idea of a cumulative hierachy of universes is due to Per Martin-Löf [7] and a such extended calculus of constructions was first investigated by Zhaohui Luo [5]. In the following we present a simplified version of the calculus of constuuction with universes. We call this system **CC_ω**. Our presentation is based on an excellent book by Luo [6].

To obtain a first understanding of CC_ω, it's best to see it as typed functional system extending the simply typed lambda calculus. In CC_ω types have types, and terms are used both for types and their inhabitants. There are three kinds of objects in CC_ω: *functions*, *function types*, and *universes*. Universes serve as the types of function types and as the types of universes.

The universes are $\mathbf{U}_0$, $\mathbf{U}_1$, $\mathbf{U}_2$, and so on. We have $\mathbf{U}_n : \mathbf{U}_{n+1}$ for every natural number n. The hierarchy of universes is cumulative in that every object that has type $\mathbf{U}_n$ also has type $\mathbf{U}_{n+1}$.

A function type takes the form $\forall x{:}s.\,t$ where s and t are types and x is a variable. It is inhabited by functions that take arguments of type s to results of type t. If x occurs in t we have a *dependent function type*, otherwise we have a plain function type that we may write more conventionally as $s \to t$. For instance, we may have the type $\forall x{:}\mathbf{U}_0.\,x \to x$. This type is inhabited by functions that take an argument $t : \mathbf{U}_0$ and an argument $s : t$ and return a result in t. One such function is the polymorphic identity function $\lambda x{:}\mathbf{U}_0.\lambda y{:}x.\,y$.

We now start the formal presentation of CC_ω. We assume a countably infinite set of **variables** (x, y, z) and define **universes**, **terms**, and **contexts** as follows:

$$
\begin{aligned}
U &::= \mathbf{U}_0 \mid \mathbf{U}_1 \mid \mathbf{U}_2 \mid \ldots \\
s &::= x \mid \lambda x{:}s.\,s \mid ss \mid U \mid \forall x{:}s.\,s \\
\Gamma &::= () \mid \Gamma, x : s
\end{aligned}
$$

There is no syntactic constraint on the construction of contexts since functionality will be enforced by the typing rules. **Abstractions** $\lambda x{:}s.\,t$ and **function types** $\forall x{:}s.\,t$ have the same binding structure: The bound variable x ranges over t but not over s. For instance, given the term $\lambda x{:}x.x$, the second occurrence of x is free but the third occurrence of x is bound. We write $s \to t$ as abbreviation for a function type $\forall x{:}s.t$ where x does not occur free in t.

A **sequent** is a triple $\Gamma \Rightarrow s : t$ consisting of a context Γ and two terms s and t. The typing rules for CC_ω derive sequents from sequents and appear in Figure 7.4. Rules Ax and Cum establish the cumulative hierachy of universes. Rule Gam makes it possible to construct well-typed contexts starting from the empty context. The side condition of Gam ensures functionality of contexts. Rule Var makes it possible to infer infer typings that are assumed in the context. The rules Fun provide for the construction of well-typed function types. Given two types s and t in $\mathbf{U}_i$ and a variable x, the function type $\forall x : s.\, t$ is in $\mathbf{U}_i$. In other words, every universe is closed under taking function types. Rule Fun_0 closes $\mathbf{U}_0$ under all function types $\forall x : s.\, t$ such that s is in some universe and t is in $\mathbf{U}_0$. Rule Lam populates function types with abstractions. Rule App provides application terms with types. Rule Alpha provides for alpha renaming (i.e., for consistent renaming of bound variables). Rule Beta provides beta conversion for terms that act as types.

Figure 7.5 shows three derivations in CC_ω. Here are examples of derivable sequents.

1. $\vdash \mathbf{U}_0 \to \mathbf{U}_0 : \mathbf{U}_1$
2. $\vdash (\mathbf{U}_0 \to \mathbf{U}_0) \to \mathbf{U}_0 : \mathbf{U}_1$
3. $\vdash \forall x : \mathbf{U}_0.\, x : \mathbf{U}_1$
4. $\vdash \forall x : \mathbf{U}_0.\, x \to x : \mathbf{U}_1$
5. $\vdash \lambda x : \mathbf{U}_0.\, \lambda y : x.\, y : \forall x : \mathbf{U}_0.\, x \to x$
6. $\vdash (\lambda x : \mathbf{U}_1.\, x)\mathbf{U}_0 : \mathbf{U}_1$
7. $\vdash \mathbf{U}_0 : (\lambda x : \mathbf{U}_2.\, x)\mathbf{U}_1$

The original calculus of constructions is the subsystem of CC_ω where sequents can only employ the universes $\mathbf{U}_0$ and $\mathbf{U}_1$. Girard's polymorphic lambda calculus is obtained by the further restriction that function types $\forall x : s.\, t$ where $s : \mathbf{U}_0$ and $t : \mathbf{U}_1$ are disallowed. The simply typed lambda calculus is obtained by only admitting function types $\forall x : s.\, t$ where $s : \mathbf{U}_0$ and $t : \mathbf{U}_0$.

The terms of type $\mathbf{U}_0$ can be seen as propositions. If we just admit the first three universes $\mathbf{U}_0$, $\mathbf{U}_1$, $\mathbf{U}_2$ in derivations, we obtain a proof system for a higher-order logic that is more expressive than simple type theory (e.g., one can quantify over types). In Coq, *Prop* stands for $\mathbf{U}_0$ and *Type* stands for all universes $\mathbf{U}_i$ with $i \geq 1$.

7.4 Properties of CC_ω

We state some properties of CC_ω. Proofs of related properties can be found in Luo [6].

$$\text{Ax}\quad \frac{}{\Rightarrow \mathbf{U}_i : \mathbf{U}_{i+1}}$$

$$\text{Cum}\quad \frac{\Gamma \Rightarrow s : \mathbf{U}_i}{\Gamma \Rightarrow s : \mathbf{U}_{i+1}}$$

$$\text{Gam}\quad \frac{\Gamma \Rightarrow s : \mathbf{U}_k}{\Gamma, x : s \Rightarrow \mathbf{U}_i : \mathbf{U}_{i+1}}\quad x \notin \mathrm{FV}\,\Gamma$$

$$\text{Var}\quad \frac{\Gamma, x : s, \Gamma' \Rightarrow \mathbf{U}_0 : \mathbf{U}_1}{\Gamma, x : s, \Gamma' \Rightarrow x : s}$$

$$\text{Fun}\quad \frac{\Gamma \Rightarrow s : \mathbf{U}_i \qquad \Gamma, x : s \Rightarrow t : \mathbf{U}_i}{\Gamma \Rightarrow \forall x : s.\, t : \mathbf{U}_i}$$

$$\text{Fun}_0\quad \frac{\Gamma, x : s \Rightarrow t : \mathbf{U}_0}{\Gamma \Rightarrow \forall x : s.\, t : \mathbf{U}_0}$$

$$\text{Lam}\quad \frac{\Gamma, x : s \Rightarrow t : u}{\Gamma \Rightarrow \lambda x : s.\, t : \forall x : s.\, u}$$

$$\text{App}\quad \frac{\Gamma \Rightarrow s : \forall x : u.\, v \qquad \Gamma \Rightarrow t : u}{\Gamma \Rightarrow st : v^x_t}$$

$$\text{Alpha}\quad \frac{\Gamma \Rightarrow s : t}{\Gamma \Rightarrow s' : t'}\quad s \sim_\alpha s' \text{ and } t \sim_\alpha t'$$

$$\text{Beta}\quad \frac{\Gamma \Rightarrow s : t \qquad \Gamma \Rightarrow t' : \mathbf{U}_i}{\Gamma \Rightarrow s : t'}\quad t \sim_\beta t'$$

Figure 7.4: Typing rules for CC_ω

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow \mathbf{U}_0 : \mathbf{U}_1}\,\mathrm{Ax}}{x : \mathbf{U}_0 \Rightarrow \mathbf{U}_0 : \mathbf{U}_1}\,\mathrm{Gam}}{x : \mathbf{U}_0 \Rightarrow x : \mathbf{U}_0}\,\mathrm{Var}}{x : \mathbf{U}_0, y : x \Rightarrow \mathbf{U}_0 : \mathbf{U}_1}\,\mathrm{Gam}}{x : \mathbf{U}_0, y : x \Rightarrow x : \mathbf{U}_0}\,\mathrm{Var}}{x : \mathbf{U}_0 \Rightarrow x \to x : \mathbf{U}_0}\,\mathrm{Fun}_0}{\Rightarrow \forall x{:}\mathbf{U}_0.\, x \to x : \mathbf{U}_0}\,\mathrm{Fun}_0$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow \mathbf{U}_1 : \mathbf{U}_2}\,\mathrm{Ax}}{x : \mathbf{U}_1 \Rightarrow \mathbf{U}_0 : \mathbf{U}_1}\,\mathrm{Gam}}{x : \mathbf{U}_1 \Rightarrow x : \mathbf{U}_1}\,\mathrm{Var}}{x : \mathbf{U}_1, y : x \Rightarrow \mathbf{U}_0 : \mathbf{U}_1}\,\mathrm{Gam}}{x : \mathbf{U}_1, y : x \Rightarrow x : \mathbf{U}_1}\,\mathrm{Var}}{x : \mathbf{U}_1, y : x \Rightarrow x : \mathbf{U}_2}\,\mathrm{Cum}$$

$$\dfrac{\dfrac{}{\Rightarrow \mathbf{U}_1 : \mathbf{U}_2}\,\mathrm{Ax} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow \mathbf{U}_1 : \mathbf{U}_2}\,\mathrm{Ax}}{x : \mathbf{U}_1 \Rightarrow \mathbf{U}_0 : \mathbf{U}_1}\,\mathrm{Gam}}{x : \mathbf{U}_1 \Rightarrow x : \mathbf{U}_1}\,\mathrm{Var} \qquad x : \mathbf{U}_1, y : x \Rightarrow x : \mathbf{U}_2}{x : \mathbf{U}_1 \Rightarrow x \to x : \mathbf{U}_2}\,\mathrm{Fun}}{\Rightarrow \forall x{:}\mathbf{U}_1.\, x \to x : \mathbf{U}_2}\,\mathrm{Fun}$$

Figure 7.5: Derivations in CC_ω

Proposition 7.4.1 (Church-Rosser) Let s and t be terms. Then $s \sim_\beta t$ if and only if there are terms s' and t' such that $s \to_\beta^* s'$, $t \to_\beta^* t'$, and $s' \sim_\alpha t'$.

A term is **canonical** if it is either an abstraction, or a function type, or a universe.

Proposition 7.4.2 (Canonical Terms) Let $\vdash s : t$ and let s and t be β-normal. Then s and t are canonical and t is not an abstraction. Moreover:

1. s is an abstraction if and only if t is a function type.
2. If s is a universe, then t is a higher universe.
3. If t is a universe, then s is a lower universe or a function type.

Proposition 7.4.3 If $\Gamma \vdash s : t$, then $\Gamma \vdash t : \mathbf{U}_i$ for some i.

Proposition 7.4.4 If $\Gamma, x : s \vdash t : u$, then $\Gamma \vdash s : \mathbf{U}_i$ for some i.

Proposition 7.4.5 If $\Gamma \vdash s : t$ and $\Gamma = (), x_1 : u_1, \ldots, x_n : u_n$, then $x_1, \ldots, x_n$ are pairwise distinct and the free variable of s and t appear in $x_1, \ldots, x_n$.

Proposition 7.4.6 (Type Preservation) If $\Gamma \vdash s : t$ and $s \to_\beta s'$, then $\Gamma \vdash s' : t$.

Proposition 7.4.7 (Termination) If $\Gamma \vdash s : t$, then β-reduction terminates on s.

7.5 Consistency

Coq employs CC_ω as a proof system. $\mathbf{U}_0$ serves as the type of propositions. The other universes are commonly displayed as Type. Implication and universal quantification are expressed as function types. The other logical connectives can be expressed via the Prawitz codings. Recall that falsity can be expressed as $\bot := \forall x : \mathbf{U}_0 . x$.

Proposition 7.5.1 (Consistency) There is no term s such that $\vdash s : \bot$.

Proof The prrof is by contradiction and uses Type preservation and termination. See Luo [6] (Theorem 5.4) for more information. ∎

Consistency is a necessary property so that CC_ω can serve as a proof system. Several extensions of CC_ω are known to be inconsistent. Girard [4] proved that adding the rule

$$\text{Poly} \quad \frac{\Gamma, x : \mathbf{U}_1 \Rightarrow t : \mathbf{U}_1}{\Gamma \Rightarrow \forall x : \mathbf{U}_1 . t : \mathbf{U}_1}$$

yields an inconsistent system even if derivations can only use the first three universes. From that it follows that adding

$$\text{Circ}_1 \quad \frac{}{\Rightarrow \mathbf{U}_1 : \mathbf{U}_1}$$

also yields an inconsistent system. This explains why CC_ω uses a non-circular hierachy of universes.

What happens if we just have a single universe $\mathbf{U}_0$ and a rule that asserts $\Rightarrow \mathbf{U}_0 : \mathbf{U}_0$? Once again we get an inconsistent system. To know more about consistency consult Barendregt [1].

7.6 Church-Girard Programming

Church discovered that the boolean values and the natural numbers can be expressed as functions in the untyped lambda calculus. The idea is straightforward.

$$\begin{aligned}
\mathit{false} &:= \lambda xy.\, y \\
\mathit{true} &:= \lambda xy.\, x \\
0 &:= \lambda fx.\, x \\
1 &:= \lambda fx.\, fx \\
2 &:= \lambda fx.\, f(fx)
\end{aligned}$$

It turns out that all computable functions on natural numbers can be expressed in the untyped lambda calculus. The terms coding the numbers are called **Church numerals**.

Girard showed that Church's codings carry over to the polymorphic lambda calculus. One obtains a terminating functional programming system that is more expressive than T. The boolean values and the numbers now appear as polymorphic functions. Here is the coding of bool.

$$\begin{aligned}
\mathit{bool} &:= \forall X{:}\mathbf{U}_0.\, X \to X \to X \\
\mathit{false} &:= \lambda X{:}\mathbf{U}_0.\, \lambda x{:}X.\, \lambda y{:}X.\, y \\
\mathit{true} &:= \lambda X{:}\mathbf{U}_0.\, \lambda x{:}X.\, \lambda y{:}X.\, x
\end{aligned}$$

One can show that *true* and *false* are the only closed β-normal terms of type *bool* (up to α-equivalence).

The coding of the natural numbers is as follows.

$$\mathit{nat} := \forall X{:}\mathbf{U}_0.\, (X \to X) \to X \to X$$

$$
\begin{aligned}
0 &:= \lambda X{:}\mathbf{U}_0.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ x \\
1 &:= \lambda X{:}\mathbf{U}_0.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ fx \\
2 &:= \lambda X{:}\mathbf{U}_0.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ f(fx)
\end{aligned}
$$

One can show that the Church numerals are the only closed β-normal terms of type *nat* (up to α-equivalence). The successor and the iteration function can be expressed as follows.

$$
\begin{aligned}
S &:= \lambda n{:}nat.\ \lambda X{:}\mathbf{U}_0.\ \lambda f{:}X{\to}X.\ \lambda x{:}X.\ nXf(fx) \\
iter &:= \lambda X{:}\mathbf{U}_0.\ \lambda n{:}nat.\ nX
\end{aligned}
$$

Addition, multiplication, and exponention are straightforward:

$$
\begin{aligned}
add &:= \lambda m{:}nat.\ \lambda n{:}nat.\ m\,nat\,S\,n \\
mul &:= \lambda m{:}nat.\ \lambda n{:}nat.\ m\,nat\,(add\,n)\,0 \\
exp &:= \lambda m{:}nat.\ \lambda n{:}nat.\ \lambda X{:}\mathbf{U}_0.\ n(X{\to}X)(mX)
\end{aligned}
$$

At first view this may look like magic. Use Coq to get used to it. Use the command

```
Eval compute in exp (S(S(S O))) (S(S O)).
```

to see the Church numeral the term $exp\ (S(S(SO)))\ (S(SO))$ β-reduces to.

There are two problems with the Curch-Girard codings:

1. Selectors like predecessor for natural numbers don't have the right complexity (linear rather than constant).
2. Basic properties are not provable, for instance, $0 \neq S0$.

For this reason Coq has inductive types that avoid both problems.

Exercise 7.6.1 Give a term of type $\forall X{:}\mathbf{U}_0.\ bool \to X \to X \to X$ that acts as conditional.

7.6.1 Products and Sums

One can define a type constructor $prod : \mathbf{U}_0 \to \mathbf{U}_0 \to \mathbf{U}_0$ that yields for two types s and t a type that represents the pairs in $s \times t$.

$$
prod := \lambda X{:}\mathbf{U}_0.\ \lambda Y{:}\mathbf{U}_0.\ \forall Z{:}\mathbf{U}_0.\ (X \to Y \to Z) \to Z
$$

The constructor and the projections for pairs can be expressed as follows:

$$
\begin{aligned}
pair &:= \lambda X{:}\mathbf{U}_0.\ \lambda Y{:}\mathbf{U}_0.\ \lambda x{:}X.\ \lambda y{:}Y.\ \lambda Z{:}\mathbf{U}_0.\ \lambda f{:}X{\to}Y{\to}Z.\ fxy \\
fst &:= \lambda X{:}\mathbf{U}_0.\ \lambda Y{:}\mathbf{U}_0.\ \lambda p{:}prod\,X\,Y.\ pY(\lambda x{:}X.\ \lambda y{:}Y.\ x)
\end{aligned}
$$

$$snd := \lambda X{:}\mathbf{U}_0.\ \lambda Y{:}\mathbf{U}_0.\ \lambda p{:}prod\,X\,Y.\ p\,Y(\lambda x{:}X.\ \lambda y{:}Y.\ x)$$

Note that $prod\,s\,t$ is the Prawitz coding of the conjunction $s \wedge t$, and that *pair*, *fst*, and *snd* are proof terms for propositions that state the introduction and elimination rules for conjunction.

$$\begin{aligned}
prod &: \mathbf{U}_0 \to \mathbf{U}_0 \to \mathbf{U}_0 \\
pair &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ X \to Y \to prod\,X\,Y \\
fst &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ prod\,X\,Y \to X \\
snd &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ prod\,X\,Y \to Y
\end{aligned}$$

Experiment with the codings in Coq. Here is one way they can be written in Coq.

```
Definition prod : Prop -> Prop -> Prop
:= fun X Y => forall Z : Prop, (X -> Y -> Z) -> Z.
Definition pair : forall X Y : Prop, X -> Y-> prod X Y
:= fun X Y x y  Z f => f x y.
Definition fst : forall X Y : Prop, prod X Y -> X
:= fun X Y p => p X (fun x y => x).
Definition snd : forall X Y : Prop, prod X Y -> Y
:= fun X Y p => p Y (fun x y => y).
```

Exercise 7.6.2 Express sum types in CC_ω and in Coq. That is, give closed terms of the following types.

$$\begin{aligned}
sum &: \mathbf{U}_0 \to \mathbf{U}_0 \to \mathbf{U}_0 \\
inl &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ X \to sum\,X\,Y \\
inr &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ Y \to sum\,X\,Y \\
case &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ \forall Z{:}\mathbf{U}_0.\ sum\,X\,Y \to (X \to Z) \to (Y \to Z) \to Z
\end{aligned}$$

Exercise 7.6.3 Express polymorphic lists in CC_ω and in Coq. That is, give closed terms of the following types.

$$\begin{aligned}
list &: \mathbf{U}_0 \to \mathbf{U}_0 \\
nil &: \forall X{:}\mathbf{U}_0.\ list\,X \\
cons &: \forall X{:}\mathbf{U}_0.\ X \to list\,X \to list\,X \\
foldl &: \forall X{:}\mathbf{U}_0.\ \forall Y{:}\mathbf{U}_0.\ (X \to Y \to Y) \to Y \to list\,X \to Y
\end{aligned}$$

Exercise 7.6.4 Express the predecessor function for *nat* in CC_ω. The trick is to iterate through the pairs $(0,0),\ (1,0),\ (2,1),\ \ldots,\ (n, n-1)$.

Exercise 7.6.5 Express primitive recursion for *nat* in CC_ω.

Bibliography

[1] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[2] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[3] Gerhard Gentzen. Untersuchungen über das natürliche Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[4] J.Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. *Thèse de doctorat d'etat, Université Paris VII*, 1972.

[5] Zhaohui Luo. ECC, an extended calculus of constructions. In *Logic in Computer Science (LICS)*, pages 386–395, 1989.

[6] Zhaohui Luo. *Computation and reasoning: a type theory for Computer Science*. Oxford University Press, 1994.

[7] Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[8] Dag Prawitz. *Natural deduction*. Dover Publications, 2006. Reprint of the dissertation from 1965.