Lecture Notes for Semantics (WS 07/08)

Gert Smolka and **Jan Schwinghammer**
Saarland University

February 18, 2008

The course will mostly follow the book *Types and Programming Languages* by Benjamin Pierce, which is required reading. Most of the additional material presented in the lectures will be covered in these notes. You are advised to take your own notes during lectures.

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

$$\begin{aligned} a &= 14 \\ b &= 20 \end{aligned}$$

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 $c = 50$. 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

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

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:

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

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

$$\begin{aligned}&cas : (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma\\&car : (\alpha \to \beta \to \gamma) \to \alpha * \beta \to \gamma\end{aligned}$$

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:

$$\begin{aligned}&f^0 x = x\\&f^n x = f^{n-1}(fx) \qquad \text{if } n > 0\end{aligned}$$

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

$$\begin{aligned}&iter\ 0\ x\ f = x\\&iter\ n\ x\ f = iter\ (n-1)\ (fx)\ f \qquad \text{if } n > 0\end{aligned}$$

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:

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

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

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

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 $\mathit{fib} : \mathit{int} \to \mathit{int}$ that satisfies the equation $\mathit{fib}\ n = \text{if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2)$.

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

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 : ((α → β) → α → β) → α → β

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:

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

Length and concatenation can be computed recursively:

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

$$
\begin{aligned}
nil\,@\,ys &= ys \\
(x :: xr)\,@\,ys &= x :: (xr\,@\,ys)
\end{aligned}
$$

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.

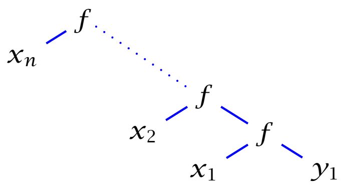
```
fun len xs = fix (fn f => (fn nil => 0 | x::xr => 1 + f xr)) xs
```

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* → α *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 $mapr\ f\ [x_1, \dots, x_n] = [f\,x_n, \dots, f\,x_1]$. 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 $list\ n\ f = [f\ 1, \dots f\ n]$. Use *iter* to realize the necessary recursion.

2 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.

2.1 Abstract Syntax

Figure 2.1 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 $\mathit{Var} \subseteq \mathit{Val} \subseteq \mathit{Ter}$. 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 instance, an abstraction $\lambda x{:}T.\,t$ is represented as the pair $\langle 8, \langle x, T, t\rangle\rangle$, where 8 is the variant number and $\langle x, T, t\rangle$ is the constituent tuple. Abstractions have the variant number 9 since this is what we obtain if we number the syntactic variants introduced in Figure 2.1 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.

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

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

Figure 2.1: PCF: Abstract syntax

```
datatype ty = Bool | Int | Proc of ty * ty
type var = string
datatype ter =
    False | True | If of ter*ter*ter
  | 0 | 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 0 = 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 2.1.1 (Values) Write a procedure *value* : *term* → *bool* that tests whether a term is a value.

Exercise 2.1.2 (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 2.2: Inference rules defining the typing relation

2.2 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 y. 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 2.2.

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

Proposition 2.2.2 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 2.3: 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 2.2.3 (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
```

2.3 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 2.3. 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 2.1.2).

The rules defining the evaluation relation are agorithmic 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 2.3.1 (Determinism) Given t, there is at most one v such that $t \Downarrow v$.

Proposition 2.3.2 (Type Preservation) If $\Gamma \vdash t : T$ and $t \Downarrow v$, then $\Gamma \vdash v : T$.

Exercise 2.3.3 (Evaluation) Write a procedure $\mathit{eval} : \mathit{ter} \to \mathit{ter}$ 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 2.3.4 Given a well-typed closed term t for which there is no value v such that $t \Downarrow v$.

Exercise 2.3.5 The evaluation relation is defined for all terms, not just well-typed terms. Explain why the term $(\lambda x{:}\mathit{int}.xx)(\lambda x{:}\mathit{int}.xx)$ does not evaluate to a value.

2.4 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** $\to \in \mathit{Ter} \times \mathit{Ter}$ 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 2.4. 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 2.4: 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 2.4 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 2.4 we obtain a deterministic reduction relation.

Proposition 2.4.1 (Determinism) Given t, there is at most one term t' such that $t \to t'$.

Proposition 2.4.2 (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 2.4.3 (Progress) Every closed and well-typed term is either a value or reducible.

Proposition 2.4.4 (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 2.4.5 (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.

2.5 Evaluation Contexts

A context is term with a hole. **Evaluation contexts** are defined as follows:

$$C \ ::= \ [] \ | \ \mathit{if}\ C\ \mathit{then}\ t\ \mathit{else}\ t \ | \ \mathit{succ}\ C \ | \ \mathit{pred}\ C \ | \ \mathit{iszero}\ C \ | \ C\ t \ | \ v\ C \ | \ \mathit{fix}\ C$$

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** $\to_0 \subseteq \mathit{Ter} \times \mathit{Ter}$ is the relation defined by the proper reduction rules. Note that $\to_0 \subseteq \to$.

A term t is called a **redex** if there is a term t' such that $t \to_0 t'$. An evaluation context C is called a **reduction context for a term** t if there exists a **redex** s such that $t = C[s]$.

Proposition 2.5.1 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 2.5.2 For all terms t, t':

$$t \to t' \iff \exists s, s', C\colon\ t = C[s] \ \wedge \ s \to_0 s' \ \wedge \ C[s'] = t'$$

2.6 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 start with the definition of a set of **semantic values**:

SV	$:=$	B of $\{0,1\}$	Boolean value
	$\mid$	N of $\mathbb{N}$	natural number
	$\mid$	C of $Var \times Ter \times (Var \rightharpoonup SV)$	closure
	$\mid$	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, $\mathsf{B}\,0 = (1,0)$ and $\mathsf{N}\,n = (2,n)$.

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 $(Var \rightharpoonup SV)\times Ter\times SV$. We write $\phi \vdash t \triangleright w$ if the triple (ϕ,t,w) is an element of the closure-evaluation relation. We define the closure-evaluation relation such that $\phi \vdash t \triangleright w$ if and only if the term t evaluates to the semantic value w in the environment ϕ. Figure 2.5 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 2.5, *fix* can only be applied to double abstractions. More general uses of *fix* can be compiled as follows

$$\mathit{fix}\ t \ \rightsquigarrow\ (\lambda p.\mathit{fix}(\lambda f.\lambda x.p\,f\,x))t$$

$$\frac{}{\phi \vdash x \rhd w}\ (x,w) \in \phi \qquad \frac{}{\phi \vdash \mathit{false} \rhd \mathsf{B}0} \qquad \frac{}{\phi \vdash \mathit{true} \rhd \mathsf{B}1}$$

$$\frac{\phi \vdash t_0 \rhd \mathsf{B}0 \qquad \phi \vdash t_2 \rhd w}{\phi \vdash \mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 \rhd w} \qquad \frac{\phi \vdash t_0 \rhd \mathsf{B}1 \qquad \phi \vdash t_1 \rhd w}{\phi \vdash \mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 \rhd w}$$

$$\frac{}{\phi \vdash 0 \rhd \mathsf{N}0} \qquad \frac{\phi \vdash t \rhd \mathsf{N}n}{\phi \vdash \mathit{succ}\ t \rhd \mathsf{N}n}\ m = n + 1$$

$$\frac{\phi \vdash t \rhd \mathsf{N}n}{\phi \vdash \mathit{pred}\ t \rhd \mathsf{N}m}\ m = \mathit{if}\ n = \mathit{0}\ \mathit{then}\ \mathit{0}\ \mathit{else}\ n - \mathit{1}$$

$$\frac{\phi \vdash t \rhd \mathsf{N}n}{\phi \vdash \mathit{iszero}\ t \rhd \mathsf{B}m}\ m = \mathit{if}\ n = \mathit{0}\ \mathit{then}\ \mathit{1}\ \mathit{else}\ \mathit{0}$$

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

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

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

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

Figure 2.5: Inference rules defining the closure-evaluation relation

The closure semantics agrees with the big step semantics if *fix* is only applied to double abstractions. Since the two semantics use different representations of values, expressing this statement precisely (what exactly means agree?) is tedious. However, its easy to state the following:

Proposition 2.6.1 For every closed term t that applies *fix* only to double abstractions, we have the following: $(\exists v:\ t \Downarrow v) \iff (\exists w:\ \emptyset \vdash t \rhd w)$.

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

2.7 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:

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

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 $s \sim t \Rightarrow C[s] \sim C[t]$ 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 2.7.1 If $\sim$ is a compatible congruence, then:

1. $s \rightarrow t \Rightarrow s \sim t$
2. $s \Downarrow v \Rightarrow s \sim v$

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:

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

Proposition 2.7.2 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 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* $\rightarrow_0$ *0*.

Proposition 2.7.3 $s \not\sim t$ 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 2.7.4 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 2.7.5 Find a context that separates

a) *false* and *true*

b) *succ*(*succ 0*) and *succ*(*succ*(*succ 0*))

Exercise 2.7.6 Find two terms $s \sim t$ such that $\emptyset \vdash s : nat$ and $\emptyset \nvdash t : nat$.

Exercise 2.7.7 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 2.7.8 Show that if $t \Uparrow$ then $t \sim \Omega$, where $\Omega = \mathit{fix}\ \lambda x{:}T.x$. **Hint:** Recall that if s converges, then $s \rightarrow \ldots \rightarrow v$ 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 2.7.9 Suppose $t \sim t'$. Then, for all x and v, $[x := v]t \sim [x := v]t'$.

2.8 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

$$v \;::=\; \cdots \;\mid\; \mathit{lazy}\ t$$

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 $t_1\ (\mathit{lazy}\ t_2)$. The idea behind thunks can be made precise with two evaluation relations $\Downarrow$ (**lazy evaluation**) and $\dot{\Downarrow}$ (**eager evaluation**), which are defined in Figure 2.6 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 2.6 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:

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

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

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

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

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

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

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

Figure 2.6: Lazy and eager evaluation

```
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 2.8.1 (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.

2.9 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 2.9.1 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.

3 Untyped Lambda Calculus

We consider the untyped call-by-value λ-calculus. This is a sublanguage of untyped PCF where procedures are the only closed values.

3.1 Terms and Substitutions

The **variables** and **terms** of the untyped λ-calculus are defined as follows:

$$\begin{aligned} x, y, z \in \mathit{Var} &::= \mathbb{N} \\ t, s \in \mathit{Ter} &::= x \mid t\,t \mid \lambda x.t \end{aligned}$$

Contexts are defined in the canonical way:

$$C ::= [] \mid Ct \mid tC \mid \lambda x.C$$

A term s is a **subterm** of a term t if there exists a context C such that $t = C[s]$. A term is **combinatorial** if none of its subterms is an abstraction (i.e., of the form $\lambda x.t$). **Size** and **free variables** of terms are defined as follows:

$$\begin{aligned} |x| &= 1 & \qquad \mathit{FV} &= \{x\} \\ |s\,t| &= 1 + |s| + |t| & \mathit{FV}(s\,t) &= \mathit{FV}\,s \cup \mathit{FV}\,t \\ |\lambda x.t| &= 1 + |t| & \mathit{FV}(\lambda x.t) &= \mathit{FV}\,t - \{x\} \end{aligned}$$

A term is **closed** if it has no free variables, and **open** otherwise. Closed abstractions are called **combinators** or **procedures**. The simplest procedure is the **identity**

$$I := \lambda x.x$$

The definition of the substitution operation $[x{:=}s]t$ is straightforward if s is closed or t is combinatorial: Replace every free occurrence of x in t with s. Otherwise, substitution must be defined such that **capture** is avoided by **renaming of bound variables**. For instance, given two distinct variables x and y, the substitution $[x{:=}y](\lambda y.x)$ must not yield $\lambda y.y$ but $\lambda z.y$ for some variable $z \neq y$.

Because of the renaming, things become easier if we work with a generalized notion of substitution. From now on, a substitution will be a function $\mathit{Var} \to \mathit{Ter}$:

$$\theta \in \mathit{Sub} := (\mathit{Var} \to \mathit{Ter}) \qquad\qquad \textbf{substitution}$$

$$\begin{aligned} \epsilon &:= \lambda x{\in}Var.\,x && \textbf{identity substitution} \\ [x{:=}s] &:= \epsilon[x{:=}s] && \\ Ker\,\theta &:= \{\, x \in Var \mid \theta x \neq x \,\} && \textbf{kernel} \end{aligned}$$

A **substitution operator** is a function S: $Sub \to Ter \to Ter$ that applies a substitution θ to a term t. The result should be a term that can be obtained from t by replacing the free variable occurrences in t according to θ, where capture is avoided by renaming of bound variables. This leads us to the following definition (recursion on term size):

$$\begin{aligned} S\theta x &= \theta x \\ S\theta(st) &= (\theta s)(\theta t) \\ S\theta(\lambda x.t) &= \lambda y.\,S(\theta[x{:=}y])t \qquad \text{where } y = \rho(x, V) \\ & \qquad\qquad\qquad\qquad V = \cup\{\, FV(\theta y) \mid y \in FV(\lambda x.t) \,\} \end{aligned}$$

The interesting part is the potential renaming of the bound variable x in the third equation. First note that the finite set V contains exactly those variables that would capture variable occurrences if used as argument variable. The **choice function** ρ will chose a variable that is not in V.

There are different possibilities for the choice function. A choice function ρ is **admissible** if $\rho(x, V) \in (Var - V)$ for all x and V. Every admissible choice function yields a substitution operator that does what it is supposed to do. We will only consider **substitution operators** that are based on admissible choice functions. The most obvious admissible choice function is

$$\rho_0(x, V) \; := \; \min(Var - V)$$

We use S_0 for the substitution operator obtained with ρ_0. For hand calculation ρ_0 has the (aesthetic) drawback that it renames argument variables even if this is not necessary. For instance, $S_0\epsilon(\lambda x.x) = \lambda 0.0$ for every variable x.

Here is a choice function that avoids renaming whenever possible:

$$\rho_1(x, V) \; := \; \mathit{if}\ x \notin V\ \mathit{then}\ x\ \mathit{else}\ \min(Var - V)$$

We use S_1 for the substitution operator obtained with ρ_1. We write θt for $S_1\theta t$ and $[x{:=}s]t$ for $S_1([x{:=}s])t$.

Proposition 3.1.1 (Coincidence) For every substitution operator S:
$(\forall x \in FV\,t\colon\ \theta x = \theta' x) \implies S\theta t = S\theta' t$

Proposition 3.1.2 (Free Variables) For every substitution operator S:
$FV(S\theta t) = \cup\{\, FV(\theta y) \mid y \in FV(\lambda x.t) \,\}$

Proposition 3.1.3 S_1 satisfies the following properties:

1. $\theta t = t$ if $\forall x \in FV\,t\colon\ \theta x = x$
2. $\epsilon t = t$
3. $[x{:=}s](\lambda x.t) = \lambda x.t$
4. $[x{:=}s]t = t$ if $x \notin FV\,t$
5. $[x{:=}s](\lambda y.t) = \lambda y.[x{:=}s]t$ if $x \neq y$ and $y \notin FV\,s$

3.2 Alpha Equivalence

Informally, two terms are α-equivalent if they are equal up to consistent renaming of bound variables. Formally, we define **α-equivalence** as follows (by recursion on term size):

$$\begin{aligned} x \sim_\alpha x' &\iff x = x' \\ st \sim_\alpha s't' &\iff s \sim_\alpha s' \ \wedge\ t \sim_\alpha t' \\ \lambda x.t \sim_\alpha \lambda x'.t' &\iff x' \notin FV(\lambda x.t) \ \wedge\ [x{:=}x']t \sim_\alpha t' \end{aligned}$$

Proposition 3.2.1 All substitution operators yield the same relation $\sim_\alpha$.

Proposition 3.2.2 $\sim_\alpha$ is a congruence such that $s \sim_\alpha t \Longrightarrow \theta s \sim_\alpha \theta t$.

Proposition 3.2.3 For all terms s and t, the following statements are equivalent:

1. $\lambda x.t \sim_\alpha \lambda x'.t'$
2. $\exists y \notin FV(\lambda x.t) \cup FV(\lambda x'.t')\colon\ [x{:=}y]t \sim_\alpha [x'{:=}y]t'$
3. $\forall y \notin FV(\lambda x.t) \cup FV(\lambda x'.t')\colon\ [x{:=}y]t \sim_\alpha [x'{:=}y]t'$

Proposition 3.2.4

1. $t \sim_\alpha S_0\epsilon t$
2. $S_0\epsilon(S_0\epsilon t) = S_0\epsilon t$
3. $s \sim_\alpha t \iff S_0\epsilon s = S_0\epsilon t$

The last statement of the proposition says that based on S_0 we can give a very straightforward definition of α-equivalence. A careful study of S_0 with complete proofs of its basic properties can be found in Allen Stoughton, Substitution Revisited, Theoretical Computer Science, 59:317-325, 1988.[1]

Exercise 3.2.5 (Free variables) Show that $t \sim_\alpha t'$ implies $FV(t) = FV(t')$. Hint: Use Proposition 3.1.2 and consider the identity substitution ϵ applied with S_0.

[1] http://people.cis.ksu.edu/~stough/research/subst.ps

3.3 Call-by-Value Reduction

In the following, we will give definitions for values and contextual equivalence that are different from the definitions we gave for PCF. In retrospect, the definitions for PCF should be changed so that they agree with the definitions for the λ-calculus.

We start with the definition of **values**:

$$v \in Val \subseteq Ter \ ::= \ \lambda x.t$$

This time we exclude variables. This has the advantage that values are closed under substitution, i.e, $t \in Val \Longrightarrow \theta t \in Val$. The **top level reduction relation $\to_0$** is defined by a single proper reduction rule:

$$(\lambda x.t)v \to_0 [x{:=}v]t$$

The **reduction relation**

$$t \to t' \ :\Longleftrightarrow \ \exists s, s', E\colon \ t = E[s] \ \wedge \ s \to_0 s' \ \wedge \ E[s'] = t'$$

is then obtained with the **evaluation contexts**

$$E \ ::= \ [] \ | \ Et \ | \ vE$$

Proposition 3.3.1

1. *Progress:* Every closed term that is not a procedure is reducible.
2. *Stability:* $t \to t' \Longrightarrow \theta t \to \theta t'$
3. *Compatibility:* $s \to s' \wedge s \sim_\alpha t \ \Longrightarrow \ \exists t'\colon \ t \to t' \wedge s' \sim_\alpha t'$

Consider the terms

$$\omega \ := \ \lambda x.xx$$
$$\Omega \ := \ \omega\omega$$

Obviously, $\Omega \to \Omega$. This means that Ω is a closed term whose reduction does not terminate.

The **evaluation relation** $\Downarrow$ is defined as follows:

$$t \Downarrow v \ :\Longleftrightarrow \ t \to \cdots \to v$$

We write $t\Downarrow$ and say that t **converges** if $t \Downarrow v$ for some value v. **Contextual equivalence** is defined as follows:

$$s \sim t \ :\Longleftrightarrow \ \forall C\colon \ C[s] \text{ and } C[t] \text{ closed} \ \Longrightarrow \ (C[s]\Downarrow \ \Longleftrightarrow \ C[s]\Downarrow)$$

The closedness conditions accounts for the fact that what really matters is the evaluation of closed terms. The closedness condition makes a difference in that it yields more equivalences. For instance, given the above definition of contextual equivalence, we have

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

We refer to this equivalence as **ν-equivalence**.

Exercise 3.3.2 Show that the above equivalence doesn't hold if contextual equivalence is defined without the closedness condition.

Exercise 3.3.3 Find s, t and θ such that $s \sim t$ and $\theta s \not\sim \theta t$.

Proposition 3.3.4 Contextual equicalence is a congruence relation with the following properties:

1. *Compatibility:* $\rightarrow \subseteq \sim$ and $\sim_\alpha \subseteq \sim$
2. *Stability:* $(\forall x \in FV(st)\colon\ \theta x \in Val) \ \wedge\ s \sim t \implies \theta s \sim \theta t$

Since every value of the λ-calculus is an abstraction, it is not surprising that we have the equivalence

$$x \sim \lambda y.xy \qquad \text{if } x \neq y$$

We refer to this equivalence as **η-equivalence**.

3.4 Church Arithmetic

Church discovered that the natural numbers and the intuitively computable functions on natural numbers can be represented in the λ-calculus. In fact, the λ-calculus was the first system used to *define* the set of computable functions. In 1936, Church's students Kleene and Turing published papers that showed that computability in the λ-calculus coincides with computability in their systems based on μ-recursion and Turing Machines.

The Booleans are represented as procedures that return one of two arguments:

$$\begin{aligned} \mathit{false} &:= \lambda xy.y \\ \mathit{true} &:= \lambda xy.x \\ \mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 &:= t_0(\lambda x.t_1)(\lambda x.t_2)(\lambda x.x) \qquad \text{where } x \notin FV(t_1t_2) \end{aligned}$$

Pairs are represented as procedures that return the first component if applied to **true** and the second component if applied to **false**:

$$\begin{aligned} (t_1, t_2) &:= (\lambda xyf.\, fxy)\, t_1\, t_2 \\ \mathit{fst}\, t &:= t\, \mathit{true} \\ \mathit{snd}\, t &:= t\, \mathit{false} \end{aligned}$$

For the representation of the natural numbers, we first define a notation for the **n-times application** of a term s to a term t:

$$\begin{aligned} s^0 t &:= t \\ s^n t &:= s^{n-1}(st) \quad \text{if } n > 0 \end{aligned}$$

For instance, $s^3 t$ is the term $s(s(st))$. Now a natural number n is represented as a procedure that applies a given procedure f n-times to a given x:

$$c_n := \lambda fx.\, f^n x$$

The procedures c_n are called **Church numerals**. Here are some procedures for natural numbers:

$$\begin{aligned} \mathit{succ} &:= \lambda nfx.\, nf(fx) \\ \mathit{iszero} &:= \lambda n.\, n(\lambda x.\, \mathit{false})\mathit{true} \\ \mathit{pred} &:= \lambda n.\, \mathit{fst}\,(n\,(\lambda p.(\mathit{snd}\,p,\ \mathit{succ}(\mathit{snd}\,p)))\,(c_0, c_0)) \\ \mathit{plus} &:= \lambda mnfx.\, mf(nfx) \\ \mathit{times} &:= \lambda mnf.\, m(nf) \\ \mathit{power} &:= \lambda mn.\, nm \end{aligned}$$

Due to the coding of natural numbers as procedures, we can obtain many operations for the natural numbers without recursion.

At this point we can ask whether we can formally specify what procedures like *succ*, *plus*, and *times* are supposed to do. In fact, we can specify these procedures with the equivalences

$$\begin{aligned} \mathit{succ}\, c_n &\sim c_{n+1} \\ \mathit{plus}\, c_m\, c_n &\sim c_{m+n} \\ \mathit{times}\, c_m\, c_n &\sim c_{m \cdot n} \end{aligned}$$

which are supposed to hold for all natural numbers m and n. That this is the case can be shown by induction on n.

3.5 Recursion Operator

The λ-calculus has procedures that act as recursion operators. To derive one of those, suppose we have a term D_f with one free variable f such that

$$D_f\,D_f \to f(\lambda x.D_f\,D_f\,x)$$

Then we can define a recursion operator as follows:

$$\mathit{fix} \;:=\; \lambda f.D_f\,D_f$$

It's not difficult to find a term D_f with the required properties:

$$D_f \;:=\; \lambda d.f(\lambda x.ddx)$$

The specification of a recursion operation is given by the the equivalence

$$\mathit{fix}\,f \;\sim\; f(\lambda x.(\mathit{fix}\,f)x)$$

3.6 Observing Church Arithmetic

Since the λ-calculus is Turing-complete, its contextual equivalence is not semi-decidable. In particular, given a number n and a term t, it is not semi-decidable whether $c_n \sim t$. Since Church Arithmetic works modulo contextual equivalence (e.g., $\mathit{succ}\,c_0 \sim c_1$ but $\mathit{succ}\,c_0 \Downarrow c_1$ does not hold), we cannot effectively observe whether a computation renders a certain number or not.

The problem can be solved by extending the λ-calculus with **output primitives**. We choose as output primitives observable natural numbers with a successor operation:

$$\begin{aligned}
&n \in \mathbb{N}\\
&t \in \mathit{Ter} \;::=\; \cdots \;\mid\; n \;\mid\; S\,t\\
&v \in \mathit{Val} \subseteq \mathit{Ter} \;::=\; \cdots \;\mid\; n\\
&E \;::=\; \cdots \;\mid\; S\,E
\end{aligned}$$

For the successor operation we need an additional proper reduction rule:

$$S\,n \to_0 n+1$$

We call a term **pure** if it is a term of the pure (i.e., unextended) calculus. Of course, every pure term is also a term of the extended calculus, but not vice versa. We use $\approx$ to denote the contextual equivalence of the extended calculus. The main difference between $\sim$ and $\approx$ is that with $\sim$ we can assume that everything is a procedure while with $\approx$ we cannot.

Exercise 3.6.1 Show $x \not\approx \lambda y.xy$.

Proposition 3.6.2 If s, t are pure terms, then $s \approx t \implies s \sim t$. The converse does not hold.

Proposition 3.6.3 In the extended calculus the following holds:

1. $\forall m, n \in \mathbb{N}\colon\ m \neq n \iff c_m \not\approx c_n$
2. $t \approx c_n \implies t(\lambda x.\mathrm{S}x)0 \Downarrow n$

Statement (2) of the proposition tells us how we can observe whether a computation renders a certain number. The various procedures defined in the previous section are all correct with respect to $\approx$. For instance, we have the following equivalences:

$$\begin{aligned} \mathit{succ}\, c_n &\approx c_{n+1} \\ \mathit{plus}\, c_m\, c_n &\approx c_{m+n} \\ \mathit{times}\, c_m\, c_n &\approx c_{m\cdot n} \end{aligned}$$

3.7 Higher-Order Abstract Syntax

Higher-order abstract syntax (HOAS) is an implementation technique that represents abstractions $\lambda x.t$ as procedures $\mathit{ter} \to \mathit{ter}$, where the procedure for an abstraction $\lambda x.t$ returns for a term s the term $[x{:=}s]t$. The closed terms of our extended λ-calculus can be implemented as follows:

```
datatype ter =  A of ter * ter | L of ter -> ter | I of int | S of ter
```

The beauty of HOAS is the fact that we can write an eval procedure without implementing substitution. This is the case since every abstraction has built-in the substitution needed for its application.

HOAS is more than just an implementation technique. Its one of the key features of the logic programming language Twelf[2] developed by Frank Pfenning and his students at CMU.

Exercise 3.7.1

a) Write declarations for the terms *succ*, *plus*, and *times*.

b) Write a procedure $\mathit{eval} : \mathit{ter} \to \mathit{ter}$ that evaluates terms.

c) Write procedures $\mathit{church} : \mathit{int} \to \mathit{ter}$ and $\mathit{dechurch} : \mathit{ter} \to \mathit{int}$ such that $\mathit{church}\ n$ yields c_n and $\mathit{dechurch}\ t$ yields n if $t \sim c_n$.

[2] http://twelf.plparty.org

Exercise 3.7.2

a) Write a procedure $size : ter \to int$ that yields the size of a term.

b) Write a procedure $occurs : int \to ter \to bool$ that tests whether a number occurs in a term.

c) Write a procedure $new : ter \to int$ that yields a number that doesn't occur in a term.

d) Write a procedure $equal : ter \to ter \to bool$ that tests whether two terms are equal (i.e., α-equivalent).

3.8 De Bruijn Terms

De Bruijn devised a representation for terms that represents abstractions without argument variables. Instead, an argument reference is done through a natural number n that says how many λ's on the path to the root must be skipped until one arrives at the λ that introduces the argument. Here are examples:

$$\begin{aligned} \lambda x.x &\rightsquigarrow \lambda 0 \\ \lambda xy.x &\rightsquigarrow \lambda(\lambda 1) \\ \lambda fxy.fyx &\rightsquigarrow \lambda(\lambda(\lambda(2(0)(1)))) \end{aligned}$$

The idea extends to open terms:

$$\begin{aligned} fx &\rightsquigarrow fx \\ \lambda x.y &\rightsquigarrow \lambda(y+1) \\ \lambda xy.fyx &\rightsquigarrow \lambda(\lambda((f+2)(0)(1))) \end{aligned}$$

Formally, we define de Bruijn terms as follows:

$$\begin{aligned} n \in Var &:= \mathbb{N} \\ \tau \in DB &::= n \mid \tau\,\tau \mid \lambda\tau \end{aligned}$$

Note that de Bruijn terms are simpler than ordinary terms since de Bruijn abstractions have only one constituent. The translation of terms to de Bruijn terms is defined as follows:

$$\begin{aligned} &db \in Ter \to DB \\ &db\,x = x \\ &db(s\,t) = (db\,s)(db\,t) \\ &db(\lambda x.t) = \lambda(db(S(\lambda y{\in}Var.\ if\ y = x\ then\ 0\ else\ y+1)\,t)) \end{aligned}$$

Proposition 3.8.1 $s \sim_\alpha t \iff db\,s = db\,t$

We can map substitutions to de Bruin substitutions:

$$db\,\theta \;:=\; \lambda x{\in}Var.\,db(\theta x)$$

Moreover, we can define a de Bruin substitution operator $\hat{S}$ such that the following proposition holds.

Proposition 3.8.2 $db(S\theta t) \;=\; \hat{S}(db\,\theta)(db\,t)$

Here is the definition of $\hat{S}$:

$$\begin{aligned}
\hat{S}\psi\tau &= \hat{S}'0\psi\tau \\
\hat{S}'d\psi n &= \mathit{if}\ n < d\ \mathit{then}\ n\ \mathit{else}\ \mathit{up}\,d\,(\psi(n-d)) \\
\hat{S}'d\psi(\tau_1\tau_2) &= (\hat{S}'d\psi\tau_1)(\hat{S}'d\psi\tau_2) \\
\hat{S}'d\psi(\lambda\tau) &= \lambda(\hat{S}'(d+1)\psi\tau) \\
\mathit{up}\,d\,n &= n+d \\
\mathit{up}\,d\,\tau &= \hat{S}(\lambda n{\in}Var.\,n+d)\tau \qquad \text{if } \tau \notin Var
\end{aligned}$$

The auxiliary function $\hat{S}'$ takes an additional argument d (the λ-depth) that says how many λ's are on the path to the root. The auxiliary function *up* raises all free variable occurrences of a term by a given number.

Exercise 3.8.3 (De Bruijn Terms) We implement ordinary and de Bruijn terms as follows:

```
type var = int
datatype ter = V of var | A of ter * ter | L of var * ter
datatype dbt = DV of var | DA of dbt * dbt | DL of dbt
```

a) Write a procedure $\mathit{free} : \mathit{var} \to \mathit{dbt} \to \mathit{bool}$ that test whether a variable occurs free in a de Bruijn term.

b) Write a procedure $\mathit{subst} : (\mathit{var} \to \mathit{dbt}) \to \mathit{dbt} \to \mathit{dbt}$ that applies a de Bruijn substitution to a de Bruijn term.

c) Write a procedure $\mathit{db} : \mathit{ter} \to \mathit{dbt}$ that translates a term into a de Bruijn term.

4 Simply Typed Lambda Calculus

The simply typed λ-calculus is obtained from the untyped λ-calculus by imposing a straightforward type discipline. Computationally, the resulting system is rather weak. However, full computational power can be regained by adding a recursion operator and operations on base types. One such extended system is PCF. Alternatively, full computational power can be regained by extending the type system with unit, product types, sum types and recursive types.

The simply typed lambda calculus will also serve us as the basis for the study subtyping and type reconstruction.

Originally, the simply typed λ-calculus was invented as a logical sytem (by Church, first publication in 1940). For logical applications, the simply typed λ-calculus is equipped with a model-theoretic semantics and the resulting system is referred to as *simple type theory*. From the perspective of logic, simple type theory is a very expressive since it subsumes propositional logic, predicate logic as well as modal and temporal logics. Moreover, simple type theory is the canonical base for higher-order logic.

There is another important use of the simply typed λ-calculus in logic which runs under the name Curry-Howard correspondence. Here types serve as formulas and terms serve as proofs. This use of the simply typed λ-calculus was not anticipated by Church. It was first made explicit in a paper by Howard that appeared in 1980.

4.1 Basic Definitions

The terms of the simply typed λ-calculus are obtained from the terms of the untyped λ-calculus by equipping abstractions with types for their arguments. Types are either **basic** or **functional**. No particular assumptions are made about the base types, which are provided through a nonempty set. Here are the basic definitions:

$X, Y \in BT$	**base type**
$T, S \in Ty ::= X \mid T \to T$	**type**
$x, y, z \in Var := \mathbb{N}$	**variable**
$t, s \in Ter ::= x \mid t\,t \mid \lambda x{:}T.t$	**term**

$C ::= [] \mid Ct \mid tC \mid \lambda x{:}T.C$	**context**
$\theta \in Var \to Ter$	**substitution**
$\Gamma \in Var \rightharpoonup Ty$	**type environment**

We assume that there is at leat one base type. **Subterms**, **size of terms**, **free variables** and the **application of substitutions** are defined as in the untyped case. The **α-equivalence** relation is defined as one would expect from the untyped case, where it is important that α-equivalent abstractions are required to have the same argument type:

$$\begin{aligned} x \sim_\alpha x' &\iff x = x' \\ st \sim_\alpha s't' &\iff s \sim_\alpha s' \ \wedge\ t \sim_\alpha t' \\ \lambda x{:}T.t \sim_\alpha \lambda x'{:}T.t' &\iff x' \notin FV(\lambda x{:}T.t)\ \wedge\ t' \sim_\alpha [x{:=}x']t \end{aligned}$$

The typing relation $\Gamma \vdash t : T$ is defined recursively by the following inference rules, which we have seen before for PCF.

$$\frac{}{\Gamma \vdash x : T}\ (x,T) \in \Gamma$$

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

The following proposition provides a more explicit characterisation of the typing relation. It could be used as an alternative definition of the typing relation.

Proposition 4.1.1 (Inversion)

1. $\Gamma \vdash x : T \iff (x,T) \in \Gamma$
2. $\Gamma \vdash t_1 t_2 : T \iff \exists T_2{:}\ \Gamma \vdash t_1 : T_2 \to T \ \wedge\ \Gamma \vdash t_2 : T_2$
3. $\Gamma \vdash \lambda x{:}T_1.t : T \iff \exists T_2{:}\ T = T_1 \to T_2 \ \wedge\ \Gamma[x{:=}T_1] \vdash t : T_2$

Proof Follows from the fact that $\Gamma \vdash t : T$ is always established by one of the three inference rules defining the typing relation. ■

4.2 Basic Properties

We now state the basic properties of the typing relations. Typically, they can be proven by induction on term size. Make sure that you can do the proofs.

Proposition 4.2.1 (Unique Type)
$\Gamma \vdash t : T \ \wedge\ \Gamma \vdash t : T' \implies T = T'$

Proof By induction on the size of t. ■

Proposition 4.2.2 (Coincidence)
$(\forall x \in FV\,t\colon\ \Gamma x = \Gamma' x) \ \wedge\ \Gamma \vdash t : T \ \Longrightarrow\ \Gamma' \vdash t : T$

Proof By induction on the size of t. ■

Proposition 4.2.3 (Relevance) $\Gamma \vdash t : T \Longrightarrow FV\,t \subseteq Dom\,\Gamma$

Proof By induction on the size of t. ■

Proposition 4.2.4 (Replacement)
$(\forall \Gamma, T\colon\ \Gamma \vdash t : T \Longrightarrow \Gamma \vdash t' : T) \ \wedge\ \Gamma \vdash C[t] : T \ \Longrightarrow\ \Gamma \vdash C[t'] : T$

Proof By induction on the size of C. ■

Proposition 4.2.5 (Substitution)
$(\forall x \in FV\,t\colon\ \Gamma' \vdash \theta x : \Gamma x) \ \wedge\ \Gamma \vdash t : T \ \Longrightarrow\ \Gamma' \vdash \theta t : T$

Proof By induction on the size of t. The proof holds for every substitution operator. Let

(1) $\forall x \in FV\,t\colon\ \Gamma' \vdash \theta x : \Gamma x$

(2) $\Gamma \vdash t : T$

We show $\Gamma' \vdash \theta t : T$ by case analysis.

Case $t = x$. Then $x \in FV\,t$ and $\Gamma x = T$ by (2). Hence $\Gamma' \vdash \theta t : T$ by (1).

Case $t = t_1 t_2$. Then $\Gamma \vdash t_1 : T_2 \to T$ and $\Gamma \vdash t_2 : T_2$ by (2) and Inversion. Hence $\Gamma' \vdash \theta t_1 : T_2 \to T$ and $\Gamma' \vdash \theta t_2 : T_2$ by (1) and induction. Hence $\Gamma' \vdash \theta t : T$.

Case $t = \lambda x{:}T_1.\,t_2$. Then by (2) and Inversion:

(3) $T = T_1 \to T_2$ and $\Gamma[x{:=}T_1] \vdash t_2 : T_2$

(4) $\theta' := \theta[x{:=}y]$ and $\theta t = \lambda y{:}T_1.\,\theta' t_2$

(5) $y \notin \cup\{\, FV(\theta z) \mid z \in FV\,t_2 - \{x\}\,\}$

First we show

(6) $\forall z \in FV\,t_2\colon\ \Gamma'[y{:=}T_1] \vdash \theta' z : (\Gamma[x{:=}T_1])z$

Let $z \in FV\,t_2$. If $z = x$, then $\theta' z = y$ by (4) and $(\Gamma[x{:=}T_1])z = T_1$. Hence $\Gamma'[y{:=}T_1] \vdash \theta' z : (\Gamma[x{:=}T_1])z$. Now let $z \neq x$. By (1) we have $\Gamma' \vdash \theta z : \Gamma z$. By (5) and Coincidence we have $\Gamma'[y{:=}T_1] \vdash \theta z : \Gamma z$. Now the claim follows by $\theta' z = \theta z$ (using (4)) and $(\Gamma[x{:=}T_1])z = \Gamma z$.

By (6) and (3) and induction we now have $\Gamma'[y{:=}T_1] \vdash \theta' t_2 : T_2$. Hence $\Gamma' \vdash \theta t : T$ by (4) and (3). ■

Proposition 4.2.6 (Compatibility with β-Reduction)
$\Gamma \vdash C[(\lambda x{:}T_1.t_1)t_2] : T \implies \Gamma \vdash C[[x{:=}t_2]t_1] : T$

Proof Because of Replacement it suffices to show the claim for $C = [\,]$. For $C = [\,]$ the claim follows from Substitution. ∎

Proposition 4.2.7 (Compatibility with α-Equivalence)
$t \sim_\alpha t' \wedge \Gamma \vdash t : T \implies \Gamma \vdash t' : T$

In other words, all α-equivalent terms have the same behaviour with respect to typing.

Proof By induction on the size of t'. The cases where $t \in \mathit{Var}$ or $t = t_1 t_2$ are easy. If $t = \lambda x{:}T_1.s$, then by definition of α-equivalence, $t' = \lambda y{:}T_1.s'$ for some $y \notin \mathit{FV}(t)$ such that $s' \sim_\alpha [x := y]s$. By inversion, $\Gamma[x := T_1] \vdash s : T_2$ for some T_2 such that $T = T_1 \to T_2$.

Let $\theta = [x := y]$, then clearly $\Gamma[y := T_1] \vdash \theta x : T_1$. Moreover, if $z \in \mathit{FV}(t)$ then $z \neq x$, $z \neq y$ and $\theta z = z$. By the Relevance Lemma, $z \in \mathit{Dom}\,\Gamma$. Hence, $(z, \Gamma z) \in \Gamma$ implies $\Gamma[y := T_1] \vdash \theta z : (\Gamma[x := T_1])z$. Combining these cases, and using $\mathit{FV}(s) \subseteq \mathit{FV}(t) \cup \{x\}$, we therefore have

$$\forall z \in \mathit{FV}(s) : \ \Gamma[y := T_1] \vdash \theta z : (\Gamma[x := T_1])z$$

Since $\Gamma[x := T_1] \vdash s : T_2$, an application of the Substitution Lemma yields

$$\Gamma[y := T_1] \vdash \theta s : T_2$$

which by definition of θ is just $\Gamma[y := T_1] \vdash [x := y]s : T_2$. Because $[x := y]s \sim_\alpha s'$ and s' is strictly smaller than t', the induction hypothesis gives

$$\Gamma[y := T_1] \vdash s' : T_2$$

from which the required $\Gamma \vdash t' : T_2$ follows by T-Abs. ∎

Exercise 4.2.8 (Compatibility with α-Equivalence) Complete the inductive proof of Lemma 4.2.7 by carefully spelling out the missing cases $t \in \mathit{Var}$ and $t = t_1 t_2$.

4.3 Reduction

We define two reduction relations. **Call-by-value Reduction** $\overset{v}{\to}$ is what we know from PCF and the untyped λ-calculus. It's defined as follows:

$v \in \mathit{Val} \subseteq \mathit{Ter} \ ::= \ \lambda x{:}T.t$ **value**

$$(\lambda x{:}T.t)v \overset{\text{v}}{\to}_0 [x{:=}v]t$$
$$E \ ::= \ [\,] \ \mid \ Et \ \mid \ vE$$
$$t \overset{\text{v}}{\to} t' \ :\Longleftrightarrow \ \exists E, s, s'\colon \ t = E[s] \ \wedge \ s \overset{\text{v}}{\to}_0 s' \ \wedge \ E[s'] = t'$$

Full reduction $\to$ generalizes call-by value reduction in that every β-redex can be reduced:

$$(\lambda x{:}T.t)s \to_0 [x{:=}s]t$$
$$t \to t' \ :\Longleftrightarrow \ \exists C, s, s'\colon \ t = C[s] \ \wedge \ s \to_0 s' \ \wedge \ C[s'] = t'$$

Proposition 4.3.1 $\overset{\text{v}}{\to} \subseteq \to$

Proposition 4.3.2 (Preservation) $t \to t' \ \wedge \ \Gamma \vdash t : T \ \Longrightarrow \ \Gamma \vdash t' : T$

Proof Follows from Compatibility with β-Reduction. ■

Preservation says that types are invariants for the terms of a reduction chain $t_0 \to t_1 \to t_2 \to \cdots$.

Proposition 4.3.3 (Progress)
$\emptyset \vdash t : T \ \wedge \ t$ not a value $\Longrightarrow \ \exists t'\colon \ t \overset{\text{v}}{\to} t'$

Proof By induction on the size of t. ■

Progress says that every closed and well-typed term that is not a value is call-by-value reducible. Together, Preservation and Progress are referred to as **type soundness**. Type soundness means that the reduction of a well-typed and closed term either does not terminate or terminates with a closed value that has the type of the initial term. Type soundness is a key requirement for typed programming languages.

Exercise 4.3.4 Make sure that you understand the definitions and the basic properties of the simply typed λ-calculus. In particular, you should be able to define or state the following:

a) Types and terms.
b) The typing relation.
c) Inversion.
d) Unique Types, Coincidence, and Relevance.
e) Replacement, Substitution, and Compatibility with β and α.
f) Call-by-value and full reduction.
g) Preservation and Progress.

$$\begin{aligned} T &::= \;\cdots \mid T \times T \\ i &\in \{1,2\} \\ t &::= \;\cdots \mid (t,t) \mid t.i \end{aligned}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1,t_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.i : T_i}$$

$$(t_1,t_2).i \to_0 t_i$$

$$\begin{aligned} v &::= \;\cdots \mid (v,v) \\ E &::= \;\cdots \mid (E,t) \mid (v,E) \end{aligned}$$

$$(v_1,v_2).i \overset{v}{\to}_0 v_i$$

Figure 4.1: Extension of the simply typed λ-calculus with product types

4.4 Binary Products and Sums

Consider the following type declaration in Standard ML:

```
datatype ter = A of ter * ter | L of ter -> ter
```

It involves several primitives:

- A product type (i.e., $ter * ter$).
- A function type (i.e., $ter \to ter$).
- A sum type (i.e, $(ter * ter) + (ter \to ter)$).
- Recursion (i.e., type ter is defined recursively).
- Naming at term level (A and L) and at type level (ter).

In this section we will explain product and sum types by adding them to the simply typed λ-calculus.

4.4.1 Product Types

The values of a **product type** $T_1 \times T_2$ are pairs (v_1, v_2) where v_1 has type T_1 and v_2 has type T_2. There are operations $t.1$ and $t.2$ called **selections** that yield the first and second component of the pair a term t evaluates to. This motivates the definitions shown in Figure 4.1. The extensions are such that all properties stated in § 4.2 and § 4.3 remain valid.

$$T \ ::= \ \cdots \ | \ T + T$$
$$i \ \in \{1, 2\}$$
$$t \ ::= \ \cdots \ | \ (i, t, T) \ | \ \text{case}\, t\, t\, t$$

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash (i, t, T_1 + T_2) : T_1 + T_2}$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma \vdash t_1 : T_1 \to T \qquad \Gamma \vdash t_2 : T_2 \to T}{\Gamma \vdash \text{case } t_0\ t_1\ t_2 : T}$$

$$\text{case } (i, t, T)\ t_1\ t_2 \ \to_0 \ t_i\ t$$

$$v \ ::= \ \cdots \ | \ (i, v, T)$$
$$E \ ::= \ \cdots \ | \ (i, E, T)$$

$$\text{case } (i, v, T)\ t_1\ t_2 \ \overset{\text{v}}{\to}_0 \ t_i\ v$$

Figure 4.2: Extension of the simply typed λ-calculus with sum types

Note that type soundness now really is a non-trivial property. It makes sure that the reduction of well-typed terms will not lead to terms that select a component of a procedure or apply a pair to a procedure.

4.4.2 Sum Types

Sum types provide for the disjoint union of types. The values of a **sum type** $T_1 + T_2$ are triples $(i, v, T_1 + T_2)$ called **variants**, where the **value** v of the variant has the type T_i determined by the **tag** $i \in \{1, 2\}$ and the **type** $T_1 + T_2$ of the variant. There is a **case construct** case $t_0\, t_1\, t_2$ that applies t_i to the value v of the variant (i, v, T) the term t_0 evaluates to. Variants carry their type so that the unique type property of the simply typed λ-calculus is preserved. This motivates the definitions shown in Figure 4.2. The extensions are such that all properties stated in § 4.2 and § 4.3 remain valid. If one is willing to give up the unique type propertx, the type components of variants can be omitted.

The notational inconvenience of variants carrying types can be avoided by using **constructors** as in Standard ML. For instance, for a sum type $T_1 + T_2$ we may introduce the constructors

$$\begin{aligned} &\text{let } && L = \lambda x{:}T_1.\ (1, x, T_1 + T_2) \\ & && R = \lambda x{:}T_2.\ (2, x, T_1 + T_2) \\ &\text{in } && t \end{aligned}$$

In desugared form, this looks as follows:

$$\begin{aligned} &(\lambda L{:}T_1 \to T_1 + T_2.\ \lambda R{:}T_2 \to T_1 + T_2.\ t) \\ &(\lambda x{:}T_1.\ (1, x, T_1 + T_2)) \\ &(\lambda x{:}T_2.\ (2, x, T_1 + T_2)) \end{aligned}$$

4.4.3 Unit and Bool

The potential of sum types becomes apparent once we extend the λ-calculus with an additional type 1 with exactly one value (). Both the type 1 and the value () are pronounced **unit**. Here are the definitions:

$$\begin{aligned} T &::= \cdots \mid 1 \\ t &::= \cdots \mid () \end{aligned}$$

$$\frac{}{\Gamma \vdash () : 1}$$

$$v ::= \cdots \mid ()$$

The type bool can now be expressed as follows:

$$\begin{aligned} \text{bool} &:= 1 + 1 \\ \text{false} &:= (1, (), \text{bool}) \\ \text{true} &:= (2, (), \text{bool}) \\ \text{if } t_0 \text{ then } t_1 \text{ else } t_2 &:= \text{case } t_0\ (\lambda x{:}1.\ t_1)\ (\lambda x{:}1.\ t_2) \qquad \text{where } x \notin FV(t_1 t_2) \end{aligned}$$

4.5 Recursive Types and FPC

The simply typed λ-calculus gains full computational power once it is equipped with recursive types. The so extended calculus can express the natural numbers and procedural recursion operators. With recursive types, the simply typed λ-calculus can also embed the untyped λ-calculus.

Let’s start with some familiar recursive types expressed in Standard ML:

```
datatype nat = Z | S of nat
datatype 'a list = N | C of 'a * 'a list
datatype 'a stream = S of 'a * (unit -> 'a stream)
```

In the λ-calculus we express these types with a type variable (i.e., a base type) and the μ-operator:

$$\begin{aligned} \mathit{nat} &:= \mu X.\, 1 + X \\ \mathit{list}_T &:= \mu X.\, 1 + (T \times X) \\ \mathit{stream}_T &:= \mu X.\, T \times (1 \to X) \end{aligned}$$

Similar to the λ-operator, μ-operator binds the type variable following it. Thus $\mu X.\ 1 + X$ is a closed type and $\mu Y.\ X \times (1 \to Y)$ is a type with one free type variable X.

The values of a recursive type $T = \mu X.S$ are obtained with a constructor $R_T : [X{:=}T]S \to T$. For instance, the first three values of *nat* are obtained as follows:

$$\begin{aligned} \mathit{zero} &:= \mathrm{R}_{\mathit{nat}}\,(1,\ (),\ 1 + \mathit{nat}) \\ \mathit{one} &:= \mathrm{R}_{\mathit{nat}}\,(2,\ \mathit{zero},\ 1 + \mathit{nat}) \\ \mathit{two} &:= \mathrm{R}_{\mathit{nat}}\,(2,\ \mathit{one},\ 1 + \mathit{nat}) \end{aligned}$$

From Standard ML we know that this awkward notation can be hidden by employing constructors:

$$\begin{aligned} \mathit{zero} &:= \mathrm{R}_{\mathit{nat}}\,(1,\ (),\ 1 + \mathit{nat}) \\ \mathit{succ} &:= \lambda n{:}\mathit{nat}.\ \mathrm{R}_{\mathit{nat}}\,(2,\ n,\ 1 + \mathit{nat}) \\ \mathit{one} &:= \mathit{succ}\ \mathit{zero} \\ \mathit{two} &:= \mathit{succ}(\mathit{succ}\ \mathit{zero}) \end{aligned}$$

Now let's see how we can express the predecessor function. In Standard ML we use pattern matching to do case analysis and get rid of constructors:

```
fun pred Z = Z
  | pred (S n) = n
```

In the λ-calculus we use the U-operator to get rid of the R-constructor and do the case analysis with the case primitive:

$$\mathit{pred} := \lambda n{:}\mathit{nat}.\ \mathrm{case}\ (\mathrm{U}\, n)\ (\lambda x{:}1.\ \mathit{zero})\ (\lambda m{:}\mathit{nat}.\ m)$$

Given a recursive type $T = \mu X.S$, the operators R_T and U realize a bijection between the recursive type and its **unfolding** $[X{:=}T]S$:

$$T \ ::= \ \cdots \ \mid \ \mu X.T$$
$$t \ ::= \ \cdots \ \mid \ \mathrm{R}_T\, t \ \mid \ \mathrm{U}\, t$$

$$\frac{\Gamma \vdash t : [X{:=}T]S}{\Gamma \vdash \mathrm{R}_T\, t : T} \ T = \mu X.S \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathrm{U}\, t : [X{:=}T]S} \ T = \mu X.S$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : T'} \ T \sim_\alpha T'$$

$$v \ ::= \ \cdots \ \mid \ \mathrm{R}_T\, v$$
$$E \ ::= \ \cdots \ \mid \ \mathrm{R}_T\, E \ \mid \ \mathrm{U}\, E$$
$$\mathrm{U}(\mathrm{R}_T\, v) \ \overset{v}{\to}_0 \ v$$

Figure 4.3: Extension of the simply typed λ-calculus with recursive types

$$\begin{tikzcd} T = \mu X.S \arrow[r, shift right, "\mathrm{U}"'] & {[X{:=}T]S} \arrow[l, shift right, "\mathrm{R}_T"'] \end{tikzcd}$$

For $nat = \mu X.\ 1 + X$ we obtain the following situation:

$$\begin{tikzcd} nat \arrow[r, shift right, "\mathrm{U}"'] & 1 + nat \arrow[l, shift right, "\mathrm{R}_{nat}"'] \end{tikzcd}$$

Figure 4.3 summarizes the extension of the simply typed λ-calculus with recursive types. The third typing rule ensures that the typing relation is invariant with respect to α-equivalence of types. This rule is needed so that the typing $[x := \mu X.1 + X] \vdash (\lambda y : \mu Y.1 + Y.\ 1)\, x : 1$ holds.

The simply typed λ-calculus extended with unit, products, sums, and recursive types is known as **FPC** (fixed point calculus). It seems to have originated with Plotkin's lecture notes (1985). FPC preserves the basic properties listed in § 4.2 and § 4.3 for the pure simply typed λ-calculus: Unique Type, Coincidence, Relevance, Replacement, Substitution, Compatibility with α-equivalence, Preservation, and Progress.

Exercise 4.5.1 Given a type in FPC whose values represent binary trees whose nodes are marked with natural numbers.

Exercise 4.5.2 Give a type in FPC that corresponds to the following type of Standard ML: *datatype tree = T of tree list*.

4.5.1 Procedural Recursion Operators

To see how recursive types can express procedural recursion, let's start with the definition of the recursion operator in the untyped λ-calculus (we use a variant of the operator in § 3.5):

$$\begin{aligned} D_f &:= \lambda dx.\, f(dd)x \\ \mathit{fix} &:= \lambda f.\, D_f\, D_f \end{aligned}$$

Now let's look at the types. The goal is that *fix* provides for the recursive definition of procedures $S \to T$. We know that *fix* is applied to the scheme for the recursive procedure (see § 1.5). Hence we know the types of the variables f and x:

$$\begin{aligned} &f : (S \to T) \to S \to T \\ &x : S \end{aligned}$$

Now we come to the type for d. Because of the self application a single type for d is not good enough. Everything would be fine if we could give d two types, namely *Fix* and $\mathit{Fix} \to S \to T$. This is not really possible, but let's look at the recursive type

$$\mathit{Fix} := \mu X.\, X \to S \to T \qquad \text{where } X \text{ not free in } S \text{ or } T$$

With U and R we can go back and forth from *Fix* to $\mathit{Fix} \to S \to T$. This gives us all we need. We arrange

$$d : \mathit{Fix}$$

and augment the definitions of D_f and *fix* with the conversions $\mathrm{R}_{\mathit{Fix}}$ and U:

$$\begin{aligned} D_f &:= \lambda dx.\, f(\mathrm{U}dd)x \\ \mathit{fix} &:= \lambda f.\, D_f\, (\mathrm{R}_{\mathit{Fix}}\, D_f) \end{aligned}$$

We have now arrived at a well-typed definition of a recursion operator *fix*. Based on this definition, it is now straightforward to declare a polymorphic recursion operation in Standard ML that does not use procedural recursion:

```
datatype ('a,'b) Fix = R of ('a,'b) Fix -> 'a -> 'b
fun U (R p) = p
fun D f d x = f (U d d) x
fun fix f = D f (R (D f))
```
val fix : $((\alpha \to \beta) \to \alpha \to \beta) \to \alpha \to \beta$

Exercise 4.5.3 Here is the untyped recursion operator from § 3.5:
$\lambda f.\,(\lambda d.\,f(\lambda x.\,ddx))\,(\lambda d.\,f(\lambda x.\,ddx))$.

a) Translate this term into FPC such that it takes for given types S, T the type $((S \to T) \to S \to T) \to S \to T$.

b) Translate this term into a polymorphic procedure in Standard ML that doesn't employ procedural recursion.

Exercise 4.5.4 Show that for every type T of FPC there is a closed term t such that $\emptyset \vdash t : T$.

4.5.2 Embedding the Untyped λ-Calculus

Consider the recursive type

$$\Lambda \;:=\; \mu X.\, X \to X$$

The values of this type are procedures that take procedures as arguments and return procedures as result (up to conversion between Λ and its unfolding $\Lambda \to \Lambda$). The terms of the pure untyped λ-calculus translate into well-typed terms of type Λ as follows:

$$\begin{aligned} \tau x &= x \\ \tau(st) &= \mathrm{U}\,(\tau s)\,(\tau t) \\ \tau(\lambda x.t) &= \mathrm{R}_\Lambda(\lambda x : \Lambda.\, \tau t) \end{aligned}$$

The translation is such that untyped values are translated into typed values, and that untyped reductions translate into typed reductions. More specifically, for an untyped reduction $s \to t$ there always exists a typed term s' such that $\tau s \to s' \to \tau t$. For instance, $(\lambda x.x)(\lambda x.x) \to \lambda x.x$ on the untyped side translates into

$$\begin{aligned} \mathrm{U}\,(\mathrm{R}_\Lambda(\lambda x : \Lambda.\, x))\,(\mathrm{R}_\Lambda(\lambda x : \Lambda.\, x)) \;&\to\; (\lambda x : \Lambda.\, x)\,(\mathrm{R}_\Lambda(\lambda x : \Lambda.\, x)) \\ &\to\; (\mathrm{R}_\Lambda(\lambda x : \Lambda.\, x)) \end{aligned}$$

So τ gives us a faithful translation of the pure untyped λ-calculus into FPC.

Exercise 4.5.5 Give a closed and well-typed term in FPC whose reduction doesn't terminate.

Exercise 4.5.6 Give a translation of the untyped λ-calculus with observable natural numbers into FPC. Do this by extending the definitions of Λ and τ. Explain what happens to the stuck untyped terms on the typed side.

4.6 Curry-Howard Correspondence

One can interpret the non-recursive types of the simply typed λ-calculus as propositional formulas:

- A base type X is a propositional variable that can take the values 0 or 1.
- A functional type $T_1 \to T_2$ is an implication.
- A product type $T_1 * T_2$ is a conjunction.
- A sum type $T_1 + T_2$ is a disjunction.
- The type 1 is the formula 1.

This interpretation is of interest since one can show that $\emptyset \vdash t : T$ implies that T is a valid formula (i.e., as a formula, T yields 1 no matter how we choose the values of its variables). This means that a term t such that $\emptyset \vdash t : T$ is a proof of the validity of the formula T. As it happens, the proof system given by the typing rules corresponds to a prominent propositional proof system known as **natural deduction** (ND) that has been proposed by Gerhard Gentzen in 1935. The correspondence between simply-typed lambda calculus and natural deduction is known as **Curry-Howard correspondence** and was first made explicit in a paper by Howard that appeared in 1980.

As is, the correspondence is not perfect since the typing rules cannot prove all valid formulas, and types cannot express the formula 0 and negated formulas. Both problems can be solved by the following, logically motivated extension:

$$\begin{aligned} T &::= \;\cdots \mid 0 \\ t &::= \;\cdots \mid \delta t \end{aligned}$$

$$\frac{\Gamma \vdash t : (T \to 0) \to 0}{\Gamma \vdash \delta t : T}$$

Negations $\neg T$ can now be expressed as implications $T \to 0$, and one can show that T is valid as a formula if and only if there is term t such that $\emptyset \vdash t : T$. In fact, a more general correspondence holds: A formula $T_1 \to \cdots \to T_n \to T$ is valid if and only if there exist Γ and t such that $\Gamma \vdash t : T$ and $\mathit{Ran}\,\Gamma = \{T_1, \ldots, T_n\}$. Note that the typing rule for δ expresses the logical rule $\dfrac{\neg(\neg T)}{T}$.

Figure 4.4 shows the syntax, the typing rules, and the proper reduction rules for the simply typed λ-calculus ND that corresponds to natural deduction.

Here are examples for straightforward proofs in ND:

1. $\lambda x{:}X.\, x$ proves $X \to X$
2. $\lambda p{:}X * Y.\ (p.2)$ proves $X * Y \to Y$
3. $\lambda n{:}0.\ \delta(\lambda g{:}X{\to}0.\ n)$ proves $0 \to X$

$$X \in \mathit{TV} := \mathbb{N}$$

$$T \in \mathit{Ty} \ ::= \ X \ | \ T \to T \ | \ T \times T \ | \ T + T \ | \ 1 \ | \ 0$$

$$x \in \mathit{Var} := \mathbb{N}$$

$$i \in \{1, 2\}$$

$$t \in \mathit{Ter} \ ::= \ x \ | \ \lambda x{:}T.t \ | \ t\,t \ | \ (t, t) \ | \ t.i \ | \ (i, t) \text{ as } T \ | \ \text{case } t\ t\ t \ | \ () \ | \ \delta\, t$$

$$\Gamma \in \mathit{Var} \rightharpoonup \mathit{Ty}$$

$$\frac{\Gamma x = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma[x := T] \vdash t : T'}{\Gamma \vdash \lambda x{:}T.t : T \to T'} \qquad \frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\, t_2 : T'}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.i : T_i}$$

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash (i, t, T_1 + T_2) : T_1 + T_2}$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma \vdash t_1 : T_1 \to T \qquad \Gamma \vdash t_2 : T_2 \to T}{\Gamma \vdash \text{case } t_0\ t_1\ t_2 : T}$$

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash t : (T \to 0) \to 0}{\delta\ t : T}$$

$$(\lambda x{:}T.t)t' \ \to_0 \ t[x := t']$$

$$(t_1, t_2).i \ \to_0 \ t_i$$

$$\text{case } (i, t, T)\ t_1\ t_2 \ \to_0 \ t_i\ t$$

Figure 4.4: The simply-typed λ-calculus ND

We use the abbreviation $\overline{X} \;:=\; X \to 0$ for negation. Here is a proof for $X + \overline{X}$:

$$\delta(\lambda f.\ f(2,\ \lambda x.f(1,x)))$$

$$\text{where}\ \ f : \overline{X + \overline{X}}\ \ \text{and}\ \ x : X$$

To keep the proof readable, we state the types of the argument variables separately and omit the types of the variants. The types of the variants can be derived from the type of f. Both variants have the type $X + \overline{X}$.

For ND we employ full reduction as defined by the proper reduction rules in Figure 4.4. Since Preservation holds for ND, we can use reduction to simplify proof terms. In fact, Gentzen's so-called cut elimination is reduction in disguised form.

We summarize the Curry-Howard correspondence as follows:

- Types are propositional formulas.
- Closed well-typed terms are proofs for the validity of formulas.
- The typing rules are proof verification rules.
- Proofs identify the formula whose validity they prove (Unique Type).
- Reduction normalizes proofs. Preservation means that a reduced proof of T is still a proof of T.

As it turns out, the Curry-Howard correspondence extends to formulas with quantifiers. For this one has types with quantifiers. Quantified types turn out to be computationally significant. They provide for polymorphism and abstract types. The standard system for quantified types is the polymorphic λ-calculus, which we shall study later.

Exercise 4.6.1 (Natural Deduction) Find ND-proofs for the following formulas:

a) $((X \to Y) \times X) \to Y$

b) $(X + Y) \to \overline{\overline{X} \times \overline{Y}}$

c) $(X \times Y) \to \overline{\overline{X} + \overline{Y}}$

d) $\overline{X \times \overline{X}}$

e) $\overline{X + Y} \to (\overline{X} \times \overline{Y})$

f) $(\overline{X} \times \overline{Y}) \to \overline{X + Y}$

g) $0 \to X$

h) $(\overline{X} \to \overline{Y}) \to (Y \to X)$

Exercise 4.6.2 (Peirce's Law) Peirce's Law is the Boolean formula

$$((X \to Y) \to X) \to X$$

This formula is valid. Hence it can be proven in ND. One can show that every proof for Peirce's law in ND must involve a subterm formed with δ. This is somewhat surprising since Peirce's law just employs implication while δ must be used with terms whose type involves 0.

a) Find a proof for Peirce's Law in ND.

b) Read the Wikipedia entry for Peirce's law.

Exercise 4.6.3 (Fix) If we extend ND with a recursion operator fix with the usual typing rule

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

we can prove everything.

a) Let T be a type. Find a proof for T in ND extended with fix.

b) Let T be a type. Find a proof for T in ND extended with fix that applies fix only to terms of the form $\lambda x_1{:}T_1.\lambda x_2{:}T_2.t$.

Exercise 4.6.4 (Proof Checker) A SML interpreter provides a proof checker for ND. Type variables, procedure types, products and 1 (unit) are built in. Sum types can be obtained with

```
datatype ('a,'b) sum = L of 'a | R of 'b
```

The type 0 can be realized as follows:

```
datatype null = N of null
val delta : (('a -> null) -> null) -> 'a = fn _ => raise Match
```

Now we have a proof checker for ND. First we try a proof for $0 \to X$:

```
fn n:null => delta (fn f:'a->null => n)
```
fn : null → α

Since SML has type reconstruction, proofs can be written without type annotations:

```
fn n => delta (fn _ => n)
```
fn : null → α

If we bind the proof to an identifier p

```
val p = fn n => delta (fn _ => n)
```
val p : null → α

we obtain a polymorphic proof of $0 \to T$ for all types T. Here is a proof for $((X + Y) \times (\overline{X} + Z)) \to (Y + Z)$ that exploits SML's pattern matching and the polymorphic proof p:

$$
\begin{array}{llll}
X \in \mathit{Sor} & := & \mathbb{N} & \text{sort} \\
T \in \mathit{Ty} & ::= & \mathbf{B} \mid X \mid T \to T & \text{type} \\
x \in \mathit{Nam} & := & \mathbb{N} \times \mathit{Ty} & \text{name} \\
t \in \mathit{Ter} & ::= & x \mid t = t \mid t\,t \mid \lambda x.t & \text{term}
\end{array}
$$

$$
\frac{}{x : T} \quad x = (n, T) \qquad \frac{t_1 : T \quad t_2 : T}{t_1 = t_2 : \mathbf{B}}
$$

$$
\frac{t_1 : T_2 \to T \quad t_2 : T_2}{t_1 t_2 : T} \qquad \frac{x : T_1 \quad t : T_2}{\lambda x.t : T_1 \to T_2}
$$

Figure 4.5: The simply-typed λ-calculus STT

```
fn (R y, _) => L y
 | (_, R z) => R z
 | (L x, L f) => p (f x)
```
*fn : (α, β) sum * (α → null, γ) sum → (β, γ) sum*

Write your proofs for Exercise 4.6.1 in SML and check them with an interpreter.

4.7 Simple Type Theory

Simple type theory is an expressive logic language that takes a simply typed λ-calculus as its syntactic base. We employ the **calculus STT** shown in Figure 4.5. STT comes with a special base type **B** (read Boole) and special terms $s = t$ called **equations**. There are base types different from **B** that are called **sorts**. The variables of STT are called **names** and come with a built-in type. This way, abstractions don't need to specify their argument type and the typing relation can be defined without type environments (as done in Figure 4.5). A term t is **well-typed** if there is a type T such that $t : T$. Every well-typed term has exactly one type. For the purposes of simple type theory only well-typed terms are considered.

The types of STT are **interpreted** as non-empty sets. Every interpretation must interpret **B** as the set $\{0, 1\}$. A function type $S \to T$ must be interpreted as the set of all total functions from the interpretation of S to the interpretation of T. Hence an interpretation is determined for all types if it is determined for all sorts.

A well-typed term is **interpreted** as an element of the set that is the interpretation of its type. An interpretation is determined for all terms if it is determined for all sorts and all names.

Given an interpretation, we say that a syntactic object **denotes** its interpretation. Moreover, we call the interpretation of a syntactic object its **denotation**.

An equation $s = t$ denotes 1 if s and t denote the same value, and 0 otherwise. An application st denotes the value obtained by applying the function denoted by s to the value denoted by t. Finally, an abstraction $\lambda x.t$ denotes the total function from the set denoted by the type of x to the set denoted by the type of t that for a value v yields the value denoted by t if the argument variable x is interpreted as v.

It is not difficult to formalize the notion of interpretation. We require that Ty and Ter are disjoint sets and that $Sor \subseteq Ty$ and $Nam \subseteq Ter$. Now an **interpretation** is a function $\mathcal{I}$ such that:

1. $Dom\,\mathcal{I} = Ty \cup Nam$
2. $\mathcal{I}\mathbf{B} = \{0, 1\}$
3. $\mathcal{I}(S \to T) = \{\, \varphi \mid \varphi \text{ function } \mathcal{I}S \to \mathcal{I}T \,\}$ for all types S, T
4. $x : T \implies \mathcal{I}x \in \mathcal{I}T$ for all names x and all types T

Exercise 4.7.1 Which condition of the above definition ensures that sorts are interpreted as non-empty sets?

Proposition 4.7.2 (Coincidence) Two interpretations are equal if they agree on all sorts and all names.

Given an interpretation $\mathcal{I}$, a name and a type $x : T$, and a value $v \in \mathcal{I}T$, we use $\mathcal{I}_{x,v}$ to denote the interpretation $\mathcal{I}[x{:=}v]$.

Proposition 4.7.3 (Evaluation) For every interpretation $\mathcal{I}$ there exists exactly one function $\hat{\mathcal{I}}$ such that:

1. $Dom\ \hat{\mathcal{I}} = Ter$
2. $t : T \implies \hat{\mathcal{I}}t \in \mathcal{I}T$ for all terms t and all types T
3. $\hat{\mathcal{I}}x = \mathcal{I}x$ for all names x
4. $\hat{\mathcal{I}}(s = t) = \text{if } \hat{\mathcal{I}}s = \hat{\mathcal{I}}t \text{ then } 1 \text{ else } 0$ for all equations $s = t$
5. $\hat{\mathcal{I}}(st) = (\hat{\mathcal{I}}s)(\hat{\mathcal{I}}t)$ for all applications st
6. $\hat{\mathcal{I}}(\lambda x.t) = \lambda v{\in}\mathcal{I}T.\ \hat{\mathcal{I}}_{x,v}t$ for all abstractions $\lambda x.t$ and types $x : T$

A **formula** is a term of type $\mathbf{B}$. An interpretation $\mathcal{I}$ **satisfies** a formula t if $\hat{\mathcal{I}}t = 1$. A formula is **valid** if it is satisfied by every interpretation. A formula is **satisfiable** if it is satisfied by at least one interpretation.

4.7.1 Boolean Connectives and Quantifiers

There are closed terms that denote in every interpretation the Boolean connectives and the quantifiers. We start with the truth values (0 and 1) and the negation function:

$$\begin{aligned} \top &:= (\lambda x.x) = (\lambda x.x) && \text{where } x : \mathbf{B} \\ \bot &:= (\lambda x.x) = (\lambda x.\top) && \text{where } x : \mathbf{B} \\ \neg &:= \lambda x.\, x = \bot && \text{where } x : \mathbf{B} \end{aligned}$$

Let T be a type. A **property for T** is a term of type $T \to \mathbf{B}$. The **universal quantifier** $\forall_T : (T \to \mathbf{B}) \to \mathbf{B}$ checks whether a property holds for all elements of T, and the **existential quantifier** $\exists_T : (T \to \mathbf{B}) \to \mathbf{B}$ checks whether a property holds for at least one element of T. The quantifiers can be expressed in STT as follows:

$$\begin{aligned} \forall_T &:= \lambda f.\, f = (\lambda z.\top) && \text{where } f : T \to \mathbf{B} \\ \exists_T &:= \lambda f.\, \neg(f = (\lambda z.\bot)) && \text{where } f : T \to \mathbf{B} \end{aligned}$$

The usual notation is recovered as follows:

$$\begin{aligned} \forall x.t &:= \forall_T(\lambda x.t) && \text{where } x : T \\ \exists x.t &:= \exists_T(\lambda x.t) && \text{where } x : T \end{aligned}$$

Now its easy to express conjunction, disjunction and implication:

$$\begin{aligned} \wedge &:= \lambda xy.\, \forall g.\, gxy = g\top\top && \text{where } g : \mathbf{B} \to \mathbf{B} \to \mathbf{B} \\ \vee &:= \lambda xy.\, \neg(\forall g.\, gxy = g\bot\bot) && \text{where } g : \mathbf{B} \to \mathbf{B} \to \mathbf{B} \\ \to &:= \lambda xy.\, \neg(\forall g.\, gxy = g\top\bot) && \text{where } g : \mathbf{B} \to \mathbf{B} \to \mathbf{B} \end{aligned}$$

We say that an interpretation interprets a name $\mathrm{C} : (T \to \mathbf{B}) \to T$ as **choice for T** if it satisfies the formula $f(\mathrm{C}f) = \exists_T f$. Choices are useful if we want to describe an object by a property that characterizes it uniquely. For instance, if we know that N is interpreted as $\mathbb{N}$, $+ : N \to N \to N$ as addition, and C as choice for N, we can describe 0 as $\mathrm{C}(\lambda x.\, x = x + x)$.

Exercise 4.7.4 (Numbers) Let $\mathcal{I}$ be an interpretation that interprets the sort N and the names $+, \cdot : N \to N \to N$ as the natural numbers with addition and multiplication. In addition, assume that $\mathcal{I}$ interprets C as choice for N. Find terms that denote in $\mathcal{I}$ the following:

a) The number 0.

b) The number 1.

c) Subtraction $- : N \to N \to N$. For $x < y$, you can choose $x - y$ as you like.

d) Integer division $\div : N \to N \to N$. You can choose $x \div 0$ as you like.

e) Less or equal $\leq : N \to N \to B$.

Exercise 4.7.5 (Conditional) Let $\mathcal{I}$ be an interpretation. Find a term that denotes in $\mathcal{I}$ the function $\lambda b{\in}\mathbb{B}.\ \lambda u{\in}\mathcal{I}X.\ \lambda v{\in}\mathcal{I}X.\ \mathit{if}\ b\ \mathit{then}\ u\ \mathit{else}\ v$.

a) Assume that $\mathcal{I}$ interprets C as choice for N.

b) Assume that $\mathcal{I}$ interprets C as choice for $\mathbf{B} \to N \to N \to N$.

Exercise 4.7.6 (Termination) Find a formula that is satisfied by an interpretation if and only if it interprets the name $r : T \to T \to \mathbf{B}$ as a terminating relation. A relation *terminates* if there is no infinite sequence a_0, a_1, a_2, ... such that the pair $(a_i, a_i + 1)$ is in the relation for all $i \in \mathbb{N}$. You formula should not use the natural numbers. Hint: Specify non-termination of r first.

4.7.2 Specification of the Natural Numbers

We can specify the natural numbers with a formula. For this we choose a sort N and names $0 : N$ and $\sigma : N \to N$. Now the specifying formula t should satisfy the following conditions:

1. Every interpretation that interprets N as $\mathbb{N}$, 0 as 0, and σ as $\lambda n{\in}\mathbb{N}.\ n + 1$ satisfies t.
2. If an interpretation satisfies t, then it interprets N as $\mathbb{N}$, 0 as 0, and σ as $\lambda n{\in}\mathbb{N}.\ n + 1$ (up to isomorphism).

The "up to isomorphism" relaxation of the second condition accounts for the fact that there are many equivalent constructions of the natural numbers.

A specification of the natural numbers was first given by Peano in 1889. It consists of three conditions:

1. Every element of N is reachable from 0 by finitely many applications of σ.
2. σ is an injective function.
3. σ never yields 0.

The second and third condition are easy to formalize:

$$\forall xy.\ (\sigma x = \sigma y) = (x = y)$$
$$\neg(\exists x.\ \sigma x = 0)$$

The first condition is formalized as the famous induction axiom, which says that a property that holds for 0 and is closed under σ must hold for all natural numbers:

$$\forall f.\ f0 \wedge (\forall x. fx \to f(\sigma x)) \ \to\ \forall x. fx$$

The induction axiom forces the reachability condition since the reachable elements of N yield a property that holds for 0 and is closed under σ.

It's easy to extend the specification of the natural numbers with addition and multiplication since these operations can be defined by recursion with respect to 0 and σ. For addition the specifying formula looks as follows:

$$\forall xy.\, 0 + y = y \ \wedge\ \sigma x + y = x + \sigma y$$

If there is a choice C for $N \to N \to N$ available, there is even a term that denotes addition:

$$\mathrm{C}(\lambda f.\, \forall xy.\, f0y = y \ \wedge\ f(\sigma x)y = fx(\sigma y))$$

Exercise 4.7.7 (Multiplication)

a) Give a formula that specifies multiplication. You may use 0, σ and $+$.

b) Give a term that denotes multiplication. You may use 0, σ, $+$ and a choice C for $N \to N \to N$.

Exercise 4.7.8 (Pairs) Let the sorts X, Y, Z and the names $\mathit{pair} : X \to Y \to Z$, $\mathit{fst} : Z \to X$, $\mathit{snd} : Z \to Y$ be given. Find a formula that is satisfied by an interpretation if and only if Z denotes (up to isomorphism) the cartesion product of the denotations of X and Y, and *pair*, *fst*, and *snd* denote the pairing and projection functions.

4.7.3 Semantic equivalence

Semantic equivalence of terms is defined as follows:

$$s \sim t \ :\Longleftrightarrow\ \exists T\colon\ s : T \ \wedge\ t : T \ \wedge\ s = t \text{ valid}$$

Proposition 4.7.9 Semantic equivalence $\sim$ satisfies the following properties:

1. $\sim$ is an equivalence relation on the set of well-typed terms
2. $s \sim t \implies C[s] \sim C[t]$ if $C[s] : T$ and $C[t] : T$
3. $s \sim t \implies \theta s \sim \theta t$ if $\forall x\colon\ x : T \Longrightarrow \theta x : T$
4. $(\lambda x.s)t \sim [x{:=}t]s$ if $(\lambda x.s)t$ well-typed
5. $\lambda x.tx \sim t$ if $x \notin \mathit{FV}\, t$ and tx well-typed
6. $\lambda x.t \sim \lambda y.\,[x{:=}y]t$ if $y \notin \mathit{FV}\, t$, x and y have same type, and t well-typed

Proposition 4.7.10 Semantic equivalence is not semi-decidable.

Proof Follows from the fact that simple type theory can specify Turing machines and the halting problem. ∎

4.7.4 Historical Remarks

Simple type theory originated with Church [1940] and was further developped by Henkin [1950, 1963]. A recent textbook is Andrews [2002]. Simple type theory is the logical base of the interactive theorem provers HOL and Isabelle.

Church didn't like types. His goal was to develop the untyped λ-calculus into a foundation for both computation and logic. Church wanted this foundation to be defined without the use of set theory. In 1934 his students Kleene and Rosser published a paper showing that Church's original set-up of logic was inconsistent. They showed that for any term $\neg$ representing negation there is a term t such that t and $\neg t$ are deductively equivalent, which is logically inconsistent. The problem is caused by terms whose reduction doesn't terminate. Hence it goes away in the simply typed lambda-calculus STT.

In retrospect, what Kleene and Rosser discovered is a trivialiy. They took Russel's antinomy that had been used before to show the inconsistency of Cantor's and Frege's systems and translated it into λ-calculus. Russell's antinomy is based on the assumption that $\{ x \mid x \notin x \}$ is a set (which it is not in modern set theory). The claimed set translates to the term $\lambda x. \neg(xx)$ in the λ-calculus. The rest we leave as an exercise.

If you want to know more, you may read J. Barkley Rosser's paper *Highlights of the History of the Lambda-Calculus* [1982].

Exercise 4.7.11 Let s be a term of the untyped λ-calculus. Find a term t such that t reduces in one step to st.

4.8 Termination

Theorem 4.8.1 (Termination) Reduction in the pure simply typed λ-calculus terminates.

5 Computational Effects: State, I/O & Friends

So far, we have investigated semantics and types of various ‘pure’ languages, which we took as idealized models of functional programming. Purity manifests itself in the principle of **referential transparency** that was valid in all the previous lambda calculi that we considered: identifiers can be freely replaced by their respective bindings. Essentially, referential transparency boils down to the following property:

$$t \Downarrow v \implies t \sim v \tag{5.1}$$

which means that evaluation of terms has no observable effect.

Most practical languages include also ‘impure’, ’side-effecting’ features to access the underlying machine state or communicate with the outside world. Examples of such **computational effects** are

- mutable state
- I/O
- exceptions, jumps, and continuations
- synchronization and non-determinism, as arising from concurrency

and generally they will invalidate (5.1). Sometimes even divergence is considered a computational effect: it makes the difference between various reduction strategies observable.

In this chapter we will see how the framework of lambda calculus can be extended to capture computational effects. We look at the case of an output operation first, and then consider mutable state in the form of ML-style references. Further examples include a nondeterministic choice operator and a simple concurrent lambda calculus. Finally we will consider control flow operators.

Since (most) computational effects make the order of evaluation observable, it is useful to have a form of sequencing. Recall that in the lambda calculus, sequencing is already available as ‘syntactic sugar’:

$\mathit{let}\ x = t\ \mathit{in}\ t'$	for $(\lambda x{:}T.t')t$	(appropriate T)
$t; t'$	for $\mathit{let}\ x = t\ \mathit{in}\ t'$	$(x \notin FV(t'))$

Required Reading: Pierce, TAPL, Chapters 13 and 14. Introduces reference types and exceptions. Provides further discussion of design decisions.

5.1 Printing

Fix a finite set Σ (the alphabet) and let σ range over Σ^* (finite strings over Σ). By ϵ we denote the empty string, and $\sigma \cdot \sigma'$ (or simply $\sigma\sigma'$) stands for the concatenation of $\sigma, \sigma' \in \Sigma^*$.

We now extend the set of lambda terms by new terms *print* σ, one for each $\sigma \in \Sigma^*$:

$$t ::= \ldots \mid \mathit{print}\ \sigma$$

(The sets of types and values do not change.) The intended behaviour is that *print* σ simply returns $()$, but in addition outputs the string σ. Thus, while both *print* σ; *print* σ' and *print* σ'; *print* σ return $()$, their semantics generally differs because the former outputs $\sigma\sigma'$ whereas the latter prints $\sigma'\sigma$.

Exercise 5.1.1 Argue similarly that the terms

$$\mathit{let}\ x = \mathit{print}\ \sigma\ \mathit{in}\ (x; x) \quad \text{and} \quad \mathit{let}\ x = (\lambda y{:}1.\mathit{print}\ \sigma)\ \mathit{in}\ (x(); x())$$

behave differently although they both evaluate to $()$. (Contrast this with the case of, e.g. PCF, where $\mathit{let}\ x = t\ \mathit{in}\ (x; x) \sim \mathit{let}\ x = (\lambda y{:}1.t)\ \mathit{in}\ (x(); x())$ holds for all t.)

Semantics

We extend the semantics to record the output that occurs during evaluation. To this end, the big-step semantics $t \Downarrow v$ becomes a ternary relation between terms, values and strings which we write $t \Downarrow v|\sigma$. It is defined inductively by the rules in Figure 5.1. Note how the rules for values and applications are obtained straightforwardly from those of simply typed lambda calculus. Also observe how the case $t_1 t_2 \Downarrow v|\sigma$ now determines the left-to-right evaluation of applications – this was unspecified in the corresponding rule.

It is also possible to give a small-step semantics to the language. This is most naturally given in the form of a relation $t|\sigma \to t'|\sigma'$ between pairs of terms and strings. The second component of these 'configurations' $t|\sigma$ can be thought of as collecting the output that has already occurred. The proper reduction rules that define top-level reduction are:

$$\begin{aligned} (\lambda x{:}T.t)v|\sigma &\to_0 [x := v]t|\sigma \\ \mathit{print}\ \sigma'|\sigma &\to_0 ()|\sigma\sigma' \end{aligned}$$

$$\frac{}{\mathit{print}\ \sigma \Downarrow () | \sigma} \qquad \frac{}{v \Downarrow v | \epsilon}$$

$$\frac{t_1 \Downarrow \lambda x{:}T.t | \sigma_1 \quad t_2 \Downarrow v_2 | \sigma_2 \quad [x := v]t \Downarrow v | \sigma_3}{t_1 t_2 \Downarrow v | \sigma} \ \sigma = \sigma_1\sigma_2\sigma_3$$

Figure 5.1: Big-step semantics for printing

and the reduction relation is obtained with the help of evaluation contexts:

$$t|\sigma \to t'|\sigma' \ :\Longleftrightarrow\ \exists s, s', E\colon\ t = E[s]\ \wedge\ s|\sigma \to_0 s'|\sigma'\ \wedge\ E[s'] = t'$$

The big-step and small-step semantics are both deterministic, and agree in the following sense:

Proposition 5.1.2 (Coincidence) For every term t, string σ and every value v, we have $t \Downarrow v|\sigma$ if and only if $t|\epsilon \to \cdots \to v|\sigma$.

Exercise 5.1.3 Prove Proposition 5.1.2.

To obtain a natural notion of semantic equivalence of programs that may print, we refine contextual equivalence to take the output into account:

$$s \sim_{\mathit{print}} t \ :\Longleftrightarrow\ \forall C \forall \sigma\colon\ (\exists v\colon\ C[s] \Downarrow v|\sigma \iff \exists v\colon\ C[t] \Downarrow v|\sigma)$$

That is, s and t have the same termination behaviour and generate the same output, in all contexts C.

Exercise 5.1.4 Give separating contexts for the following programs:

a) $\mathit{print}\ \sigma$ and $\mathit{print}\ \sigma'$, where $\sigma \neq \sigma'$

b) $x()$ and $x();x()$

Show that $t \Downarrow v|\sigma \ \Rightarrow\ t \sim_{\mathit{print}} v$ does *not* hold.

Typing and Type Safety

Consider the evident typing rule

$$\frac{}{\Gamma \vdash \mathit{print}\ \sigma : 1}$$

suggested by the semantics of $\mathit{print}\ \sigma$. With its help we can prove progress and type preservation, as straightforward adaptations of the previously stated propositions for PCF and the simply typed lambda calculus. Together they yield type safety.

Proposition 5.1.5 (Progress) Every closed and well-typed term t is either a value, or else for any σ there exists σ', t' such that $t|\sigma \to t'|\sigma'$.

Proposition 5.1.6 (Type Preservation) For all σ and σ', if $\Gamma \vdash t : T$ and $t|\sigma \to t'|\sigma'$, then $\Gamma \vdash t' : T$.

Exercise 5.1.7 Modify the language so that strings are a basic type:

$$T ::= \ldots \mid \mathit{String} \quad \text{and} \quad t ::= \ldots \mid \sigma \mid \mathit{print}\ t \mid t \cdot t$$

For this language, state

- the big-step semantics,
- an equivalent small-step semantics, and
- the rules defining the typing relation.

Exercise 5.1.8 Suggest a way of modelling user input.

5.2 References

This section complements Chapter 13 of TAPL. The combination of dynamically created references and higher-order functions gives rise to an expressive and semantically rich language, as the following exercises show.

Exercise 5.2.1 (Sequence generators in SML) By a *generator for a sequence of natural numbers* $a = (a_i)_{i \in \mathbb{N}}$ we mean a procedure of type $\mathit{unit} \to \mathit{int}$ which returns a_i upon the i-th call. In Standard ML, implement the following procedures:

a) a generator *squares* for the sequence of square numbers $0, 1, 4, 9, 16, \ldots$

b) a (non-recursive) generator *fac* for the factorial numbers $1, 1, 2, 6, 24, \ldots$

c) a generator *fibs* for the sequence of Fibonacci numbers $0, 1, 1, 2, 3, \ldots$

d) a procedure $\mathit{newGen} : (\mathit{int} \to \mathit{int}) \to \mathit{unit} \to \mathit{int}$ that returns a generator for the sequence $f\ 0, f\ 1, f\ 2, \ldots$ when given f.

We can approximate some aspects of object-oriented programming:

Exercise 5.2.2 (Resettable counters in SML)

a) Complete the following SML declaration

```
    val (count, inc, reset) =
```

such that it yields an encapsulated counter with initial value 0 and three procedures to access and modify the counter as follows:

- $\mathit{count} : \mathit{unit} \to \mathit{int}$ returns the value of the counter.
- $\mathit{inc} : \mathit{unit} \to \mathit{unit}$ increases the value of the counter by one.

$$\begin{array}{rcl}
T \in Ty & ::= & \ldots \mid Ref\ T \\
l \in Loc & & \\
v \in Val & ::= & \ldots \mid l \\
t \in Ter & ::= & \ldots \mid ref\ t \mid t := t \mid !t \\
E \in EC & ::= & \ldots \mid ref\ E \mid E := t \mid v := E \mid !E
\end{array}$$

Figure 5.2: Syntax of lambda calculus with references

- *reset* : *unit* → *unit* resets the counter to its initial state.

b) Write a procedure

$$newCounter : int \to (unit \to int) \times (unit \to unit) \times (unit \to unit)$$

such that each call with argument n yields a new counter with initial value n.

Small-step Semantics

The semantics of lambda calculus with references can be given in the form of proper reduction rules and evaluation contexts. Similar to the printing case, → becomes a relation between 'configurations', which in this case are pairs of terms and stores. The extension to the syntax of terms and evaluation contexts are summarized in Figure 5.2. Stores are modelled as finite maps μ from *Loc* to *Val*, and the proper reduction rules associated with the store operations are given by

$$\begin{array}{rcll}
ref\ v|\mu & \to_0 & l|\mu[l := v] & \quad l \notin Dom(\mu) \\
l := v|\mu & \to_0 & ()|\mu[l := v] & \quad l \in Dom(\mu) \\
!l|\mu & \to_0 & \mu(l)|\mu & \quad l \in Dom(\mu)
\end{array}$$

The reduction relation is obtained from these by

$$t|\mu \to t'|\mu' \ :\Longleftrightarrow\ \exists s, s', E\colon\ t = E[s] \ \wedge\ s|\mu \to_0 s'|\mu' \ \wedge\ E[s'] = t'$$

Note that the side-condition of the first proper reduction rule can always be satisfied because $Dom(\mu)$ is a *finite* subset of the infinite set *Loc*. Also note that reduction is now non-deterministic, due to the arbitrarily chosen location $l \notin Dom(\mu)$. However, this non-determinism is benign and does not cause any problems when proving type safety. In fact it could be avoided, e.g. by taking $Loc = \mathbb{N}$ and then always choosing the *smallest* $l \in \mathbb{N}$ such that $l \notin Dom(\mu)$.

(The situation would be different in a language with pointer arithmetic, where the non-deterministic allocator becomes observable.)

As for printing, we need to make a small modification to the definition of contextual equivalence. Here it is due to the fact that generally evaluation depends on an initial store. However, since we are only interested in the equivalence of programs written in the 'surface' language , we can restrict attention to terms (and contexts) that do not contain locations. Since such programs never access the store (outside the region allocated by the program itself), it suffices to define the observations with respect to the *empty* store. We define $t \Downarrow$ to mean that there exist v and μ such that $t|\emptyset \Downarrow v|\mu$, and let

$$t \sim_{ref} t' \quad :\iff \quad \forall C\colon\ Locs(C[t]) = \emptyset = Locs(C[t']) \ \Rightarrow\ (C[t]\Downarrow \iff C[t']\Downarrow)$$

Exercise 5.2.3 (Semantics of lambda calculus with references)

a) Make sure that you can state the proper reduction rules, reduction contexts, contextual equivalence, typing relation and the progress and preservation properties for the lambda calculus with references.

b) Give an equivalent big-step semantics.

c) Give an example of a well-typed, closed term such that its evaluation creates a heap μ with a cycle $\mu(l) = \lambda x{:}Unit.(!l)x$, for some l.

d) Find Γ, μ, and $\Sigma_1 \neq \Sigma_2$ such that $\Gamma|\Sigma_i \vdash \mu$ holds for $i = 1, 2$.

e) At some types it is not necessary to add the comparison operator $r = s$ on references as a new primitive: give a closed term t (depending on r, s) such that t evaluates to *true* if r and s are bound to the same location, and to *false* otherwise, where

- r and s have type $Ref\ Int$;
- r and s have type $Ref\ Bool$.

Use SML to test your terms.

Exercise 5.2.4 (Recursion)

a) Show that the reduction relation is not normalizing on well-typed terms of the lambda calculus with references. **Hint:** Use a reference to a location of type $1 \to 1$. Test your term using SML.

b) Let T be any type. Find a well-typed term $Fix : ((T \to T) \to T \to T) \to T \to T$ that evaluates to a procedure that behaves like a recursion operator. **Hint:** for any type T it is possible to allocate a reference of type $(T \to T)\ Ref$. Use SML to test your terms.

c) Consider the simply typed lambda calculus extended with only $Unit$ references, (i.e. where the types are given by $T ::= Unit \mid Int \mid T \to T \mid Ref\ Unit$).

$$
\begin{array}{rcl}
t \in \mathit{Ter} & ::= & \ldots \mid t \oplus t \\
E \in \mathit{EC} & ::= & \ldots \mid E \oplus t \mid v \oplus E
\end{array}
$$

$$v_1 \oplus v_2 \to_0 v_1$$

$$v_1 \oplus v_2 \to_0 v_2$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 \oplus t_2 : T}$$

Figure 5.3: Lambda calculus with choice

Show that reduction is normalizing. **Hint:** You may assume that reduction is normalizing in the simply typed lambda calculus over base types *Unit* and *Int*. Use this to simulate reduction steps in the calculus with references.

Exercise 5.2.5 (Contextual equivalence) Consider the simply typed lambda calculus with pairs and references, over base types *Int* and *Bool*. Find separating contexts for the following expressions.

a) $(\mathit{ref}\ 0, \mathit{ref}\ 0)$ and $(\lambda r.(r, r))\ (\mathit{ref}\ 0)$

b) $r := 1; s := 2$ and $s := 2; r := 1$

c) $r := 1; r :=!s$ and $r :=!s$

d) $\mathit{let}\ r = \mathit{ref}\ 0\ \mathit{in}\ \mathit{let}\ s = \mathit{ref}\ 0\ \mathit{in}\ \ \lambda x.\mathit{if}\ x = r\ \mathit{then}\ r\ \mathit{else}\ s$ and
$\mathit{let}\ r = \mathit{ref}\ 0\ \mathit{in}\ \mathit{let}\ s = \mathit{ref}\ 0\ \mathit{in}\ \ \lambda x.\mathit{if}\ x = s\ \mathit{then}\ r\ \mathit{else}\ s$

Note that the last inequivalence is interesting, since contexts (initially) have no access to the locally declared references r and s. It is due to Andy Pitts and Ian Stark.

5.3 Nondeterminism and Concurrency

Extending the lambda calculus with a nondeterministic choice construct introduces observable nondeterminism. Figure 5.3 summarizes syntax, operational semantics and typing of such an extension. There are few surprises: we have 2 new proper reduction rules, which entail that $\to$ is no longer deterministic. Since either branch t_1, t_2 may be the result of the choice $t_1 \oplus t_2$, the typing rule ensures that the respective types agree. Note however that the branches t_1, t_2 must be fully evaluated before the choice is taken (i.e., we do not discard *computations* but only *values*).

For example, the term

$$t \;:=\; (\lambda x{:}\mathit{1}.\mathit{false}) \oplus (\lambda x{:}\mathit{1}.\mathit{true})$$

may reduce to either the constantly *true* or constantly *false* function of type $\mathit{1} \to \mathit{Bool}$. Consequently, $t()$ can yield either *true* or *false*. Similarly,

$$\lambda x{:}T.\lambda y{:}T.x \oplus y$$

is a procedure that returns either its first or second argument.

Exercise 5.3.1 Develop an equivalent big-step operational semantics for lambda calculus with choice.

Next, let us consider the definition of contextual equivalence. If we adopted the usual notion this would lead to an inconsistency: let Ω be a divergent term, and consider the term

$$t \;:=\; ((\lambda x{:}\mathit{1}.\mathit{true}) \oplus (\lambda x{:}\mathit{1}.\Omega))()$$

Depending on the choice, this term may reduce to the result *true*. So we have

$$t \sim \mathit{true} \tag{5.2}$$

But t may also reduce to Ω, and thus

$$t \sim \Omega \tag{5.3}$$

Combining these equivalences (by transitivity of $\sim$), we obtain $\mathit{true} \sim \Omega$. But clearly this latter equivalence does not make sense. The solution is to refine the notion of convergence so that it becomes appropriate for a nondeterministic setting: we write $t\downarrow$ if t **may converge**, i.e. if there is *some* finite reduction sequence $t \to \ldots \to t'$ where t' is irreducible, and we write $t \Downarrow$ if t **must converge**, i.e. if there is *no* infinite reduction sequence $t = t_0 \to t_1 \to t_2 \to \ldots$ starting from t. We can then define both may- and must-contextual equivalence,

$$\begin{aligned} t \sim_{may} t' \quad &:\Longleftrightarrow \quad \forall C{:}\; C[t]\downarrow \iff C[t']\downarrow \\ t \sim_{must} t' \quad &:\Longleftrightarrow \quad \forall C{:}\; C[t]\Downarrow \iff C[t']\Downarrow \end{aligned}$$

and obtain contextual equivalence $t \sim t'$ as their intersection:

$$t \sim t' \quad :\Longleftrightarrow \quad t \sim_{may} t' \;\wedge\; t \sim_{must} t'$$

Note that (5.2) no longer holds because the terms are not must-contextually equivalent, and that (5.3) fails because the terms are not may-contextually equivalent. Moreover, note that this definition is a generalization of the one from PCF, since in deterministic languages may- and must-convergence coincide.

The introduction of nondeterminism may seem like a needless complication, but it arises quite naturally if one extends the lambda calculus with primitives for concurrent computation. There, concurrently running threads will usually compete for access to some shared 'resources'. A simple example is obtained by extending the lambda calculus with references by a construct

$$\mathit{cobegin}\ t_1 \| \ldots \| t_n\ \mathit{end}$$

The intended meaning is that upon entering the block, $t_1, \ldots, t_n$ are reduced concurrently. One possible formalization is in terms of **interleaving**, e.g. by introducing evaluation contexts

$$\mathit{cobegin}\ t_1 \| \ldots \| t_{i-1} \| E \| t_{i+1} \| \ldots \| t_n\ \mathit{end}$$

for $i = 1, \ldots, n$. If all t_i have been fully reduced, we simply replace the whole block by $()$.

Exercise 5.3.2 Several alternatives to this decision are similarly sensible. Discuss (and formalize) a few of them.

Exercise 5.3.3 Give a typing rule for $\mathit{cobegin}\ t_1 \| \ldots \| t_n\ \mathit{end}$.

Exercise 5.3.4 Give an example that shows how nondeterminism arises. **Hint:** consider a block where the t_i have access to a shared reference.

5.4 Exceptions and Continuations

This section contains some additional observations, extending TAPL, Chapter 14. If you want to know more about type and effect systems have a look at the book *Principles of Program Analysis* by Nielson, Nielson and Hankin. More on continuations and their connection to classical logic can be found in Robert Harper's lecture notes.

5.4.1 Error Propagation and Exceptions

The intuition that *error* propagates by popping frames from the control stack can be made precise by considering **stack frames**

$$F \in \mathit{Frm} ::= [\,]\ t \mid v\ [\,]$$

which are the minimal building blocks for evaluation contexts. It is easily seen that the proper reduction rules for *error* given in TAPL are exactly those of the form

$$F[\mathit{error}] \to_0 \mathit{error}$$

The advantage of using frames is that this formulation more directly extends to richer languages.

Exercise 5.4.1 Show that every evaluation context E can be uniquely decomposed into frames $F_1, \ldots, F_n$ such that $E = F_n[F_{n-1}[\ldots[F_1]\ldots]]$, for some $n \geq 0$.

Exercise 5.4.2 Define the set of stack frames corresponding to the evaluation contexts of the lambda calculus with pairs.

An alternative semantics of *error* could be to 'throw away' the evaluation context in one go, rather than popping off individual stack frames one by one. More precisely, one defines

$$\begin{aligned} t \to t' \;:\Longleftrightarrow\; & \exists s, s', E\colon\; t = E[s] \;\wedge\; s \to_0 s' \;\wedge\; E[s'] = t' && (5.4)\\ & \vee\; \exists E \neq [\,]\colon\; t = E[\mathit{error}] \;\wedge\; t' = \mathit{error} && (5.5) \end{aligned}$$

Exercise 5.4.3 Once exceptions can be handled (ie. after adding the *try t with t′* construct), not every context is 'propagating'. Adapt (5.4) accordingly, by defining which evaluation contexts propagate *error* and exceptions *raise v*, respectively. Reduction shall remain deterministic.

Exercise 5.4.4 (Semantics of exceptions) Consider the simply typed lambda calculus extended with exceptions that carry values.

a) Make sure that you can state the (proper) reduction rules, typing rules, and the revised progress property.

b) State the inference rules that define an equivalent big-step semantics $t \Downarrow r$ for this language. Here r is a *result*, which can either be a value or a term of the form *raise v*. Take care to treat exceptions correctly in the rule for applications.

c) Let T_1, T_2 be any types. Find a well-typed term that evaluates to a procedure that behaves like a recursion operator $\mathit{fix} : ((T_1 \to T_2) \to T_1 \to T_2) \to T_1 \to T_2$. Use SML to test your term.

Exercise 5.4.5 (Recursion) Consider the simply typed lambda calculus extended with exceptions that carry values.

a) Show that reduction is not normalizing on well-typed terms of this calculus.

b) Let T_1, T_2 be any types. Find a well-typed term that evaluates to a procedure that behaves like a recursion operator $\mathit{fix} : ((T_1 \to T_2) \to T_1 \to T_2) \to T_1 \to T_2$.

Use SML to test your terms.

5.4.2 Exception Analysis: A Type and Effect-System

As we've seen already, the presence of computational effects invalidates many useful program transformations. For example, the *dead code elimination*

$$\mathit{let}\ x = t_1\ \mathit{in}\ t_2,\ x \notin FV(t_2) \quad \longmapsto \quad t_2$$

is valid only if evaluation of t_1 is effect-free. Consequently, one is often interested in establishing the absence of effects. In the case of exceptions, we may therefore ask:

> Are any exceptions raised during evaluation of term t? And if so, which ones?

Like most such questions, this is undecidable in general, and we will have to look for a (safe) approximation: a set ε of exception names such that

1. if t raises exn then $exn \in \varepsilon$,
2. ε may possibly contain more exceptions.

We will obtain such an approximation by *refining the type system* for exceptions with information about possibly raised exceptions, defining a relation

$$\Gamma \vdash t : T : \varepsilon$$

where, intuitively, $\emptyset \subseteq \varepsilon \subseteq T_{exn}$. (For simplicity, let us assume that $T_{exn} = < exn_1 : T_1, \dots, exn_n : T_n >$, i.e., T_{exn} corresponds to the sum type $T_1 + \dots + T_n$ and we will write the injection of $t : T_i$ into the i-th component simply as $exn_i\ t$; see TAPL, Chap. 11.10.) This gives us an **effect system**, which is roughly a type system with additional annotations. The benefit of this approach is that

- the analysis is **compositional**: effect information about a compound term is derived from effect information of its component terms, i.e., it is structurally inductive just like a type system; and
- the **soundness of the analysis** may be established with the methods we already know: it follows from preservation and progress properties.

Figure 5.4 presents the rules that define the effect system. Formally, **(effect) annotations** and **annotated types** are given by the grammar

$$\varepsilon ::= \emptyset \ \mid\ \{exn_i\} \ \mid\ \varepsilon \cup \varepsilon$$

$$\frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool} : \emptyset} \qquad \frac{(x, T) \in \Gamma}{\Gamma \vdash x : T : \emptyset} \qquad \frac{\Gamma[x := T] \vdash t : T' : \varepsilon}{\Gamma \vdash \lambda x{:}T.t : T \xrightarrow{\varepsilon} T' : \emptyset}$$

$$\frac{\Gamma \vdash t_1 : T' \xrightarrow{\varepsilon} T : \varepsilon_1 \qquad \Gamma \vdash t_2 : T : \varepsilon_2}{\Gamma \vdash t_1 t_2 : T : \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon}$$

$$\frac{\Gamma \vdash t : \mathit{Bool} : \varepsilon \qquad \Gamma \vdash t_1 : T : \varepsilon_1 \qquad \Gamma \vdash t_2 : T : \varepsilon_2}{\Gamma \vdash \mathit{if}\ t\ \mathit{then}\ t_1\ \mathit{else}\ t_2 : T : \varepsilon \cup \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Gamma \vdash t : T_i : \varepsilon}{\Gamma \vdash \mathit{raise}\ t : T : \varepsilon \cup \{\mathit{exn}_i\}} \qquad \frac{\Gamma \vdash t_1 : T : \varepsilon_1 \qquad \Gamma \vdash t_2 : T_{exn} \xrightarrow{\varepsilon} T : \varepsilon_2}{\Gamma \vdash \mathit{try}\ t_1\ \mathit{with}\ t_2 : T : \varepsilon_2 \cup \varepsilon}$$

$$\frac{\Gamma \vdash t : T : \varepsilon \qquad \varepsilon \subseteq \varepsilon'}{\Gamma \vdash t : T : \varepsilon'}$$

Figure 5.4: Effect system for exception analysis

$$T ::= X \mid T \xrightarrow{\varepsilon} T \mid T + T \mid \ \ldots$$

Since values are already fully evaluated, they cannot raise any exceptions. This explains why we can soundly set $\varepsilon = \emptyset$ in the case of basic values like *true*, *false*,... Similarly, since in a call-by-value language variables will only be bound to values, $\varepsilon = \emptyset$ also in the case of variables. The case for abstractions is more interesting, as any exceptions that may be raised when evaluating the procedure body will not occur until the time when the procedure is applied. Thus the function type is equipped with an additional annotation to transfer this information from the point of definition to the point of use of the function. In fact, this refinement of function types is characteristic of effect systems (not only the one for exception analysis).

The rule for applications $t_1 t_2$ collects not only the exceptions that may be raised by evaluation of the components, i.e., those of t_1 and t_2, but also the set of exceptions ε arising from evaluating the function body. Similarly, the rule for conditionals collects the information obtained by the component terms t, t_1, t_2. In order to obtain the preservation property, we must add (why?) a rule for 'subeffecting',

$$\Gamma \vdash t : T : \varepsilon \ \wedge \ \varepsilon \subseteq \varepsilon' \implies \Gamma \vdash t : T : \varepsilon'$$

The rule for *raise* t is the only one where the set ε is actually extended. Dually, ε may become smaller when passing through a handler: any exception raised by t_1 in *try* t_1 *with* t_2 will be captured, so it need not appear in the annotation unless it is re-raised during the evaluation of the handler.

Exercise 5.4.6 (Exception analysis) Prove soundness of the inference system for exception analysis: Suppose $\emptyset \vdash t : T : \varepsilon$. Then

$$t \to \ldots \to t' \text{ where } t' \text{ is irreducible } \implies t' \in \mathit{Val} \ \vee \ \exists v\colon \ (t' = \mathit{exn}_i v \wedge \mathit{exn}_i \in \varepsilon)$$

You will need to adapt the Progress and Preservation theorems accordingly.

Exercise 5.4.7 Complete the effect system by stating rules for

a) the *case*-construct

b) the *fix*-construct, for recursive function definitions

Extend the soundness proof accordingly.

Exercise 5.4.8 (Conservativity) Show that the effect system *conservatively* extends the type system: Let us write $|T|$ for the simple type obtained by stripping off all annotations, and extend this to contexts and terms in the evident way. Then $\Gamma \vdash t : T : \varepsilon$ for some ε if and only if $|\Gamma| \vdash |t| : |T|$.

5.4.3 Abstract Machine and Continuations

In order to discuss control flow and continuations it is helpful to consider an execution model for the lambda calculus that is slightly closer to an implementation. In order to make explicit the search for the next redex position (i.e., the splitting of a reducible term t into E and s such that $t = E[s]$ and s is the left hand side of a proper reduction rule), we will consider an **abstract machine** that performs this operation explicitly.

The machine operates on configurations $(t, k) \in \mathit{Ter} \times \mathit{Stk}$ where t is a term and k is a list of stack frames $F_n{::}\ldots{::}F_1{::}\mathit{nil}$. We may sometimes use list notation $[F_n, \ldots F_1]$ for this. As a consequence of Exercise 5.4.1, such lists are in one-one correspondence with evaluation contexts. The initial state of the machine will be configurations of the form (t, nil), the final ones those of the form (v, nil). A key difference to the previous operational semantics is that now every reduction rule is proper.

Obviously we would like a good fit between the earlier semantics, and the new one given by the abstract machine:

Proposition 5.4.9 (Coincidence of small-step and abstract machine semantics) For all t and v, $t \Downarrow v$ iff $(t, \mathit{nil}) \mapsto \ldots \mapsto (v, \mathit{nil})$.

Exercise 5.4.10 Prove Proposition 5.4.9.

- For '$\Rightarrow$', you need to prove the more general property that if $t \Downarrow v$ then $\forall k\colon \ (t, k) \mapsto \ldots \mapsto (v, k)$.

Types, terms and stacks

$$\begin{aligned}
T \in Ty &::= X \mid T \to T \\
t \in Ter &::= x \mid tt \mid \lambda x{:}T.t \\
v \in Val &::= \lambda x{:}T.t \\
F \in Frm &::= [\,]\ t \mid v\ [\,] \\
k \in Stk &::= nil \mid F :: k
\end{aligned}$$

Reduction

$$\begin{aligned}
(t\,t', k) &\mapsto (t, ([\,]\ t')::fs) \\
(v, ([\,]\ t)::fs) &\mapsto (t, (v\ [\,])::fs) \\
(v, ((\lambda x{:}T.t)\ [\,])::fs) &\mapsto ([x := v]t, fs)
\end{aligned}$$

Typing (in addition to the usual inference rules for terms)

$$\frac{}{\Gamma \vdash nil : T} \qquad \frac{\Gamma \vdash k : T' \quad \Gamma \vdash F : T \multimap T'}{\Gamma \vdash F :: k : T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash [\,]\ t : (T \to T') \multimap T'} \qquad \frac{\Gamma \vdash v : T \to T'}{\Gamma \vdash v\ [\,] : T}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash k : T}{\Gamma \vdash (t, k) : ok}$$

Figure 5.5: An abstract machine for lambda calculus

- For '⇐', define the 'dismantling' $k \bullet t$ of a stack k onto a term t inductively by $nil \bullet t \ := \ t$ and $(F{::}k) \bullet t \ := \ k \bullet F[t]$. Then prove (1) that $t \to t'$ implies $\forall k\colon\ k \bullet t \to k \bullet t'$, and (2) that $\forall k\colon\ (t, k) \mapsto (t', k')$ then $k \bullet t \to \ldots \to k' \bullet t'$. You can then use the coincidence result between small-step and big-step semantics to infer the proposition.

  There are typing rules for all syntactic categories:
- $\Gamma \vdash t : T$ for terms (which is standard and omitted in Figure 5.5),
- $\Gamma \vdash F : T \multimap T'$ for frames,
- $\Gamma \vdash k : T$ for stacks, and
- $\Gamma \vdash (t, k) : ok$ for states of the machine.

For terms, T describes the type result type whereas for stacks, T describes the type of the 'hole' of the evaluation context represented by the stack. The result type of the stack is not needed for our purposes; intuitively because nothing is returned after passing control to the stack. Correspondingly a state (t, k) is well-typed, written $\Gamma \vdash (t, k) : ok$ whenever both $\Gamma \vdash t : T$ and $\Gamma \vdash k : T$ match, for some type T. The result type of the overall computation is not important, because it is never plugged into another context.

As for the other semantics of simply typed lambda calculus, we can infer type safety from preservation and progress properties:

Theorem 5.4.11 (Type safety)

1. If $\Gamma \vdash (t, k) : ok$ and $(t, k) \mapsto (t', k')$ then $\Gamma \vdash (t, k) : ok$.
2. If $\emptyset \vdash (t, k) : ok$ and $(t, k) \not\mapsto$ then (t, k) is a final state.

It is easy to extend the machine with the abort primitive *error* considered earlier: The set of terms is now $t ::= \ldots \mid \mathit{error}$, and there is a single new reduction

$$(\mathit{error}, F :: k) \;\mapsto\; (\mathit{error}, k)$$

that propagates *error* by popping frames off the stack. Alternatively, the stack may be discarded at once:

$$(\mathit{error}, F :: k) \;\mapsto\; (\mathit{error}, \mathit{nil})$$

Note that without the restriction to non-empty stacks on the lhs, reduction on error-states would not be terminating.

Exercise 5.4.12 Extend the abstract machine to error handling.

Exercise 5.4.13 (Abstract machine for PCF and control)

a) Simulate (using pencil and paper) the abstract machine on the terms
   - $(\lambda xy.y)(\lambda xy.x)$ and
   - $(\lambda xy.y)(\mathit{callcc}(\lambda k.\mathit{throw}(\lambda xy.x, k)))$.

b) We implement terms and frames of the pure λ calculus in SML as follows:

```
type var = int
datatype ter = V of var | A of ter * ter | L of var * ter
and      frm = AppL of ter | AppR of ter   (* [] t and v [], resp *)
```

   Write a procedure $\mathit{run} : \mathit{ter} * \mathit{frm\ list} \to \mathit{ter} * \mathit{frm\ list}$ that implements reduction in the abstract machine for λ calculus.

c) Extend your implementation from (b) to PCF.

d) Extend your implementation with *error* and error handling.

Many control constructs can be expressed in terms of **continuations** (in particular, jumps, exceptions, backtracking, coroutines). Roughly, a continuation is a 'reified' control stack, i.e., a representation of the control stack can be used as an ordinary value. In particular, it can be

- stored and passed around,
- duplicated or discarded, and
- restored.

As Harper puts it, they provide a means of "unlimited time travel". To look at the semantics of continuations, we'll extend the abstract machine model by reified stacks *cont* k, a new operator *callcc* t to bind the current continuation in the term t, and a construct $\mathit{throw}(t_1, t_2)$ to restore the continuation represented by t_2, with (the value of) t_1:

$$\begin{aligned} v \in \mathit{Val} &::= \ \dots \ \mid \ \mathit{cont}\ k \\ t \in \mathit{Ter} &::= \ \dots \ \mid \ \mathit{callcc}\ t \ \mid \ \mathit{throw}(t_1, t_2) \\ F \in \mathit{Frm} &::= \ \dots \ \mid \ \mathit{callcc}\ [\,] \ \mid \ \mathit{throw}([\,], t) \ \mid \ \mathit{throw}(v, [\,]) \end{aligned}$$

There are 3 new rules that simply coordinate the evaluation of the subterms of *callcc* and $\mathit{throw}(t_1, t_2)$:

$$\begin{aligned} (\mathit{callcc}\ t, k) &\mapsto (t, \mathit{callcc}\ [\,]::k) \\ (\mathit{throw}(t, t'), k) &\mapsto (t, \mathit{throw}([\,], t')::k) \\ (v, \mathit{throw}([\,], t)::k) &\mapsto (t, \mathit{throw}(v, [\,])::k) \end{aligned}$$

The rule

$$(\mathit{cont}\ k, \mathit{throw}(v, [\,])::k') \mapsto (v, k)$$

replaces the current stack k' by the continuation k. The rule for *callcc*

$$(\lambda x{:}T.t, (\mathit{callcc}\ [\,])::k) \mapsto ([x := \mathit{cont}\ k]t, k)$$

binds the variable x to the continuation k that is currently in use.

Example 5.4.14

1. The term $t = \mathit{callcc}(\lambda h.(\lambda x.y)(\mathit{throw}(\lambda z.z, h)))$ yields $\lambda z.z$.
2. The term $\lambda x.\mathit{throw}(x, \mathit{cont}\ \mathit{nil})$ behaves like a value-carrying version of *error*. □

The set of types is extended with a type of 'T-continuations':

$$T \in \mathit{Ty} ::= \ \dots \ \mid \ \mathit{Cont}\ T$$

$$\frac{\Gamma \vdash k : T}{\Gamma \vdash \mathit{cont}\ k : \mathit{Cont}\ T} \qquad \frac{\Gamma \vdash t : (\mathit{Cont}\ T) \to T}{\Gamma \vdash \mathit{callcc}\ t : T}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : \mathit{Cont}\ T}{\Gamma \vdash \mathit{throw}(t_1, t_2) : T'}$$

$$\frac{}{\Gamma \vdash \mathit{callcc}\ [\,] : ((\mathit{Cont}\ T) \to T) \multimap T} \qquad \frac{\Gamma \vdash t : \mathit{Cont}\ T}{\Gamma \vdash \mathit{throw}([\,], t) : T \multimap T'}$$

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \mathit{throw}(v, [\,]) : \mathit{Cont}\ T \multimap T'}$$

Figure 5.6: Typing rules for control

They will stand for the continuations that correspond to contexts where values of type T may be plugged into the hole. Figure 5.6 contains the typing rules for the new terms and frames.

Continuations and their typing have interesting connections to classical logic, via the Curry-Howard isomorphism. Indeed, with the intuition that unlike with function calls, when reinstating a continuation via *throw* no value is returned, one may identify the type *Cont* T with $T \to 0$. Sometimes, for emphasis, *Cont* T is also written $\neg T$. One then sees that the type of *callcc* is Peirces Law! A tight connection to classical logic can be obtained by also introducing the type 0 and considering an 'abort' operator $\mathcal{A}\ t$ that has arbitrary type T, whenever $\Gamma \vdash t : 0$. It can be seen as a proof of the fact that from false we can conclude anything. Its operational behaviour is given by

$$(v, (\mathcal{A}[\,])::k) \ \mapsto\ (v, \mathit{nil})$$

so that it behaves like the second term from Example 5.4.14 (with argument type restricted to 0). (Closed) terms that do not contain stacks can then be viewed as proof terms for an inference system of propositional logic.

Exercise 5.4.15 (Control operators and Curry-Howard correspondence) Use the lambda calculus with *callcc* to obtain proof terms for the following propositions, where for emphasis we write $\neg T$ for *Cont* T:

a) $(\neg T \to T) \to T$

b) $(\neg T \to \neg T') \to T' \to T$

c) $(T \to T') \to \neg T' \to \neg T$

d) $0 \to T$

e) $\neg(\neg T) \to T$

6 Polymorphic Lambda Calculus

Let **S** be the simply typed lambda calculus introduced in Section 4.1. We have considered several extensions of S:

$$\begin{aligned}\text{PCF} &= S(\mathit{bool}, \mathit{nat}, \mathit{fix})\\ \text{FPC} &= S(1, \times, +, \mu)\\ \text{ND} &= S(1, \times, +, 0, \delta)\end{aligned}$$

PCF and FPC are Turing-complete programming languages and ND is a complete proof system for propositional logic. ND is terminating, PCF and FPC are not. We will now extend S with procedures that take types as arguments. Such procedures are called **polymorphic**. The resulting calculus is known as **polymorphic lambda calculus** and was invented by the French logician Jean-Yves Girard [1972]. Girard's goal was the extension of the Curry-Howard correspondence to logics with quantifiers. The polymorphic lambda calculus was reinvented by the American computer scientist John Reynolds [1974] who wanted to explain polymorphic procedures in programming languages. Polymorphic lambda calculus contributes to the foundational theory of ML. When work on ML started in 1974, the connection with the polymorphic lambda calculus was not known.

6.1 System F

We start with the first-order polymorphic lambda calculus, which is known as **System F**. Procedures taking types as arguments have types of the form $\forall X.S$ where the type variable X is bound. If a procedure of type $\forall X.S$ is applied to a type T, it yields a result of type S^X_T, where S^X_T is obtained from S by capture-free replacement of X with T.

Figure 6.1 shows the definition of System F. The notation s^x_t stands for $[x := t]s$. The constraint $t \sim_\alpha t'$ of Rule Equiv says that the terms t and t' are equal up to α-renaming of bound type variables. Likewise, $T \sim_\alpha T'$ of Rule Equiv says that the types T and T' are equal up to α-renaming of bound type variables. Here are examples of terms and types:

$$\begin{aligned}\mathit{id} &:= \lambda X.\lambda x{:}X.\,x && : \forall X.\, X \to X\\ \mathit{id}' &:= \mathit{id}(X\to X)(\mathit{id}\ X) && : X \to X\end{aligned}$$

X, Y	$::= \mathbb{N}$	**type variable**
S, T	$::= X \mid T \to T \mid \forall X.T$	**type**
x, y	$::= \mathbb{N}$	**term variable**
s, t	$::= x \mid tt \mid \lambda x{:}T.t \mid \lambda X.t \mid tT$	**term**
Γ	is function $x \mapsto T$	**type environment**

$$(\lambda x.s)t \to_0 s^x_t$$
$$(\lambda X.s)T \to_0 s^X_T$$

$$\frac{}{\Gamma \vdash x : T}\ \Gamma x = T$$

$$\frac{\Gamma \vdash t_1 : S \to T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\, t_2 : T} \qquad \frac{\Gamma[x := S] \vdash t : T}{\Gamma \vdash \lambda x{:}S.t : S \to T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \lambda X.t : \forall X.T}\ X \text{ not free in } \Gamma \qquad \frac{\Gamma \vdash s : \forall X.S}{\Gamma \vdash sT : S^X_T}$$

$$\textbf{Equiv}\quad \frac{\Gamma \vdash t : T}{\Gamma \vdash t' : T'}\ t \sim_\alpha t' \wedge T \sim_\alpha T'$$

Figure 6.1: System F

$$\begin{aligned} twice &:= \lambda X.\lambda f{:}X{\to}X.\lambda x{:}X.f(fx) &&: \forall X.\,(X \to X) \to (X \to X) \\ quad &:= \lambda X.\lambda f{:}X{\to}X.\,twice\,(X{\to}X)(twice\,(X{\to}X)f) &&: \forall X.\,(X \to X) \to (X \to X) \end{aligned}$$

Obviously, F is an extension of S. Although F is much more expressive than S, it still has all the essential properties of S:

- unique types (up to alpha equivalence)
- type preservation
- progress
- termination

F can type the Church encodings of the types 1, *bool* and*nat* as well as the encodings of products and sums of types. In other words, the respective types can be expressed in F. Figure 6.2 shows how this is done. The single value $()$ of 1 can be expressed as follows: $() = \lambda X.\lambda x{:}X.x$. We leave the expression of the other values as exercises. If you need help, you may consult [Pierce].

$$\begin{aligned}
1 &:= \forall X.\ X \to X \\
bool &:= \forall X.\ X \to X \to X \\
nat &:= \forall X.\ (X \to X) \to X \to X \\
S \times T &:= \forall X.\ (S \to T \to X) \to X && \text{where } X \text{ not free in } S \to T \\
S + T &:= \forall X.\ (S \to X) \to (T \to X) \to X && \text{where } X \text{ not free in } S \to T
\end{aligned}$$

Figure 6.2: Basic types and basic type operations in F

Exercise 6.1.1 (Booleans) Express the following values in F:

$$\begin{aligned}
false &: bool \\
true &: bool \\
if &: \forall X\ Y.\ bool \to (1 \to X) \to (1 \to X) \to X
\end{aligned}$$

Exercise 6.1.2 (Natural Numbers) Express the following values in F:

$$\begin{aligned}
0 &: nat \\
succ &: nat \to nat \\
iter &: \forall X.\ nat \to (X \to X) \to X \to X \\
plus &: nat \to nat \to nat \\
power &: nat \to nat \to nat
\end{aligned}$$

Exercise 6.1.3 (Lists) Express lists in F. First find a type construction *list* S and then express the following operations:

$$\begin{aligned}
nil &: \forall X.\ list\ X \\
cons &: \forall X.\ X \to list\ X \to list\ X \\
foldl &: \forall X\ Y.\ (X \to Y \to Y) \to Y \to list\ X \to Y \\
length &: \forall X.\ list\ X \to nat
\end{aligned}$$

Exercise 6.1.4 (Products) Express the following values in F:

$$\begin{aligned}
pair &: \forall X\ Y.\ X \to Y \to X \times Y \\
fst &: \forall X\ Y.\ X \times Y \to X \\
snd &: \forall X\ Y.\ X \times Y \to Y
\end{aligned}$$

Exercise 6.1.5 (Sums) Express the following values in F:

$$\begin{aligned} \mathit{inl} &: \forall X\,Y.\,X \to X + Y \\ \mathit{inr} &: \forall X\,Y.\,Y \to X + Y \\ \mathit{case} &: \forall X\,Y\,Z.\,X + Y \to (X \to Z) \to (Y \to Z) \to Z \end{aligned}$$

Exercise 6.1.6 (Freshness Constraint) Consider the closed term

$$t \;=\; \lambda X.\,\lambda x{:}X.\;\lambda X.\,\lambda f{:}X{\to}X.\;fx$$

a) Convince yourself that t is ill-typed.

b) Convince yourself that the term becomes well-typed if the freshness constraint "Y not free in Γ or t" in the typing rule for polymorphic procedures is dropped.

c) Convince yourself that t *bool true nat succ* $\to^*$ *succ true*. Why does this reduction show that t must be ill-typed?

Exercise 6.1.7 (Equiv) Explain why the rule Equiv is needed to derive the well-typedness of the term $\lambda X.\,\lambda x{:}X.\lambda X.\,\lambda f{:}X{\to}X.\;\lambda x{:}X.\,fx$.

6.2 Curry-Howard Correspondence

We discussed the Curry-Howard correspondence for the simply-typed λ-calculus ND in Section 4.6. The correspondence also holds for F. A polymorphic type $\forall X.S$ represents a universally quantified propositional formula where X ranges of the two truth values 0 and 1. We have allready seen that F can express 1 (true), $\times$ (conjunction) and $+$ (disjunction). For false we take

$$0 \;:=\; \forall X.X$$

As before one can show that $\emptyset \vdash t : T$ implies that T is valid. Hence F is a proof system for quantified propositional formulas. Due to the De Morgan dualism F can also express existential quantification:

$$\exists X.S \;:=\; \neg\forall X.\neg S$$

With the current set-up, F cannot prove all valid formulas. It can only prove the so-called **intuitionistic fragment**. However, if we assume that $\neg\neg X \to X$ is valid, which intuitionisticly it is not, F can prove all (classically) valid formulas. A term t such that $\emptyset \vdash t : T$ is called an **intuitionistic proof** of T, and a term t such that $\delta : \forall X.\neg\neg X \to X \vdash t : T$ is called an **classic proof** of T.

Theorem 6.2.1 A type of F is valid as a formula if and only if it has a classic proof in F.

The difference between classical and intuitionistic proofs can be seen as follows. There is a straightforward intuitionistic proof for $(S + T) \to (T + S)$. Disjunction can also be expressed as $S \vee T \;:=\; \overline{S} \to T$. We now observe that $(S \vee T) \to (T \vee S)$ has a classic proof but no intuitionistic proof. So the two codings of disjunction, $S + T$ and $S \vee T$, lead to radically different proofs. While $S + T$ is an intuitionistic coding of disjunction, $S \vee T$ is a classical coding. The built-in sums of ND are intuitionistic.

F is a very expressive system. As is it provides a complete proof system for quantified propositional formulas. This is in sharp contrast to the simply typed λ-calculus, where even for unquantified propositional formulas numerous extensions are needed (Calculus ND in Figure 4.4).

Exercise 6.2.2 (Intuitionistic Proofs) Find intuitionistic proofs for the following formulas:

a) $0 \to X$

b) $(\forall X.S) \to S^X_T$

c) $S^X_T \to \overline{\forall X.\overline{S}}$

d) $(S + T) \to (T + S)$

e) $((X \to Y) \times X) \to Y$

f) $(X + Y) \to \overline{\overline{X} \times \overline{Y}}$

g) $(X \times Y) \to \overline{\overline{X} + \overline{Y}}$

h) $\overline{X \times \overline{X}}$

i) $\overline{X + Y} \to (\overline{X} \times \overline{Y})$

j) $(\overline{X} \times \overline{Y}) \to \overline{X + Y}$

Exercise 6.2.3 (Classic Proofs) Find classic proofs for the following formulas:

a) $\forall X.\, X + \overline{X}$

b) $(\overline{X} \to \overline{Y}) \to (Y \to X)$

c) $((X \to Y) \to X) \to X$

6.3 System F_ω

Consider the type constructor *list*: Given a type T it returns a type *list* T. Thus *list* may be modelled as a function from types to types. If we take the idea of type functions serious, a type sytem for types is needed. Let's call the types of types **kinds**. We shall use the following kinds:

$$K ::= * \mid K \to K$$

The kind $*$ represents the ordinary types we are already used to. Kinds of the form $K \to K$ represent type functions. For instance, the type functions that yield list types and product types have the following kinds:

$$\begin{aligned} \mathit{list} &: * \to * \\ \mathit{product} &: * \to * \to * \end{aligned}$$

It is common to call everything that has a kind a type. Types of kind $*$ are called **proper**. We call a type **functional** if it is has the form $S \to T$ or if it has a functional kind. Note that our definition is such that there are proper and non-proper functional types.

The **higher-order polymorphic lambda calculus** F_ω is obtained from F by admitting type functions. There are three consequences:

- Binding occurences of type variables must be qualified with a kind.
- The syntax for types must provide for application and abstraction of types.
- There is a non-trivial equivalence of types. For instance, $(\lambda X{:}K.\,X)X$ and X are equivalent types.

There are several possibilities for the definition of type equivalence. We use $T \sim_{\alpha\beta} T'$, which holds if and only if T and T' have the same β-normal form up to α-equivalence. The **β-normal form** of a type is obtained by applying the β-rule $(\lambda X.S)T \to S^X_T$ as long as it is applicable and wherever it is applicable. One can prove that this process terminates and yields a result that is unique up to α-equivalence.

The definition of F_ω appears in Figure 6.3. We write $\lambda X.t$, $\lambda X.T$ and $\forall X.T$ as abbreviations for the terms $\lambda X{:}*.\,t$, $\lambda X{:}*.\,T$ and $\forall X{:}*.T$.

Exercise 6.3.1 (Well-Formed Environments) F has environments that map type variables to kinds and term variables to types. An environment is **well-formed** if the types of the term variables are well-kinded with respect to the kinds of the type variables. Give a formal definition of well-formed finite environments. Use recursion on the size of the environment (number of variables introduced).

Exercise 6.3.2 (Polymorphic Lists) Express the following objects in F_ω:

$$\begin{aligned} \mathit{list} &: * \to * \\ \mathit{nil} &: \forall X.\ \mathit{list}\ X \\ \mathit{cons} &: \forall X.\ X \to \mathit{list}\ X \to \mathit{list}\ X \\ \mathit{foldl} &: \forall X\ Y.\ (X \to Y \to Y) \to Y \to \mathit{list}\ X \to Y \\ \mathit{length} &: \forall X.\ \mathit{list}\ X \to \mathit{nat} \end{aligned}$$

K	$::= \ * \mid K \to K$	**kind**
X, Y	$::= \ \mathbb{N}$	**type variable**
S, T	$::= \ X \mid T \to T \mid \forall X{:}K.T \mid T\,T \mid \lambda X{:}K.\,T$	**type**
x, y	$::= \ \mathbb{N}$	**term variable**
s, t	$::= \ x \mid t\,t \mid \lambda x{:}T.t \mid \lambda X{:}K.t \mid tT$	**term**
Γ	is a function $X \mapsto K$ and $x \mapsto T$	**environment**

$$(\lambda x.s)t \to_0 s^x_t$$
$$(\lambda X.s)T \to_0 s^X_T$$

$$\frac{\Gamma \vdash T : *}{\Gamma \vdash x : T}\ \Gamma x = T \qquad \frac{\Gamma \vdash t_1 : S \to T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\, t_2 : T}$$

$$\frac{\Gamma \vdash S : * \quad \Gamma[x{:=}S] \vdash t : T}{\Gamma \vdash \lambda x{:}S.\,t : S \to T} \qquad \frac{\Gamma \vdash s : \forall X{:}K.S \quad \Gamma \vdash T : K}{\Gamma \vdash sT : S^X_T}$$

$$\frac{\Gamma[X{:=}K] \vdash t : T}{\Gamma \vdash \lambda X{:}K.t : \forall X{:}K.T}\ X \notin Dom\,\Gamma$$

$$\textbf{Equiv}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash t' : T'}\ t \sim_\alpha t' \ \wedge\ T \sim_{\alpha\beta} T' \ \wedge\ \Gamma \vdash T' : *$$

$$\frac{}{\Gamma \vdash X : K}\ \Gamma X = K \qquad \frac{\Gamma \vdash s : * \quad \Gamma \vdash T : *}{S \to T : *}$$

$$\frac{\Gamma[X{:=}K] \vdash T : *}{\Gamma \vdash \forall X{:}K.T : *}\ X \notin Dom\,\Gamma$$

$$\frac{\Gamma \vdash S : K' \to K \quad \Gamma \vdash T : K'}{\Gamma \vdash S\,T : K} \qquad \frac{\Gamma[X{:=}K] \vdash T : K'}{\Gamma \vdash \lambda X{:}K.\,T : K \to K'}\ X \notin Dom\,\Gamma$$

Figure 6.3: System F$_\omega$

6.4 Abstract Data Types

One of the cornerstones of Software Engineering is the notion of an **abstract data type (ADT)**. One distinguishes between **implementations** and **users** of an ADT, where the interface between the two is specified through a **signature**. As example we take an ADT NAT providing natural numbers. The signature of NAT specifies four objects:

$$\begin{aligned} Nat &: * \\ 0 &: Nat \\ S &: Nat \to Nat \\ iter &: \forall X.\, Nat \to (X \to X) \to (X \to X) \end{aligned}$$

From the signature it is clear what implementations of NAT have to provide and what users of NAT can expect. Type checking ensures that users can access an implementation only to the extend that is licensed by the signature, and that implementations provide the required objects in the form announced by the signature.

We have used the kinds and types of F_ω for the specification of the objects of NAT. We shall now see that F_ω can express the entire signature as a type function and users and implementations as procedures. The basic idea is to express users of an ADT as procedures that take the objects declared by the signature of the ADT as arguments. With this idea we can model a signature as a type function

$$Sig : * \to *$$

that for a type T yields the type of a user that yields a result of type T. For instance, the signature of NAT is represented as the following type function:

$$\lambda Z.\, \forall N.\, N \to (N \to N) \to (\forall X.\, N \to (X \to X) \to (X \to X)) \to Z$$

An implementation of an ADT is expressed as a procedure that takes a user (i.e., a procedure) as argument and applies it to the implementations of the objects declared by the signature:

$$impl \;:\; \forall Z.\, Sig\, Z \to Z$$

For instance, an implementation $impl_0$ of NAT can be obtained as follows:

$$\begin{aligned}
\mathit{nat}_0 &:= \forall X.\,(X \to X) \to X \to X \\
\mathit{impl}_0 &:= \lambda Z.\,\lambda\, \mathit{user} : \mathit{Sig}\ Z. \\
&\quad\ \mathit{user}\ \mathit{nat}_0 \\
&\qquad\quad (\lambda X.\,\lambda f{:}X{\to}X.\ \lambda x{:}X.\ x) \\
&\qquad\quad (\lambda n{:}\mathit{nat}_0.\,\lambda X.\,\lambda f{:}X{\to}X.\ \lambda x{:}X.\ nXf(fx)) \\
&\qquad\quad (\lambda X.\,\lambda n{:}\mathit{nat}_0.\ nX)
\end{aligned}$$

In summary, we have

$$\begin{aligned}
\mathit{Sig} &: * \to * \\
\mathit{user} &: \mathit{Sig}\ T \\
\mathit{impl} &: \forall Z.\ \mathit{Sig}\ Z \to Z \\
\mathit{code} &= \mathit{impl}\ T\ \mathit{user}
\end{aligned}$$

where *code* is a term that connects an implementation *impl* with a user *user*.

Exercise 6.4.1 (Lists over nat) Consider an ADT that implements lists over *nat*:

$$\begin{aligned}
\mathit{list} &: * \\
\mathit{nil} &: \mathit{list} \\
\mathit{cons} &: \mathit{nat} \to \mathit{list} \to \mathit{list} \\
\mathit{foldl} &: \mathit{list} \to \forall Z.\,(\mathit{nat} \to Z \to Z) \to Z \to Z
\end{aligned}$$

a) Represent the signature of the ADT as a type function.

b) Write a procedure *impl* that implements the ADT. Use the objects from exercise 6.3.2.

Exercise 6.4.2 (Polymorphic Lists) Consider an ADT that implements polymorphic lists as in Exercise 6.3.2 (omit *length*).

a) Represent the signature of the ADT as a type function.

b) Write a procedure *impl* that implements the ADT. Use the objects from exercise 6.3.2.

7 A Calculus of Primitive Objects

In this chapter we will take a closer look at the semantic foundations of object-oriented languages. Their combination of recursion (both on the level of terms and types), subtyping, polymorphism, and internal state draws on many of the ideas that we've seen already.

There are two alternatives to proceed:

1. explain object-oriented features by *translation* into sufficiently rich λ-calculi, which we already understand and where a set of tools is available (type safety, methods for establishing contextual equivalences,...)
2. develop a calculus where objects are *primitive*, and study object-oriented features more directly.

Since the known encodings of objects into λ-calculi are fairly intricate, we'll follow the second alternative. This has the additional benefit that we can see how most of the *techniques* that we developed for λ-calculi carry over to similar formal systems.

The article *A Theory of Primitive Objects (Untyped and First-order Systems)* by M. Abadi and L. Cardelli, is recommended for further reading. It is available from http://lucacardelli.name/Papers/PrimObj1stOrder.A4.pdf.

7.1 Untyped Sigma Calculus

The calculus we consider is called **ς-calculus**, and was introduced by Martín Abadi and Luca Cardelli [1994]. It takes objects as primitive, and directly supports invocation and overriding of methods. Syntactically it is very simple: the terms are given by the grammar

$x, y ::= \mathbb{N}$	**term variables**
$a, b ::= x$	**terms**
$\mid\ [l_i = \varsigma(x_i)b_i{}^{i=1\dots n}]$	**object**
$\mid\ a.l$	**method invocation/field selection**
$\mid\ a.l \Leftarrow \varsigma(y)b$	**method override/field update**

An object $[l_i = \varsigma(x_i)b_i{}^{i=1\dots n}]$ is a collection of methods $\varsigma(x_i)b_i$, with associated names l_i. The variable x_i is bound in the **method body** b_i by the **self binder**

$$v ::= [l_i = \varsigma(x_i)b_i{}^{i=1\ldots n}] \qquad \textbf{values}$$
$$E ::= [\,] \mid E.l \mid E.l \Leftarrow \varsigma(y)b \qquad \textbf{evaluation contexts}$$

$$\begin{array}{rcl} a.l_j & \to_0 & [x_j := a]b_j \\ & & \text{where } a = [l_i = \varsigma(x_i)b_i{}^{i=1\ldots n}] \text{ and } 1 \leq j \leq n \\ a.l_j \Leftarrow \varsigma(y)b & \to_0 & [l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i{}^{i=1\ldots n \mid i \neq j}] \\ & & \text{where } a = [l_i = \varsigma(x_i)b_i{}^{i=1\ldots n}] \text{ and } 1 \leq j \leq n \end{array}$$

Figure 7.1: Operational semantics of ς-calculus

$\varsigma(\cdot)$. The intended meaning is that, dynamically, x_i refers to the host object of the method. It models the **self parameter** that is found in realistic object-oriented languages, where it is usually called **this** or **self**.

Since $\varsigma(\cdot)$ is a binder, the same issues about α-renaming apply as for λ-bound variables in λ-calculus. It may be a useful exercise to give formal definitions of the set $\mathrm{FV}(a)$ of free variables of a, and of the capture-avoiding substitution $[x := a]b$ of a for x in b. (Note that Abadi and Cardelli use the notation $b\{x \leftarrow a\}$ in their article.)

The operational semantics of method invocation makes this dependence of methods on the host object explicit. The self parameter is substituted by the complete host object: if a is $[l_i = \varsigma(x_i)b_i{}^{i=1\ldots n}]$ and $1 \leq j \leq n$, then

$$a.l_j \to [x_j := a]b_j$$

Method override simply replaces an old by a new method:

$$a.l_j \Leftarrow \varsigma(y)b \to [l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i{}^{i=1\ldots n \mid i \neq j}]$$

There are 2 points to note here: firstly, method override assumes that a method named l_j exists already, so objects are not *extensible*. Secondly, method override has a *functional* interpretation, i.e., it creates a new object.

The operational semantics is given formally in Fig. 7.1. Objects are the only values, and the proper reduction rules are the two rules for invocation and update described above. Our choice of evaluation contexts then gives rise to a deterministic reduction relation $\to$. As usual, we do not permit reductions under binders.

Notation. We may view **fields** as methods that do not use their self parameter. It is sometimes convenient to use the notation $[l = b, \ldots]$ and $a.l := b$ to denote the field l containing b, and field update of l by b, resp. Keep in mind that this is just shorthand for $[l = \varsigma(x)b, \ldots]$ and $a.l \Leftarrow \varsigma(x)b$ where $x \notin \mathrm{FV}(b)$.

Exercise 7.1.1 (Big-step operational semantics) Give an equivalent big-step semantics.

7.2 Examples

Despite being small, the ς-calculus is very expressive. For instance, by making the self parameter available in method bodies one has a means of implementing (mutual) recursion. That is, a method may invoke any of its sibling methods, as well as call itself:

$$[l = \varsigma(x)x.l]$$

Note that this expression is a value, but upon invoking its l-method it will result in a diverging computation.

We will discuss some further examples below.

Objects with backup Consider objects of the form

$$[\mathsf{retrieve} = \varsigma(x)x, \mathsf{backup} = \varsigma(x)x.\mathsf{retrieve} \Leftarrow \varsigma(y)x, \ldots]$$

The object allows us to store the object in its current form, by calling backup, and afterwards to restore this version, by calling retrieve. In fact, this can be iterated: we may store a whole trail, and then move back to previous versions by repeated calls to retrieve. The technique relies on two features of the ς-calculus: the ability to return self (i.e., the use of x in the retrieve methods), and the ability to have several self parameters in scope (i.e., the use of both x and y in the backup method).

Booleans The idea behind many encoding in object calculus is to treat data as "active", similar to the Church encodings of various datatypes in untyped lambda calculus. For instance, to represent booleans, the constants *true* and *false* will take the decision of which branch to use when placed inside a conditional:

$$\begin{array}{rcl} \mathit{true} & := & [\mathsf{then} = \varsigma(x)x.\mathsf{then}, \mathsf{else} = \varsigma(x)x.\mathsf{else}, \mathsf{val} = \varsigma(x)x.\mathsf{then}] \\ \mathit{false} & := & [\mathsf{then} = \varsigma(x)x.\mathsf{then}, \mathsf{else} = \varsigma(x)x.\mathsf{else}, \mathsf{val} = \varsigma(x)x.\mathsf{else}] \\ \mathit{if}\ t\ \mathit{then}\ a\ \mathit{else}\ b & := & ((t.\mathsf{then} := a).\mathsf{else} := b).\mathsf{val} \end{array}$$

The method $\mathsf{then} = \varsigma(x)x.\mathsf{then}$ (and similarly else) is used simply as a way to initialize the object. Any method body would have done, but the choice above has the advantage of integrating smoothly with types (see Exercise 7.3.2 below).

Natural numbers We will represent natural numbers as objects that support 3 operations: *iszero*, *pred*, and *succ*. Here is the representation of 0 (using the notation for fields, and booleans as introduced in the previous example):

$$zero := [\mathsf{iszero} = true, \mathsf{prev} = \varsigma(x)x, \mathsf{succ} = \varsigma(x)(x.\mathsf{iszero} := false).\mathsf{pred} := x]$$

Untyped lambda calculus We restricted the calculus to methods that take self as the *only* argument. To see that this is no restriction, we show how lambda calculus can be expressed inside the ς-calculus – methods with additional parameters can then be written as $\varsigma(x)\lambda x_1 \dots \lambda x_n.a$. The encoding is given as a translation $(\cdot)^*$ that maps untyped lambda terms to object calculus terms:

$$\begin{aligned} x^* &:= x \\ (t_1 t_2)^* &:= (t_1^*.\mathsf{arg} := t_2^*).\mathsf{val} \\ (\lambda x.t)^* &:= [\mathsf{arg} = \varsigma(x)x.\mathsf{arg}, \mathsf{val} = \varsigma(x)([x := x.\mathsf{arg}]t^*)] \end{aligned}$$

The basic idea is to use an additional field to pass the argument. The (translated) function body can access this argument through self.

Exercise 7.2.1 (Beta reduction on values) Verify that $((\lambda x.x)y)^* \to \dots \to y^*$.

Exercise 7.2.2 (Untyped natural numbers objects)
Extend the object-oriented natural numbers by

a) a method $\mathsf{add} = \varsigma(x)\lambda(n)\dots$ that adds another number to the number represented by the object;

b) a method $\mathsf{mult} = \varsigma(x)\lambda(n)\dots$ that multiplies another number to the number represented by the object;

c) a method $\mathsf{fac} = \varsigma(x)\dots$ that returns the factorial of the number represented by the object.

Feel free to use booleans and lambda notation as shorthand whenever necessary.

7.3 Simply Typed Sigma Calculus

In the basic system, there is just one way of forming types:

$$A, B ::= [l_i : A_i{}^{i=1\dots n}] \qquad \textbf{object type}$$

An object type $[l_i : A_i{}^{i=1\dots n}]$ describes the set of objects that provide methods named l_i that return values of type A_i, resp., for all $i = 1, \dots, n$. The syntax of terms is changed slightly: in order to obtain the unique typing property for

$$\frac{(x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\forall i = 1, \ldots, n:\ \Gamma[x_i := A] \vdash b_i : A_i}{\Gamma \vdash [l_i = \varsigma(x_i{:}A) b_i{}^{i=1\ldots n}] : A} \quad (A = [l_i : A_i{}^{i=1\ldots n}])$$

$$\frac{\Gamma \vdash a : [l_i : A_i{}^{i=1\ldots n}]}{\Gamma \vdash a.l_j : A_j} \quad (1 \leq j \leq n)$$

$$\frac{\Gamma \vdash a : A \quad \Gamma[x := A] \vdash b : A_j}{\Gamma \vdash a.l_j \Leftarrow \varsigma(x{:}A) b : A} \quad (A = [l_i : A_i{}^{i=1\ldots n}] \ \wedge \ 1 \leq j \leq n)$$

Figure 7.2: Simple types for ς-calculus

$$\frac{}{[l_i : A_i{}^{i=1\ldots n+k}] <: [l_i : A_i{}^{i=1\ldots n}]}$$

$$\frac{}{A <: A}$$

$$\frac{A <: C \quad C <: B}{A <: B}$$

$$\frac{\Gamma \vdash a : A \quad A <: B}{\Gamma \vdash a : B}$$

Figure 7.3: Subtyping for ς-calculus

ς-calculus, binding occurrences of each variable must be annotated with a type. (Recall that we did the same when moving from untyped to typed λ calculus.)

The typing rules are collected in Fig. 7.2, where Γ is a finite map from term variables to object types. The rules in Fig. 7.3 extend the system with subtyping. The subtype relation is generated by a single rule that allows us to "forget" the existence of methods in an object. Additionally, there are rules to ensure that $<:$ is transitive and reflexive.[1] As usual, subtyping is then connected to the typing relation through a subsumption rule.

It is possible to prove a substitution lemma analogous to the one for lambda calculus:

Proposition 7.3.1 (Substitution)

[1] One could also introduce a top type *Top* without causing problems.

$(\forall x \in FV\, a\colon\ \Gamma' \vdash \theta x : \Gamma x)\ \wedge\ \Gamma \vdash a : A\ \Longrightarrow\ \Gamma' \vdash \theta a : A$

From this, it is not difficult to show progress and preservation lemmas, resulting in a statement of type safety for ς-calculus.

Exercise 7.3.2 (Simple types)

a) Show that $\emptyset \vdash [l = \varsigma(x{:}A)x.l] : [l{:}A]$, for any type A.

b) Assume that the ς-calculus has been extended with a base type *int*, and let $P_1 := [\mathsf{x} : \mathit{int}]$ and $P_2 := [\mathsf{x} : \mathit{int}, \mathsf{y} : \mathit{int}]$ be the types of 1- and 2-dimensional points, resp. Show $\emptyset \vdash [\mathsf{x} = 1] : P_1$ and $\emptyset \vdash [\mathsf{x} = 1, \mathsf{y} = 1] : P_2$.

c) Assume that arithmetic operations $+$, $(\ )^2$ and *sqrt* are given, with their usual types. Convince yourself that, for $B := [\mathsf{p} : P_2, \mathsf{dist} : \mathit{int}]$,

$$\emptyset \vdash [\mathsf{p} = [\mathsf{x} = 1, \mathsf{y} = 1],\ \mathsf{dist} = \varsigma(s{:}B)\mathit{sqrt}((s.\mathsf{p}.\mathsf{x})^2 + (s.\mathsf{p}.\mathsf{y})^2)] : B$$

d) Use (b) and (c) to show that (covariant) subtyping in depth is not sound:

$$\frac{A_i <: B_i \quad \forall i = 1 \ldots n}{[l_i{:}A_i{}^{i=1\ldots n+k}] <: [l_i{:}B_i{}^{i=1\ldots n}]} \qquad \text{(wrong!!!)}$$

Exercise 7.3.3 (Encoding simply typed lambda calculus) Show that the encoding of lambda terms as objects carries over to the typed case, where $S \to T$ is translated to $(S \to T)^* = [\mathsf{arg} : S^*, \mathsf{val} : T^*]$. More precisely, prove that the encoding maps well-typed terms of the simply typed lambda calculus to well-typed terms of ς-calculus.

Is the statement also true for simply typed lambda calculus with subtyping?

7.4 Representing Classes and Inheritance