Computability via PCF
Lecture Notes

Peter Dybjer

3 februari 2004

1 PCF

PCF is essentially a sublanguage of Haskell and we will often use Haskell syntax to write PCF-programs. The main difference between our version of PCF and Plotkin's[1] original one is that ours has a polymorphic type system.

1.1 Types

The *PCF-types* are generated by the following context-free grammar:

```
<type>    ::= Bool | Nat | (<type> -> <type>) | <typevar>
<typevar> ::= a | b | c | ...
```

If we omit the rule for type variables we get the *monomorphic PCF-types.*

We may omit outer parentheses in type expressions and also use the convention that the function arrow associates to the right. Thus we may write

```
a -> b -> c
a -> b -> c -> d
...
```

as abbreviations of the "official" PCF-types

```
(a -> (b -> c))
(a -> (b -> (c -> d)))
...
```

which are the ones which can be generated by the grammar.

Examples of types are

```
Bool, Nat, a, Nat -> Nat, a -> Nat, (Bool -> Bool) -> Bool, ...
```

The base types `Bool` and `Nat` can be defined by the following datatype declarations in Haskell:

```
data Bool = True | False
data Nat  = Zero | Succ Nat
```

Note that `Bool` is defined in the standard prelude but that `Nat`, the type of *natural numbers* or *non-negative integers*, differs from the built-in datatype `Int` of Haskell. (Sometimes it may be convenient to use `Int` as a substitute for `Nat` when writing PCF-programs in Haskell. This is ok, provided one bears in mind that one may only use the non-negative integers and the functions programmable in PCF.)

[1]Gordon D. Plotkin, LCF considered as a programming language, *Theoretical Computer Science* 5, 1977, 223-255

1.2 Terms

The *PCF-terms* (or synonymously, the *PCF-expressions*) are generated by the following context-free grammar:

```
<term> ::= <var> | (<term> <term>) | (\<var> -> <term>) |
           True | False | Zero | Succ |
           if | pred | isZero | fix

<var>  ::= x | y | z | ...
```

Here we have used Haskell's version of λ-notation:

```
(\x -> e)
```

In later chapters we will mostly use the usual $\lambda x.e$.

We will also use the ordinary conventions for omitting outer parentheses, letting application associate to the left, and replacing several consecutive λ's by one (see KP[2] p 153).

The constants `True, False, Zero, Succ, if, pred, isZero, fix` are called *PCF-combinators.* The *constructors* `True, False, Zero, Succ` are provided directly by the definitions of `Bool` and `Nat`, whereas the *selectors* and *destructors* `if, pred, isZero` can be defined by

```
if True  d e = d
if False d e = e

pred Zero     = Zero
pred (Succ n) = n

isZero Zero     = True
isZero (Succ n) = False
```

(The reader who wishes to implement the PCF-programs in these notes in Haskell should be aware that there may be name clashes with programs from the standard prelude. It may also be convenient to use built-in constructs from Haskell such as the `if-then-else-` construct which gives a readable version of `if`.)

The *fixed point* combinator `fix`, which is used for implementing *general recursion*, is defined by

```
fix f = f (fix f)
```

The reason for its name is that it returns a *fixed point* of the function `f`, that is, a solution

```
x = fix f
```

to the equation

```
x = f x
```

Examples of PCF-terms are

```
Zero, Succ Zero, x, Succ x, pred y, fix pred, ...
```

but also ill-typed terms such as

```
Succ True, pred if, fix isZero, ...
```

The notions of *free* and *bound* variable occurrences are defined in KP 5.3. As usual, a *closed* term is one without free variable occurrences, whereas an *open* term may have such occurrences. A closed PCF-term is also called a *PCF-program.*

[2]Kent Petersson, *Beräkningsbarhet för dataloger: från λ till P*, Bokförlaget Aquila 1987

1.3 Abbreviations

When programming in PCF it is convenient to introduce abbreviations, such as the numerals

```
0 = Zero
1 = Succ 0
2 = Succ 1
...
```

and various other constants

```
id       = \x -> x
loop     = fix id
infinity = fix Succ
...
```

These abbreviations are always *explicit definitions* or *macros* of the form

```
f = e
```

where `f` is a new identifier and `e` is a PCF-expression possibly containing other abbreviations.

It is easy to obtain an official PCF-expression (as generated by the syntax) by successively replacing the LHS (the `f`) of each definition by the RHS (the `e`). This process is called *elimination of explicit definitions* or *macro expansion*. For example,

```
Succ 2
```

is a PCF-expression containing the abbreviation 2. By performing macro expansion we successively get the expressions

```
Succ (Succ 1)
Succ (Succ (Succ 0))
Succ (Succ (Succ Zero))
```

If we finally insert the outer parentheses, which we have omitted by convention, we get the official PCF-expression

```
(Succ (Succ (Succ Zero)))
```

generated by the grammar.

We may introduce abbreviations for types in a similar manner.

1.4 How to transform a Haskell program to a PCF program

The fact that all definitions are explicit (macros) is an important difference between PCF and Haskell.

Firstly, in Haskell, one may introduce new identifiers by function definitions such as

```
id      x = x
plustwo n = Succ (Succ n)
```

In PCF we must instead write

```
id      = \x -> x
plustwo = \n -> Succ (Succ n)
```

Secondly, recursive definitions are allowed in Haskell, for example

```
infinity = Succ infinity
loop     = loop
```

In PCF we use the fixed point combinator to turn a recursive definition into an explicit one:

```
infinity = fix Succ
loop     = fix id
```

where we have used that

```
loop     = id loop
```

The general principle for eliminating a recursive definition is the following. Assume that we have a recursive definition of the form

```
foo = ... foo ...
```

where the RHS is an expression which may contain `foo`. We first transform it into the form

```
foo = (\f -> ... f ...) foo
```

so that we can use the fixed point combinator

```
foo = fix (\f -> ... f ...)
```

In Haskell one may define functions by *pattern matching* such as

```
equiv True  True  = True
equiv True  False = False
equiv False True  = False
equiv False False = True
```

Instead of pattern matching on `True` and `False` we use `if`:

```
equiv m n = if m (if n True False) (if n False True)
```

Then we can λ-abstract

```
equiv = \m n -> if m (if n True False) (if n False True)
```

to get a PCF-program.

In Haskell pattern matching can be combined with recursion:

```
double Zero     = Zero
double (Succ n) = Succ (Succ (double n))
```

To transform this to a PCF program we first eliminate the pattern matching using `if, isZero` and `pred`:

```
double n = if (isZero n)
              Zero
              (Succ (Succ (double (pred n))))
```

Then we λ-abstract with respect to `n`:

```
double = \n -> if (isZero n)
               Zero
               (Succ (Succ (double (pred n))))
```

Then we transform the RHS to get it on the form `double` $= f$ `double`:

```
double = (\d n-> if (isZero n)
                    Zero
                    (Succ (Succ (d (pred n)))))
         double
```

Finally, we can solve this equation by using `fix`

```
double = fix (\d n-> if (isZero n)
                        Zero
                        (Succ (Succ (d (pred n)))))
```

1.5 More recursive datatypes

PCF can be seen as the core of a standard functional language such as ML or Haskell. The most important feature of such languages which is missing in PCF is the type of lists and other recursive datatypes.

A recursive datatype which we shall use later in the course is the type of binary trees

```
data Bin a = Leaf a | Branch (Bin a) (Bin a)
```

This means that we declare two constructors `Leaf` and `Branch`. When we define functions on binary trees we will use a selector `isLeaf` which is analogous to `isZero`. It is defined by

```
isLeaf (Leaf a)       = True
isLeaf (Branch as bs) = False
```

Furthermore, we will use destructors `left, right`, and `peel` which are analogous to `pred`. They are defined by

```
left  (Branch as bs) = as
right (Branch as bs) = bs
peel  (Leaf a)       = a
```

Note that these are all partial functions: `left (Leaf a)`, `right (Leaf a)`, and `peel (Branch as bs)` are all undefined.

Exercise. Extend the grammars for PCF-types and PCF-terms in 1.1 to include binary trees and the operations on them.

Exercise. The type of lists is usually a built in datatype, but could otherwise be given by the following datatype declaration in Haskell:

```
List a = Nil | Cons a (List a)
```

What are suitable selectors and destructors for lists? Extend the grammars for PCF-types and PCF-terms in 1.1 to include lists and the operations on them.

1.6 Reduction rules for PCF (small step)

Before reading this section it is useful to have read KP 5.3 and 5.3.1. There you will find a detailed discussion of *substitution* and *β-reduction* of pure λ-terms. KP discusses full β-reduction of *open* terms, where one may reduce an arbitrary β-redex. This is different from the notion of reduction used in functional programming languages, where one only reduces *closed* terms, and where one chooses a certain *reduction strategy* so that a term can only be reduced in one way (so called *deterministic* reduction). (See also the discussion in KP p 164-166 on reduction strategies and reduction in real functional languages.)

There are two principal reduction strategies: *applicative* order reduction and *normal* order reduction. We choose the latter which is the reduction strategy used for "lazy" functional languages such as Haskell. To emphasize the analogy with parameter passing mechanisms in imperative languages applicative order is often referred to as *call-by-value* and normal order as *call-by-name.*

However, the natural numbers in PCF are not the "lazy" natural numbers you get in Haskell from the data type declaration

```
data Nat = Zero | Succ Nat
```

Instead, for reasons of simplicity, we assume that natural numbers are always evaluated fully, that is, the computation of a natural number will not terminate until it has reached $0 = \texttt{Zero}, 1 = \texttt{Succ Zero}, 2 = \texttt{Succ (Succ Zero)}, \ldots$ This is the way integers (of type `Int`) in Haskell are computed. To give a complete explanation of lazy evaluation of natural numbers is more complicated, see Section 1.9 which is optional reading.

We first give the rules for *one-step reduction* and write $e \rightarrow_1 e'$ if e can be reduced to e' in one step. From this we can define *many-step reduction* and write $e \rightarrow_i e'$ if e can be reduced to e' in i steps by adding the rules

$$e \rightarrow_0 e$$

which states that e reduces to itself in 0 steps (!), and the rule

$$\frac{e \rightarrow_i e' \qquad e' \rightarrow_j e''}{e \rightarrow_{i+j} e''}$$

which states that if e reduces in i steps to e' and e' reduces in j steps to e'' then e reduces to e'' in $i+j$ steps.

We now give the rules for one-step reduction between closed PCF-terms. In general in this section a, b, c, d, e, f (possibly with primes) stand for arbitrary PCF-terms.

First, we have rules for computing function applications $f\ a$. The first rule states that we reduce the function part f if possible:

$$\frac{f \rightarrow_1 f'}{f\ a \rightarrow_1 f'\ a}$$

The second rule is *β-reduction*

$$(\lambda x.e)\ a \rightarrow_1 e[x := a]$$

which specifies that when the function part has reached a canonical form $\lambda x.e$ we can substitute the argument a for the variable x in e. (The general notion of a canonical form is defined below.)

The first rule for conditional expressions states that we reduce the condition b if possible:

$$\frac{b \rightarrow_1 b'}{\texttt{if}\ b\ d\ e \rightarrow_1 \texttt{if}\ b'\ d\ e}$$

The other rules state what to do when the condition is a canonical truth-value:

$$\begin{array}{rcl} \texttt{if True}\ d\ e & \rightarrow_1 & d \\ \texttt{if False}\ d\ e & \rightarrow_1 & e \end{array}$$

We then specify the rules for computing the operations on `Nat`. First we have rules specifying that we compute the argument of `Succ`, `pred`, and `isZero`, whenever possible.

$$\frac{e \rightarrow_1 e'}{\texttt{Succ}\ e \rightarrow_1 \texttt{Succ}\ e'}$$

$$\frac{e \rightarrow_1 e'}{\texttt{pred}\ e \rightarrow_1 \texttt{pred}\ e'}$$

$$\frac{e \rightarrow_1 e'}{\texttt{isZero}\ e \rightarrow_1 \texttt{isZero}\ e'}$$

Then we have rules which specify the result of the operation when the argument is a canonical natural number (note that n below has to be a fully evaluated natural number $0 = \texttt{Zero}, 1 = \texttt{Succ Zero}, 2 = \texttt{Succ (Succ Zero)}, \ldots$):

$$\begin{array}{rcl} \texttt{pred Zero} & \rightarrow_1 & \texttt{Zero} \\ \texttt{pred (Succ}\ n) & \rightarrow_1 & n \\ \texttt{isZero Zero} & \rightarrow_1 & \texttt{True} \\ \texttt{isZero (Succ}\ n) & \rightarrow_1 & \texttt{False} \end{array}$$

Finally we have the rule for computing the fixed point combinator:

$$\texttt{fix}\ f \quad \rightarrow_1 \quad f\ (\texttt{fix}\ f)$$

When computing a PCF-term e we perform repeated one-step reductions

$$\begin{array}{rcl} e & \to_1 & e_1 \\ e_1 & \to_1 & e_2 \\ e_2 & \to_1 & e_3 \\ & \vdots & \end{array}$$

Such *reduction-sequences* are usually written in an abbreviated way:

$$e \to_1 e_1 \to_1 e_2 \to_1 e_3 \to_1 \cdots$$

Three things can happen when we perform step-wise reduction of a term:

- The reduction of e terminates with a *canonical form* v:

  $$e \to_1 e_1 \to_1 e_2 \to_1 e_3 \cdots \to_1 v$$

  The canonical forms cannot be reduced further, that is, there is no e' such that $v \to_1 e'$. Moreover, they are the "good" results of a terminating computation. They include the canonical truth values `True`, `False`; the canonical (or fully evaluated) natural numbers $0 = \texttt{Zero}, 1 = \texttt{Succ Zero}, 2 = \texttt{Succ (Succ Zero)}, \ldots$; the closed λ-expressions of the form $\lambda x.e$, where e can be an arbitrary term (the so called *weak head normal forms*); the function constants `Succ`, `pred`, `isZero`, `fix` and `if`, as well as expressions of the form `if` b and `if` $b\ d$, in which `if` has too few arguments to be reduced.

- The reduction of e terminates with an e' which cannot be reduced further

  $$e \to_1 e_1 \to_1 e_2 \to_1 e_3 \to_1 \cdots \to_1 e'$$

  but the final term e' in the sequence is still not a canonical form, because it is not a "good" value. For example, the term `Succ True` is not a canonical form even though it cannot be reduced.

- The reduction does not terminate:

  $$e \to_1 e_1 \to_1 e_2 \to_1 e_3 \to_1 \cdots \quad \text{(ad infinitum)}$$

  For example, the reductions of `loop` and `infinity` do not terminate.

As in KP we write $e \to e'$ if $e \to_i e'$ for some i, that is, if e can be reduced in some number of steps to e', and we do not care how many. It follows that $\to$ is a reflexive and transitive relation (why?). One says that $\to$ is the *reflexive-transitive closure* of $\to_1$, since it is the smallest reflexive and transitive relation containing $\to_1$.

Examples. We show that `equiv True True` $\to_4$ `True`:

```
   equiv True True
=  (\b b’ -> if b (if b’ True False) (if b’ False True)) True True
-> (\b’ -> if True (if b’ True False) (if b’ False True)) True
-> if True (if True True False) (if True False True)
-> if True True False
-> True
```

(For typographic reasons we use Haskell-style λ and drop the index on the arrows indicating one-step reductions. Note the difference between $=$ and $\to$: the former indicates macro expansion whereas the latter indicates one-step reduction.)

We then show the reduction sequences for `double` applied to 0 and 1. For the sake of readability we introduce the abbreviations

```
double    = fix doubledef
doubledef = (\d n-> if (isZero n)
                       0
                       (Succ (Succ (d (pred n)))))
```

First we show double $0 \to_5 0$:

```
   double 0
=  fix doubledef 0
-> doubledef (fix doubledef) 0
=  doubledef double 0
=  (\d n-> if (isZero n)
           0
           (Succ (Succ (d (pred n))))) double 0
-> (\n -> if (isZero n)
          0
          (Succ (Succ (double (pred n))))) 0
-> if (isZero 0)
      0
      (Succ (Succ (double (pred 0))))
-> if True
      0
      (Succ (Succ (double (pred 0))))
-> 0
```

Then we show double $1 \to_{11} 2$:

```
   double 1
=  fix doubledef 1
-> doubledef double 1
=  (\d n-> if (isZero n)
           0
           (Succ (Succ (d (pred n))))) double 1
-> (\n -> if (isZero n)
          0
          (Succ (Succ (double (pred n))))) 1
-> if (isZero 1)
      0
      (Succ (Succ (double (pred 1))))
-> if False
      0
      (Succ (Succ (double (pred 1))))
-> Succ (Succ (double (pred 1)))
-> Succ (Succ (double 0))
... (in 5 steps, see above)
-> Succ (Succ 0)
=  2
```

1.7 Rules for evaluation to canonical form (big step)

An alternative way to define the operational semantics of PCF is to give rules relating a PCF-term e to its canonical form v, written $e \Rightarrow v$. This is often called “big step semantics” since $\Rightarrow$ is like a “big step” to the canonical form.

If we want to include a complexity measure, we write $e \Rightarrow_i v$ if the PCF-term e reaches its canonical form v in i steps. We here give a big step semantics including such a complexity measure.

First, we have a rule that states that expressions beginning with a λ are canonical

$$\lambda x.e \Rightarrow_0 \lambda x.e$$

Similarly, the function constants are canonical if they are applied to too few arguments:

$$\texttt{if} \Rightarrow_0 \texttt{if} \qquad \texttt{if}\ b \Rightarrow_0 \texttt{if}\ b \qquad \texttt{if}\ b\ d \Rightarrow_0 \texttt{if}\ b\ d$$

$$\texttt{Succ} \Rightarrow_0 \texttt{Succ} \qquad \texttt{pred} \Rightarrow_0 \texttt{pred} \qquad \texttt{isZero} \Rightarrow_0 \texttt{isZero}$$

$$\texttt{fix} \Rightarrow_0 \texttt{fix}$$

Then there is a rule for evaluating applications stating that the canonical form of $f\ a$ can be obtained by first evaluting f to canonical form $\lambda x.e$, then performing the β-reduction to get $e[x := a]$, which thereafter is evaluated to canonical form:

$$\frac{f \Rightarrow_i \lambda x.e \qquad e[x := a] \Rightarrow_j v}{f\ a \Rightarrow_{i+1+j} v}$$

We also have rules for truth-values. First we state that `True` and `False` are canonical:

$$\texttt{True} \Rightarrow_0 \texttt{True} \qquad \texttt{False} \Rightarrow_0 \texttt{False}$$

Then we have rules for evaluting conditional expressions:

$$\frac{b \Rightarrow_i \texttt{True} \qquad d \Rightarrow_j v}{\texttt{if}\ b\ d\ e \Rightarrow_{i+1+j} v} \qquad \frac{b \Rightarrow_i \texttt{False} \qquad e \Rightarrow_j v}{\texttt{if}\ b\ d\ e \Rightarrow_{i+1+j} v}$$

Moreover, we have rules for natural numbers (here n ranges over canonical natural numbers $0, 1, 2, \ldots$):

$$\texttt{Zero} \Rightarrow_0 \texttt{Zero} \qquad \frac{e \Rightarrow_i n}{\texttt{Succ}\ e \Rightarrow_i \texttt{Succ}\ n}$$

$$\frac{e \Rightarrow_i \texttt{Zero}}{\texttt{pred}\ e \Rightarrow_{i+1} \texttt{Zero}} \qquad \frac{e \Rightarrow_i \texttt{Succ}\ n}{\texttt{pred}\ e \Rightarrow_{i+1} n}$$

$$\frac{e \Rightarrow_i \texttt{Zero}}{\texttt{isZero}\ e \Rightarrow_{i+1} \texttt{True}} \qquad \frac{e \Rightarrow_i \texttt{Succ}\ n}{\texttt{isZero}\ e \Rightarrow_{i+1} \texttt{False}}$$

Finally, there is a rule for fixed points

$$\frac{f\ (\texttt{fix}\ f) \Rightarrow_i v}{\texttt{fix}\ f \Rightarrow_{i+1} v}$$

We should now be able to prove that $e \Rightarrow_j v$ iff $e \rightarrow_j v$ and v is a canonical form, but we omit the proof of this. (The proof is by so-called "rule induction": both $\Rightarrow_i$ and $\rightarrow_i$ are *inductively defined*, that is, they are the smallest relations satisfying their respective rules.)

1.8 Applicative order evaluation

To obtain applicative order evaluation we just need to change the rule of β-reduction, which now should only be performed when the argument is in canonical form ("call-by-value"). So the reduction of an application starts as before by reducing the function part to canonical form. But then the argument is reduced before the β-reduction is performed.

We therefore need to replace the rule of normal order β-reduction (call-by-name)

$$(\lambda x.e)\ a \rightarrow_1 e[x := a]$$

by the following two rules for applicative order (call-by-value).

The first rule states that you can reduce the argument of an application provided the function part is in canonical form:

$$\frac{a \rightarrow_1 a'}{(\lambda x.e)\ a \rightarrow_1 (\lambda x.e)\ a'}$$

The second rule states that if both the function part and the argument are in canonical form, then you perform β-reduction:

$$(\lambda x.e)\ v \rightarrow_1 e[x := v]$$

Note that v ranges over canonical forms in this rule.

Alternatively, we can modify the rules for evaluation to canonical form (big step). Here we just need to replace the rule for normal order evaluation of an application

$$\frac{f \Rightarrow_i \lambda x.e \qquad e[x := a] \Rightarrow_j v}{f\ a \Rightarrow_{i+1+j} v}$$

to the one for applicative order evaluation:

$$\frac{f \Rightarrow_i \lambda x.e \qquad a \Rightarrow_k u \qquad e[x := u] \Rightarrow_j v}{f\ a \Rightarrow_{i+k+1+j} v}$$

Remark. As is clear from the big step semantics of applicative order evaluation, it does not matter if one chooses to reduce the function f or the argument a first in an application $f\ a$. In our small step semantics above we chose to begin by reducing the function, but in the SECD-machine (described in KP 5.4) which implements applicative order evaluation the argument is computed before the function.

1.9 Lazy natural numbers (optional reading)

As mentioned before, we have assumed that natural numbers in PCF are always evaluated fully. This means that the reduction of a natural number does not terminate until it has reached one of $0, 1, 2, \ldots$. (One says that such natural numbers are "strict" as opposed to "lazy".) So if we define

```
infinity = fix Succ
```

and compute `infinity` then this reduces in one step to

```
Succ (fix Succ)
```

However, this has not yet terminated and we have to reduce `fix Succ`, etc. so we never terminate.

However, when we define the type `Nat` in Haskell by

```
data Nat = Zero | Succ Nat
```

we get "lazy" natural numbers. This means that when we evaluate a natural number we will output a `Succ` as soon as we have reached an expression `Succ` e even if e is not itself fully evaluated. For example, if we compute `Succ loop` we will output `Succ` even though its argument `loop` will never terminate.

So if we compute `infinity` for lazy natural numbers this will reduce in one step to

```
Succ (fix Succ)
```

and we will immediately output a `Succ`. Then its argument `fix Succ` will be computed and similarly output another `Succ`. The result is that we will compute an infinite sequence

```
Succ (Succ (Succ (Succ ... )))
```

To get computation rules for lazy natural numbers we first change the rules for computing `pred` and `isZero` so that they do not fully evaluate their arguments. We get the rules

$$\begin{aligned}
\texttt{pred}\,\texttt{Zero} &\rightarrow_1 \texttt{Zero} \\
\texttt{pred}\,(\texttt{Succ}\,e) &\rightarrow_1 e \\
\texttt{isZero}\,\texttt{Zero} &\rightarrow_1 \texttt{True} \\
\texttt{isZero}\,(\texttt{Succ}\,e) &\rightarrow_1 \texttt{False}
\end{aligned}$$

where e now is an arbitrary expression, not necessarily a fully evaluated natural number.

Furthermore, we remove the rule

$$\frac{e \rightarrow_1 e'}{\texttt{Succ}\,e \rightarrow_1 \texttt{Succ}\,e'}$$

which expresses that `Succ` should evaluate its argument. It should *not* do so *until it has terminated and output a* `Succ`. This may seem strange, because `Succ` will indeed evaluate its argument, but the point is that this will happen only *after* it has produced its output. The explanation is that we should no longer understand $\rightarrow_1$ as one step of the reduction until it *finally* terminates, but only one step of the reduction until it produces its first output.

1.10 Typing rules

We are used to reading typing statements of the form

$$e :: \tau$$

where e is a *closed* term and τ is a type expression. Examples are the typing rules for the PCF-combinators, but also typings for compound expressions such as

$$\begin{aligned} \texttt{isZero}\,(\texttt{Succ}\,\texttt{Zero}) &:: \texttt{Bool} \\ \texttt{fix}\,\texttt{Succ} &:: \texttt{Nat} \end{aligned}$$

However, in order to specify the type system for PCF we must also deal with the types of *open* terms. For example, what is the type of the expression `Succ x`? We cannot answer this unless we know the type of `x`! If $\texttt{x} :: \texttt{Bool}$, then `Succ x` is ill-typed, but if $\texttt{x} :: \texttt{Nat}$, then $\texttt{Succ x} :: \texttt{Nat}$. We express the latter correct conditional typing statement by writing

$$\texttt{x} :: \texttt{Nat} \vdash \texttt{Succ}\,\texttt{x} :: \texttt{Nat}$$

In general we shall write $\Gamma \vdash e :: \tau$, where Γ is a *type assignment*, that is, a list of assignments of PCF-types to variables. So $\texttt{x} :: \texttt{Nat}$ is a type assignment and $\texttt{x} :: \texttt{Nat}, \texttt{y} :: \texttt{Bool}$ is another. It can also be the case that the same variable gets typed twice: $\texttt{x} :: \texttt{Nat}, \texttt{x} :: \texttt{Bool}$ is also a valid type assignment.

The typing rule for application is

$$\frac{\Gamma \vdash f :: \tau \to \tau' \qquad \Gamma \vdash a :: \tau}{\Gamma \vdash f\ a :: \tau'}$$

which states that if f has a function type $\tau \to \tau'$ and a has type τ, then $f\ a$ has the type τ'. The Γ in the rules expresses that this is the case for any type assignment.

Then there is the rule for λ-abstraction:

$$\frac{\Gamma, x :: \tau \vdash e :: \tau'}{\Gamma \vdash \lambda x.e :: \tau \to \tau'}$$

which says that if e is an expression of type τ' which may contain a free variable x of type τ, then $\lambda x.e$ has the type $\tau \to \tau'$.

Then there is the rule for variables:

$$\Gamma \vdash x :: \tau$$

provided $x :: \tau$ is the last assignment of a type to x in Γ. To explain why we are only allowed to use the last assignment consider the following correct derivation:

```
   x :: a, x :: b |- x :: b
  x :: a |- \x -> x :: b -> b
 |- \x -> \x -> x :: a -> b -> b
```

Then compare it with the following erroneous derivation:

```
   x :: a, x :: b |- x :: a
  x :: a |- \x -> x :: b -> a
 |- \x -> \x -> x :: a -> b -> a
```

The remaining rules are the typing rules for the PCF-combinators:

$$\begin{aligned}
\Gamma &\vdash \texttt{True} :: \texttt{Bool} \\
\Gamma &\vdash \texttt{False} :: \texttt{Bool} \\
\Gamma &\vdash \texttt{Zero} :: \texttt{Nat} \\
\Gamma &\vdash \texttt{Succ} :: \texttt{Nat} \to \texttt{Nat} \\
\Gamma &\vdash \texttt{if} :: \texttt{Bool} \to \tau \to \tau \to \tau \\
\Gamma &\vdash \texttt{pred} :: \texttt{Nat} \to \texttt{Nat} \\
\Gamma &\vdash \texttt{isZero} :: \texttt{Nat} \to \texttt{Bool} \\
\Gamma &\vdash \texttt{fix} :: (\tau \to \tau) \to \tau
\end{aligned}$$

Example derivation. Usually type derivations are presented as trees, much like the derivation trees for computations in P discussed in KP 3.7. For typographic reasons we here instead use indentation to indicate the tree structure: the two (say) premises `premise1, premise2` of a typing statement `conclusion` can be found above and one step indented:

```
   .
   .
  premise1
   .
   .
  premise2
 conclusion
  .
  .
```

Using this notation the derivation of

```
|- equiv :: Bool -> Bool -> Bool
```

is as follows

```
        b :: Bool, b' :: Bool |- if :: Bool -> Bool -> Bool -> Bool
        b :: Bool, b' :: Bool |- b :: Bool
       b :: Bool, b' :: Bool |- if b :: Bool -> Bool -> Bool
          b :: Bool, b' :: Bool |- if :: Bool -> Bool -> Bool -> Bool
          b :: Bool, b' :: Bool |- b' :: Bool
         b :: Bool, b' :: Bool |- if b' :: Bool -> Bool -> Bool
         b :: Bool, b' :: Bool |- True :: Bool
        b :: Bool, b' :: Bool |- if b' True :: Bool -> Bool
        b :: Bool, b' :: Bool |- False :: Bool
       b :: Bool, b' :: Bool |- if b' True False :: Bool
      b :: Bool, b' :: Bool |- if b (if b' True False) :: Bool -> Bool
         b :: Bool, b' :: Bool |- if :: Bool -> Bool -> Bool -> Bool
         b :: Bool, b' :: Bool |- b' :: Bool
        b :: Bool, b' :: Bool |- if b' :: Bool -> Bool -> Bool
        b :: Bool, b' :: Bool |- False :: Bool
       b :: Bool, b' :: Bool |- if b' False :: Bool -> Bool
       b :: Bool, b' :: Bool |- True :: Bool
      b :: Bool, b' :: Bool |- if b' False True :: Bool
     b :: Bool, b' :: Bool |- if b (if b' True False) (if b' False True)) :: Bool
    b :: Bool |- \b' -> if b (if b' True False) (if b' False True)) :: Bool -> Bool
   |- (\b b' -> if b (if b' True False) (if b' False True)) :: Bool -> Bool -> Bool
```

In the derivation of the type for `equiv` we have written out every step according to the typing rules. This makes it quite long and tedious. A way to make it shorter is to introduce *derived rules* such as

$$\frac{\Gamma \vdash b :: \texttt{Bool} \qquad \Gamma \vdash d :: \tau \qquad \Gamma \vdash e :: \tau}{\Gamma \vdash \texttt{if}\ b\ d\ e :: \tau}$$

It is clear that this rule can be derived by using the type of `if` and the rule for typing an application three times. Using the derived rule the derivation of `equiv` becomes shorter and more readable:

```
       b :: Bool, b' :: Bool |- b :: Bool
        b :: Bool, b' :: Bool |- b' :: Bool
        b :: Bool, b' :: Bool |- True :: Bool
        b :: Bool, b' :: Bool |- False :: Bool
       b :: Bool, b' :: Bool |- if b' True False :: Bool
        b :: Bool, b' :: Bool |- b' :: Bool
        b :: Bool, b' :: Bool |- False :: Bool
        b :: Bool, b' :: Bool |- True :: Bool
       b :: Bool, b' :: Bool |- if b' False True :: Bool
      b :: Bool, b' :: Bool |- if b (if b' True False) (if b' False True)) :: Bool
     b :: Bool |- \b' -> if b (if b' True False) (if b' False True)) :: Bool -> Bool
    |- (\b b' -> if b (if b' True False) (if b' False True)) :: Bool -> Bool -> Bool
```

For the same reason it is useful to introduce derived rules for the other PCF-combinators. Do this as an exercise!

2 Computability

We shall now discuss some problems which cannot be computed by any PCF-program. To this end it will be useful to consider *all* PCF-programs, not only the ones which have a type.

For the sake of readibility we will write our PCF-programs in Haskell-style, and rely on the reader to apply the technique of Section 1.4 to transform them into the proper form.

For example, we shall write

```
id   x = x
loop   = loop
liar x = if (halts (x x)) loop True
```

as more readable versions of the proper PCF-programs

```
id   = \x -> x
loop = fix id
liar = \x -> if (halts (x x)) loop True
```

2.1 Some incomputable problems about PCF-programs

We begin by proving the following simple version of the halting problem:

Theorem 1 *There is no PCF-program* `halts`*, such that*

$$\texttt{halts}\,e \quad\Rightarrow\quad \begin{cases} \texttt{True}, & \textit{iff } e \textit{ has canonical form} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Proof: Assume that there is such a PCF-program `halts`. Then we can define the PCF-program

```
liar x = if (halts (x x)) loop True
```

Now, `True` is canonical but `loop` does not have canonical form. So apply `liar` to itself to get a contradiction: `liar liar` has a canonical form iff `liar liar` does not have a canonical form!
□

Some variations which are proved in essentially the same way:

Theorem 2 *There is no PCF-program* `isBool`*, such that*

$$\texttt{isBool}\,e \quad\Rightarrow\quad \begin{cases} \texttt{True}, & \textit{iff } e \Rightarrow \texttt{True} \textit{ or } e \Rightarrow \texttt{False} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Theorem 3 *There is no PCF-program* `isTrue`*, such that*

$$\texttt{isTrue}\,e \quad\Rightarrow\quad \begin{cases} \texttt{True}, & \textit{iff } e \Rightarrow \texttt{True} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Theorem 4 *There is no PCF-program* `isNat`*, such that*

$$\texttt{isNat}\,e \quad\Rightarrow\quad \begin{cases} \texttt{True}, & \textit{iff } e \Rightarrow n \textit{ for some canonical natural number } n \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

A PCF-program f computes a *constant function* on natural numbers iff there is an n :: `Nat` such that $f\ m \Rightarrow n$ for all m :: `Nat`, that is, if it outputs the same n irrespectively of the input m. It would be wasteful to compute the argument of f, since the result doesn't depend on it. This kind of knowledge is useful for optimizing implementations of functional programs.

Theorem 5 *There is no PCF-program* `isConstFun` *such that*

$$\texttt{isConstFun}\,e \quad\Rightarrow\quad \begin{cases} \texttt{True}, & \textit{iff } e \textit{ is a constant function on natural numbers} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Proof. We assume that there is such a PCF-program `isConstFun` and show that this entails that there is a PCF-program `isNat` which contradicts theorem 4.

First, consider the PCF-program `strictZero`

```
strictZero x = if (isZero x) 0 (strictZero (pred x))
```

This program has the property that `strictZero` $e \Rightarrow 0$ iff $e \Rightarrow n$ for some canonical natural number n.

But now we can define `isNat`:

$$\texttt{isNat}\,\texttt{x} = \texttt{isConstFun}\,(\lambda\texttt{y}.\texttt{strictZero}\,\texttt{x})$$

Assume first that $e \Rightarrow n$ for some canonical natural number n. Then `strictZero` $e \Rightarrow 0$ and thus λ`y.strictZero` e is a constant function on natural numbers. Hence, `isNat` $e \Rightarrow$ `True`.

Assume then that there is no canonical natural number n such that $e \Rightarrow n$. Then `strictZero` e does not reduce to a canonical natural number and thus λ`y.strictZero` e is not a constant function on natural numbers. Hence, `isNat` $e \Rightarrow$ `False`.

□

Two PCF-programs f and f' compute the same function on natural numbers iff for all $m, n ::$ `Nat` it is the case that $f\ m \Rightarrow n$ iff $f'\ m \Rightarrow n$, that is, f and f' give the same output n (if any) for an arbitrary input m.

The following theorem states that it is futile to write a test for deciding whether two PCF-programs compute the same function. This is the reason why function types are not instances of the equality class `Eq` in Haskell.

Theorem 6 *There is no PCF-program* `eqFun` *such that*

$$\texttt{eqFun}\,f\ f' \quad\Rightarrow\quad \begin{cases} \texttt{True}, & \textit{if } f \textit{ and } f' \textit{ compute the same function on natural numbers} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Proof. The proof is similar to the previous. If we can test whether two function are equal then we can test whether a function f is equal to the function which always returns 0, that is, we can test whether $f\ m \Rightarrow 0$ for all canonical natural numbers m. So if we assume that `eqFun` exists, then we can write a program for `isNat`! (How?)

□.

2.2 Encoding PCF-programs as natural numbers

Above we could show that some problems about PCF-programs do not have computable solutions. It was possible to do so in a simple way by using the fact that we can apply a PCF-program e to itself to get a new PCF-program $e\ e$.

Traditional computability theory usually uses Turing machines or recursive functions for defining a notion of computable function on natural numbers. In KP the small imperative language P is used for the same purpose. In neither of these models of computation self-application is possible. Instead one first encodes for example Turing machines as natural numbers. Then one can give the code of a Turing machine as input to the Turing machine itself. The development is analogous if one uses recursive functions or P-programs.

We shall now show how to encode PCF-programs as natural numbers, and then reformulate theorem 1-6 as statements about such codes.

To this end we introduce CPCF, a combinatory version of PCF. Since CPCF-programs have no variables they will be easier to encode as natural numbers than PCF-programs.

The *CPCF-terms* are generated by the following context-free grammar:

```
<cterm> ::= (<cterm> <cterm>) | k | s |
            True | False | Zero | Succ |
            if | pred | isZero | fix
```

and again we may omit parentheses in the usual way. CPCF is also a Haskell-subset, since we can define the combinators `k` and `s` by

```
k x y   = x
s x y z = x z (y z)
```

One can translate back and forth between PCF and CPCF. A CPCF-program can easily be translated into a PCF-program by using the definitions of `k` and `s` as λ-expressions. A PCF-program on the other hand can be translated into a CPCF-program by using the algorithm given in exercise 5.13 in KP (lab 4). So CPCF-programs should be viewed as "machine codes" for PCF-programs and are there only for coding purposes.

We shall now show how to encode a PCF-program as a natural number. This natural number can be represented as an element of the type `Nat` in PCF in the usual way. (Therefore we can talk about a "self-encoding" of PCF inside itself.) To do this we first translate the PCF-program into the corresponding CPCF-program. To this CPCF-program e we associate a natural number $e^{\#}$ in PCF as follows:

$$\begin{aligned}
\texttt{k}^{\#} &= 0\\
\texttt{s}^{\#} &= 1\\
\texttt{True}^{\#} &= 2\\
\texttt{False}^{\#} &= 3\\
\texttt{Zero}^{\#} &= 4\\
\texttt{Succ}^{\#} &= 5\\
\texttt{if}^{\#} &= 6\\
\texttt{pred}^{\#} &= 7\\
\texttt{isZero}^{\#} &= 8\\
\texttt{fix}^{\#} &= 9\\
(f\ a)^{\#} &= 10 + (\texttt{paircode}\ f^{\#}\ a^{\#})
\end{aligned}$$

where

```
paircode :: Nat -> Nat -> Nat

paircode m n = (m + n + 1)(m + n)/2 + m
```

is a PCF-program, where we have used infix $+$ for PCF-addition.

Note that there is a bijective correspondence between CPCF-programs e and elements $e^{\#}$ of `Nat`. Note also that the function which maps a program e to its code $e^{\#}$ is not itself a PCF-program. (In fact, it can be shown that there is no PCF-program `encode` such that `encode` $e = e^{\#}$ for all e. So $-^{\#}$ is another example of an uncomputable function.)

2.3 The self interpreter

A self interpreter for PCF is a PCF-program `eval` which inputs a natural number and returns the value of the PCF-program it encodes, that is,

$$\texttt{eval}\ e^{\#} \Rightarrow v \text{ iff } e \Rightarrow\ v$$

for an abitrary PCF-program e. In particular assume that $f :: \texttt{Nat} \to \texttt{Nat}$, then we also have

$$\texttt{eval}\ f^{\#}\ m \Rightarrow n \text{ iff } f\ m \Rightarrow\ n$$

for all $m, n :: \texttt{Nat}$. Similarly, for a binary function $f :: \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$, we have

$$\texttt{eval}\ f^{\#}\ m\ m' \Rightarrow n \text{ iff } f\ m\ m' \Rightarrow\ n$$

for all $m, m', n :: \texttt{Nat}$. Etc.

The self interpreter is a PCF-program which computes the inverse of the encoding function. Given a natural number n it returns the CPCF-program n encodes. If $n = 0$ we return `k`, if $n = 1$ we return `s`, etc. If $n \geq 10$ then we encode an application. Thus $n - 10$ is a code for a pair of natural numbers p, q where p encodes the function part of the application and q encodes the argument part. We thus need two auxiliary functions

```
funpart :: Nat -> Nat
argpart :: Nat -> Nat
```

such that

```
n = 10 + (paircode (funpart n) (argpart n))
```

Here is a PCF-program for `eval`:

```
eval n =  if (n == 0) k
         (if (n == 1) s
         (if (n == 2) True
         (if (n == 3) False
         (if (n == 4) Zero
         (if (n == 5) Succ
         (if (n == 6) if
         (if (n == 7) pred
         (if (n == 8) isZero
         (if (n == 9) fix
                      (eval (funpart n)) (eval (argpart n))))))))))))
```

We have used Haskell's infix equality sign (==) for equality of natural numbers in PCF.

Remark. A self interpreter for Turing machines is usually called a *universal Turing machine*. This is a Turing machine `eval` which given a code $t^{\#}$ for an arbitrary Turing machine t and a natural number m as input on its tape, computes the same natural number n (if any) as t does with input m. Note the similarity between a universal Turing machine and a computer with a stored program.

2.4 Some incomputable problems about codes for PCF-programs

In the first version of the halting problem we showed that it was impossible to find a PCF-program which takes an arbitrary PCF-program as input and decides whether it has a canonical form. This formulation is not so satisfactory because such a program must be defined on an arbitrary input, so cannot be typable.

In the second formulation we instead show that there can be no PCF-program

```
haltsc :: Nat -> Bool
```

which takes a PCF-program *coded as a natural number* and decides whether the PCF-program has a canonical form.

Theorem 7 *There is no PCF-program* `haltsc`*, such that*

$$\texttt{haltsc}\, e^{\#} \quad \Rightarrow \quad \begin{cases} \texttt{True}, & \textit{iff } e \textit{ has canonical form} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Proof: The proof is much like the proof of Theorem 1: we assume that there is such a PCF-program `haltsc`, and derive a contradiction. However, we need a modified version `liarc` of `liar` which is applied to a *code* of itself rather than to itself.

A first try would be

$$\texttt{liarc}\,\texttt{n} = \texttt{if}\,(\texttt{haltsc}\,(\texttt{eval}\,\texttt{n}\,\texttt{n})^{\#})\,\texttt{loop}\,\texttt{True}$$

but $(\texttt{eval}\,\texttt{n}\,\texttt{n})^{\#}$ is unfortunately not a PCF-program (as remarked above the function $e^{\#}$ which computes a code for an arbitrary PCF-program e is not computable).

However, it is sufficient to compute $(\texttt{eval}\,\texttt{n}\,\texttt{n})^{\#}$ when `n` is a natural number in PCF, and this can easily be done by following the definition of the encoding function in Section 2.2. Call this program `selfevalc`:

```
selfevalc n    = applyc (applyc evalc (natc n)) (natc n)

applyc m n     = 10 + (paircode m n)

natc Zero      = 4
natc (Succ n)  = applyc 5 (natc n)
```

where `evalc` $=$ `eval`$^{\#}$. It follows immediately that

$$\texttt{selfevalc n} \Rightarrow (\texttt{eval n n})^{\#}$$

for all natural numbers `n` in PCF.

Now we are ready for the correct definition of `liarc`

```
liarc n = if (haltsc (selfevalc n)) loop True
```

To derive the contadiction we reason as in the first version of the halting problem, except that we instead apply `liarc` to a *code* for `liarc`. Recall that `eval` $e^{\#}$ $e^{\#}$ has canonical form iff e $e^{\#}$ does. Therefore we conclude that `liarc liarc`$^{\#}$ has a canonical form iff `selfevalc liarc`$^{\#}$ does not have a canonical form iff `liarc liarc`$^{\#}$ does not have a canonical form. We have a contradiction, so we must conclude that there is no PCF-program `haltsc` with the above specification.

□

Again, there are some variations which are proved in essentially the same way:

Theorem 8 *There is no PCF-program* `isBoolc`*, such that*

$$\texttt{isBoolc}\, e^{\#} \quad \Rightarrow \quad \begin{cases} \texttt{True}, & \textit{iff } e \Rightarrow \texttt{True} \textit{ or } e \Rightarrow \texttt{False} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Theorem 9 *There is no PCF-program* `isTruec`*, such that*

$$\texttt{isTruec}\, e^{\#} \quad \Rightarrow \quad \begin{cases} \texttt{True}, & \textit{iff } e \Rightarrow \texttt{True} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Theorem 10 *There is no PCF-program* `isNatc`*, such that*

$$\texttt{isNatc}\, e^{\#} \quad \Rightarrow \quad \begin{cases} \texttt{True}, & \textit{iff } e \Rightarrow n \textit{ for some canonical natural number } n \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Moreover, we have analogous versions of theorem 5 and 6:

Theorem 11 *There is no PCF-program* `isConstFunc` *such that*

$$\texttt{isConstFunc}\, e^{\#} \quad \Rightarrow \quad \begin{cases} \texttt{True}, & \textit{iff } e \textit{ is a constant function on natural numbers} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

Theorem 12 *There is no PCF-program* `eqFunc` *such that*

$$\texttt{eqFunc}\, f^{\#}\, f'^{\#} \quad \Rightarrow \quad \begin{cases} \texttt{True}, & \textit{if } f \textit{ and } f' \textit{ compute the same function on natural numbers} \\ \texttt{False}, & \textit{otherwise} \end{cases}$$

The proofs of these two theorems use the same ideas as the proofs of theorem 5 and 6. However, this idea needs to be adapted to codes for PCF-programs. The key step is that we need to show how to code programs with a free variable. We omit the details of these proofs.

2.5 PCF-computable functions

We shall now define when a partial function $g : \mathbf{N} \xrightarrow{\sim} \mathbf{N}$ is PCF-computable. By "function" we here mean "mathematical function", that is, a function in the set-theoretic sense. (Partial functions are defined in Definition 2.1 in KP and total functions on the following page.) Moreover, $\mathbf{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers. We shall distinguish between such natural numbers and their representatives in PCF, that is, the canonical elements of the type `Nat`, that is, $0 =$ `Zero`, $1 =$ `Succ Zero`, $2 =$ `Succ (Succ Zero)`, …. If $n \in \mathbf{N}$ then we write $\overline{n}$ :: `Nat` for its represenation in PCF.

With this notation we say that a PCF-program f *computes* the partial function $g : \mathbf{N} \xrightarrow{\sim} \mathbf{N}$ provided

$$f\ \overline{m} \Rightarrow \overline{n} \text{ iff } g(m) = n$$

for all $m, n \in \mathbf{N}$. Moreover, we say that g is PCF-computable provided there is *some* PCF-program f which computes it.

Similarly, if $\mathbf{Bool} = \{\mathit{true}, \mathit{false}\}$ are the truth-values, so that $\overline{\mathit{true}} =$ `True` and $\overline{\mathit{false}} =$ `False`, then the PCF-program f *computes* the partial function $g : \mathbf{N} \xrightarrow{\sim} \mathbf{Bool}$ provided

$$f\ \overline{m} \Rightarrow \overline{b} \text{ iff } g(m) = b$$

for all $m \in \mathbf{N}$ and $b \in \mathbf{Bool}$. Moreover, we say that g is PCF-computable provided there is *some* PCF-program f which computes it.

Before, we introduced the notation $e^{\#}$ for the code of the program e. But, since we pedantically distinguish between members of the set $\mathbf{N}$ and their reprsentations as members of the type `Nat`, we must distinguish between $e^{\#}$ :: `Nat` and the corresponding element $e^{\natural} \in \mathbf{N}$.

For example, using the notion of computable problem theorem 10 can be reformulated in the following way:

Theorem 13 *The function isNatc*

$$\mathit{isNatc}(e^{\natural}) \quad = \quad \begin{cases} \mathit{true}, & \mathit{iff}\ e \Rightarrow n \text{ for some canonical natural number } n \\ \mathit{false}, & \mathit{otherwise} \end{cases}$$

is not computable.

Proof. If *isNatc* is computable, then there is a PCF-program `isNatc` as in theorem 10. But we showed that this led to a contradiction.
□

Theorem 14 *The partial function pNatc*

$$\mathit{pNatc}(e^{\natural}) \quad = \quad \begin{cases} \mathit{true}, & \mathit{iff}\ e \Rightarrow n \text{ for some canonical natural number } n \\ \mathit{undefined}, & \mathit{otherwise} \end{cases}$$

is computable.

Proof. It is computed by the PCF-program

```
pNatc n = pNat (eval n)

pNat  e = if (isZero e) True (pNat (pred e))
```
□

A total function $g : \mathbf{N} \to \mathbf{Bool}$ is also called a *decision problem*. If g is computable then the problem is said to be *decidable*. So the problem of deciding whether a PCF-term reduces to a canonical natural number is an example of an *incomputable problem*. However, one says that this problem is *semi-decidable*, since the partial function *pNatc* is computable. The only difference between *pNatc* and *isNatc* is that *pNatc* is undefined whenever the argument does not reduce to a canonical canonical natural number, whereas *isNatc* is required to return *false* in that case.

The notions of *λ-definable function* (KP Definition 5.11-5.12), *Turing-machine computable function* (KP Definition 6.5), *recursive function* (KP Definition 6.7), *P-computable function* (KP Definition 3.15), etc. have definitions which are analogous to our definition of PCF-computable function.

We have the following theorem:

Theorem 15 *A partial function on natural numbers is PCF-computable iff it is λ-definable iff it is Turing-machine computable iff it is recursive iff it is P-computable.*

To prove this we need to show how to translate between programs (machines) in the different models in a meaning preserving way. This means for example that if the PCF-program f computes the partial function g and is translated into the Turing machine t, then t should also compute g. (This requirement on the translation is the general criterion for compiler correctness: the compiled target program should compute the same function as the source program.)

Church's thesis. Theorem 15 can be extended to many other models of computation. The belief that there is no good model of computation which gives rise to more computable functions than the λ-definable (etc) ones is called "Church's thesis" or "Church-Turing's thesis". This is not a theorem and can never become one, since there is no precise way to define what a "good" model of computation is. Nevertheless, Church's thesis is a meaningful statement. It is generally believed to be true, since noone has been able to refute it during the 60 years which have elapsed since it was first formulated.

2.6 Encoding PCF-programs as binary trees

If we add binary trees to PCF as in 1.5, then it becomes very easy to write a self-interpreter.

So let PCF+ be PCF extended with the type of binary trees and the constants `Leaf, Branch, isLeaf, left, right`. Similarly, we let CPCF+ be the combinatory version of PCF+.

The *CPCF+-terms* are generated by the following context-free grammar:

```
<cterm> ::= (<cterm> <cterm>) | k | s |
            True | False | Zero | Succ | Leaf | Branch |
            if | pred | isZero | isLeaf | left | right | peel |
            fix
```

We shall now show how to encode a CPCF+-program as a binary tree in CPCF+:

$$\begin{aligned}
\texttt{k}^{\#} &= \texttt{Leaf}\ 0\\
\texttt{s}^{\#} &= \texttt{Leaf}\ 1\\
\texttt{True}^{\#} &= \texttt{Leaf}\ 2\\
\texttt{False}^{\#} &= \texttt{Leaf}\ 3\\
\texttt{Zero}^{\#} &= \texttt{Leaf}\ 4\\
\texttt{Succ}^{\#} &= \texttt{Leaf}\ 5\\
\texttt{Leaf}^{\#} &= \texttt{Leaf}\ 6\\
\texttt{Branch}^{\#} &= \texttt{Leaf}\ 7\\
\texttt{if}^{\#} &= \texttt{Leaf}\ 8\\
\texttt{pred}^{\#} &= \texttt{Leaf}\ 9\\
\texttt{isZero}^{\#} &= \texttt{Leaf}\ 10\\
\texttt{isLeaf}^{\#} &= \texttt{Leaf}\ 11\\
\texttt{left}^{\#} &= \texttt{Leaf}\ 12\\
\texttt{right}^{\#} &= \texttt{Leaf}\ 13\\
\texttt{peel}^{\#} &= \texttt{Leaf}\ 14\\
\texttt{fix}^{\#} &= \texttt{Leaf}\ 15\\
(f\ a)^{\#} &= \texttt{Branch}\ f^{\#}\ a^{\#}
\end{aligned}$$

2.7 A self interpreter for (C)PCF+

We write the CPCF+ interpreter as a Haskell program and leave it to the reader to transform it to a CPCF+ program

```
eval (Leaf  0)      = k
eval (Leaf  1)      = s
eval (Leaf  2)      = True
eval (Leaf  3)      = False
eval (Leaf  4)      = Zero
eval (Leaf  5)      = Succ
eval (Leaf  6)      = Leaf
eval (Leaf  7)      = Branch
eval (Leaf  8)      = if
eval (Leaf  9)      = pred
eval (Leaf 10)      = isZero
eval (Leaf 11)      = isLeaf
eval (Leaf 12)      = left
eval (Leaf 13)      = right
eval (Leaf 14)      = peel
eval (Leaf 15)      = fix
eval (Branch as bs) = (eval as) (eval bs)
```

Remark. PCF+ is quite closely related to the language LISP. This is an untyped functional languages where all data is coded as binary trees, called S-expressions. There is an empty tree called `NIL` and our `Branch` is called `CONS`. Moreover, our `left` is called `CAR` and our `right` is called `CDR`. LISP also has a built in coding operation called `QUOTE` and a self-interpreter `EVAL`.