Implementing PCF in Idris

Alex Keizer Jeremy Kirn Simon Lemal

Saturday 5th February, 2022

Abstract

PCF is a small extension of the simply typed lambda calculus that can be thought of as a basic programming language. We give a brief introduction to PCF, including its small-step reduction rules, and then give the details of two of our implementations of PCF in Idris, a dependently typed programming language similar to Haskell. The second implementation more heavily utilizes the dependent types of Idris.

Contents

1 The Language PCF

PCF stands for "programming with computable functions." It is based on LCF, the "logic of computable functions," which was introduced by Dana Scott in [Sco69] as a restricted formalism for investigating computation, one better suited to this task than set theory. Gordon Plotkin then introduced PCF in [Plo75] in order to investigate the semantics of programming languages. He named this language PCF because it was inspired by the syntax of LCF. In what follows, we give a quick introduction to the language PCF. The reader may consult [Sel13] for a more detailed introduction.

1.1 Types and Terms

PCF is a programming language that is very similar to the simply-typed lambda calculus. Instead of type variables, it has concrete types **bool** and **nat**. It also has the classical type constructors $\to$, $\times$ and 1. Formally, types are defined by the following BNF:

$$A, B := \mathbf{bool} \mid \mathbf{nat} \mid A \to B \mid A \times B \mid 1.$$

The terms for PCF are those of the simply-typed lambda calculus, plus some concrete terms. In BNF:

$$\begin{aligned} M, N, P := x &\mid MN \mid \lambda x^A.M \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M \mid * \\ &\mid \mathbf{T} \mid \mathbf{F} \mid \mathbf{zero} \mid \mathbf{succ}\,(M) \mid \mathbf{pred}\,(M) \\ &\mid \mathbf{iszero}\,(M) \mid \mathbf{if}\, M\, \mathbf{then}\, N\, \mathbf{else}\, P \mid \mathbf{Y}(M). \end{aligned}$$

The **Y** constructor returns a fixpoint of the given term. It is what makes the language Turing complete. Without adding **Y** to PCF, we would be able to prove that every well-typed term eventually reduces to a term that cannot be reduced further. That is, without **Y**, every program written in PCF halts, but there are computable functions that do not halt on some inputs, so PCF without **Y** cannot be Turing complete. An example of how **Y** produces non-halting PCF programs is given by $\mathbf{Y}(\lambda x^{\mathbf{nat}}.\mathbf{succ}\ x)$, whose reduction never halts:

$$\mathbf{Y}(\lambda x^{\mathbf{nat}}.\mathbf{succ}\ x) \to^* \mathbf{succ}(\mathbf{Y}(\lambda x^{\mathbf{nat}}.\mathbf{succ}\ x)) \to^* \mathbf{succ}(\mathbf{succ}(\mathbf{Y}(\lambda x^{\mathbf{nat}}.\mathbf{succ}\ x))) \to^* \ldots.$$

One may think of **Y** as a way to introduce recursion into PCF. For example, the addition function is most naturally defined by recursion, and it can be defined using **Y** as follows:

$$+ := \mathbf{Y}(\lambda f^{\mathbf{nat}\to\mathbf{nat}} m^{\mathbf{nat}} n^{\mathbf{nat}}.\mathbf{if}\ \mathbf{iszero}(m)\ \mathbf{then}\ n\ \mathbf{else}\ \mathbf{succ}(f\mathbf{pred}(m)n)).$$

1.2 Typing Rules

The reader may notice that among the well-formed terms, some do not make any sense. Such terms are, for example, $\mathbf{iszero}(\mathbf{T})$ or $\pi_1(\lambda x^{\mathbf{nat}}.x)$. We should thus specify a type for each term and specify which constructions can be applied to which types. We start by giving the typing rules for the fragment of PCF that is the simply-typed lambda calculus with finite products:

$$\frac{}{\Gamma \vdash * : 1} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B}$$

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A}$$

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

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B.}$$

We now give the typing rules for the rest of PCF:

$$\frac{}{\Gamma \vdash \mathbf{T} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{F} : \mathbf{bool}}$$

$$\frac{}{\Gamma \vdash \mathbf{zero} : \mathbf{nat}} \qquad \frac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash \mathbf{succ}(M) : \mathbf{nat}}$$

$$\frac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash \mathbf{pred}(M) : \mathbf{nat}} \qquad \frac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash \mathbf{iszero}(M) : \mathbf{bool}}$$

$$\frac{\Gamma \vdash M : \mathbf{bool} \qquad \Gamma \vdash N : A \qquad \Gamma \vdash P : A}{\Gamma \vdash \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P : A} \qquad \frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathbf{Y}(M) : A.}$$

1.3 Reduction

PCF is a programming language in which PCF-terms are programs. How do we run such a program? We reduce it to its simplest form. Note that the syntax of a PCF-term is not sufficient to determine its reduction behavior. For example, we have two sensible ways to reduce the term $\mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero}))$, as shown below:

$$\begin{tikzcd}
 & \mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero})) \arrow[dl] \arrow[dr] & \\
(\lambda x^{\mathbf{nat}}.x)\mathbf{zero} \arrow[dr] & & \mathbf{pred}(\mathbf{succ}(\mathbf{zero})) \arrow[dl] \\
 & \mathbf{zero} &
\end{tikzcd}$$

In order to view PCF-terms as programs, we need to specify a reduction order for PCF-terms so that they can be run deterministically. This reduction order is what makes PCF a programming language, rather than just a calculus like the simply-typed lambda calculus. Below, we specify this reduction order in in the small-step reduction style, while noting that it is also possible to do this in a big-step reduction style.

Small-step reduction specifies how to reduce a PCF-term one step at a time. For example, the reduction $\mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero})) \to \mathbf{pred}(\mathbf{succ}(\mathbf{zero}))$ is one small-step reduction that the rules below specify. If we reduce once more, $\mathbf{pred}(\mathbf{succ}(\mathbf{zero})) \to \mathbf{zero}$, we arrive at the *value* $\mathbf{zero}$, which we think of as the result of running the program $\mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero}))$. In general, the form of values is given by the following BNF, where M and N are any PCF-terms:

$$V := \mathbf{T} \mid \mathbf{F} \mid \mathbf{zero} \mid \mathbf{succ}(V) \mid * \mid \langle M, N \rangle \mid \lambda x^A.M.$$

We begin by giving the small-step reduction rules for the fragment of PCF that is the simply-typed lambda calculus with finite products:

$$\frac{M : 1 \qquad M \neq *}{M \to *} \qquad\qquad \frac{}{\pi_1\langle M_1, M_2\rangle \to M_1}$$

$$\frac{}{(\lambda x^A.M)N \to M[N/x]} \qquad\qquad \frac{}{\pi_2\langle M_1, M_2\rangle \to M_2}$$

$$\frac{M \to N}{MP \to NP} \qquad\qquad \frac{M \to N}{\pi_i M \to \pi_i N.}$$

We now give the small-step reduction rules for the rest of PCF:

$$\frac{}{\mathbf{Y}(M) \to M(\mathbf{Y}(M))} \qquad\qquad \frac{}{\mathbf{pred}(\mathbf{succ}(V)) \to V}$$

$$\frac{M \to N}{\mathbf{succ}(M) \to \mathbf{succ}(N)} \qquad\qquad \frac{}{\mathbf{iszero}(\mathbf{zero}) \to \mathbf{T}}$$

$$\frac{}{\mathbf{pred}(\mathbf{zero}) \to \mathbf{zero}}$$

$$\frac{M \to N}{\mathbf{pred}(M) \to \mathbf{pred}(N)} \qquad\qquad \frac{M \to N}{\mathbf{iszero}(M) \to \mathbf{iszero}(N)}$$

$$\frac{}{\mathbf{iszero}(\mathbf{succ}(V)) \to \mathbf{F}} \qquad \frac{}{\mathbf{if\ F\ then}\ N\ \mathbf{else}\ P \to P}$$

$$\frac{}{\mathbf{if\ T\ then}\ N\ \mathbf{else}\ P \to N}$$

$$\frac{M \to M'}{\mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ P \to \mathbf{if}\ M'\ \mathbf{then}\ N\ \mathbf{else}\ P.}$$

Small-steps can be concatenated. For example, the small-steps $\mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero})) \to \mathbf{pred}(\mathbf{succ}(\mathbf{zero})) \to \mathbf{zero}$ can be concatenated to $\mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero})) \to^* \mathbf{zero}$, where $\to^*$ denotes finitely many small-step reductions. Since **zero** is a value, we say that $\mathbf{pred}(\mathbf{succ}((\lambda x^{\mathbf{nat}}.x)\mathbf{zero}))$ *evaluates* to **zero**. In general, if $M \to^* V$, where V is a value, we say that M evaluates to V.

The notion of evaluating to a value is what gives us the big-step style of specifying reduction order. We can give rules that directly axiomatize big-step reduction, usually denoted by $\Downarrow$, with the intention that $\Downarrow$ coincides with evaluating to a value via $\to^*$. We will not give the rules for $\Downarrow$, however, because our implementation of big-step reduction in Idris closely follows the idea that a big-step is composed of multiple small-steps that result in a value.

2 A Prime Implementation of PCF

In this section, we explore a way of implementing PCF in Idris. The use of dependent types in this approach is quite mild, and it should not be too dissimilar to how PCF might be implemented in Haskell.

2.1 Implementing Types

Defining a data type for PCF types is just a matter of translating the formal definition. We use the symbol `~>` to represent the arrow constructor.

```
public export
data PCFType = PCFBool
             | PCFNat
             | PCFUnit
             | (~>) PCFType PCFType
             | (*) PCFType PCFType

infixr 10 ~>
```

We want our types to be comparable. Sadly, Idris does not have an equivalent of Haskell's "deriving" statement, so we'll have to implement equality ourselves. We omit the details here and in any other similarly trivial implementation blocks.

We also want our types to be displayed nicely. Once again, we do not display this rather trivial implementation of the `show` function.

2.2 Implementing Terms

We now want to define terms. We use de Bruijn indices to represent bound variables. This is an elegant way to deal with α-equivalence.

```
public export
Var : Nat -> Type
Var = Fin
```

Instead of having a ton of different constructors, we define a symbol data type and then define a single constructor that works for all symbols.

```
namespace Symbol
  public export
  data Symbol : (0 ar : Nat) -> Type where
    IfElse : Symbol 3       -- if-then-else construct
    App    : Symbol 2       -- application
    Pair   : Symbol 2       -- pairing
    Fst    : Symbol 1       -- first projection
    Snd    : Symbol 1       -- second projection
    Succ   : Symbol 1       -- successor
    Pred   : Symbol 1       -- predecessor
    IsZero : Symbol 1       -- is zero predicate
    Y      : Symbol 1       -- fixpoint / Y-combinator
    T      : Symbol 0       -- true
    F      : Symbol 0       -- false
    Zero   : Symbol 0       -- zero value
    Unit   : Symbol 0       -- unit value (*)
```

We also keep track of (an upper bound on) the number of free variables in the type: `PCFTerm n` encodes terms with at most n free variables.

```
public export
data PCFTerm : Nat -> Type where
  V    : Var k -> PCFTerm k                             -- variables
  L    : PCFType    -> PCFTerm (S k) -> PCFTerm k         -- lambda
  S    : Symbol ar -> Vect ar (PCFTerm k) -> PCFTerm k  -- other symbols
```

Of special interest are the closed terms, those without any free variables

```
public export
ClosedPCFTerm : Type
ClosedPCFTerm = PCFTerm 0
```

Remember that the type only gives an upper bound, so an inhabitant of say `PCFType 3` might still be closed.

The following will try to strengthen any such term. This really is just a wrapper around `Fin.strengthen`, with straightforward recursive cases, so we detail only variables and lambdas.

```
strengthen : {k : _} -> PCFTerm (S k) -> Maybe (PCFTerm k)
strengthenVect : {k : _} -> Vect n (PCFTerm (S k)) -> Maybe (Vect n (PCFTerm k))
```

The `strengthenVect` function is useful for the symbol case.

```
strengthen (V v)    = Fin.strengthen v >>= Just . V
strengthen (L t m)  = strengthen m     >>= Just . L t
strengthen (S s ar) = Just (S s !(strengthenVect ar))

strengthenVect (m::ms) = [| (strengthen m) :: (strengthenVect ms) |]
strengthenVect []      = Just []
```

```
public export
tryClose : {k : _} -> PCFTerm k -> Maybe ClosedPCFTerm
tryClose {k} t = case k of
                  0      => Just t
                  (S k') => strengthen t >>= tryClose
```

As for types, we want terms to be comparable. The important case is lambda abstraction. We are using de Bruijn indices, which make comparing terms very easy. We once again skip the other trivial cases.

```
public export partial
implementation Eq (PCFTerm k) where
  V v          == V w          = v == w
  L a m        == L b n        = a == b && m == n
```

We can also implement a `show` function. The implementation in itself is not interesting, so we omit the details.

2.3 Type Checking

We are now ready to define a type infering function. Such a function takes as arguments a context and a term, and it returns a type if the term is typeable in the given context and `Nothing` otherwise.

We've been keeping track of free variables in the type of terms, so we'd like to restrict to contexts that actually provide a type for all (potential) free variables

```
public export
Context : Nat -> Type
Context n = Vect n PCFType
```

Type checking occurs in two mutually recursive functions. `TypeOf` infers the type of a single term, while `typeOfVect` handles lists of terms.

```
public export
total typeOf : Context k -> PCFTerm k -> Maybe PCFType
total typeOfVect : Context k -> Vect n (PCFTerm k) -> Maybe (Vect n PCFType)

typeOf con (V v)      = Just (index v con)
typeOf con (L t m)    = typeOf (t::con) m >>= Just . ( t ~> )
typeOf con (S s ms)   = case (s,  !(typeOfVect con ms)) of
  (IfElse,  [PCFBool, a, b]) => if a == b
                                  then Just a
                                else Nothing
  (App,     [(a ~> b), c])   => if a == c
                                  then Just b
                                else Nothing
  (Pair,    [a, b])          => Just (a * b)
  (Fst,     [a * _])         => Just a
  (Snd,     [_ * b])         => Just b
  (Succ,    [PCFNat])        => Just PCFNat
  (Pred,    [PCFNat])        => Just PCFNat
  (IsZero,  [PCFNat])        => Just PCFBool
  (Y,       [a ~> b])        => if a == b
                                  then Just a
                                else Nothing
  (T,       [])              => Just PCFBool
  (F,       [])              => Just PCFBool
  (Zero,    [])              => Just PCFNat
  (Unit,    [])              => Just PCFUnit
  (_, _)                     => Nothing

typeOfVect x (y :: ys) = [| (typeOf x y) :: (typeOfVect x ys) |]
typeOfVect _ []        = Just []
```

Closed terms are typeable exactly when they are typeable with an empty context.

```
public export
typeOfClosed : ClosedPCFTerm -> Maybe PCFType
typeOfClosed = typeOf []
```

2.4 Substitution

In order to define small-step reduction, we must be able to substitute a term for a variable in another term. We only allow the maximal variable, indicated `k` in the following signature, to be substituted, so that we can decrease that upper bound by one for the return type and maintain a sharp upper bound.

When substituting a term inside another, we might need to rename (increase) free variables. The following function does this. The depth argument keeps track of how many lambda's have been encoutered, while the types reflect that the upper bound on free variables also increases.

Because of the totality checker, we have to give a `*Vect` version of the function as well, following the same pattern as we did while type checking.

```
total incFreeVar : Nat -> PCFTerm k -> PCFTerm (S k)
total incFreeVarVect : Nat -> Vect n (PCFTerm k) -> Vect n (PCFTerm (S k))

incFreeVar depth (V v)    = if (finToNat v) < depth
                              then (V (weaken v))
                            else (V (FS v))
incFreeVar depth (L t m)  = L t (incFreeVar (S depth) m)
incFreeVar depth (S s ms) = S s (incFreeVarVect depth ms)

incFreeVarVect depth (m::ms) = (incFreeVar depth m) :: (incFreeVarVect depth ms)
incFreeVarVect _     []      = []
```

We now define substitution.

```
public export
total substitute : {k : _} -> PCFTerm k -> PCFTerm (S k) -> PCFTerm k
total substituteVect : {k : _} -> PCFTerm k -> Vect n (PCFTerm (S k)) -> Vect n (PCFTerm k)
```

We try to strengthen (i.e., decrement) the bound on the variable index. The only reason for this to fail is if the index is already at the upper bound; if `w == k`, thus if strengthening fails, we should substitute

```
substitute s (V v) =  case Fin.strengthen v of
                        Nothing => s
                        Just w  => V w
```

Recall that the body of a lambda has one more (potential) free-variable, thus the upper bound is automatically incremented

```
substitute s (L t m) = L t (substitute (incFreeVar 0 s) m)
```

All the other cases are straightforward, once again, the substitution is just passed on.

```
substitute s (S sym ms) = S sym (substituteVect s ms)

substituteVect s (m::ms) = (substitute s m) :: (substituteVect s ms)
substituteVect _ [] = []
```

2.5 Reduction

We can now define reduction. We begin with small-step reduction.

2.5.1 Small-step Reduction

Not all terms can reduce, it is thus important that the result is of type `Maybe PCFTerm`.

```
public export
smallStep : {k : _} -> PCFTerm k -> Maybe (PCFTerm k)
```

Neither variables nor top-level lambdas are reducable, so we restrict attention to the symbols

```
smallStep (S s arg) = ssSym s arg where
  ssSym : Symbol ar -> Vect ar (PCFTerm k) -> Maybe (PCFTerm k)
```

We start with the top-level reducts. It is worth mentioning that the reduction rule for **pred** is slightly different from the one exposed in section 1.

```
  ssSym Pred [S Zero []]      = Just $ S Zero []
  ssSym Pred [S Succ [m]]     = Just m

  ssSym IsZero [S Zero _]     = Just $ S T []
  ssSym IsZero [S Succ _]     = Just $ S F []

  ssSym App [(L _ m), n]      = Just (substitute n m)

  ssSym Fst [S Pair [m, _]]   = Just m
  ssSym Snd [S Pair [_, n]]   = Just n

  ssSym IfElse [S T _, m, _] = Just m
  ssSym IfElse [S F _, _, n] = Just n

  ssSym Y [m]                 = Just $ S App [m, (S Y [m])]
```

If a term had no top-level reducts, then we try to reduce its first argument. The only exception is `Pair`, which does not evaluate its arguments at all.

```
  ssSym Pair _     = Nothing
  ssSym s (m::ms) = Just $ S s (!(smallStep m) :: ms)
```

Alternatively, if the term is of the unit type, but not the unit value itself, we reduce it to the unit value

```
  ssSym Unit _ = Nothing
  ssSym s arg  = let m = (S s arg) in
                 case tryClose m >>= typeOfClosed of
                   Just PCFUnit => Just (S Unit [])
                   _            => Nothing

smallStep _ = Nothing
```

2.5.2 Values

A certain subset of terms are called *values*. Those include constants, successors of values, pairs and abstractions.

```
public export
data PCFValue : Nat -> Type where
  C     : Symbol 0   -> PCFValue k
  Succ  : PCFValue k -> PCFValue k
  Pair  : PCFTerm k  -> PCFTerm k     -> PCFValue k
  L     : PCFType    -> PCFTerm (S k) -> PCFValue k
```

Of course we are mainly interested in closed values.

```
public export
ClosedPCFValue : Type
ClosedPCFValue = PCFValue 0
```

Some terms are values. We want to be able to convert those into the `PCFValue` type.

```
public export
toValue : PCFTerm k -> Maybe (PCFValue k)
toValue (L t m)         = Just $ L t m
toValue (S s [])        = Just $ C s
toValue (S Succ [m])    = Just $ Succ !(toValue m)
toValue (S Pair [m, n]) = Just $ Pair m n
toValue _               = Nothing
```

Conversely, all values are terms, and we must be able to convert `PCFValues` to `PCFTerms`.

```
public export
toTerm : PCFValue k -> PCFTerm k
toTerm (C s)      = S s []
toTerm (Succ v)   = S Succ [toTerm v]
toTerm (Pair m n) = S Pair [m, n]
toTerm (L t m)    = L t m
```

Finally, we want to check whether or not a term is a value.

```
public export
isValue : PCFTerm k -> Bool
isValue (L t m)         = True
isValue (S s [])        = True
isValue (S Succ [m])    = isValue m
isValue (S Pair [m, n]) = True
isValue _               = False
```

Values are exactly the normal forms for small-step reduction, that is, values are the terms that cannot be reduced further.

2.5.3 Big-step Reduction

By successively applying small-step reductions, terms can reduce to values. This is the so-called big-step reduction.

Not all terms evaluate to a value, some may enter an infinite loop. To still define `eval` as a total function, we supply it with some "fuel". This fuel acts as an upper bound on computation steps. For well-typed terms, the function will only return `Nothing` if this upper bound was reached.

```
public export
eval : Fuel -> ClosedPCFTerm -> Maybe ClosedPCFValue
```

To begin with, values reduce to themselves.

```
eval _ (L t m)         = Just $ L t m
eval _ (S Pair [m, n]) = Just $ Pair m n
eval _ (S s [])        = Just $ C s
```

Terms starting with **Y** are the terms that might not terminate. Evaluating such terms costs fuel.

```
eval (More f) (S Y [m]) = eval f (S App [m, S Y [m]])
```

For other cases, we just do one step of computation then keep evaluating.

```
eval f (S s (m :: ms)) = evSym s !(eval f m) ms where
  evSym : Symbol (1 + ar) -> ClosedPCFValue -> Vect ar ClosedPCFTerm -> Maybe
      ClosedPCFValue
  evSym IfElse  (C T)      [m, _] = eval f m
  evSym IfElse  (C F)      [_, n] = eval f n
  evSym Pred    (C Zero)   _      = Just $ C Zero
  evSym Pred    (Succ m)   _      = Just m
  evSym IsZero  (C Zero)   _      = Just $ C T
  evSym IsZero  (Succ _)   _      = Just $ C F
  evSym App     (L _ m)    [n]    = eval f (substitute n m)
  evSym Fst     (Pair m _) _      = eval f m
  evSym Snd     (Pair _ n) _      = eval f n
  evSym s       v          ms     = let m = (S s ((toTerm v) :: ms)) in  -- take the
      evaluated first arg, and recombine into a term m
                  case typeOfClosed m of
                    Just PCFUnit => Just (C Unit)
                    _            => toValue m
```

Everything that reaches that last clause will evaluate to itself, but only if it is representable as a value. This should only be the case for all well-typed terms that reach this point (in fact, every well-typed term that reaches this clause should be successor values). Notice that the argument `m` is evaluated first.

3 A Second Implementation of PCF

So far, we used a definition of terms that allowed us to express "terms with at most n free variables" on the type level. In the following section, we will illustrate how we can use dependent types to give a type such that its inhabitants are always well-typed terms, or even terms of a specific (PCF) type.

The construction goes as follows: `TermOfType k con type`, where `k` is the bound on free variables, `con` a context with k types and `type` a `PCFType`, should represent all terms `t` such that `typeOf con t == Just type`. The definition has the same constructors as standard `PCFTerms`, but now the typing rules are also incorporated into the signatures.

```
public export
data TermOfType : {k : Nat} -> (con : Context k) -> (0 t : PCFType) -> Type where
    V    : (v : Var _)  -> TermOfType con (index v con)             -- variables
    App  : TermOfType con (t1 ~> t2)  -> TermOfType con t1          -- application
           -> TermOfType con t2
    L    : (t1 : PCFType)         -> TermOfType (t1 :: con) t2     -- lambda abstraction
           -> TermOfType con (t1 ~> t2)
    Pair : TermOfType con t1 -> TermOfType con t2                  -- pairing
           -> TermOfType con (t1 * t2)
    Fst   : TermOfType con (t1 * _)                                -- first projection
           -> TermOfType con t1
    Snd   : TermOfType con (_ * t2)                                -- second projection
           -> TermOfType con t2
    T    : TermOfType _ PCFBool                                    -- true
    F    : TermOfType _ PCFBool                                    -- false
    Zero : TermOfType _ PCFNat                                     -- zero value
    Succ : TermOfType c PCFNat -> TermOfType c PCFNat              -- successor
    Pred : TermOfType c PCFNat -> TermOfType c PCFNat              -- predecessor
    IsZero : TermOfType c PCFNat -> TermOfType c PCFBool           -- is zero predicate
    IfElse : TermOfType c PCFBool -> TermOfType c t -> TermOfType c t
                 -> TermOfType c t
    Y     : TermOfType c (t ~> t) -> TermOfType c t                -- fixpoint / Y-
        combinator
    Unit  : TermOfType c PCFUnit                                   -- unit value (*)
```

Often we don't want to specify the actual type, we just want to know the term has *some* type. A `TypedTerms` is a dependent pair of a type and a term of that type.

```
public export
TypedTerm : {k : Nat} -> (con : Context k) -> Type
TypedTerm con = DPair PCFType (\t => TermOfType con t)
```

From the description so far, one might (wrongly) expect that the following two signatures are equivalent.

```
-- foo : TermOfType con type -> ...
-- bar : TypedTerm con    -> ...
```

After all, the `type` in `foo` is universally quantified, so both functions take well-formed terms of arbitrary types. The difference is subtle: in `foo` the type exists purely in the type-checker and will be erased, at run-time `foo` cannot inspect `type`. Conversely, in `bar` the type is stored in the dependent pair and is thus fully available at run-time.

For both notions, we give an alias for closed terms.

```
public export
ClosedTermOfType : (0 _ : PCFType) -> Type
ClosedTermOfType = TermOfType []

public export
ClosedTypedTerm : Type
ClosedTypedTerm = TypedTerm []
```

Type checking now means to translate a `PCFTerm` to a `TypedTerm`.

```
public export
typeCheck : (con : Context k) -> PCFTerm k -> Maybe (TypedTerm con)

public export
typeCheckVect : (con: Context k) -> Vect n (PCFTerm k) -> Maybe (Vect n (TypedTerm con))
typeCheckVect x (y :: ys) = [| (typeCheck x y) :: (typeCheckVect x ys) |]
typeCheckVect _ []        = Just []

||| A useful alias that will automatically infer the type
JustT : {type : PCFType} -> TermOfType con type -> Maybe (TypedTerm con)
JustT m = Just (type ** m)

typeCheck con (V v)    = JustT (V v)
typeCheck con (L t m)  = JustT (L t (snd !(typeCheck (t::con) m) ))
typeCheck con (S s ms) = case ( s,  !(typeCheckVect con ms) ) of
```

DecEq is an interface (cf. Haskells type-classes) for “decidable equality”. That is, the implementation does not only return whether two things are equal, it should provide a proof of (in)equality. The implementation of DecEq for PCFType is quite straightforward, the interested reader can find it in the Lib.Types.DecEq module.

Recall that an IfElse term is well-typed, if the then-branch and else-branch have the same type. To construct a TypedTerm with the IfElse constructor, we have to convince the type-checker of this equality, and indeed that is where the proof provided by decEq is used.

```
  (IfElse,  [(PCFBool ** p), (a ** m), (b ** n)])
      => case (decEq a b) of
            Yes eq => let n = (rewrite eq in n) in JustT (IfElse p m n)
            No  _  => Nothing
  (App,     [((a ~> b) ** m), (c ** n)])
      => case (decEq a c) of
            Yes eq => let n = (rewrite eq in n) in JustT (App m n)
            No  _  => Nothing
  (Pair,    [(_ ** m), (_ ** n)] )  => JustT (Pair m n)
  (Fst,     [((_ * _) ** m)])       => JustT (Fst m)
  (Snd,     [((_ * _) ** m)])       => JustT (Snd m)
  (Succ,    [(PCFNat ** m)])        => JustT (Succ m)
  (Pred,    [(PCFNat ** m)])        => JustT (Pred m)
  (IsZero,  [(PCFNat ** m)])        => JustT (IsZero m)
  (Y,       [(a ~> b ** m)])        => case (decEq a b) of
                                         Yes eq => Just (a ** (Y (rewrite cong (a ~> ) eq
                                             in m)))
                                         No  _  => Nothing
  (T,       [])                     => JustT T
  (F,       [])                     => JustT F
  (Zero,    [])                     => JustT Zero
  (Unit,    [])                     => JustT Unit
  (_, _)                            => Nothing
```

```
public export
typeCheckClosed : ClosedPCFTerm -> Maybe (TypedTerm [])
typeCheckClosed m = typeCheck [] m
```

```
public export
typeOf : TypedTerm k -> PCFType
typeOf = fst
```

Now we have two notions of typeability: the original typeOf function and this new typeCheck function. The following “function” expresses this equality, and indeed giving a definition that satisfies the type checker would be a proof.

```
-- typeOfMatchesCheck : (con : Context k) -> (m : PCFTerm k)
--                         -> (typeOf con m) = (typeCheck con m >>= Just . TypedTerms.typeOf
    )
```

However, Idris is explicitly not a proof assistant, and lacks many of the features that helps users write proofs in such tools. As such, we were unsuccessful in our attempts to prove the above in Idris.

Of course, every well-typed term is still a term, so we can forget the extra structure and go back to the more general PCFTerm type.

```
public export
forgetType : {con: Context k} -> TermOfType con t -> PCFTerm k
forgetType (V v)      = V v
forgetType (L t1 x)   = L t1 (forgetType x)
forgetType (App x y)  = S App   [forgetType x, forgetType y]
forgetType (Pair x y) = S Pair  [forgetType x, forgetType y]
forgetType (Fst x)    = S Fst   [forgetType x]
forgetType (Snd x)    = S Snd   [forgetType x]
forgetType (Succ x)   = S Succ  [forgetType x]
forgetType (Pred x)   = S Pred  [forgetType x]
forgetType (IsZero x) = S IsZero [forgetType x]
forgetType (Y x)      = S Y     [forgetType x]
forgetType (IfElse x y z) = S IfElse [forgetType x, forgetType y, forgetType z]
forgetType T          = S T []
forgetType F          = S F []
forgetType Zero       = S Zero []
forgetType Unit       = S Unit []
```

3.1 Substitution

Substitution has the nice property that it preserves types, as long as the substituted term has the same type as the variable to be substituted had. Similarly, if term m of type t small-step reduces to term n, then the latter also is of type t.

This is true of our first implementation, but merely as an artifact of the particular way we choose to implement the functions. We could have just as well given a different (wrong) definition,which does not have this property, but still fits the type signature and would thus be accepted by Idris. Using the new definitions of the current section, however, allows us to be more precise in the type signatures.

```
substitute : TermOfType con t -> TermOfType (t::con) u -> TermOfType con u
smallStep  : TermOfType con t -> Maybe (TermOfType con t)
```

Now any implementation should also carry a proof of type preservation, and any other code knows for sure that types are preserved. Sadly, this proof-burden does make it much harder to provide an implementation, and we were not able to.

References

[Plo75] Gordon David Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 1975.

[Sco69] Dana Stewart Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 1969.

[Sel13] Peter Selinger. Lecture notes on the lambda calculus, 2013.