Typed Lambda Calculus with Fixed Points

Robin Cockett

November 16, 2005

This document describes some basic type theories which can be built on top of the λ-calculus. Perhaps one of the most basic is the type theory which underlies most functional style programming languages: this is a simply typed calculus with fixed points and some basic programming "features". We shall illustrate three basic programming features: pairs, the natural numbers, and lists. With these added features one obtains a reasonable programming language.

1 The simply typed λ-calculus with fixed points

The simply typed λ-calculus with fixed points and some basic programming features is sometimes called **PCF** (Programming language for Computable Functions). We shall introduce a variant of this language in which we use the case construct on natural numbers and lists rather than booleans with a conditional construct and the natural numbers with a test for being zero. Our language, consequently, is a slightly more usable than the standard PCF.

1.1 The basic judgments

Below, in table 1, are the inference rules for the typed λ calculus with fixed points: often a sequent, such as

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

which can be proved in a type theory such as this is referred to as a **type judgment**. It is a proof, after all, that a term may be typed in a particular way.

We have introduced a slightly modified syntax to handle the fixed points. This means that we do not have an explicit fixed point operator Y in the language. However, see the third example below, we can construct a fixed point operator.

Here some basic examples of typing judgments:

Example 1.1 (Simple typing judgments)

(1) One of the simplest terms is the identity function which may be typed as $\lambda x.x :: A \longrightarrow A$:

$$\cfrac{\cfrac{}{x :: A \vdash x :: A}\ \text{prj}}{\vdash \lambda x.x :: A \longrightarrow A}\ \text{abst}$$

$$\frac{}{x :: P, z :: \Gamma \vdash x :: P}\ \text{prj}$$

$$\frac{x :: P, z :: \Gamma \vdash t :: Q}{z :: \Gamma \vdash \lambda x.t :: P \longrightarrow Q}\ \text{abst}$$

$$\frac{z :: \Gamma \vdash f :: P \longrightarrow Q \quad z :: \Gamma \vdash t :: P}{z :: \Gamma \vdash (ft) :: Q}\ \text{app}$$

$$\frac{z :: \Gamma \vdash t :: Q \longrightarrow Q}{z :: \Gamma \vdash \mathsf{fix}[t] :: Q}\ \text{fix}$$

Table 1: Annotate type system

(2) The next simplest are the curried projections. The first projection has a typing judgment as $\lambda xy.x :: A \longrightarrow B \longrightarrow A$:

$$\cfrac{\cfrac{\cfrac{}{x :: A, y :: B \vdash x :: A}\ \text{prj}}{x :: A \vdash \lambda y, x :: B \longrightarrow A}\ \text{abst}}{\vdash \lambda x.xy :: A \longrightarrow B \longrightarrow A}\ \text{abst}$$

(3) Here is a fixed point combinator in this calculus with its typing judgment:

$$\cfrac{\cfrac{\cfrac{}{x :: A \longrightarrow A \vdash x :: A \longrightarrow A}\ \text{prj}}{x :: A \longrightarrow A \vdash \mathsf{fix}[x] :: A}\ \text{fix}}{\vdash \lambda x.\mathsf{fix}[x] :: (A \longrightarrow A) \longrightarrow A}\ \text{abst}$$

$\square$

Notice that there is something rather unsatisfactory about these proofs as where did the types A and B originally come from? The assumption behind this definition is that one has a predefined set of types from which one can choose to form a projection judgment. Formally therefore we should define this set. This can be done inductively by

$$\mathsf{type} = \mathsf{type} \longrightarrow \mathsf{type} \mid A \mid B \mid ...$$

where we allow upper case letters as basic types. Note if we do not have some basic types we cannot get going! We will replace these basic types shortly by the types introduced by the programming features we shall add.

Notice, however, with these judgments the term $\lambda x.x$ has many possible typing judgments:

$$\lambda x.x :: B \longrightarrow B \quad \lambda x.x :: (A \longrightarrow B) \longrightarrow A \longrightarrow B \quad ....$$

The effect of adding programming features is to narrow down what is a legal type as it must be built from the types introduced by these features. However, it does not alleviate the above problem, namely that a given term can have many type judgments. This problem will be revisited in the section on type inference.

$$\frac{z :: \Gamma \vdash t :: P \quad z :: \Gamma \vdash s :: Q}{z :: \Gamma \vdash (t,s) :: P \times Q}\ \text{pair}$$

$$\frac{z :: \Gamma \vdash t : P \times Q \quad z :: \Gamma, x :: P, y :: Q \vdash s :: W}{z :: \Gamma \vdash \begin{array}{l}\mathsf{case}\ t\\ \ \mathsf{of}\ (x,y) \mapsto s\end{array} :: W}\ \text{pcase}$$

$$\frac{}{z :: \Gamma \vdash () :: 1}\ \text{unit}$$

$$\frac{z :: \Gamma \vdash t : 1 \quad z :: \Gamma \vdash s :: W}{z :: \Gamma \vdash \begin{array}{l}\mathsf{case}\ t\\ \ \mathsf{of}\ () \mapsto s\end{array} :: W}\ \text{ucase}$$

Table 2: Judgments for products

Remark 1.2 (Syntax for defining recursive functions)

There is an alternative syntax for defining recursive programs which is reminiscent of recursive programming:

$$f := t.N \equiv (\lambda f.N)\mathsf{fix}[\lambda f.t]$$

Notice that we could equivalently have had a type judgment

$$\frac{f :: Y, z :: \Gamma \vdash N :: X \quad f :: Y, z :: \Gamma \vdash t : Y}{z :: \Gamma \vdash f := t.N :: X}\ \text{recdef}$$

and have defined

$$\mathsf{fix}[t] \equiv f := tf.(\lambda x.fx).$$

□

1.2 Judgments for products

One of the most basic programing constructs is the product: the ability to pair terms and to pull them apart. To introduce products (or pairing at the level of terms) we use the judgments in table 2 below. They introduce two new syntactic constructs: pairing and the case on a product.

This allows us to create some new terms. Here are some examples of type judgments using these constructs:

Example 1.3 (Simple typing judgments)

(1) We may express the projections from a pair using the following definitions:

$$\pi_0 \equiv \lambda z. \left(\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y) \mapsto x\end{array} \right)$$

$$\pi_0 \equiv \lambda z.\left(\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y)\mapsto y\end{array}\right)$$

we derive a typing judgment for the first:

$$\dfrac{\dfrac{}{z :: A\times B \vdash z :: A\times B}\ \mathrm{prj} \quad \dfrac{}{z :: A\times B, x :: A, y :: B \vdash x :: A}\ \mathrm{prj}}{\dfrac{z :: A\times B \vdash \begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y)\mapsto x\end{array} :: A}{\vdash \lambda z.\left(\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y)\mapsto x\end{array}\right) :: A\times B \longrightarrow A}\ \mathrm{app}}\ \mathrm{pcase}$$

(2) Another simple term is the symmetry map on the product which swaps the components of a pair. Here is the derivation of a typing judgment for it:

$$\dfrac{\dfrac{}{z :: A\times B \vdash z :: A\times B}\ \mathrm{prj} \quad \dfrac{\dfrac{}{z :: A\times B, x :: A, y :: B \vdash y :: B}\ \mathrm{prj} \quad \dfrac{}{z :: A\times B, x :: A, y :: B \vdash x :: A}\ \mathrm{prj}}{z :: A\times B, x :: A, y :: B \vdash (y,x) :: B\times A}\ \mathrm{pair}}{\dfrac{z :: A\times B \vdash \begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y)\mapsto (y,x)\end{array} :: B\times A}{\vdash \lambda z.\left(\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y)\mapsto (y,x)\end{array}\right) :: A\times B \longrightarrow B\times A}\ \mathrm{app}}\ \mathrm{pcase}$$

$\square$

Notice that an alternative approach to this is to use the projections to define the case construct:

$$\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y)\mapsto t\end{array} \equiv (\lambda xy.t)(\pi_0 z)(\pi_1 z).$$

An interesting question is to determine whether these are actually equal by using the rewriting semantics which will be described shortly.

For completeness we have also given the typing rules for the "unit" or final type. This type only has one element, namely ().

We can also introduce an n-ary product:

Remark 1.4 (Syntax for defining mutually recursive functions)

It is also useful to have a convenient syntax for define mutually recursive functions. To accomplish this is it is useful to use the syntax for n-ary products:

$$f_1 := t_1 \& ... \& f_n := t_n.N \equiv \left(\lambda z.\ \begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (f_1,..,f_n)\mapsto N\end{array}\right)\ \mathsf{fix}\left[\lambda z.\ \begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (f_1,..,f_n)\mapsto (t_1,..,t_n)\end{array}\right]$$

$\square$

$$\frac{z :: \Gamma \vdash t_1 :: P \quad ... \quad z :: \Gamma \vdash t_n :: P_n}{z :: \Gamma \vdash (t_1, .., t_n) :: P_1 \times ... \times P_n} \text{ tuple}$$

$$\frac{z :: \Gamma \vdash t : P_1 \times ... \times P_n \quad z :: \Gamma, x_1 :: P_1, ..., x_n :: P_n \vdash s :: W}{z :: \Gamma \vdash \begin{array}{l} \mathsf{case}\ t \\ \mathsf{of}\ (x_1, ..., x_n) \mapsto s \end{array} :: W} \text{ pcase}$$

Table 3: Judgments for n-ary products

$$\frac{}{z :: \Gamma \vdash \mathsf{succ} :: \mathbb{N} \longrightarrow \mathbb{N}} \text{ succ}$$

$$\frac{}{z :: \Gamma \vdash \mathsf{zero} :: \mathbb{N}} \text{ zero}$$

$$\frac{z :: \Gamma \vdash t :: \mathbb{N} \quad z :: \Gamma \vdash t_0 :: W \quad z :: \Gamma, n :: \mathbb{N} \vdash t_1 :: W}{z :: \Gamma \vdash \begin{array}{l} \mathsf{case}\ t \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \mapsto & t_0 \\ \mathsf{succ}\ n & \mapsto & t_1 \end{array} \right. \end{array} :: W} \ \mathbb{N} \text{ case}$$

Table 4: Judgments for the Natural Numbers

1.3 Judgments for natural numbers

To get a programming language a very basic requirement is to be able to represent numbers. Accordingly we introduce a type of natural numbers with its typing judgments. These are described in table 4. To introduce the natural numbers we introduce two constructors zero and succ together with a case statement for the natural numbers which allows one to test the form of the number and act accordingly.

Once one has the natural numbers one can write some non-trivial programs in this language. Here are some examples.

Example 1.5 (Type judgments for programs on the natural numbers)

(1) The simples program for the natural numbers is the predecessor:

$$\mathsf{pred} \equiv \lambda n. \left(\begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{zero} & \mapsto & \mathsf{zero} \\ \mathsf{succ}\ m & \mapsto & m \end{array} \right. \end{array} \right)$$

Here is the derivation of its type:

$$\dfrac{\dfrac{}{n :: \mathbb{N} \vdash n :: \mathbb{N}}\ \text{prj} \quad \dfrac{}{n :: \mathbb{N} \vdash \mathsf{zero} :: \mathbb{N}}\ \mathsf{zero} \quad \dfrac{}{n :: \mathbb{N}, m :: \mathbb{N} \vdash m :: \mathbb{N}}\ \text{prj}}{\dfrac{n :: \mathbb{N} \vdash \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{zero} & \mapsto & \mathsf{zero} \\ \mathsf{succ}\ m & \mapsto & m \end{array} \right. \end{array} :: \mathbb{N}}{\vdash \lambda n. \left(\begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{zero} & \mapsto & \mathsf{zero} \\ \mathsf{succ}\ m & \mapsto & m \end{array} \right. \end{array} \right) :: \mathbb{N} \longrightarrow \mathbb{N}}\ \text{app}}\ \mathbb{N}\ \text{case}$$

(2) One of the most basic things that one might wish to do with two numbers is to add them. This is defined using the fixed point operator as follows:

$$\mathsf{add} \equiv \mathsf{fix} \left[\lambda f z. \left(\begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (n,m) \mapsto \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{zero} & \longrightarrow & m \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (f(n',m)) \end{array} \right. \end{array} \end{array} \right) \right]$$

or using the syntax introduced for recursive functions:

$$\mathsf{add} := \lambda z. \left(\begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (n,m) \mapsto \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{zero} & \longrightarrow & m \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',m)) \end{array} \right. \end{array} \end{array} \right) .\mathsf{add}$$

Here is the type judgment for this term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}, z :: \mathbb{N}\times\mathbb{N} \vdash z :: \mathbb{N}\times\mathbb{N}}\,\mathrm{prj} \quad f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}, z :: \mathbb{N}\times\mathbb{N}, n :: \mathbb{N}, m :: \mathbb{N} \vdash \begin{array}{l}\mathsf{case}\ n\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{zero} & \mapsto & m\\ \mathsf{succ}\ n' & \mapsto & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array} :: \mathbb{N}}{f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}, z :: \mathbb{N}\times\mathbb{N} \vdash \begin{array}{l}\mathsf{case}\ z\\ \mathsf{of}\ (n,m)\mapsto \begin{array}{l}\mathsf{case}\ n\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{zero} & \mapsto & m\\ \mathsf{succ}\ n' & \mapsto & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array}\end{array} :: \mathbb{N}}\,\mathrm{pcase}}{f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N} \vdash \lambda z.(\begin{array}{l}\mathsf{case}\ z\\ \mathsf{of}\ (n,m)\mapsto \begin{array}{l}\mathsf{case}\ n\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{zero} & \mapsto & m\\ \mathsf{succ}\ n' & \mapsto & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array}\end{array}) :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}}\,\mathrm{app}}{\vdash \lambda fz.(\begin{array}{l}\mathsf{case}\ z\\ \mathsf{of}\ (n,m)\mapsto \begin{array}{l}\mathsf{case}\ n\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{zero} & \mapsto & m\\ \mathsf{succ}\ n' & \mapsto & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array}\end{array}) :: (\mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N})\longrightarrow\mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}}\,\mathrm{app}}{\vdash \mathsf{fix}[\lambda fz.(\begin{array}{l}\mathsf{case}\ z\\ \mathsf{of}\ (n,m)\mapsto \begin{array}{l}\mathsf{case}\ n\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{zero} & \mapsto & m\\ \mathsf{succ}\ n' & \mapsto & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array}\end{array})] :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}}\,\mathrm{fix}$$

and, letting $\Gamma = f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}, z :: \mathbb{N}\times\mathbb{N}, n :: \mathbb{N}, m :: \mathbb{N}, n' :: \mathbb{N}$, where

$$\dfrac{\dfrac{}{f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}, z :: \mathbb{N}\times\mathbb{N}, n :: \mathbb{N}, m :: \mathbb{N} \vdash \mathsf{zero}}\,\mathsf{zero} \quad \dfrac{\dfrac{}{\Gamma\vdash\mathsf{succ}}\,\mathsf{succ} \quad \dfrac{\dfrac{}{\Gamma\vdash f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}}\,\mathrm{prj} \quad \dfrac{\dfrac{}{\Gamma\vdash n' :: \mathbb{N}}\,\mathrm{prj} \quad \dfrac{}{\Gamma\vdash m :: \mathbb{N}}\,\mathrm{prj}}{\Gamma\vdash (n',m)}\,\mathrm{pair}}{\Gamma\vdash f(n',m)}\,\mathrm{app}}{\Gamma\vdash \mathsf{succ}\ (f(n',m))}\,\mathrm{app}}{f :: \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}, z :: \mathbb{N}\times\mathbb{N}, n :: \mathbb{N}, m :: \mathbb{N} \vdash \begin{array}{l}\mathsf{case}\ n\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{zero} & \mapsto & m\\ \mathsf{succ}\ n' & \mapsto & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array} :: \mathbb{N}}\,\mathbb{N}\ \mathrm{case}$$

$\square$

1.4 Judgments for lists

After being able to represent the natural numbers the next most important datatype is lists. Below in table 5 we introduce a type of lists with its typing judgments. The list type is itself dependent on a type which is a slightly new wrinkle. Otherwise these rules follow exactly the same pattern as those for the natural numbers.

The resulting language with these features has its types defined inductively by:

$$\mathsf{type} = \ \mathsf{type}\times\mathsf{type} \mid \mathsf{type}\longrightarrow\mathsf{type} \mid \mathbb{L}(\mathsf{type}) \mid \mathbb{N} \mid 1$$

so that all its types are built inductively from the basic types.

Example 1.6 (Programs for list)

(1) The append function is the analogue of addition:

$$\mathsf{append} := \lambda xy. \begin{array}{l}\mathsf{case}\ x\\ \mathsf{of}\left|\begin{array}{lll}\mathsf{nil} & \mapsto & y\\ \mathsf{cons}\ z & \mapsto & \begin{array}{l}\mathsf{case}\ z\\ \mathsf{of}\ (x,xs)\mapsto\mathsf{cons}\ (x,\mathsf{append}\ xs\ y)\end{array}\end{array}\right.\end{array} .\mathsf{append}$$

Derive a type judgement for this.

$$\frac{}{z :: \Gamma \vdash \mathsf{cons}\ :: A \times \mathbb{L}(A) \longrightarrow \mathbb{L}(A)}\ \text{cons}$$

$$\frac{}{z :: \Gamma \vdash \mathsf{nil}\ :: \mathbb{L}(A)}\ \text{nil}$$

$$\frac{z :: \Gamma \vdash t :: \mathbb{L}(A) \quad z :: \Gamma \vdash t_0 :: W \quad z :: \Gamma, v :: A \times \mathbb{L}(A) \vdash t_1 :: W}{z :: \Gamma \vdash \begin{array}{l} \mathsf{case}\ t \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{nil} & \longrightarrow & t_0 \\ \mathsf{cons}\ v & \longrightarrow & t_1 \end{array} \right. \end{array} :: W}\ \mathbb{L}\ \text{case}$$

Table 5: Judgments for Lists

(2) The real use of the extended syntax is to give a series of definitions as in:

$$\begin{array}{lll} \mathsf{append} & := & \lambda xy. \begin{array}{l} \mathsf{case}\ x \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{nil} & \mapsto & y \\ \mathsf{cons}\ z & \mapsto & \begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (x, xs) \mapsto \mathsf{cons}\ (x, \mathsf{append}\ xs\ y) \end{array} \end{array} \right. \end{array} . \\ \mathsf{reverse} & := & \lambda x. \begin{array}{l} \mathsf{case}\ x \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{nil} & \mapsto & \mathsf{nil} \\ \mathsf{cons}\ z & \mapsto & \begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (x, xs) \mapsto \mathsf{append}\ (\mathsf{reverse}\ xs)(\mathsf{cons}\ (x, \mathsf{nil}\ )) \end{array} \end{array} \right. \end{array} . \\ \mathsf{reverse} & & \end{array}$$

$\square$

We shall call the resulting language with all these features BPL (Basic Programming Language).

$$\frac{}{\lambda x.N \leadsto \lambda x.N} \qquad \frac{}{(N,M) \leadsto (N,M)}$$

$$\frac{}{\mathsf{zero} \leadsto \mathsf{zero}} \qquad \frac{}{\mathsf{succ}\ M \leadsto \mathsf{succ}\ M}$$

$$\frac{}{\mathsf{nil} \leadsto \mathsf{nil}} \qquad \frac{}{\mathsf{cons}\ M \leadsto \mathsf{cons}\ M}$$

$$\frac{N \leadsto \lambda x.N' \quad N'[M/x] \leadsto R}{NM \leadsto R}$$

$$\frac{M\mathsf{fix}[M] \leadsto R}{\mathsf{fix}[M] \leadsto R}$$

$$\frac{N \leadsto (N_1, N_2) \quad M[N_1/x, N_2/y] \leadsto R}{\begin{array}{l}\mathsf{case}\ N \\ \mathsf{of}\ (x,y) \mapsto M\end{array} \leadsto R}$$

$$\frac{N \leadsto \mathsf{zero} \quad M_0 \leadsto R}{\begin{array}{l}\mathsf{case}\ N \\ \mathsf{of} \left| \begin{array}{lll}\mathsf{zero} & \mapsto & M_0 \\ \mathsf{succ}\ n & \mapsto & M_1\end{array}\right.\end{array} \leadsto R} \qquad \frac{N \leadsto \mathsf{succ}\ N' \quad M_1[N'/n] \leadsto R}{\begin{array}{l}\mathsf{case}\ N \\ \mathsf{of} \left| \begin{array}{lll}\mathsf{zero} & \mapsto & M_0 \\ \mathsf{succ}\ n & \mapsto & M_1\end{array}\right.\end{array} \leadsto R}$$

$$\frac{L \leadsto \mathsf{nil} \quad M_0 \leadsto R}{\begin{array}{l}\mathsf{case}\ L \\ \mathsf{of} \left| \begin{array}{lll}\mathsf{nil} & \mapsto & M_0 \\ \mathsf{cons}\ z & \mapsto & M_1\end{array}\right.\end{array} \leadsto R} \qquad \frac{L \leadsto \mathsf{cons}\ N' \quad M_1[N'/z] \leadsto R}{\begin{array}{l}\mathsf{case}\ L \\ \mathsf{of} \left| \begin{array}{lll}\mathsf{nil} & \mapsto & M_0 \\ \mathsf{cons}\ z & \mapsto & M_1\end{array}\right.\end{array} \leadsto R}$$

Table 6: Reduction rules for BPL

2 Program evaluation in BPL

We may define a simple operational semantics for this language by describing how a term in the language can be reduced. The simplest way to do this is to describe a reduction system using inference rules much as above. The meaning of these inferences is, however, somewhat different. The strategy is basically to do a normal order reduction, however, as we have some new constructs we must define the reduction strategy for these terms as well. The reduction strategy is defined in table 6.

Notice how all of the reduction rules, except for the fixed point rule, are of the same general form as a β-reduction. These rules should be read as sequential evaluation recipes. Thus, the first rule may be read as follows: $N\ M$ reduces to R in case N reduces to $\lambda x.N'$ and when M is substituted for x in N' then this reduces to R.

Here are some simple examples of evaluation:

Example 2.1 (Reductions in BPL)

1. Here we calculate the predecessor of 2:

$$\dfrac{\dfrac{}{\mathsf{pred} \leadsto \mathsf{pred}} \qquad \dfrac{\dfrac{}{\mathsf{succ}\ (\mathsf{succ\ zero}\ ) \leadsto \mathsf{succ}\ (\mathsf{succ\ zero}\ )} \quad \dfrac{}{m[\mathsf{succ\ zero}\ /m] \leadsto \mathsf{succ\ zero}}}{\left(\begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \mapsto & \mathsf{zero} \\ \mathsf{succ}\ m & \mapsto & m \end{array}\right. \end{array}\right) [\mathsf{succ}\ (\mathsf{succ\ zero}\ )/n] \leadsto \mathsf{succ\ zero}}}{\mathsf{pred}\ (\mathsf{succ}\ (\mathsf{succ\ zero}\ )) \leadsto \mathsf{succ\ zero}}$$

2. Here we do the reduction of adding one and one!

$$\dfrac{\dfrac{\begin{array}{c} \lambda f z. \left(\begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (n,m) \mapsto \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \longrightarrow & m \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (f(n',m)) \end{array}\right. \end{array} \end{array}\right) \mathsf{add} \\ \leadsto \\ \lambda z. \left(\begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (n,m) \mapsto \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \longrightarrow & m \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',m)) \end{array}\right. \end{array} \end{array}\right) \end{array}}{\begin{array}{c} \mathsf{add} \\ \leadsto \\ \lambda z. \left(\begin{array}{l} \mathsf{case}\ z \\ \mathsf{of}\ (n,m) \mapsto \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \longrightarrow & m \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',m)) \end{array}\right. \end{array} \end{array}\right) \end{array}} \quad \dfrac{\dfrac{}{(1,1) \leadsto (1,1)} \quad \dfrac{\dfrac{}{1 \leadsto 1} \quad \dfrac{}{\mathsf{succ}\ (\mathsf{add}\ (0,1)) \leadsto \mathsf{succ}\ (\mathsf{add}\ (0,1))}}{\begin{array}{c} \begin{array}{l} \mathsf{case}\ 1 \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \longrightarrow & 1 \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',1)) \end{array}\right. \end{array} \\ \leadsto \\ \mathsf{succ}\ (\mathsf{add}\ (0,1)) \end{array}}}{\begin{array}{c} \left(\begin{array}{l} \mathsf{case}\ (1,1) \\ \mathsf{of}\ (n,m) \mapsto \begin{array}{l} \mathsf{case}\ n \\ \mathsf{of} \left| \begin{array}{lll} \mathsf{zero} & \longrightarrow & m \\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',m)) \end{array}\right. \end{array} \end{array}\right) \\ \leadsto \\ \mathsf{succ}\ (\mathsf{add}\ (0,1)) \end{array}}}{\mathsf{add}\ (1,1) \leadsto \mathsf{succ}\ (\mathsf{add}\ (0,1))}$$

   Notice that the final result has an unevaluated argument: all we really know is that the answer is succ of something.

□

The reduction rules apply only to closed term. Thus, we should show that closedness of terms is preserved by reductions: this is easily checked. More significantly, notice that the reduction rules do not refer to the type. This means that to show that the rules make sense it is necessary to establish that, despite this omission, they do actually behave well with respect to the types. Thus, a very basic requirement is that when a reduction is performed the term which results can still be judged to have the same type as the original before the reduction. The calculus is said to satisfies **subject reduction** if this is the case. To check that this is true it is necessary to do a proof by induction.

We are only interested in terms which have a head normal form: in which case to each term which has a head normal form we may attach the number of reduction inferences required to reduce it to that form. For terms which are already in head normal form this number is zero and clearly subject reduction holds for them. Now suppose the current term requires n-reduction inferences then we may take as our induction hypothesis that any term which requires $m < n$ reduction inferences satisfies subject reduction.

With this in mind consider the first rule:

$$\frac{N \leadsto \lambda x.N' \quad N'[M/x] \leadsto R}{NM \leadsto R}$$

if $NM :: Y$ then $N :: X \longrightarrow Y$ and $M :: X$. By inductive assumption as N requires strictly less reduction inferences we may conclude that $\lambda x.N' :: X \longrightarrow Y$. whence $x :: X \vdash N' :: Y$ is a type judgment. We now need the following lemma:

Lemma 2.2 (Substitutions lemma)

$$\frac{z : \Gamma \vdash M :: X \quad x :: X \vdash N' :: Y}{z : \Gamma \vdash N'[M/x] :: Y}$$

is a valid type judgment.

Proof. The proof of this is a straightforward (if tedious) induction on the structure of the term N'. One starts by assuming that N' is a variable and then considers all the term construction steps. $\square$

So this allows us to conclude that $N'[M/x] :: X$ and whence by inductive assumption that $R :: X$. Except for the fixed point rule all the remaining rules follow a similar pattern as they rely at some stage on the substitution lemma. Let us therefore look at the fixed point rule:

$$\frac{M\ \mathsf{fix}[M] \rightsquigarrow R}{\mathsf{fix}[M] \rightsquigarrow R}$$

here it suffices to observe that $M\ \mathsf{fix}[M]$ has the same type as M always so that we may use our inductive assumption on the number of reduction instances to show that this reduction satisfies subject reduction.

3 Type inference for BPL

The last aspect we consider for BPL in this document is the question of type inference. When a term has a number of different type judgments there is a type, possibly with some free type variables, which can be substituted to give all the possible type judgments. This type is called the most general type of the the term and is a **parametric** (or sometimes polymorphic) type as it may contain type variables.

The purpose of this section is to explain how this most general type can be derived. To do this we shall show how the type system allows one to derive equations for the type. These equations can then be solved using the unification algorithm.

3.1 Rules for type inference

To do this type inference it is necessary to read the inference trees backwards: this is sometimes called the proof search direction. Starting at the bottom of the proof tree the idea is to develop it while accumulating type equations. At each node in the proof tree it is possibly necessary to introduce some new type variables and some equations. We will write these as a sequent:

$$X_1, .., X_n \rhd T_1 = T_1', ..., T_m = T_m'$$

where the fresh type variables are on the left and the type equations are on the right.

In table 7 the inference rules have been modified to reflect how the most general type is derived.

The idea is that as one does the derivation of a type judgment one collects type equations which must hold the list of variables introduced. Any final type must then satisfy *all* these equations. We shall delay the discussion of how we can determine whether the equations are satisfied (this is the unification algorithm). For the moment we shall focus on seeing how these inferences allow us to collect equations.

Example 3.1 (Simple parametric type inference)

1. Consider the term $\lambda x.x$ this has the following derivation:

$$\dfrac{\dfrac{}{x :: X \vdash x :: Y}\ \text{prj} : \rhd X = Y}{\vdash \lambda x.x :: Q}\ \text{abst} : X, Y \rhd Q = X \longrightarrow Y$$

   These equations can be solved to give $Q = X \longrightarrow X$ as expected.

2. Consider the symmetry map $\lambda z.\left(\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y) \mapsto (y,x)\end{array}\right)$:

$$\dfrac{\dfrac{\dfrac{}{z :: A \vdash z :: Z}\ \text{prj} : \rhd A = Z \qquad \dfrac{\dfrac{}{z :: A, x :: C, y :: D \vdash y :: E}\ \text{prj} : \rhd E = D \qquad \dfrac{}{z :: A, x :: C, y :: D \vdash x :: F}\ \text{prj} : \rhd F = C}{z :: A, x :: C, y :: D \vdash (y,x) :: B}\ \text{pair} : E, F \rhd B = E \times F}{z :: A \vdash \begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y) \mapsto (y,x)\end{array} :: B}\ \text{pcase} : C, D \rhd Z = C \times D}{\vdash \lambda z.\left(\begin{array}{l}\mathsf{case}\ z\\ \ \mathsf{of}\ (x,y) \mapsto (y,x)\end{array}\right) :: Q}\ \text{abst} : A, B \rhd Q = A \longrightarrow B$$

   When one resolves the equations this says that

$$Q = C \times D \longrightarrow D \times C.$$

   This example shows that the number of equations generated increases (linearly) with the size of the proof. Thus the importance of having a method of cleaning up the equations is clear!

$$\frac{}{x :: P, z :: \Gamma \vdash x :: Q}\ \text{proj} : \rhd P = Q$$

$$\frac{x :: X, z :: \Gamma \vdash t :: Y}{z :: \Gamma \vdash \lambda x.t :: Q}\ \text{abst} X, Y \rhd Q = X \longrightarrow Y :$$

$$\frac{z :: \Gamma \vdash f :: Z \quad z :: \Gamma \vdash t :: X}{z :: \Gamma \vdash (ft) :: Q}\ \text{app} : X, Z \rhd Z = X \longrightarrow Q$$

$$\frac{z :: \Gamma \vdash t :: Z}{z :: \Gamma \vdash \mathsf{fix}[t] :: Q}\ \text{fix} : \mathrm{Z} \rhd \mathrm{Z} = \mathrm{Q} \longrightarrow \mathrm{Q}$$

$$\frac{z :: \Gamma \vdash t :: X \quad z :: \Gamma \vdash s :: Y}{z :: \Gamma \vdash (t, s) :: Q}\ \text{pair} : X, Y \rhd Q = X \times Y$$

$$\frac{z :: \Gamma \vdash t : Z \quad z :: \Gamma, x :: X, y :: Y \vdash s :: Q}{z :: \Gamma \vdash \begin{array}{l}\mathsf{case}\ t \\ \mathsf{of}\ (x, y) \mapsto s\end{array} :: Q}\ \text{pcase} : X, Y, Z \rhd Z = X \times Y$$

$$\frac{}{z :: \Gamma \vdash () :: Q}\ \text{unit} : \rhd Q = 1$$

$$\frac{z :: \Gamma \vdash t :: Z \quad z :: \Gamma \vdash s :: Q}{z :: \Gamma \vdash \begin{array}{l}\mathsf{case}\ t \\ \mathsf{of}\ () \mapsto s\end{array} :: Q}\ \text{ucase} : Z \rhd Z = 1$$

$$\frac{}{z :: \Gamma \vdash \mathsf{succ}\ :: Q}\ \text{succ} : \rhd Q = \mathbb{N} \longrightarrow \mathbb{N} \qquad \frac{}{z :: \Gamma \vdash \mathsf{zero}\ :: Q}\ \text{zero} : \rhd Q = \mathbb{N}$$

$$\frac{z :: \Gamma \vdash t :: X_1 \quad z :: \Gamma \vdash t_0 :: Y_1 \quad z :: \Gamma, n :: X_2 \vdash t_1 :: Y_2}{z :: \Gamma \vdash \begin{array}{l}\mathsf{case}\ t \\ \mathsf{of} \left| \begin{array}{lll}\mathsf{zero} & \mapsto & t_0 \\ \mathsf{succ}\ n & \mapsto & t_1\end{array}\right.\end{array} :: Q}\ \mathbb{N}\ \text{case} : \begin{array}{l} X_1, \\ Y_1, \\ X_2, \\ Y_2 \end{array} \rhd \begin{array}{l} X_1 = \mathbb{N}, \\ Y_1 = Q, \\ X_2 = \mathbb{N}, \\ Y_2 = Q \end{array}$$

$$\frac{}{z :: \Gamma \vdash \mathsf{cons}\ :: Q}\ \text{cons} : \rhd Q = A \times \mathbb{L}(A) \longrightarrow \mathbb{L}(A) \qquad \frac{}{z :: \Gamma \vdash \mathsf{nil}\ :: Q}\ \text{nil} : \rhd Q = \mathbb{L}(A)$$

$$\frac{z :: \Gamma \vdash t :: X_1 \quad z :: \Gamma \vdash t_0 :: Y_1 \quad z :: \Gamma, v :: X_2 \vdash t_1 :: Y_2}{z :: \Gamma \vdash \begin{array}{l}\mathsf{case}\ t \\ \mathsf{of} \left| \begin{array}{lll}\mathsf{nil} & \longrightarrow & t_0 \\ \mathsf{cons}\ v & \longrightarrow & t_1\end{array}\right.\end{array} :: Q}\ \mathbb{L}\ \text{case} : \begin{array}{l} X, \\ X_1, \\ Y_1, \\ X_2, \\ Y_2 \end{array} \rhd \begin{array}{l} X_1 = \mathbb{L}(X), \\ Y_1 = Q, \\ X_2 = X \times \mathbb{L}(X), \\ Y_2 = Q \end{array}$$

Table 7: Rules for type inference

$\square$

3.2 Solving type equations

Type inference may be divided into two parts: the first part described above involves collecting the equations which must hold between the types, while the second part is to solve these equations. This latter problem is often called the **unification** problem and fortunately there is a well-known algorithm for solving it.

The types in our system may be described inductively, for this section however we need to add type variables so the description becomes:

$$\mathsf{type} = \ \mathsf{type} \times \mathsf{type} \mid \mathsf{type} \longrightarrow \mathsf{type} \mid \mathbb{L}(\mathsf{type}) \mid \mathbb{N} \mid 1 \mid X_1 \mid X_2 \mid ...$$

where $X_1, X_2, ...$ are variables: these types are called **parametric** types as they may contain variables.

Given an equation between parametric types

$$X_1, .., X_n \rhd S = T$$

where will often we indicate the free variables by subscripting the equality, a **unifier** is a substitution list

$$Y_1, .., Y_m \rhd [T_1/X_1, ..., T_n/X_n]$$

such that

$$S[T_1/X_1, ..., T_n/X_n] \equiv_{Y_1,..,Y_m} T[T_1/X_1, ..., T_n/X_n].$$

The meaning of $\equiv$ here is that the two types are now structurally identical.

The book-keeping required to keep track of the free variables can be relaxed: often we will not wish to change the name of a variable and strictly speaking, from the above, we should then add the trivial substitution X/X. However, it is convenient to omit these trivial substitutions and treat them as understood.

A **most general unifier** for the equation $S =_\Delta T$ is a unifier $\Delta' \rhd (T_1, ..., T_n)/\Delta$ such that any other unifier $\Theta \rhd (T_1', ..., T_n')/\Delta$ can be obtained by a substitution, $[(S_1, .., S_m)/\Delta']$ of this

$$(T_1', ..., T_n') =_\Theta (T_1, ..., T_n)[(S_1, .., S_m)/\Delta'].$$

It is a useful fact that for "free terms" if two terms can be unified then there is a most general unifier. The proof of this is in its construction which we is described in table 8.

The function *unify* attempts to solve the equations $S = T$. It does so in two steps:

1. The *match* function builds a list of substitutions:

   $$[S_1/X_1, ..., S_n/X_n]$$

   such that X_i does not occur in S_i: this is the **occurs check** and is one of the two places at which the algorithm can fail.

   It builds these substititions by overlaying one term with the other: if they differ structurally, because they differ in their type constructors ($\times$, $\longrightarrow$, $\mathbb{L}$, or $\mathbb{N}$) then unification will fail. When they differ because one or the other is a variable then one obtains a substitution which goes forward if it satisfies the occurs check.

$$
\begin{array}{lcl}
\mathit{check}(X/X) & = & [] \\
\mathit{check}(T/X) & = & \begin{cases} [T/X] & \text{if } X \text{ does not occur in } T \\ \text{fail} & \text{otherwise} \end{cases} \\
\\
\mathit{match}(X,T) & = & \mathit{check}(T/X) \\
\mathit{match}(T,X) & = & \mathit{check}(T/X) \\
\mathit{match}(F(\vec{T}),G(\vec{S})) & = & \begin{cases} \mathit{flatten}(\mathit{map}\ \mathit{match}\ (\mathit{zip}\ (\vec{T},\vec{S}))) & \text{if } F \equiv G \\ \mathit{fail} & \text{otherwise} \end{cases} \\
\\
\mathit{coalesce}(T/X,[]) & = & [] \\
\mathit{coalesce}(T/X,S/Y{:}\mathit{Rest}) & = & \begin{cases} \mathit{append}(\mathit{match}(T,S),\mathit{coalesce}(T/A,\mathit{Rest})) & \text{if } X \equiv Y \\ \mathit{append}(\mathit{check}(S[T/X]/Y),\mathit{coalesce}(T/A,\mathit{Rest})) & \text{otherwise} \end{cases} \\
\\
\mathit{linearize}([]) & = & [] \\
\mathit{linearize}(S/A{:}\mathit{Rest}) & = & S/A{:}(\mathit{linearize}(\mathit{coalesce}(S/A,\mathit{Rest})) \\
\\
\mathit{unify}(S,T) & = & \mathit{linearize}(\mathit{match}(S,T))
\end{array}
$$

Table 8: The unification algorithm

2. The *linearize* function takes the output of the *match* function and produces a list of assignments
$$[S_1/X_1, ..., S_n/X_n]$$
such that $X_1, .., X_n$ are distinct variables and for each $1 \leq i \leq j \leq n$, X_i does not occur in S_j. To achieve this we remove the earlier variable from the later substitutions using the *coalesce* function. Here there are two cases to consider: when the variable is substituted a second time in the list one uses the match function recursively to remove that occurrence or when the variable occurs in another substitution, in which case one removes it by applying the given substitution.

Type inference generates a list of substitutions and the idea is to incrementally update this list as each new equation is encountered while checking that there is a solution. This can be done by adding to the current solution σ the result of resolving the current substituted equation:

$$\mathit{add}(\sigma, S = T) = \mathit{append}(\sigma, \mathit{unify}(S[\sigma], T[\sigma])$$