Basic Programming for Computable Functions: BPCF (The Typed Lambda Calculus with Fixed Points)

Robin Cockett

November 29, 2016

This document describes a basic type theories which can be built on top of the λ-calculus in order to create a simple programming language. The basic type theory which underlies functional style programming languages is the simply typed calculus with fixed points. To this one adds, in order to obtain a reasonable programming language, some basic programming "features" for pairs, natural numbers and list. The result is what we refer to as the Basic Programming (language) for Computable Functions: BPCF.

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, **BPCF** (Basic Programming Language for Computable Functions) 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. A sequent, such as

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

is called a **type judgement**. if it can be inferred (or proven) in a type theory, it is referred to as a **valid type judgment**. Thus a valid type judgement has a proof that the term, t, may be assigned the type T.

Each sequent $\Gamma \vdash t :: Q$ consists of a **type context**, Γ, and a term t together with its type Q, written as $t :: Q$. A type context is an association of types to list of *distinct* variables:

$$\Gamma := x_1 :: Q_1, x_2 :: Q_2, ..., x_n : Q_n$$

The order in which these associations is written does not matter.

The first three rules give respectively the ability to: choose a variable from the context, form an application, and form an abstraction. The last rule introduces a slightly modified syntax to

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

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

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

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

Table 1: Annotated type system

handle the fixed points. This means that we do not have an explicit fixed point operator Y in the language. However, as demonstrated below, we can construct a fixed point operator for each type.

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 $\lambda x.x :: A \longrightarrow A$:

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

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

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

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

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

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

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

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:

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

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.

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:

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

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

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

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

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

Table 2: Judgments for products

(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:

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

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:*

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

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

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

Table 3: Judgments for n-ary products

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

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

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

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:

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

Here is the derivation of its type:

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

(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:*

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

or using the syntax introduced for recursive functions:

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

Here is the type judgment for this term:

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

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*

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

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

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

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

Table 5: Judgments for Lists

1.4 Judgments for lists

After being able to represent the natural numbers the next most important data type 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:

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

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:

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

Derive a type judgment for this.

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

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

We shall call the resulting language with all these features **BPCF** (Basic Programming language for Computable Functions).

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

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

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

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

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

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

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

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

Table 6: Reduction rules for BPCF

2 Program evaluation in BPCF

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 BPCF*

1. *Here we calculate the predecessor of 2:*

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

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}{lcl}\mathsf{zero} & \longrightarrow & m\\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (f(n',m))\end{array}\right.\end{array}\end{array}\right)\mathsf{add}\\ \rightsquigarrow\\ \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)\end{array}}{\begin{array}{c}\mathsf{add}\\ \rightsquigarrow\\ \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)\end{array}} \quad \dfrac{\dfrac{}{(1,1) \rightsquigarrow (1,1)} \quad \dfrac{\dfrac{}{1 \rightsquigarrow 1} \quad \dfrac{}{\mathsf{succ}\ (\mathsf{add}\ (0,1)) \rightsquigarrow \mathsf{succ}\ (\mathsf{add}\ (0,1))}}{\begin{array}{c}\begin{array}{l}\mathsf{case}\ 1\\ \mathsf{of} \left|\begin{array}{lcl}\mathsf{zero} & \longrightarrow & 1\\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',1))\end{array}\right.\end{array}\\ \rightsquigarrow\\ \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}{lcl}\mathsf{zero} & \longrightarrow & m\\ \mathsf{succ}\ n' & \longrightarrow & \mathsf{succ}\ (\mathsf{add}\ (n',m))\end{array}\right.\end{array}\end{array}\right)\\ \rightsquigarrow\\ \mathsf{succ}\ (\mathsf{add}\ (0,1))\end{array}}}{\mathsf{add}\ (1,1) \rightsquigarrow \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 satisfy **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:

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

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. □

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 BPCF

The last aspect we consider for **BPCF** 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 some new type variables and some equations relating them to the existing type variables are introduced. We will indicate this as a list of equations in which the introduced variables are bound:

$$\exists X_1, .., X_n.T_1 = T'_1, ..., T_m = T'_m$$

where the fresh type variables are on the left and the type equations are on the right. Although we shall use the notation $\exists$ this is not to be viewed as being existential quantification (in the sense of predicate logic or existential types) it is more prosaically notation for a type binder.

In table 7 the inference rules have been modified to derive the type equations which any types for the judgment must satisfy.

The idea is that as one does the derivation of a type judgment one collects type equations which must bind 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:*

$$\dfrac{\dfrac{}{x :: X \vdash x :: Y \langle X = Y\rangle}\ \text{prj}}{\vdash \lambda x.x :: Q \quad \langle \exists X, Y.Q = X \longrightarrow Y, X = Y}\ \text{abst}$$

The resulting type equation is $\exists X, Y.Q = X \longrightarrow Y, X = Y$ and can be solved to give $\exists Y.Q = Y \longrightarrow Y$ as expected.

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

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

$$\frac{}{x :: P, \Gamma \vdash x :: Q \quad \langle P = Q\rangle}\ \text{proj}$$

$$\frac{x :: X, \Gamma \vdash \quad t :: Y\langle E\rangle}{\Gamma \vdash \lambda x.t :: Q \quad \langle \exists X, Y. Q = X \to Y, E\rangle}\ \text{abst}$$

$$\frac{\Gamma \vdash f :: Z \quad \langle E_1\rangle \quad \Gamma \vdash t :: X \quad \langle E_2\rangle}{\Gamma \vdash (f t) :: Q \quad \langle \exists X, Z. Z = X \to Q, E_1, E_2\rangle}\ \text{app}$$

$$\frac{\Gamma \vdash t :: Z \quad \langle E\rangle}{\Gamma \vdash \mathsf{fix}[t] :: Q \quad \langle \exists Z. Z = Q \to Q, E\rangle}\ \text{fix}$$

$$\frac{\Gamma \vdash t :: X \quad \langle E_1\rangle \quad \Gamma \vdash s :: Y \quad \langle E_2\rangle}{\Gamma \vdash (t, s) :: Q \quad \langle \exists X, Y. Q = X \times Y, E_1, E_2\rangle}\ \text{pair}$$

$$\frac{\Gamma \vdash t : Z \quad \langle E_1\rangle \quad z\Gamma, x :: X, y :: Y \vdash s :: Q \quad \langle E_2\rangle}{\Gamma \vdash \begin{array}{l}\mathsf{case}\ t\\ \ \mathsf{of}\ (x, y) \mapsto s\end{array} :: Q \quad \langle \exists X, Y, Z. Z = X \times Y, E_1, E_2\rangle}\ \text{pcase}$$

$$\frac{}{\Gamma \vdash () :: Q \quad \langle Q = 1\rangle}\ \text{unit}$$

$$\frac{\Gamma \vdash t :: Z \quad \langle E_1\rangle \quad \Gamma \vdash s :: Q \quad \langle E_2\rangle}{\Gamma \vdash \begin{array}{l}\mathsf{case}\ t\\ \ \mathsf{of}\ () \mapsto s\end{array} :: Q \quad \langle \exists Z. Z = 1, E_1, E_2\rangle}\ \text{ucase}$$

$$\frac{}{\Gamma \vdash \mathsf{succ}\ :: Q \quad \langle Q = \mathbb{N} \to \mathbb{N}\rangle}\ \text{succ} \qquad \frac{}{\Gamma \vdash \mathsf{zero}\ :: Q \quad \langle Q = \mathbb{N}\rangle}\ \text{zero}$$

$$\frac{\Gamma \vdash t :: X_1 \quad \langle E_1\rangle \quad \Gamma \vdash t_0 :: Y_1 \quad \langle E_2\rangle \quad \Gamma, n :: X_2 \vdash t_1 :: Y_2 \quad \langle E_3\rangle}{\Gamma \vdash \begin{array}{l}\mathsf{case}\ t\\ \mathsf{of}\ \left|\begin{array}{lll}\mathsf{zero} & \mapsto & t_0\\ \mathsf{succ}\ n & \mapsto & t_1\end{array}\right.\end{array} :: Q \quad \left\langle \exists X_1, Y_1, X_2, Y_2. \begin{array}{l} X_1 = \mathbb{N}, Y_1 = Q,\\ X_2 = \mathbb{N}, Y_2 = Q,\\ E_1, E_2, E_3\end{array}\right\rangle}\ \mathbb{N}\ \text{case}$$

$$\frac{}{\Gamma \vdash \mathsf{cons}\ :: Q \quad \langle \exists A. Q = A \times \mathbb{L}(A) \to \mathbb{L}(A)\rangle}\ \text{cons} \qquad \frac{}{\Gamma \vdash \mathsf{nil}\ :: Q \quad \langle \exists A. Q = \mathbb{L}(A)\rangle}\ \text{nil}$$

$$\frac{\Gamma \vdash t :: X_1 \quad \langle E_1\rangle \quad \Gamma \vdash t_0 :: Y_1 \quad \langle E_2\rangle \quad \Gamma, v :: X_2 \vdash t_1 :: Y_2 \quad \langle E_3\rangle}{\Gamma \vdash \begin{array}{l}\mathsf{case}\ t\\ \mathsf{of}\ \left|\begin{array}{lll}\mathsf{nil} & \to & t_0\\ \mathsf{cons}\ v & \to & t_1\end{array}\right.\end{array} :: Q \quad \left\langle \exists X, X_1, Y_1, X_2, Y_2. \begin{array}{l} X_1 = \mathbb{L}(X), Y_1 = Q,\\ X_2 = X \times \mathbb{L}(X), Y_2 = Q,\\ E_1, E_2, E_3\end{array}\right\rangle}\ \mathbb{L}\ \text{case}$$

Table 7: Rules for type inference

When one collects the equations one gets:

$$\exists A, B.Q = A \longrightarrow B(\exists C, D, Z.Z = C \times D, A = Z, (\exists E, F.B = E \times F, E = D, F = C))$$

this can be solved to give:

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

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!

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 called a **unification** problem and fortunately there is a well-known algorithm for solving it. However, it is necessary to modify the basic algorithm to fit with the type inference.

3.2.1 Basic unification

The parametric types – that is the types which include type variables – may be described inductively:

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

where $X_1, X_2, ...$ are type variables.

Given an equation between parametric types

$$S =_{X_1,..,X_n} T$$

where we indicate the free variables by sub-scripting the equality with them, a **unifier** is a substitution list

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

(where the parametric types $T_1, ..., T_n$ have free variables from $\{Y_1, .., Y_m\}$), 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 absolutely identical once the substitution is made.

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 for any variable which is not changed. 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:

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

Table 8: The unification algorithm

1. The *match* function given two terms (or parametric types) 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 substitutions 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 an **assignment** (an equality in which the first term is a variable) which is carried forward by the algorithm if it satisfies the occurs check.

2. The *linearize* function takes the output of the *match* function and produces a list of assignments

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

   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 variables from the later substitutions using the *coalesce* function. Here there are two cases to consider as one recursively builds this list by adding an assignment. If the variable occurs in the term of an earlier assignment one removes the occurrence by using the given assignment as a substitution. However, it is also possible that the variable of the assignment being added already occurs as the variable of an assignment in the list of assignments collected so far: in this case removes all occurrences of the variable by substitution (as before) until one reaches the earlier assignment occurrence at which stage one uses the match function to remove that occurrence.

3.2.2 Solving type equations

Type inference generates a list of type equations with "existential" variable bindings and when one solves the equations generated by a type inference algorithm it is worth using the extra information inherent in the variable bindings: this information not only simplifies the problem but in a real implementation can help one to locate where the typing went wrong (if it did). This is because the modified algorithm works by trying to eliminate the existentially bound variables. Thus, if one associates with each existentially bound variable the line number (and term) at which the equation collection introduced that quantified variable, then that area of the code can be identified as the source of failure when an attempt to eliminate that variable causes a failure of typing.

It should be mentioned that, in general, the cause of a type error is rather hard to track ... and indeed could have many sources which need not be particularly local. From the perspective of error reporting this does create a difficulty. The above suggestion has the merit of usually placing the problem in the right area and being relatively easy to implement.

The equation collection from the the first stage of inference returns a tree of equations in a data type of the general form:

```
data TypeEqns v = TypeExist [v] [TypeEqns v]
                | TypeEqn (TypeExp v,TypeExp v)
```

where v is the type of the variable (usually integers). The objective of the type inference is to return either a type failure or an element of the type `([v],[v],[(v,TypeExp v)])`. This consists of a list of free variables whose substitutions one is trying to determine, a list of bound variables one could not eliminate, and a list of assignments (or substitutions) for the free variables – variables not mentioned are assumed to be substituted by the identity substitution x/x.

We may express what must be done to solve these equations in various ways: one method is to give a set of rules for simplifying type equations:

1. Any occurrence of an equation of the form $x = t$ where $x \in t$ and $x \neq t$ causes FAILURE (occurs check failure).
2. Any occurrence of an equation of the form $f(t_1, .., t_n) = g(t'_1, .., t'_m)$ where either $f \neq g$ or $n \neq m$ causes FAILURE (match failure)
3. Any occurrence of an equation of the form $f(t_1, .., t_n) = f(t'_1, .., t'_n)$ can be replaced by the equations $t_1 = t'_1, ..., , t_n = t'_n$ (matching).
4. Existentially quantified empty lists of equations, $\exists x.()$, can be removed.
5. Substitutions of variables can eliminated by substituting: $\exists x.x = tE = \exists x.E[t/x]$.
6. Scope manipulations:

$$\exists x.\exists y.E = \exists y.\exists x.E \qquad E_1, \exists y.E_2 = \exists y.E_1, E_2 \quad y \notin E_1.$$

Notice that we can always remove a equation of the form $x = x$ as substitution using it will do nothing. If $x = t$ is an equation where x is a variable and $x \notin t$, then substitution will remove all occurrences of x, this, in turn, will allow the quantification over x to be removed. This is presented as two step above, however, it may be viewed as a single step allowing the complete removal of a quantified variable from the equations.

This may be translated into an algorithm: the type solving algorithm starts at the leaves of the tree of type equations and works down towards the root (that is it is a fold).

It has to perform the following actions:

1. Given any existentially quantified empty list of type equations one returns an empty substitution list and empty sets of bound and free variables.

2. Given a basic type equation `TypeEqn (TypeExp v,TypeExp v)` the expressions must be matched to obtain a list of substitutions together with the list of free variables and an empty list of bound variables.

3. To add a type equation to an already processed type equation list (which either fails or returns `([v],[v],[(v,TypeExp v)])`) one evaluates the added type equation (my matching) to the same form (a list of free and bound variables and substitutions): the resulting free variables are the union of the two sets of free variables and the bound variables are similarly the disjoint union (as they must be distinct) of the two sets of bound variables. The two substitution lists must be then be combined into one by appending the first onto the second.

4. Given an existential quantification one must go through all the bound variables and try to eliminate them. To do this one performs essentially the "linearize" step above but restricted to the bound variables – both those passed down and those newly introduced:

   (a) One locates an assignment of a bound variable and uses it to eliminate that variable from the list. This is done by substituting it into the term of all assignments (failing on occurs check) and matching it with substitutions of the same variable (failing on a match failure).

   (b) One keeps doing this until there are no more assignments associated with bound variables. The variables that remain are passed on as the new bound variables. The new free variables are the old free variables less all the variables in the new binding.

5. The final result of this may have an assignment/substitution list which still contains repeated assignments to variables (the free ones) and so it must be "linearized" with respect to the free variables and then substituted out to get the final substitution of the free variables: recall that there may still be some bound variables left and the solution is now parametric in these.

Here are the two examples we considered earlier:

Example 3.2 *Simple parametric type inference*

1. *The term $\lambda x.x$ gave the following type equations:*

$$\exists X, Y. Q = X \longrightarrow Y, X = Y$$

It remains to eliminate the quantified variables: eliminating X gives $\exists Y. Q = Y \longrightarrow Y$ and this give $\exists Y. Q = Y \longrightarrow Y$ as the answer.

2. *The term* $\lambda z.\left(\begin{array}{l}\mathsf{case}\ z\\ \mathsf{of}\ (x,y)\mapsto(y,x)\end{array}\right)$ *gives the equations:*

$$\exists A,B.Q = A \longrightarrow B(\exists C,D,Z.Z = C\times D, A = Z, (\exists E,F.B = E\times F, E = D, F = C))$$

Starting on the inside we solve $\exists E,F.B = E\times F, E = D, F = C$ *to obtain* $B = D\times C$ *so that now we must solve:*

$$\exists A,B.Q = A \longrightarrow B(\exists C,D,Z.Z = C\times D, A = Z, B = D\times C)$$

Working next on $\exists C,D,Z.Z = C\times D, A = Z, B = D\times C$ *one has no substitutions for the bound variables* C *and* D *but one can eliminate* Z *so one must now solve:*

$$\exists A,B,C,D.Q = A \longrightarrow B, A = C\times D, B = D\times C$$

Elimination applied to this yields

$$\exists C,D.Q = (C\times D) \longrightarrow (D\times C).$$

These example shows that although the number of equations generated increases (linearly) with the size of the proof, the variable elimination is very effective at keeping the number of equations actually passed small. This illustrates another important algorithmic aspect of the variable binding in this algorithm.