1.3 Arithmetic

We obtain the system of *first-order arithmetic* if we restrict quantifiers to elements of type **nat**. Recall the induction principle for natural numbers and the rules for equality $n =_N m$ and the less-than relation $n < m$ summarized in Section **??**.

As a reminder, we will prove some frequently needed properties of equality. The first is reflexivity of equality.

$$\forall x{\in}\mathbf{nat}.\ x =_N x$$

We first give the informal proof, then its translation into a formal proof language.

Proof: The proof is by induction on x.

Case: $x = \mathbf{0}$. Then $\mathbf{0} =_N \mathbf{0}$ by rule $=_N I_0$.

Case: $x = \mathbf{s}(x')$. Then

$x' =_N x'$ by induction hypothesis
$s(x') =_N s(x')$ by rule $=_N I_s$.

□

As a formal proof in linear format:

```
[ x : nat;                % assumption
  0 = 0;                  % by =I0 (base case)
  [x' : nat, x' = x';     % assumptions
   s(x') = s(x')];        % by =Is (induction step)
  x = x ];                % by induction on x
!x:nat. x = x;            % by !I
```

We can also write out the proof term that corresponds to the proof above.

$$\begin{array}{rcl} \mathit{refl} & : & \forall x{\in}\mathbf{nat}.\ x =_N x \\ & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ r(\mathbf{0}) \Rightarrow \mathbf{eq}_0 \\ & & \qquad\ \mid r(\mathbf{s}(x')) \Rightarrow \mathbf{eq}_s(r(x')) \end{array}$$

As a second example, we consider transitivity of equality.

$$\forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ \forall z{\in}\mathbf{nat}.\ x =_N y \supset y =_N z \supset x =_N z$$

This time we will give the proof in three forms: as an informal mathematical proof, as a formal proof in linear form, and as an equational specification proof term.

Proof: The proof is by induction on x. We need to distinguish subcases on y and z.

Case: $x = \mathbf{0}$. Then we distinguish subcases on y.

Case: $y = \mathbf{0}$. Then we distinguish subcases on z.

Case: $z = \mathbf{0}$. Then $\mathbf{0} =_N \mathbf{0}$ by rule $=_N I_0$.

Case: $z = \mathbf{s}(z')$. Then $y =_N z$ is impossible by rule $=_N E_{0s}$.

Case: $y = \mathbf{s}(y')$. Then $x =_N y$ is impossible by rule $=_N E_{0s}$.

Case: $x = \mathbf{s}(x')$. We assume the induction hypothesis

$$\forall y{\in}\mathbf{nat}.\, \forall z{\in}\mathbf{nat}.\, x' =_N y \supset y =_N z \supset x' =_N z$$

and distinguish subcases on y.

Case: $y = \mathbf{0}$. Then $x =_N y$ is impossible by rule $=_N E_{0s}$.

Case: $y = \mathbf{s}(y')$. Then we distinguish subcases on z.

Case: $z = \mathbf{0}$. Then $y =_N z$ is impossible by rule $=_N E_{s0}$.

Case: $z = \mathbf{s}(z')$. Then we assume $\mathbf{s}(x') =_N \mathbf{s}(y')$ and $\mathbf{s}(y') =_N \mathbf{s}(z')$ and have to show that $\mathbf{s}(x') =_N \mathbf{s}(z')$. We continue:

$x' =_N y'$	by rule $=_N E_{ss}$
$y' =_N z'$	by rule $=_N E_{ss}$
$x' =_N z'$	by universal and implication eliminations from induction hypothesis
$\mathbf{s}(x') =_N \mathbf{s}(z')$	by rule $=_N I_s$.

□

The formal proof of transitivity is a good illustration why mathematical proofs are not written as natural deductions: the granularity of the steps is too small even for relatively simple proofs.

```
[ x : nat;
  [ y : nat;
    [ z : nat;
      [ 0 = 0;
        [ 0 = 0;
          0 = 0 ];                          % eqI0
        0 = 0 => 0 = 0 ];
      0 = 0 => 0 = 0 => 0 = 0;              % case (z = 0)
      [ z' : nat, 0 = 0 => 0 = z' => 0 = z';
        [ 0 = 0;
          [ 0 = s(z');
            0 = s(z') ];                    % eqE0s
          0 = s(z') => 0 = s(z') ];
        0 = 0 => 0 = s(z') => 0 = s(z') ];  % case (z = s(z'))
      0 = 0 => 0 = z => 0 = z ];
    !z:nat. 0 = 0 => 0 = z => 0 = z;        % case (y = 0)

    [ y' : nat, !z:nat. 0 = y' => y' = z => 0 = z;
```

```
      [ z : nat;
        [ 0 = s(y');
          [ s(y') = z;
            0 = z ];                          % eqE0s
          s(y') = z => 0 = z ];
        0 = s(y') => s(y') = z => 0 = z ];
      !z:nat. 0 = s(y') => s(y') = z => 0 = z ]; % case (y = s(y'))

    !z:nat. 0 = y => y = z => 0 = z ];
  !y:nat. !z:nat. 0 = y => y = z => 0 = z;    % base case (x = 0)

  [ x' : nat, !y:nat. !z:nat. x' = y => y = z => x' = z;  % ind hyp (x)
    [ y : nat;
      [ z : nat;
        [ s(x') = 0;
          [ 0 = z;
            s(x') = z ];                      % eqEs0
          0 = z => s(x') = z ];
        s(x') = 0 => 0 = z => s(x') = z ];
      !z:nat. s(x') = 0 => 0 = z => s(x') = z;   % case (y = 0)
      [ y' : nat, !z:nat. s(x') = y' => y' = z => s(x') = z;
        [ z : nat;
          [ s(x') = s(y');
            [ s(y') = 0;
              s(x') = 0 ];                    % eqEs0
            s(y') = 0 => s(x') = 0 ];
          s(x') = s(y') => s(y') = 0 => s(x') = 0;     % case (z = 0)

          [ z' : nat, s(x') = s(y') => s(y') = z' => s(x') = z';
            [ s(x') = s(y');
              [ s(y') = s(z');
                x' = y';                      % eqEss
                y' = z';                      % eqEss
                !z:nat. x' = y' => y' = z => x' = z;
                x' = y' => y' = z' => x' = z';
                y' = z' => x' = z';
                x' = z';
                s(x') = s(z') ];              % eqIs
              s(y') = s(z') => s(x') = s(z') ];
            s(x') = s(y') => s(y') = s(z') => s(x') = s(z') ];
          s(x') = s(y') => s(y') = z => s(x') = z ];
        !z:nat. s(x') = s(y') => s(y') = z => s(x') = z ];  % case (y = s(y'))
      !z:nat. s(x') = y => y = z => s(x') = z ];
    !y:nat. !z:nat. s(x') = y => y = z => s(x') = z ];  % ind step (x = s(x'))
  !y:nat. !z:nat. x = y => y = z => x = z ];
!x:nat. !y:nat. !z:nat. x = y => y = z => x = z;
```

Instead of giving the proof term in full, we give its specification. Recall that

$$\mathit{trans} \quad : \quad \forall x{\in}\mathbf{nat}.\, \forall y{\in}\mathbf{nat}.\, \forall z{\in}\mathbf{nat}.\, x =_N y \supset y =_N z \supset x =_N z$$

and therefore *trans* is a function of five arguments: natural numbers x, y, and z and proof terms $u{:}x =_N y$ and $w{:}y =_N z$. It has to return a proof term $M : x =_N z$. The proof above corresponds to the following specification.

$$\begin{array}{llllllll}
\mathit{trans} & \mathbf{0} & \mathbf{0} & \mathbf{0} & u & w & = & \mathbf{eq}_0 \\
\mathit{trans} & \mathbf{0} & \mathbf{0} & (\mathbf{s}(z')) & u & w & = & \mathbf{eqE}_{0s}(w) \\
\mathit{trans} & \mathbf{0} & (\mathbf{s}(y')) & z & u & w & = & \mathbf{eqE}_{0s}(u) \\
\mathit{trans} & (\mathbf{s}(x')) & \mathbf{0} & z & u & w & = & \mathbf{eqE}_{s0}(u) \\
\mathit{trans} & (\mathbf{s}(x')) & (\mathbf{s}(y')) & \mathbf{0} & u & w & = & \mathbf{eqE}_{s0}(w) \\
\mathit{trans} & (\mathbf{s}(x')) & (\mathbf{s}(y')) & (\mathbf{s}(z')) & u & w & = & \\
\multicolumn{8}{r}{\mathbf{eq}_s(\mathit{trans}\ x'\ y'\ z'\ (\mathbf{eqE}_{ss}(u))\ (\mathbf{eqE}_{ss}(w)))}
\end{array}$$

Note that all but the first and the last case are impossible, for which we provide evidence by applying the right elimination rule to either u or w. We can also see that the first argument to the recursive call to *trans* is at x' and the specificaton above therefore satisfies the restriction on primitive recursion. By comparing this to the formal proof (and also the omitted proof term) we can see the programming with equational specifications of this kind is much simpler and more concise than many other representations. There is ongoing research on directly verifying and compiling specifications, which is close to actual programming practice in languages such as ML or Haskell.

Symmetry of equality can be proven in a similar way. This proof and the corresponding specification and proof term are left as an exercise to the reader.

A second class of example moves us closer the extraction of functional programs on natural numbers from constructive proofs. Keeping in mind the constructive interpretation of the existential quantifier, how could we specify the predecessor operation? There are many possible answers to this. Here we would like express that the predecessor should only be applied to positive natural numbers.

$$\forall x{\in}\mathbf{nat}.\, \neg x =_N \mathbf{0} \supset \exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N x$$

We can prove this by cases on x. Formally, this takes the form of an induction in which the induction hypothesis is not used.

Proof: The proof proceeds by cases on x.

Case: $x = \mathbf{0}$. Then the assumption $\neg 0 =_N 0$ is contradictory.

Case: $x = \mathbf{s}(x')$. Assume $\neg \mathbf{s}(x') =_N 0$. We have to show that $\exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N \mathbf{s}(x')$. This follows with the witness x' for y since $\mathbf{s}(x') =_N \mathbf{s}(x')$ by reflexivity of equality.

□

Here is the same proof in the linear notation for natural deductions.

```
[ x : nat;
  [ ~ 0 = 0;
    0 = 0;
    F;
    ?y:nat. s(y) = 0 ];
  ~ 0 = 0 => ?y:nat. s(y) = 0;    % case (x = 0)
  [ x' : nat, ~ x' = 0 => ?y:nat. s(y) = x';
    [ ~ s(x') = 0;
      !z:nat. z = z;              % reflexivity lemma
      s(x') : nat;
      s(x') = s(x');
      ?y:nat. s(y) = s(x') ];
    ~ s(x') = 0 => ?y:nat. s(y) = s(x') ];  % case (x = s(x'))
  ~ x = 0 => ?y:nat. s(y) = x ];
!x:nat. ~ x = 0 => ?y:nat. s(y) = x;
```

Next we give the equational specification of the function $pred'$ corresponding to this proof. Note that the function takes two arguments: x and a proof u of $\neg x =_N \mathbf{0}$. It returns a pair consisting of a witness n and proof that $\mathbf{s}(n) =_N x$.

$$\begin{array}{llcl} pred' & \mathbf{0} \quad u & = & \mathbf{abort}(u\ \mathbf{eq}_0) \\ pred' & (\mathbf{s}(x')) \quad u & = & \langle x', refl\,(\mathbf{s}(x')) \end{array}$$

Note that in the case of $x = \mathbf{0}$ we do not explicitly construct a pair, but abort the computation, which may have any type. This specification can be written as a program rather directly.

$$\begin{array}{lcl} pred' & : & \forall x{\in}\mathbf{nat}.\, \neg x =_N \mathbf{0} \supset \exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N x \\ pred' & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda u.\, \mathbf{abort}\,(u\ \mathbf{eq}_0)) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda u.\, \langle x', refl(\mathbf{s}(x'))\rangle) \end{array}$$

If we erase the parts of this term that are concerned purely with propositions and leave only data types we obtain

$$\begin{array}{lcl} pred' & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \_\_ \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow x' \end{array}$$

which is close to our earlier implementation of the predecessor. Erasing the **abort** clause from the impossible case has left a hole, which we denoted by __. We return to a more detailed specification of this erasure process in the next section.

First, we discuss and alternative way to connect arithmetic to functional programming. This is to write the program first and prove its properties. Recall the definition of *pred*:

$$\begin{array}{lcl} pred & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow x' \end{array}$$

Now we can prove that

$$\forall x \in \mathbf{nat}.\ \neg x =_N \mathbf{0} \supset \mathbf{s}(\mathit{pred}(x)) =_N x$$

Proof: The proof is by cases over x.

Case: $x = \mathbf{0}$. Then $\neg\mathbf{0} =_N \mathbf{0}$ is contradictory.

Case: $x = \mathbf{s}(x')$. Then

$$\begin{array}{l}\mathbf{s}(\mathit{pred}(\mathbf{s}(x'))) \\ \quad\Longrightarrow \mathbf{s}(\mathbf{rec}\ \mathbf{s}(x') \\ \qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{0} \\ \qquad\qquad\quad \mid f(\mathbf{s}(x')) \Rightarrow x') \\ \quad\Longrightarrow \mathbf{s}(x')\end{array}$$

and $\mathbf{s}(x') =_N \mathbf{s}(x')$ by reflexivity.

□

This shows that we must be able to use the rules of computation when reasoning about functions. This is not a property particular to natural numbers, but we might have to reason about functions at arbitrary types or proofs of arbitrary propositions. Reduction therefore has to be an integral part of the type theory. We will use two rules of the form

$$\frac{\Gamma \vdash M : A \qquad A \Longrightarrow A' \qquad \Gamma \vdash A\ \mathit{prop}}{\Gamma \vdash M : A'}\ \mathit{conv}$$

$$\frac{\Gamma \vdash M : A' \qquad A \Longrightarrow A' \qquad \Gamma \vdash A\ \mathit{prop}}{\Gamma \vdash M : A}\ \mathit{conv}'$$

where $A \Longrightarrow A'$ allows the reduction of a term occurring in A. A unified form of this rule where $A \Longleftrightarrow A'$ allows an arbitrary number of reduction and expansion steps in both directions is called *type conversion*. While reduction generally preserves well-formedness (see Theorem **??**), the converse does not. Generally, this is implied either from the premise or the conclusion, depending on whether we are reasoning backward or forward. Note that the conversion rules are "silent" in that the proof term M does not change.

In the formal proof, computations are omitted. They are carried out implicitly by the type checker. The question whether the resulting checking problem is decidable varies, depending on the underlying notion of computation. We return to this question when discussing the operational semantics in Section **??**.

We close this section with the formal version of the proof above. Note the use of the conversion rule *conv'*.

```
[ x : nat;
  [ ~ 0 = 0; 0 = 0; F;
    s(pred(0)) = 0 ];
  ~ 0 = 0 => s(pred(0)) = 0;    % case (x = 0)
  [ x' : nat, ~ x' = 0 => s(pred(x')) = x';
    [ ~ s(x') = 0;
      !z:nat. z = z;            % reflexivity lemma
      s(x') : nat;
      s(pred(s(x'))) = s(x') ]; % since pred(s(x')) ==> x'
    ~ s(x') = 0 => s(pred(s(x'))) = s(x') ];  % case (x = s(x'))
  ~ x = 0 => s(pred(x)) = x ];
!x:nat. ~ x = 0 => s(pred(x)) = x
```

1.4 Contracting Proofs to Programs

In this section we return to an early idea behind the computational interpretation of constructive logic: a proof of $\forall x{\in}\tau.\ \exists y{\in}\sigma.\ A(x,y)$ should describe a function f from elements of type τ to elements of type σ such that $A(x, f(x))$ is true for all x. The proof terms for intuitionistic logic and arithmetic do not quite fill this role. This is because if M is a proof term for $\forall x{\in}\tau.\ \exists y{\in}\sigma.\ A(x,y)$, then it describes a function that returns not only an appropriate term t, but also a proof term that certifies $A(x,t)$.

Thus we would like to contract proofs to programs, ignoring those parts of a proof term that are not of interest. Of course, what is and what is not of interest depends on the application. To illustrate this point and the process of erasing parts of a proof term, we consider the example of even and odd numbers. We define the addition function (in slight variation to the definition in Section **??**) and the predicates *even* and *odd*.

$$\begin{array}{rcl}
\mathit{plus} & : & \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \\
\mathit{plus} & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\
 & & \qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow \lambda y.\ y \\
 & & \qquad\ \mid\ p(\mathbf{s}(x')) \Rightarrow \lambda y.\ \mathbf{s}(p(x')\,y) \\
\mathit{even}(x) & = & \exists y{\in}\mathbf{nat}.\ \mathit{plus}\,y\,y =_N x \\
\mathit{odd}(x) & = & \exists y{\in}\mathbf{nat}.\ \mathbf{s}(\mathit{plus}\,y\,y) =_N x
\end{array}$$

For the rest of this section, we will use the more familiar notation $m + n$ for $\mathit{plus}\,m\,n$.

We can now prove that every natural number is either even or odd. First, the informal proof. For this we need a lemma (whose proof is left to the reader)

$$\mathit{lmps} \quad : \quad \forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ x + \mathbf{s}(y) =_N \mathbf{s}(x + y)$$