Constructive Logic

Frank Pfenning
Carnegie Mellon University

Draft of February 5, 2003

Material for the course *Constructive Logic* at Carnegie Mellon University, Fall 2000. Material for this course is available at
`http://www.cs.cmu.edu/~fp/courses/logic/`.

Please send comments to `fp@cs.cmu.edu`

This material is in rough draft form and is likely to contain errors. Furthermore, citations are in no way adequate or complete. Please do not cite or distribute this document.

This work was supported in part by the University Education Council at Carnegie Mellon University and by NSF Grant CCR-9619684.

Contents

Chapter 3

Proofs as Programs

In this chapter we investigate a computational interpretation of constructive proofs and relate it to functional programming. On the propositional fragment of logic this is referred to as the Curry-Howard isomorphism [How80]. From the very outset of the development of constructive logic and mathematics, a central idea has been that proofs ought to represent constructions. The Curry-Howard isomorphism is only a particularly poignant and beautiful realization of this idea. In a highly influential subsequent paper, Martin-Löf [ML80] developed it further into a more expressive calculus called *type theory*.

3.1 Propositions as Types

In order to illustrate the relationship between proofs and programs we introduce a new judgment:

$M : A$ $\quad$ M is a proof term for proposition A

We presuppose that A is a proposition when we write this judgment. We will also interpret $M : A$ as "*M is a program of type A*". These dual interpretations of the same judgment is the core of the Curry-Howard isomorphism. We either think of M as a term that represents the proof of A *true*, or we think of A as the type of the program M. As we discuss each connective, we give both readings of the rules to emphasize the analogy.

We intend that if $M : A$ then A *true*. Conversely, if A *true* then $M : A$. But we want something more: every deduction of $M : A$ should correspond to a deduction of A *true* with an identical structure and vice versa. In other words we annotate the inference rules of natural deduction with proof terms. The property above should then be obvious.

Conjunction. Constructively, we think of a proof of $A \wedge B$ *true* as a pair of proofs: one for A *true* and one for B *true*.

$$\frac{M : A \qquad N : B}{\langle M, N\rangle : A \wedge B}\,\wedge I$$

The elimination rules correspond to the projections from a pair to its first and second elements.

$$\frac{M : A \wedge B}{\mathbf{fst}\, M : A}\wedge E_L \qquad \frac{M : A \wedge B}{\mathbf{snd}\, M : B}\wedge E_R$$

Hence conjunction $A \wedge B$ corresponds to the product type $A \times B$.

Truth. Constructively, we think of a proof of $\top$ *true* as a unit element that carries now information.

$$\frac{}{\langle\,\rangle : \top}\top I$$

Hence $\top$ corresponds to the unit type $\mathbf{1}$ with one element. There is no elimination rule and hence no further proof term constructs for truth.

Implication. Constructively, we think of a proof of $A \supset B$ *true* as a function which transforms a proof of A *true* into a proof of B *true*.

In mathematics and many programming languages, we define a function f of a variable x by writing $f(x) = \ldots$ where the right-hand side "$\ldots$" depends on x. For example, we might write $f(x) = x^2 + x - 1$. In functional programming, we can instead write $f = \lambda x.\ x^2 + x - 1$, that is, we explicitly form a functional object by λ-*abstraction* of a variable (x, in the example).

We now use the notation of λ-abstraction to annotate the rule of implication introduction with proof terms. In the official syntax, we label the abstraction with a proposition (writing $\lambda u{:}A$) in order to specify the domain of a function unambiguously. In practice we will often omit the label to make expressions shorter—usually (but not always!) it can be determined from the context.

$$\begin{array}{c}\overline{u : A}\,u\\ \vdots\\ M : B\end{array}\over{\lambda u{:}A.\ M : A \supset B}\supset I^u$$

The hypothesis label u acts as a variable, and any use of the hypothesis labeled u in the proof of B corresponds to an occurrence of u in M.

As a concrete example, consider the (trivial) proof of $A \supset A$ *true*:

$$\frac{\begin{array}{c}\overline{A\ \mathit{true}}\,u\end{array}}{A \supset A\ \mathit{true}}\supset I^u$$

If we annotate the deduction with proof terms, we obtain

$$\frac{\begin{array}{c}\overline{u : A}\,u\end{array}}{(\lambda u{:}A.\ u) : A \supset A}\supset I^u$$

So our proof corresponds to the identity function id at type A which simply returns its argument. It can be defined with $\mathrm{id}(u) = u$ or $\mathrm{id} = (\lambda u{:}A.\ u)$.

The rule for implication elimination corresponds to function application. Following the convention in functional programming, we write $M\,N$ for the application of the function M to argument N, rather than the more verbose $M(N)$.

$$\frac{M : A \supset B \qquad N : A}{M\,N : B}\,{\supset}E$$

What is the meaning of $A \supset B$ as a type? From the discussion above it should be clear that it can be interpreted as a function type $A \to B$. The introduction and elimination rules for implication can also be viewed as formation rules for functional abstraction $\lambda u{:}A.\ M$ and application $M\,N$.

Note that we obtain the usual introduction and elimination rules for implication if we erase the proof terms. This will continue to be true for all rules in the remainder of this section and is immediate evidence for the soundness of the proof term calculus, that is, if $M : A$ then A *true*.

As a second example we consider a proof of $(A \wedge B) \supset (B \wedge A)$ *true*.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{B\ \mathit{true}}\,{\wedge}E_R \quad \dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{A\ \mathit{true}}\,{\wedge}E_L}{B \wedge A\ \mathit{true}}\,{\wedge}I}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\,{\supset}I^u$$

When we annotate this derivation with proof terms, we obtain a function which takes a pair $\langle M, N\rangle$ and returns the reverse pair $\langle N, M\rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathbf{snd}\,u : B}\,{\wedge}E_R \quad \dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathbf{fst}\,u : A}\,{\wedge}E_L}{\langle \mathbf{snd}\,u, \mathbf{fst}\,u\rangle : B \wedge A}\,{\wedge}I}{(\lambda u.\ \langle \mathbf{snd}\,u, \mathbf{fst}\,u\rangle) : (A \wedge B) \supset (B \wedge A)}\,{\supset}I^u$$

Disjunction. Constructively, we think of a proof of $A \vee B$ *true* as either a proof of A *true* or B *true*. Disjunction therefore corresponds to a disjoint sum type $A + B$, and the two introduction rules correspond to the left and right injection into a sum type.

$$\frac{M : A}{\mathbf{inl}^B\,M : A \vee B}\,{\vee}I_L \qquad \frac{N : B}{\mathbf{inr}^A\,N : A \vee B}\,{\vee}I_R$$

In the official syntax, we have annotated the injections **inl** and **inr** with propositions B and A, again so that a (valid) proof term has an unambiguous type. In

writing actual programs we usually omit this annotation. The elimination rule corresponds to a case construct which discriminates between a left and right injection into a sum types.

$$\frac{\begin{array}{ccc} & \overline{}\,u & \overline{}\,w \\ & u : A & w : B \\ & \vdots & \vdots \\ M : A \vee B & N : C & O : C \end{array}}{\mathbf{case}\, M\, \mathbf{of}\, \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O : C}\, \vee E^{u,w}$$

Recall that the hypothesis labeled u is available only in the proof of the second premise and the hypothesis labeled w only in the proof of the third premise. This means that the scope of the variable u is N, while the scope of the variable w is O.

Falsehood. There is no introduction rule for falsehood ($\bot$). We can therefore view it as the empty type $\mathbf{0}$. The corresponding elimination rule allows a term of $\bot$ to stand for an expression of any type when wrapped with **abort**. However, there is no computation rule for it, which means during computation of a valid program we will never try to evaluate a term of the form $\mathbf{abort}\, M$.

$$\frac{M : \bot}{\mathbf{abort}^C\, M : C}\, \bot E$$

As before, the annotation C which disambiguates the type of $\mathbf{abort}\, M$ will often be omitted.

This completes our assignment of proof terms to the logical inference rules. Now we can interpret the interaction laws we introduced early as programming exercises. Consider the left-to-right direction of (L11)

(L11a) $\quad (A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C)$ *true*

Interpreted constructively, this assignment can be read as:

> Write a function which, when given a function from A to pairs of type $B \wedge C$, returns two functions: one which maps A to B and one which maps A to C.

This is satisfied by the following function:

$$\lambda u.\, \langle (\lambda w.\, \mathbf{fst}\, (u\, w)), (\lambda v.\, \mathbf{snd}\, (u\, v)) \rangle$$

The following deduction provides the evidence:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{w : A}\,w}{u\,w : B \wedge C}\supset E}{\mathbf{fst}\,(u\,w) : B}\wedge E_L}{\lambda w.\ \mathbf{fst}\,(u\,w) : A \supset B}\supset I^w \quad \dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{v : A}\,v}{u\,v : B \wedge C}\supset E}{\mathbf{snd}\,(u\,v) : C}\wedge E_R}{\lambda v.\ \mathbf{snd}\,(u\,v) : A \supset C}\supset I^v}{\langle(\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v))\rangle : (A \supset B) \wedge (A \supset C)}\wedge I}{\lambda u.\ \langle(\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v))\rangle : (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))}\supset I^u$$

Programs in constructive propositional logic are somewhat uninteresting in that they do not manipulate basic data types such as natural numbers, integers, lists, trees, etc. We introduce such data types in Section 3.5, following the same method we have used in the development of logic.

To close this section we recall the guiding principles behind the assignment of proof terms to deductions.

1. For every deduction of A *true* there is a proof term M and deduction of $M : A$.

2. For every deduction of $M : A$ there is a deduction of A *true*

3. The correspondence between proof terms M and deductions of A *true* is a bijection.

We will prove these in Section 3.4.

3.2 Reduction

In the preceding section, we have introduced the assignment of proof terms to natural deductions. If proofs are programs then we need to explain how proofs are to be executed, and which results may be returned by a computation.

We explain the operational interpretation of proofs in two steps. In the first step we introduce a judgment of *reduction* $M \Longrightarrow M'$, read "M *reduces to* M'". A computation then proceeds by a sequence of reductions $M \Longrightarrow M_1 \Longrightarrow M_2 \ldots$, according to a fixed strategy, until we reach a value which is the result of the computation. In this section we cover reduction; we return to reduction strategies in Section **??**.

As in the development of propositional logic, we discuss each of the connectives separately, taking care to make sure the explanations are independent. This means we can consider various sublanguages and we can later extend our logic or programming language without invalidating the results from this section. Furthermore, it greatly simplifies the analysis of properties of the reduction rules.

In general, we think of the proof terms corresponding to the introduction rules as the *constructors* and the proof terms corresponding to the elimination rules as the *destructors*.

Conjunction. The constructor forms a pair, while the destructors are the left and right projections. The reduction rules prescribe the actions of the projections.

$$\begin{array}{rcl} \mathbf{fst}\,\langle M, N\rangle & \Longrightarrow & M \\ \mathbf{snd}\,\langle M, N\rangle & \Longrightarrow & N \end{array}$$

Truth. The constructor just forms the unit element, $\langle\,\rangle$. Since there is no destructor, there is no reduction rule.

Implication. The constructor forms a function by λ-abstraction, while the destructor applies the function to an argument. In general, the application of a function to an argument is computed by *substitution*. As a simple example from mathematics, consider the following equivalent definitions

$$f(x) = x^2 + x - 1 \qquad f = \lambda x.\, x^2 + x - 1$$

and the computation

$$f(3) = (\lambda x.\, x^2 + x - 1)(3) = [3/x](x^2 + x - 1) = 3^2 + 3 - 1 = 11$$

In the second step, we substitute 3 for occurrences of x in $x^2 + x - 1$, the *body of the λ-expression*. We write $[3/x](x^2 + x - 1) = 3^2 + 3 - 1$.

In general, the notation for the substitution of N for occurrences of u in M is $[N/u]M$. We therefore write the reduction rule as

$$(\lambda u{:}A.\, M)\, N \quad\Longrightarrow\quad [N/u]M$$

We have to be somewhat careful so that substitution behaves correctly. In particular, no variable in N should be bound in M in order to avoid conflict. We can always achieve this by renaming bound variables—an operation which clearly does not change the meaning of a proof term.

Disjunction. The constructors inject into a sum types; the destructor distinguishes cases. We need to use substitution again.

$$\begin{array}{rcl} \mathbf{case}\,\mathbf{inl}^B\, M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/u]N \\ \mathbf{case}\,\mathbf{inr}^A\, M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/w]O \end{array}$$

Falsehood. Since there is no constructor for the empty type there is no reduction rule for falsehood.

This concludes the definition of the reduction judgment. In the next section we will prove some of its properties.

As an example we consider a simple program for the composition of two functions. It takes a pair of two functions, one from A to B and one from B to C and returns their composition which maps A directly to C.

$$\mathsf{comp} \quad : \quad ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$$

We transform the following implicit definition into our notation step-by-step:

$$\begin{array}{rcl} \mathsf{comp}\,\langle f, g\rangle\,(w) & = & g(f(w)) \\ \mathsf{comp}\,\langle f, g\rangle & = & \lambda w.\ g(f(w)) \\ \mathsf{comp}\,u & = & \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)(w)) \\ \mathsf{comp} & = & \lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) \end{array}$$

The final definition represents a correct proof term, as witnessed by the following deduction.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{snd}\,u : B \supset C}\wedge E_R \qquad \dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{fst}\,u : A \supset B}\wedge E_L \quad \dfrac{}{w : A}\,w}{(\mathbf{fst}\,u)\,w : B}\supset E}{(\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : C}\supset E}{\lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : A \supset C}\supset I^w}{(\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w)) : ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)}\supset I^u$$

We now verify that the composition of two identity functions reduces again to the identity function. First, we verify the typing of this application.

$$(\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w))\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle \quad : \quad A \supset A$$

Now we show a possible sequence of reduction steps. This is by no means uniquely determined.

$$\begin{array}{rl} & (\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w))\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle \\ \Longrightarrow & \lambda w.\ (\mathbf{snd}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,((\mathbf{fst}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,w) \\ \Longrightarrow & \lambda w.\ (\lambda y.\ y)\,((\mathbf{fst}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,w) \\ \Longrightarrow & \lambda w.\ (\lambda y.\ y)\,((\lambda x.\ x)\,w) \\ \Longrightarrow & \lambda w.\ (\lambda y.\ y)\,w \\ \Longrightarrow & \lambda w.\ w \end{array}$$

We see that we may need to apply reduction steps to subterms in order to reduce a proof term to a form in which it can no longer be reduced. We postpone a more detailed discussion of this until we discuss the operational semantics in full.

3.3 Summary of Proof Terms

Judgments.

$M : A$	M is a proof term for proposition A
$M \Longrightarrow M'$	M reduces to M'

Proof Term Assignment.

Constructors	Destructors
$\dfrac{M : A \qquad N : B}{\langle M, N\rangle : A \wedge B}\,\wedge I$	$\dfrac{M : A \wedge B}{\mathbf{fst}\, M : A}\,\wedge E_L$ $\dfrac{M : A \wedge B}{\mathbf{snd}\, M : B}\,\wedge E_R$
$\dfrac{}{\langle\,\rangle : \top}\,\top I$	no destructor for $\top$
$\dfrac{\begin{array}{c}\overline{u : A}\,u \\ \vdots \\ M : B\end{array}}{\lambda u{:}A.\, M : A \supset B}\,\supset I^u$	$\dfrac{M : A \supset B \qquad N : A}{M\, N : B}\,\supset E$
$\dfrac{M : A}{\mathbf{inl}^B\, M : A \vee B}\,\vee I_L$ $\dfrac{N : B}{\mathbf{inr}^A\, N : A \vee B}\,\vee I_R$	$\dfrac{M : A \vee B \qquad \begin{array}{c}\overline{u : A}\,u \\ \vdots \\ N : C\end{array} \qquad \begin{array}{c}\overline{w : B}\,w \\ \vdots \\ O : C\end{array}}{\mathbf{case}\, M\, \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O : C}\,\vee E^{u,w}$
no constructor for $\bot$	$\dfrac{M : \bot}{\mathbf{abort}^C\, M : C}\,\bot E$

Reductions.

$$\begin{array}{rcl}
\mathbf{fst}\,\langle M, N\rangle & \Longrightarrow & M \\
\mathbf{snd}\,\langle M, N\rangle & \Longrightarrow & N \\
\text{no reduction for } \langle\,\rangle & & \\
(\lambda u{:}A.\ M)\,N & \Longrightarrow & [N/u]M \\
\mathbf{case}\,\mathbf{inl}^B\,M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/u]N \\
\mathbf{case}\,\mathbf{inr}^A\,M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/w]O \\
\text{no reduction for } \mathbf{abort} & &
\end{array}$$

Concrete Syntax. The concrete syntax for proof terms used in the mechanical proof checker has some minor differences to the form we presented above.

u	`u`	Variable
$\langle M, N\rangle$	`(M,N)`	Pair
$\mathbf{fst}\,M$	`fst M`	First projection
$\mathbf{snd}\,M$	`snd M`	Second projection
$\langle\,\rangle$	`()`	Unit element
$\lambda u{:}A.\ M$	`fn u => M`	Abstraction
$M\,N$	`M N`	Application
$\mathbf{inl}^B\,M$	`inl M`	Left injection
$\mathbf{inr}^A\,N$	`inr N`	Right injection
$\mathbf{case}\,M$	`case M`	Case analysis
$\quad\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N$	`  of inl u => N`	
$\quad\mid \mathbf{inr}\,w \Rightarrow O$	`   | inr w => O`	
	`end`	
$\mathbf{abort}^C\,M$	`abort M`	Abort

Pairs and unit element are delimited by parentheses '`(`' and '`)`' instead of angle brackets $\langle$ and $\rangle$. The `case` constructs requires an `end` token to mark the end of the a sequence of cases.

Type annotations are generally omitted, but a whole term can explicitly be given a type. The proof checker (which here is also a type checker) infers the missing information. Occasionally, an explicit type ascription `M : A` is necessary as a hint to the type checker.

For rules of operator precedence, the reader is refered to the on-line documentation of the proof checking software available with the course material. Generally, parentheses can be used to disambiguate or override the standard rules.

As an example, we show the proof term implementing function composition.

```
term comp : (A => B) & (B => C) => (A => C) =
fn u => fn x => (snd u) ((fst u) x);
```

We also allow annotated deductions, where each line is annotated with a proof term. This is a direct transcription of deduction for judgments of the form $M : A$. As an example, we show the proof that $A \vee B \supset B \vee A$, first in the pure form.

```
proof orcomm : A | B => B | A =
begin
[ A | B;
  [ A;
    B | A];
  [ B;
    B | A];
  B | A ];
A | B => B | A
end;
```

Now we systematically annotate each line and obtain

```
annotated proof orcomm : A | B => B | A =
begin
[ u : A | B;
  [ v : A;
    inr v : B | A];
  [ w : B;
    inl w : B | A];
  case u
    of inl v => inr v
     | inr w => inl w
   end : B | A ];
fn u => case u
          of inl v => inr v
           | inr w => inl w
         end : A | B => B | A
end;
```

3.4 Properties of Proof Terms

In this section we analyze and verify various properties of proof terms. Rather than concentrate on reasoning within the logical calculi we introduced, we now want to reason about them. The techniques are very similar—they echo the ones we have introduced so far in natural deduction. This should not be surprising. After all, natural deduction was introduced to model mathematical reasoning, and we now engage in some mathematical reasoning about proof terms, propositions, and deductions. We refer to this as *meta-logical reasoning*.

First, we need some more formal definitions for certain operations on proof terms, to be used in our meta-logical analysis. One rather intuitive property of is that variable names should not matter. For example, the identity function at type A can be written as $\lambda u{:}A.\ u$ or $\lambda w{:}A.\ w$ or $\lambda u'{:}A.\ u'$, etc. They all denote the same function and the same proof. We therefore identify terms which differ only in the names of variables (here called u) bound in $\lambda u{:}A.\ M$, $\mathbf{inl}\, u \Rightarrow M$ or $\mathbf{inr}\, u \Rightarrow O$. But there are pitfalls with this convention: variables have to be renamed *consistently* so that every variable refers to the same binder before and after the renaming. For example (omitting type labels for brevity):

$$\begin{array}{rcl} \lambda u.\ u & = & \lambda w.\ w \\ \lambda u.\ \lambda w.\ u & = & \lambda u'.\ \lambda w.\ u' \\ \lambda u.\ \lambda w.\ u & \neq & \lambda u.\ \lambda w.\ w \\ \lambda u.\ \lambda w.\ u & \neq & \lambda w.\ \lambda w.\ w \\ \lambda u.\ \lambda w.\ w & = & \lambda w.\ \lambda w.\ w \end{array}$$

The convention to identify terms which differ only in the naming of their bound variables goes back to the first papers on the λ-calculus by Church and Rosser [CR36], is called the "*variable name convention*" and is pervasive in the literature on programming languages and λ-calculi. The term λ-calculus typically refers to a pure calculus of functions formed with λ-abstraction. Our proof term calculus is called a *typed λ-calculus* because of the presence of propositions (which an be viewed as types).

Following the variable name convention, we may silently rename when convenient. A particular instance where this is helpful is substitution. Consider

$$[u/w](\lambda u.\ w\, u)$$

that is, we substitute u for w in $\lambda u.\ w\, u$. Note that u is a variable visible on the outside, but also bound by λu. By the variable name convention we have

$$[u/w](\lambda u.\ w\, u) = [u/w](\lambda u'.\ w\, u') = \lambda u'.\ u\, u'$$

which is correct. But we cannot substitute without renaming, since

$$[u/w](\lambda u.\ w\, u) \neq \lambda u.\ u\, u$$

In fact, the right hand side below is invalid, while the left-hand side makes perfect sense. We say that u is *captured* by the binder λu. If we assume a hypothesis $u{:}\top \supset A$ then

$$[u/w](\lambda u{:}\top.\ w\, u) : A$$

but

$$\lambda u{:}\top.\ u\, u$$

is not well-typed since the first occurrence of u would have to be of type $\top \supset A$ but instead has type $\top$.

So when we carry out substitution $[M/u]N$ we need to make sure that no variable in M is *captured* by a binder in N, leading to an incorrect result.

Fortunately we can always achieve that by renaming some bound variables in N if necessary. We could now write down a formal definition of substitution, based on the cases for the term we are substituting into. However, we hope that the notion is sufficiently clear that this is not necessary.

Instead we revisit the substitution principle for hypothetical judgments. It states that if we have a hypothetical proof of C *true* from A *true* and we have a proof of A *true*, we can substitute the proof of A *true* for uses of the hypothesis A *true* and obtain a (non-hypothetical) proof of A *true*. In order to state this more precisely in the presence of several hypotheses, we recall that

$$\begin{array}{c} A_1\ \mathit{true} \ldots A_n\ \mathit{true} \\ \vdots \\ C\ \mathit{true} \end{array}$$

can be written as

$$\underbrace{A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}}_{\Delta} \vdash C\ \mathit{true}$$

Generally we abbreviate several hypotheses by Δ. We then have the following properties, evident from the very definition of hypothetical judgments and hypothetical proofs

Weakening: If $\Delta \vdash C\ \mathit{true}$ then $\Delta, \Delta' \vdash C\ \mathit{true}$.

Substitution: If $\Delta, A\ \mathit{true}, \Delta' \vdash C\ \mathit{true}$ and $\Delta \vdash A\ \mathit{true}$ then $\Delta, \Delta' \vdash C\ \mathit{true}$.

As indicated above, weakening is realized by adjoining unused hypotheses, substitutions is realized by substitution of proofs for hypotheses.

For the proof term judgment, $M : A$, we use the same notation and write

$$\begin{array}{c} u_1{:}A_1\ \ldots\ u_n{:}A_n \\ \vdots \\ N : C \end{array}$$

as

$$\underbrace{u_1{:}A_1, \ldots, u_n{:}A_n}_{\Gamma} \vdash N : C$$

We use Γ to refer to collections of hypotheses $u_i{:}A_i$. In the deduction of $N : C$, each u_i stands for an unknown proof term for A_i, simply assumed to exist. If we actually find a proof $M_i{:}A_i$ we can eliminate this assumption, again by substitution. However, this time, the substitution has to perform two operations: we have to substitute M_i for u_i (the unknown proof term variable), and the deduction of $M_i : A_i$ for uses of the hypothesis $u_i{:}A_i$. More precisely, we have the following two properties:

Weakening: If $\Gamma \vdash N : C$ then $\Gamma, \Gamma' \vdash N : C$.

Substitution: If $\Gamma, u{:}A, \Gamma' \vdash N : C$ and $\Gamma \vdash M : A$ then $\Gamma, \Gamma' \vdash [M/u]N : C$.

Now we are in a position to state and prove our second meta-theorem, that is, a theorem about the logic under consideration. The theorem is called *subject reduction* because is concerns the *subject* M of the judgment $M : A$. It states that reduction preserves the type of an object. We make the hypotheses explicit as we have done in the explanations above.

Theorem 3.1 (Subject Reduction)
If $\Gamma \vdash M : A$ *and* $M \Longrightarrow M'$ *then* $\Gamma \vdash M' : A$.

Proof: We consider each case in the definition of $M \Longrightarrow M'$ in turn and show that the property holds. This is simply an instance of *proof by cases*.

Case: $\mathbf{fst}\,\langle M_1, M_2\rangle \Longrightarrow M_1$. By assumption we also know that

$$\Gamma \vdash \mathbf{fst}\,\langle M_1, M_2\rangle : A.$$

We need to show that $\Gamma \vdash M_1 : A$.

Now we inspect all inference rules for the judgment $M : A$ and we see that there is only one way how the judgment above could have been inferred: by $\wedge E_L$ from

$$\Gamma \vdash \langle M_1, M_2\rangle : A \wedge A_2$$

for some A_2. This step is called *inversion*, since we infer the premises from the conclusion of the rule. But we have to be extremely careful to inspect all possibilities for derivations so that we do not forget any cases.

Next, we apply inversion again: the judgment above could only have been inferred by $\wedge I$ from the two premises

$$\Gamma \vdash M_1 : A$$

and

$$\Gamma \vdash M_2 : A_2$$

But the first of these is what we had to prove in this case and we are done.

Case: $\mathbf{snd}\,\langle M_1, M_2\rangle \Longrightarrow M_2$. This is symmetric to the previous case. We write it an abbreviated form.

$\Gamma \vdash \mathbf{snd}\,\langle M_1, M_2\rangle : A$	Assumption
$\Gamma \vdash \langle M_1, M_2\rangle : A_1 \wedge A$ for some A_1	By inversion
$\Gamma \vdash M_1 : A_1$ and	
$\Gamma \vdash M_2 : A$	By inversion

Here the last judgment is what we were trying to prove.

Case: There is no reduction for $\top$ since there is no elimination rule and hence no destructor.

Case: $(\lambda u{:}A_1.\ M_2)\, M_1 \Longrightarrow [M_1/u]M_2$. By assumption we also know that

$$\Gamma \vdash (\lambda u{:}A_1.\ M_2)\, M_1 : A.$$

We need to show that $\Gamma \vdash [M_1/u]M_2 : A$.

Since there is only one inference rule for function application, namely implication elimination ($\supset E$), we can apply inversion and find that

$$\Gamma \vdash (\lambda u{:}A_1.\ M_2) : A_1' \supset A$$

and

$$\Gamma \vdash M_1 : A_1'$$

for some A_1'. Now we repeat inversion on the first of these and conclude that

$$\Gamma, u{:}A_1 \vdash M_2 : A$$

and, moreover, that $A_1 = A_1'$. Hence

$$\Gamma \vdash M_1 : A_1$$

Now we can apply the substitution property to these to judgments to conclude

$$\Gamma \vdash [M_1/u]M_2 : A$$

which is what we needed to show.

Case: $(\mathbf{case}\,\mathbf{inl}^C\, M_1\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow [M_1/u]N$. By assumption we also know that

$$\Gamma \vdash (\mathbf{case}\,\mathbf{inl}^C\, M_1\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) : A$$

Again we apply inversion and obtain three judgments

$$\begin{array}{l}\Gamma \vdash \mathbf{inl}^C\, M_1 : B' \vee C' \\ \Gamma, u{:}B' \vdash N : A \\ \Gamma, w{:}C' \vdash O : A\end{array}$$

for some B' and C'.

Again by inversion on the first of these, we find

$$\Gamma \vdash M_1 : B'$$

and also $C' = C$. Hence we can apply the substitution property to get

$$\Gamma \vdash [M_1/u]N : A$$

which is what we needed to show.

Case: $(\mathbf{case}\,\mathbf{inr}^B\, M_1\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow [M_1/u]N$. This is symmetric to the previous case and left as an exercise.

Case: There is no introduction rule for $\bot$ and hence no reduction rule.

□

The important techniques introduced in the proof above are *proof by cases* and *inversion*. In a proof by cases we simply consider all possibilities for why a judgment could be evident and show the property we want to establish in each case. Inversion is very similar: from the shape of the judgment we see it could have been inferred only in one possible way, so we know the premises of this rule must also be evident. We see that these are just two slightly different forms of the same kind of reasoning.

If we look back at our early example computation, we saw that the reduction step does not always take place at the top level, but that the redex may be embedded in the term. In order to allow this, we need to introduce some additional ways to establish that $M \Longrightarrow M'$ when the actual reduction takes place *inside* M. This is accomplished by so-called *congruence rules*.

Conjunction. As usual, conjunction is the simplest.

$$\frac{M \Longrightarrow M'}{\langle M, N\rangle \Longrightarrow \langle M', N\rangle} \qquad \frac{N \Longrightarrow N'}{\langle M, N\rangle \Longrightarrow \langle M, N'\rangle}$$

$$\frac{M \Longrightarrow M'}{\mathbf{fst}\, M \Longrightarrow \mathbf{fst}\, M'} \qquad \frac{M \Longrightarrow M'}{\mathbf{snd}\, M \Longrightarrow \mathbf{snd}\, M'}$$

Note that there is one rule for each subterm for each construct in the language of proof terms, just in case the reduction might take place in that subterm.

Truth. There are no rules for truth, since $\langle\rangle$ has no subterms and therefore permits no reduction inside.

Implication. This is similar to conjunction.

$$\frac{M \Longrightarrow M'}{M\, N \Longrightarrow M'\, N} \qquad \frac{N \Longrightarrow N'}{M\, N \Longrightarrow M\, N'}$$

$$\frac{M \Longrightarrow M'}{(\lambda u{:}A.\ M) \Longrightarrow (\lambda u{:}A.\ M')}$$

Disjunction. This requires no new ideas, just more cases.

$$\frac{M \Longrightarrow M'}{\mathbf{inl}^B\, M \Longrightarrow \mathbf{inl}^B\, M'} \qquad \frac{N \Longrightarrow N'}{\mathbf{inr}^A\, N \Longrightarrow \mathbf{inr}^A\, N'}$$

$$\frac{M \Longrightarrow M'}{(\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow (\mathbf{case}\, M'\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O)}$$

$$\frac{N \Longrightarrow N'}{(\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow (\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N' \mid \mathbf{inr}\, w \Rightarrow O)}$$

$$\frac{O \Longrightarrow O'}{(\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow (\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O')}$$

Falsehood. Finally, there *is* a congruence rule for falsehood, since the proof term constructor has a subterm.

$$\frac{M \Longrightarrow M'}{\mathbf{abort}^C\, M \Longrightarrow \mathbf{abort}^C\, M'}$$

We now extend the theorem to the general case of reduction on subterms. A proof by cases is now no longer sufficient, since the congruence rules have premises, for which we would have to analyze cases again, and again, etc.

Instead we use a technique called *structural induction* on proofs. In structural induction we analyse each inference rule, assuming the desired property for the premises, proving that they hold for the conclusion. If that is the case for all inference rules, the conclusion of each deduction must have the property.

Theorem 3.2 (Subterm Subject Reduction)
If $\Gamma \vdash M : A$ *and* $M \Longrightarrow M'$ *then* $\Gamma \vdash M' : A$ *where* $M \Longrightarrow M'$ *refers to the congruent interpretation of reduction.*

Proof: The cases where the reduction takes place at the top level of the term M, the cases in the proof of Theorem 3.1 still apply. The new cases are all very similar, and we only show one.

Case: The derivation of $M \Longrightarrow M'$ has the form

$$\frac{M_1 \Longrightarrow M_1'}{\langle M_1, M_2\rangle \Longrightarrow \langle M_1', M_2\rangle}$$

We also know that $\Gamma \vdash \langle M_1, M_2\rangle : A$. We need to show that

$$\Gamma \vdash \langle M_1', M_2\rangle : A$$

By inversion,

$$\Gamma \vdash M_1 : A_1$$

and

$$\Gamma \vdash M_2 : A_2$$

and $A = A_1 \wedge A_2$.

Since we are proving the theorem by structural induction and we have a deduction of $\Gamma \vdash M_1 : A_1$ we can now apply the induction hypothesis to $M_1 \Longrightarrow M_1'$. This yields

$$\Gamma \vdash M_1' : A_1$$

and we can construct the deduction

$$\frac{\Gamma \vdash M_1' : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1', M_2 \rangle : A_1 \wedge A_2} \wedge I$$

which is what we needed to show since $A = A_1 \wedge A_2$.

Cases: All other cases are similar and left as an exercise.

□

The importance of the technique of structural induction cannot be overemphasized in this domain. We will see it time and again, so the reader should make sure the understand each step in the proof above.

3.5 Primitive Recursion

In the preceding sections we have developed an interpretation of propositions as types. This interpretation yields function types (from implication), product types (from conjunction), unit type (from truth), sum types (from disjunction) and the empty type (from falsehood). What is missing for a reasonable programming language are basic data types such as natural numbers, integers, lists, trees, etc. There are several approaches to incorporating such types into our framework. One is to add a general definition mechanism for *recursive types* or *inductive types*. We return to this option later. Another one is to specify each type in a way which is analogous to the definitions of the logical connectives via introduction and elimination rules. This is the option we pursue in this section. A third way is to use the constructs we already have to define data. This was Church's original approach culminating in the so-called *Church numerals*. We will not discuss this idea in these notes.

After spending some time to illustrate the interpretation of propositions as types, we now introduce types as a first-class notion. This is not strictly necessary, but it avoids the question what, for example, **nat** (the type of natural numbers) means as a proposition. Accordingly, we have a new judgment τ *type* meaning "τ *is a type*". To understand the meaning of a type means to understand what elements it has. We therefore need a second judgment $t \in \tau$ (read:

"*t is an element of type* τ") that is defined by introduction rules with their corresponding elimination rules. As in the case of logical connectives, computation arises from the meeting of elimination and introduction rules. Needless to say, we will continue to use our mechanisms of hypothetical judgments.

Before introducing any actual data types, we look ahead at their use in logic. We will introduce new propositions of the form $\forall x \in \tau.\ A(x)$ (*A is true for every element x of type* τ) and $\exists x \in \tau.\ A(x)$ (*A is true some some element x of type* τ). This will be the step from propositional logic to first-order logic. This logic is called *first-order* because we can quantify (via $\forall$ and $\exists$) only over elements of data types, but not propositions themselves.

We begin our presentation of data types with the natural numbers. The formation rule is trivial: **nat** is a type.

$$\frac{}{\mathbf{nat}\ \mathit{type}}\ \mathbf{nat}F$$

Now we state two of Peano's famous axioms in judgmental form as introduction rules: (1) **0** is a natural numbers, and (2) if n is a natural number then its successor, $\mathbf{s}(n)$, is a natural number. We write $\mathbf{s}(n)$ instead of $n + 1$, since addition and the number 1 have yet to be defined.

$$\frac{}{\mathbf{0} \in \mathbf{nat}}\ \mathbf{nat}I_0 \qquad\qquad \frac{n \in \mathbf{nat}}{\mathbf{s}(n) \in \mathbf{nat}}\ \mathbf{nat}I_s$$

The elimination rule is a bit more difficult to construct. Assume have a natural number n. Now we cannot directly take its predecessor, for example, because we do not know if n was constructed using $\mathbf{nat}I_0$ or $\mathbf{nat}I_s$. This is similar to the case of disjunction, and our solution is also similar: we distinguish cases. In general, it turns out this is not sufficient, but our first approximation for an elimination rule is

$$\frac{\begin{array}{ccc} & & \dfrac{}{x \in \mathbf{nat}}\,x \\ & & \vdots \\ n \in \mathbf{nat} & t_0 \in \tau & t_s \in \tau \end{array}}{\mathbf{case}\, n\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s \in \tau}\,x$$

Note that x is introduced in the third premise; its scope is t_s. First, we rewrite this using our more concise notation for hypothetical judgments. For now, Γ contains assumptions of the form $x \in \tau$. Later, we will add logical assumptions of the form $u{:}A$.

$$\frac{\Gamma \vdash n \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat} \vdash t_s \in \tau}{\Gamma \vdash \mathbf{case}\, n\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s \in \tau}\,x$$

This elimination rule is sound, and under the computational interpretation of terms, type preservation holds. The reductions rules are

$$\begin{array}{rcl} (\mathbf{case}\, \mathbf{0}\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s) & \Longrightarrow & t_0 \\ (\mathbf{case}\, \mathbf{s}(n)\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s) & \Longrightarrow & [n/x]t_s \end{array}$$

Clearly, this is the intended reading of the case construct in programs.

In order to use this in writing programs independently of the logic developed earlier, we now introduce function types in a way that is isomorphic to implication.

$$\frac{\tau\ \mathit{type} \qquad \sigma\ \mathit{type}}{\tau \to \sigma\ \mathit{type}} \to F$$

$$\frac{\Gamma, x \in \sigma \vdash t \in \tau}{\Gamma \vdash \lambda x \in \sigma.\, t \in \sigma \to \tau} \to I^x \qquad \frac{\Gamma \vdash s \in \tau \to \sigma \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash s\, t \in \sigma} \to E$$

$$(\lambda x \in \sigma.\ s)\, t \Longrightarrow [t/x]s$$

Now we can write a function for truncated predecessor: the predecessor of $\mathbf{0}$ is defined to be $\mathbf{0}$; otherwise the predecessor of $n+1$ is simply n. We phrase this as a notational definition.

$$\mathit{pred} \quad = \quad \lambda x \in \mathbf{nat}.\ \mathbf{case}\, x\ \mathbf{of}\ \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y$$

Then $\vdash \mathit{pred} \in \mathbf{nat} \to \mathbf{nat}$ and we can formally calculate the predecessor of 2.

$$\begin{array}{rcl} \mathit{pred}(\mathbf{s}(\mathbf{s}(\mathbf{0}))) & = & (\lambda x \in \mathbf{nat}.\ \mathbf{case}\, x\ \mathbf{of}\ \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y)\, (\mathbf{s}(\mathbf{s}(\mathbf{0}))) \\ & \Longrightarrow & \mathbf{case}\, \mathbf{s}(\mathbf{s}(\mathbf{0}))\ \mathbf{of}\ \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y \\ & \Longrightarrow & \mathbf{s}(\mathbf{0}) \end{array}$$

As a next example, we consider a function which doubles its argument. The behavior of the *double* function on an argument can be specified as follows:

$$\begin{array}{rcl} \mathit{double}(\mathbf{0}) & = & \mathbf{0} \\ \mathit{double}(\mathbf{s}(n)) & = & \mathbf{s}(\mathbf{s}(\mathit{double}(n))) \end{array}$$

Unfortunately, there is no way to transcribe this definition into an application of the **case**-construct for natural numbers, since it is *recursive*: the right-hand side contains an occurrence of *double*, the function we are trying to define.

Fortunately, we can generalize the elimination construct for natural numbers to permit this kind of recursion which is called *primitive recursion*. Note that we can define the value of a function on $\mathbf{s}(n)$ only in terms of n and the value of the function on n. We write

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\ \mathbf{nat}E^{f,x}$$

Here, f may not occur in t_0 and can only occur in the form $f(x)$ in t_s to denote the result of the recursive call. Essentially, $f(x)$ is just the mnemonic name of a new variable for the result of the recursive call. Moreover, x is bound with scope t_s. The reduction rules are now recursive:

$$\begin{array}{rcl} (\mathbf{rec}\ \mathbf{0}\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & t_0 \\ (\mathbf{rec}\ \mathbf{s}(n)\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & \\ [(\mathbf{rec}\ n\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s)/f(x)]\, [n/x]\, t_s & & \end{array}$$

As an example we revisit the double function and give it as a notational definition.

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

Now $\mathit{double}\,(\mathbf{s}(\mathbf{0}))$ can be computed as follows

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

As some other examples, we consider the functions for addition and multiplication. These definitions are by no means uniquely determined. In each case we first give an implicit definition, describing the intended behavior of the function, and then the realization in our language.

$$\begin{array}{rcl} \mathit{plus}\,\mathbf{0}\,y & = & y \\ \mathit{plus}\,(\mathbf{s}(x'))\,y & = & \mathbf{s}(\mathit{plus}\,x'\,y) \end{array}$$

$$\begin{array}{lcl} \mathit{plus} & = & \lambda x \in \mathbf{nat}.\ \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow y \\ & & \qquad\ \mid p(\mathbf{s}(x')) \Rightarrow \mathbf{s}(p(x')) \end{array}$$

$$\begin{array}{rcl} \mathit{times}\,\mathbf{0}\,y & = & \mathbf{0} \\ \mathit{times}\,(\mathbf{s}(x'))\,y & = & \mathit{plus}\,y\,(\mathit{times}\,x'\,y) \end{array}$$

$$\begin{array}{lcl} \mathit{times} & = & \lambda x \in \mathbf{nat}.\ \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ t(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid t(\mathbf{s}(x')) \Rightarrow \mathit{plus}\,y\,(t(x')) \end{array}$$

The next example requires pairs in the language. We therefore introduce

pairs which are isomorphic to the proof terms for conjunction from before.

$$\frac{\Gamma \vdash s \in \sigma \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash \langle s, t\rangle \in \sigma \times \tau} \times I$$

$$\frac{\Gamma \vdash t \in \tau \times \sigma}{\Gamma \vdash \mathbf{fst}\, t \in \tau} \times E_L \qquad \frac{\Gamma \vdash t \in \tau \times \sigma}{\Gamma \vdash \mathbf{snd}\, t \in \sigma} \times E_R$$

$$\begin{array}{c} \mathbf{fst}\, \langle t, s\rangle \Longrightarrow t \\ \mathbf{snd}\, \langle t, s\rangle \Longrightarrow s \end{array}$$

Next the function *half*, rounding down if necessary. This is slightly trickier then the examples above, since we would like to count down by *two* as the following specification indicates.

$$\begin{array}{rcl} \mathit{half}\, \mathbf{0} & = & \mathbf{0} \\ \mathit{half}\, (\mathbf{s}(\mathbf{0})) & = & \mathbf{0} \\ \mathit{half}\, (\mathbf{s}(\mathbf{s}(x'))) & = & \mathbf{s}(\mathit{half}(x')) \end{array}$$

The first step is to break this function into two, each of which steps down by one.

$$\begin{array}{rcl} \mathit{half}_1\, \mathbf{0} & = & \mathbf{0} \\ \mathit{half}_1\, (\mathbf{s}(x')) & = & \mathit{half}_2(x') \end{array}$$

$$\begin{array}{rcl} \mathit{half}_2\, \mathbf{0} & = & \mathbf{0} \\ \mathit{half}_2\, (\mathbf{s}(x'')) & = & \mathbf{s}(\mathit{half}_1(x'')) \end{array}$$

Note that half_1 calls half_2 and vice versa. This is an example of so-called *mutual recursion*. This can be modeled by one function half_{12} returning a pair such that $\mathit{half}_{12}(x) = \langle \mathit{half}_1(x), \mathit{half}_2(x)\rangle$.

$$\begin{array}{rcl} \mathit{half}_{12}\, \mathbf{0} & = & \langle \mathbf{0}, \mathbf{0}\rangle \\ \mathit{half}_{12}\, (\mathbf{s}(x)) & = & \langle \mathbf{snd}\, (\mathit{half}_{12}(x)), \mathbf{s}(\mathbf{fst}\, (\mathit{half}_{12}(x))) \\ \mathit{half}\, x & = & \mathbf{fst}\, (\mathit{half}_{12}\, x) \end{array}$$

In our notation this becomes

$$\begin{array}{rcl} \mathit{half}_{12} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\, x \\ & & \quad \mathbf{of}\, h(\mathbf{0}) \Rightarrow \langle \mathbf{0}, \mathbf{0}\rangle \\ & & \quad \mid h(\mathbf{s}(x')) \Rightarrow \langle \mathbf{snd}\, (h(x)), \mathbf{s}(\mathbf{fst}\, (h(x)))\rangle \\ \mathit{half} & = & \lambda x \in \mathbf{nat}.\ \mathbf{fst}\, (\mathit{half}_{12}\, x) \end{array}$$

As a last example in the section, consider the subtraction function which cuts off at zero.

$$\begin{array}{rcl} \mathit{minus}\, \mathbf{0}\, y & = & \mathbf{0} \\ \mathit{minus}\, (\mathbf{s}(x'))\, \mathbf{0} & = & \mathbf{s}(x') \\ \mathit{minus}\, (\mathbf{s}(x'))\, (\mathbf{s}(y')) & = & \mathit{minus}\, x'\, y' \end{array}$$

To be presented in the schema of primitive recursion, this requires two nested case distinctions: the outermost one on the first argument x, the innermost one

on the second argument y. So the result of the first application of *minus* must be function, which is directly represented in the definition below.

$$\begin{array}{lcl} \mathit{minus} & = & \lambda x \in \mathbf{nat}.\, \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ m(\mathbf{0}) \Rightarrow \lambda y \in \mathbf{nat}.\, 0 \\ & & \quad \mid m(\mathbf{s}(x')) \Rightarrow \lambda y \in \mathbf{nat}.\, \mathbf{rec}\ y \\ & & \qquad\qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow \mathbf{s}(x') \\ & & \qquad\qquad \mid p(\mathbf{s}(y')) \Rightarrow (m\,(x'))\,y' \end{array}$$

Note that m is correctly applied only to x', while p is not used at all. So the inner recursion could have been written as a **case**-expression instead.

Functions defined by primitive recursion terminate. This is because the behavior of the function on $\mathbf{s}(n)$ is defined in terms of the behavior on n. We can therefore count down to $\mathbf{0}$, in which case no recursive call is allowed. An alternative approach is to take **case** as primitive and allow arbitrary recursion. In such a language it is much easier to program, but not every function terminates. We will see that for our purpose about integrating constructive reasoning and functional programming it is simpler if all functions one can write down are *total*, that is, are defined on all arguments. This is because total functions can be used to provide witnesses for propositions of the form $\forall x \in \mathbf{nat}.\, \exists y \in \mathbf{nat}.\, P(x, y)$ by showing how to compute y from x. Functions that may not return an appropriate y cannot be used in this capacity and are generally much more difficult to reason about.

3.6 Booleans

Another simple example of a data type is provided by the Boolean type with two elements **true** and **false**. This should *not* be confused with the propositions $\top$ and $\bot$. In fact, they correspond to the unit type $\mathbf{1}$ and the empty type $\mathbf{0}$. We recall their definitions first, in analogy with the propositions.

$$\frac{}{\mathbf{1}\ \mathit{type}}\,\mathbf{1}F$$

$$\frac{}{\Gamma \vdash \langle\,\rangle \in \mathbf{1}}\,\mathbf{1}I \qquad \textit{no } \mathbf{1} \textit{ elimination rule}$$

$$\frac{}{\mathbf{0}\ \mathit{type}}\,\mathbf{0}F$$

$$\textit{no } \mathbf{0} \textit{ introduction rule} \qquad \frac{\Gamma \vdash t \in \mathbf{0}}{\Gamma \vdash \mathbf{abort}^{\tau}\, t \in \tau}\,\mathbf{0}E$$

There are no reduction rules at these types.

The Boolean type, **bool**, is instead defined by two introduction rules.

$$\frac{}{\mathbf{bool}\ \mathit{type}}\,\mathbf{bool}F$$

$$\frac{}{\Gamma \vdash \mathbf{true} \in \mathbf{bool}}\,\mathbf{bool}I_1 \qquad \frac{}{\Gamma \vdash \mathbf{false} \in \mathbf{bool}}\,\mathbf{bool}I_0$$

The elimination rule follows the now familiar pattern: since there are two introduction rules, we have to distinguish two cases for a given Boolean value. This could be written as

$$\mathbf{case}\, t\, \mathbf{of}\, \mathbf{true} \Rightarrow s_1 \mid \mathbf{false} \Rightarrow s_0$$

but we typically express the same program as an $\mathbf{if}\, t\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0$.

$$\frac{\Gamma \vdash t \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\, t\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 \in \tau}\, \mathbf{bool}E$$

The reduction rules just distinguish the two cases for the subject of the **if**-expression.

$$\begin{array}{rcl} \mathbf{if}\, \mathbf{true}\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 & \Longrightarrow & s_1 \\ \mathbf{if}\, \mathbf{false}\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 & \Longrightarrow & s_0 \end{array}$$

Now we can define typical functions on booleans, such as *and*, *or*, and *not*.

$$\begin{array}{rcl} \mathit{and} & = & \lambda x \in \mathbf{bool}.\, \lambda y \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, y\, \mathbf{else}\, \mathbf{false} \\ \mathit{or} & = & \lambda x \in \mathbf{bool}.\, \lambda y \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, \mathbf{true}\, \mathbf{else}\, y \\ \mathit{not} & = & \lambda x \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, \mathbf{false}\, \mathbf{else}\, \mathbf{true} \end{array}$$

3.7 Lists

Another more interesting data type is that of lists. Lists can be created with elements from any type whatsoever, which means that $\tau\, \mathbf{list}$ is a type for any type τ.

$$\frac{\tau\ \mathit{type}}{\tau\, \mathbf{list}\ \mathit{type}}\, \mathbf{list}F$$

Lists are built up from the empty list (**nil**) with the operation :: (pronounced "cons"), written in infix notation.

$$\frac{}{\Gamma \vdash \mathbf{nil}^{\tau}\ \in \tau\, \mathbf{list}}\, \mathbf{list}I_n \qquad \frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \tau\, \mathbf{list}}{\Gamma \vdash t :: s \in \tau\, \mathbf{list}}\, \mathbf{list}I_c$$

The elimination rule implements the schema of primitive recursion over lists. It can be specified as follows:

$$\begin{array}{rcl} f\,(\mathbf{nil}) & = & s_n \\ f\,(x :: l) & = & s_c(x, l, f(l)) \end{array}$$

where we have indicated that s_c may mention x, l, and $f(l)$, but no other occurrences of f. Again this guarantees termination.

$$\frac{\Gamma \vdash t \in \tau\, \mathbf{list} \qquad \Gamma \vdash s_n \in \sigma \qquad \Gamma, x \in \tau, l \in \tau\, \mathbf{list}, f(l) \in \sigma \vdash s_c \in \sigma}{\Gamma \vdash \mathbf{rec}\, t\, \mathbf{of}\, f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c \in \sigma}\, \mathbf{list}E$$

We have overloaded the **rec** constructor here—from the type of t we can always tell if it should recurse over natural numbers or lists. The reduction rules are once again recursive, as in the case for natural numbers.

$$\begin{array}{rcl}
(\mathbf{rec}\,\mathbf{nil}\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & s_n \\
(\mathbf{rec}\,(h :: t)\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & \\
[(\mathbf{rec}\,t\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c)/f(l)]\,[h/x]\,[t/l]\,s_c &&
\end{array}$$

Now we can define typical operations on lists via primitive recursion. A simple example is the *append* function to concatenate two lists.

$$\begin{array}{rcl}
\mathit{append}\,\mathbf{nil}\,k & = & k \\
\mathit{append}\,(x :: l')\,k & = & x :: (\mathit{append}\,l'\,k)
\end{array}$$

In the notation of primitive recursion:

$$\begin{array}{rcl}
\mathit{append} & = & \lambda l \in \tau\,\mathbf{list}.\,\lambda k \in \tau\,\mathbf{list}.\,\mathbf{rec}\ l \\
& & \qquad \mathbf{of}\ a(\mathbf{nil}) \Rightarrow k \\
& & \qquad \mid\ a(x :: l') \Rightarrow x :: (a\,l') \\
\vdash \mathit{append} & \in & \tau\,\mathbf{list} \rightarrow \tau\,\mathbf{list} \rightarrow \tau\,\mathbf{list}
\end{array}$$

Note that the last judgment is parametric in τ, a situation referred to as *parametric polymorphism*. In means that the judgment is valid for every type τ. We have encountered a similar situation, for example, when we asserted that $(A \wedge B) \supset A$ *true*. This judgment is parametric in A and B, and every instance of it by propositions A and B is evident, according to our derivation.

As a second example, we consider a program to reverse a list. The idea is to take elements out of the input list l and attach them to the front of a second list a one which starts out empty. The first list has been traversed, the second has accumulated the original list in reverse. If we call this function *rev* and the original one *reverse*, it satisfies the following specification.

$$\begin{array}{rcl}
\mathit{rev} & \in & \tau\,\mathbf{list} \rightarrow \tau\,\mathbf{list} \rightarrow \tau\,\mathbf{list} \\
\mathit{rev}\,\mathbf{nil}\,a & = & a \\
\mathit{rev}\,(x :: l')\,a & = & \mathit{rev}\,l'\,(x :: a) \\
\\
\mathit{reverse} & \in & \tau\,\mathbf{list} \rightarrow \tau\,\mathbf{list} \\
\mathit{reverse}\,l & = & \mathit{rev}\,l\,\mathbf{nil}
\end{array}$$

In programs of this kind we refer to a as the *accumulator argument* since it accumulates the final result which is returned in the base case. We can see that except for the additional argument a, the *rev* function is primitive recursive. To make this more explicit we can rewrite the definition of *rev* to the following equivalent form:

$$\begin{array}{rcl}
\mathit{rev}\,\mathbf{nil} & = & \lambda a.\,a \\
\mathit{rev}\,(x :: l) & = & \lambda a.\,\mathit{rev}\,l\,(x :: a)
\end{array}$$

Now the transcription into our notation is direct.

$$\begin{array}{rcl}
\mathit{rev} & = & \lambda l \in \tau\,\mathbf{list}.\,\mathbf{rec}\ l \\
& & \qquad \mathbf{of}\ r(\mathbf{nil}) \Rightarrow \lambda a \in \tau\,\mathbf{list}.\,a \\
& & \qquad \mid\ r(x :: l') \Rightarrow \lambda a \in \tau\,\mathbf{list}.\,r\,(l')\,(x :: a) \\
\mathit{reverse}\,l & = & \mathit{rev}\,l\,\mathbf{nil}
\end{array}$$

Finally a few simple functions which mix data types. The first counts the number of elements in a list.

$$\begin{array}{rcl} \mathit{length} & \in & \tau\,\mathbf{list} \to \mathbf{nat} \\ \mathit{length}\ \mathbf{nil} & = & \mathbf{0} \\ \mathit{length}\ (x :: l') & = & \mathbf{s}(\mathit{length}\ (l')) \end{array}$$

$$\begin{array}{rcl} \mathit{length} & = & \lambda x \in \tau\,\mathbf{list}.\ \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ \mathit{le}(\mathbf{nil}) \Rightarrow \mathbf{0} \\ & & \quad \mid \mathit{le}(x :: l') \Rightarrow \mathbf{s}(\mathit{le}\ (l')) \end{array}$$

The second compares two numbers for equality.

$$\begin{array}{rcl} \mathit{eq} & \in & \mathbf{nat} \to \mathbf{nat} \to \mathbf{bool} \\ \mathit{eq}\ \mathbf{0}\ \mathbf{0} & = & \mathbf{true} \\ \mathit{eq}\ \mathbf{0}\ (\mathbf{s}(y')) & = & \mathbf{false} \\ \mathit{eq}\ (\mathbf{s}(x'))\ \mathbf{0} & = & \mathbf{false} \\ \mathit{eq}\ (\mathbf{s}(x'))\ (\mathbf{s}(y')) & = & \mathit{eq}\ x'\ y' \end{array}$$

As in the example of subtraction, we need to distinguish two levels.

$$\begin{array}{rcl} \mathit{eq} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ e(\mathbf{0}) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{true} \\ & & \qquad \mid f(\mathbf{s}(y')) \Rightarrow \mathbf{false} \\ & & \quad \mid e(\mathbf{s}(x')) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{false} \\ & & \qquad \mid f(\mathbf{s}(y')) \Rightarrow e(x')\ y' \end{array}$$

We will see more examples of primitive recursive programming as we proceed to first order logic and quantification.

3.8 Summary of Data Types

Judgments.

$\tau\ \mathit{type}$	τ is a type
$t \in \tau$	t is a term of type τ

Type Formation.

$$\frac{}{\mathbf{nat}\ \mathit{type}}\,\mathbf{nat}F \qquad \frac{}{\mathbf{bool}\ \mathit{type}}\,\mathbf{bool}F \qquad \frac{\tau\ \mathit{type}}{\tau\,\mathbf{list}\ \mathit{type}}\,\mathbf{list}F$$

Term Formation.

$$\frac{}{\mathbf{0} \in \mathbf{nat}}\,\mathbf{nat}I_0 \qquad \frac{n \in \mathbf{nat}}{\mathbf{s}(n) \in \mathbf{nat}}\,\mathbf{nat}I_s$$

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\,\mathbf{nat}E$$

$$\frac{}{\Gamma \vdash \mathbf{true} \in \mathbf{bool}}\,\mathbf{bool}I_1 \qquad \frac{}{\Gamma \vdash \mathbf{false} \in \mathbf{bool}}\,\mathbf{bool}I_0$$

$$\frac{\Gamma \vdash t \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\ t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 \in \tau}\,\mathbf{bool}E$$

$$\frac{}{\Gamma \vdash \mathbf{nil}^\tau\ \in \tau\,\mathbf{list}}\,\mathbf{list}I_n \qquad \frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \tau\,\mathbf{list}}{\Gamma \vdash t :: s \in \tau\,\mathbf{list}}\,\mathbf{list}I_c$$

$$\frac{\Gamma \vdash t \in \tau\,\mathbf{list} \qquad \Gamma \vdash s_n \in \sigma \qquad \Gamma, x \in \tau, l \in \tau\,\mathbf{list}, f(l) \in \sigma\,\mathbf{list} \vdash s_c \in \sigma}{\Gamma \vdash \mathbf{rec}\,t\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c \in \sigma}\,\mathbf{list}E$$

Reductions.

$$\begin{aligned}
(\mathbf{rec}\ \mathbf{0}\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) &\implies t_0 \\
(\mathbf{rec}\ \mathbf{s}(n)\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) &\implies \\
[(\mathbf{rec}\ n\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s)/f(x)]\,[n/x]\,t_s & \\
\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 &\implies s_1 \\
\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 &\implies s_0 \\
(\mathbf{rec}\,\mathbf{nil}\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) &\implies s_n \\
(\mathbf{rec}\,(h :: t)\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) &\implies \\
[(\mathbf{rec}\,t\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c)/f(l)]\,[h/x]\,[t/l]\,s_c &
\end{aligned}$$

3.9 Predicates on Data Types

In the preceding sections we have introduced the concept of a type which is determined by its elements. Examples were natural numbers, Booleans, and lists. In the next chapter we will explicitly quantify over elements of types. For example, we may assert that every natural number is either even or odd. Or we may claim that any two numbers possess a greatest common divisor. In order to formulate such statements we need some basic propositions concerned with data types. In this section we will define such predicates, following our usual methodology of using introduction and elimination rules to define the meaning of propositions.

We begin with $n < m$, the less-than relation between natural numbers. We have the following formation rule:

$$\frac{\Gamma \vdash m \in \mathbf{nat} \quad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash m < n \; \mathit{prop}} <F$$

Note that this formation rule for propositions relies on the judgment $t \in \tau$. Consequently, we have to permit a hypothetical judgment, in case n or m mention variables declared with their type, such as $x \in \mathbf{nat}$. Thus, in general, the question whether A *prop* may now depend on assumptions of the form $x \in \tau$.

This has a consequence for the judgment A *true*. As before, we now must allow assumptions of the form B *true*, but in addition we must permit assumptions of the form $x \in \tau$. We still call the collection of such assumptions a *context* and continue to denote it with Γ.

$$\frac{}{\Gamma \vdash \mathbf{0} < \mathbf{s}(n) \; \mathit{true}} <I_0 \qquad \frac{\Gamma \vdash m < n \; \mathit{true}}{\Gamma \vdash \mathbf{s}(m) < \mathbf{s}(n) \; \mathit{true}} <I_s$$

The second rule exhibits a new phenomenon: the relation '$<$' whose meaning we are trying to define appears in the premise as well as in the conclusion. In effect, we have not really introduced '$<$', since it already occurs. However, such a definition is still justified, since the conclusion defines the meaning of $\mathbf{s}(m) < \cdot$ in terms of $m < \cdot$. We refer to this relation as *inductively defined*. Actually we have already seen a similar phenomenon in the second "introduction" rule for **nat**:

$$\frac{\Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \mathbf{s}(n) \in \mathbf{nat}} \mathbf{nat}I_s$$

The type **nat** we are trying to define already occurs in the premise! So it may be better to think of this rule as a formation rule for the successor operation on natural numbers, rather than an introduction rule for natural numbers.

Returning to the less-than relation, we have to derive the elimination rules. What can we conclude from $\Gamma \vdash m < n \; \mathit{true}$? Since there are two introduction rules, we could try our previous approach and distinguish cases for the proof of that judgment. This, however, is somewhat awkward in this case—we postpone discussion of this option until later. Instead of distinguishing cases for the proof of the judgment, we distinguish cases for m and n. In each case, we analyse how the resulting judgment could be proven and write out the corresponding elimination rule. First, if n is zero, then the judgment can never have a normal proof, since no introduction rule applies. Therefore we are justified in concluding anything, as in the elimination rule for falsehood.

$$\frac{\Gamma \vdash m < \mathbf{0} \; \mathit{true}}{\Gamma \vdash C \; \mathit{true}} <E_0$$

If the $m = \mathbf{0}$ and $n = \mathbf{s}(n')$, then it could be inferred only by the first introduction rule $<I_0$. This yields no information, since there are no premises to this rule. This is just as in the case of the true proposition $\top$.

The last remaining possibility is that both $m = \mathbf{s}(m')$ and $n = \mathbf{s}(n')$. In that case we now that $m' < n'$, because $<I_s$ is the only rule that could have been applied.

$$\frac{\Gamma \vdash \mathbf{s}(m') < \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash m' < n'\ \mathit{true}}\,{<}E_s$$

We summarize the formation, introduction, and elimination rules.

$$\frac{\Gamma \vdash n \in \mathbf{nat} \quad \Gamma \vdash m \in \mathbf{nat}}{\Gamma \vdash n < m\ \mathit{prop}}\,{<}F$$

$$\frac{}{\Gamma \vdash \mathbf{0} < \mathbf{s}(n)\ \mathit{true}}\,{<}I_0 \qquad \frac{\Gamma \vdash m < n\ \mathit{true}}{\Gamma \vdash \mathbf{s}(m) < \mathbf{s}(n)\ \mathit{true}}\,{<}I_s$$

$$\frac{\Gamma \vdash m < \mathbf{0}\ \mathit{true}}{\Gamma \vdash C\ \mathit{true}}\,{<}E_0$$

$$\textit{no rule for } \mathbf{0} < \mathbf{s}(n') \qquad \frac{\Gamma \vdash \mathbf{s}(m') < \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash m' < n'\ \mathit{true}}\,{<}E_s$$

Now we can prove some simple relations between natural numbers. For example:

$$\dfrac{\dfrac{}{\cdot \vdash \mathbf{0} < \mathbf{s}(\mathbf{0})\ \mathit{true}}\,{<}I_0}{\cdot \vdash \mathbf{0} < \mathbf{s}(\mathbf{s}(\mathbf{0}))\ \mathit{true}}\,{<}I_s$$

We can also establish some simple parametric properties of natural numbers.

$$\dfrac{\dfrac{\dfrac{}{m \in \mathbf{nat}, m < 0\ \mathit{true} \vdash m < 0\ \mathit{true}}\,u}{m \in \mathbf{nat}, m < 0\ \mathit{true} \vdash \bot\ \mathit{true}}\,{<}E_0}{m \in \mathbf{nat} \vdash \neg(m < 0)\ \mathit{true}}\,{\supset}I^u$$

In the application of the $<E_0$ rule, we chose $C = \bot$ in order to complete the proof of $\neg(m < \mathbf{0})$. Even slightly more complicated properties, such as $m < \mathbf{s}(m)$ require a proof by induction and are therefore postponed until Section 3.10.

We introduce one further relation between natural numbers, namely equality.

We write $m =_N n$. Otherwise we follow the blueprint of the less-than relation.

$$\frac{\Gamma \vdash m \in \mathbf{nat} \qquad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash m =_N n \ \mathit{prop}} =_N F$$

$$\frac{}{\Gamma \vdash \mathbf{0} =_N \mathbf{0} \ \mathit{true}} =_N I_0 \qquad \frac{\Gamma \vdash m =_N n \ \mathit{true}}{\Gamma \vdash \mathbf{s}(m) =_N \mathbf{s}(n) \ \mathit{true}} =_N I_s$$

$$no =_N E_{00} \ \mathit{elimination\ rule} \qquad \frac{\Gamma \vdash \mathbf{0} =_N \mathbf{s}(n) \ \mathit{true}}{\Gamma \vdash C \ \mathit{true}} =_N E_{0s}$$

$$\frac{\Gamma \vdash \mathbf{s}(m) =_N \mathbf{0} \ \mathit{true}}{\Gamma \vdash C \ \mathit{true}} =_N E_{s0} \qquad \frac{\Gamma \vdash \mathbf{s}(m) =_N \mathbf{s}(n) \ \mathit{true}}{\Gamma \vdash m =_N n \ \mathit{true}} =_N E_{ss}$$

Note the difference between the *function*

$$\mathit{eq} \in \mathbf{nat} \to \mathbf{nat} \to \mathbf{bool}$$

and the *proposition*

$$m =_N n$$

The equality function provides a computation on natural numbers, always returning **true** or **false**. The proposition $m =_N n$ requires *proof*. Using induction, we can later verify a relationship between these two notions, namely that $\mathit{eq}\, n\, m$ reduces to **true** if $m =_N n$ is true, and $\mathit{eq}\, n\, m$ reduces to **false** if $\neg(m =_N n)$.

3.10 Induction

Now that we have introduced the basic propositions regarding order and equality, we can consider induction as a reasoning principle. So far, we have considered the following elimination rule for natural numbers:

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\ \mathbf{nat}E$$

This rule can be applied if we can derive $t \in \mathbf{nat}$ from our assumptions and we are trying to construct a term $s \in \tau$. But how do we use a variable or term $t \in \mathbf{nat}$ if the judgment we are trying to prove has the form $M : A$, that is, if we are trying the prove the truth of a proposition? The answer is induction. This is actually very similar to primitive recursion. The only complication is that the proposition A we are trying to prove may depend on t. We indicate this by writing $A(x)$ to mean the proposition A with one or more occurrences of a variable x. $A(t)$ is our notation for the result of substituting t for x in A. We

could also write $[t/x]A$, but this is more difficult to read. Informally, induction says that in order to prove $A(t)$ *true* for arbitrary t we have to prove $A(\mathbf{0})$ *true* (the base case), and that for every $x \in \mathbf{nat}$, if $A(x)$ *true* then $A(\mathbf{s}(x))$ *true*.

Formally this becomes:

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash A(\mathbf{0})\ \mathit{true} \qquad \Gamma, x \in \mathbf{nat}, A(x)\ \mathit{true} \vdash A(\mathbf{s}(x))\ \mathit{true}}{\Gamma \vdash A(t)\ \mathit{true}}\ \mathbf{nat}E'$$

Here, $A(x)$ is called the *induction predicate*. If t is a variable (which is frequently the case) it is called the *induction variable*. With this rule, we can now prove some more interesting properties. As a simple example we show that $m < \mathbf{s}(m)$ *true* for any natural number m. Here we use $\mathcal{D}$ to stand for the derivation of the third premise in order to overcome the typesetting difficulties.

$$\mathcal{D} = \frac{\dfrac{}{m \in \mathbf{nat}, x \in \mathbf{nat}, x < \mathbf{s}(x)\ \mathit{true} \vdash x < \mathbf{s}(x)\ \mathit{true}}}{m \in \mathbf{nat}, x \in \mathbf{nat}, x < \mathbf{s}(x)\ \mathit{true} \vdash \mathbf{s}(x) < \mathbf{s}(\mathbf{s}(x))}\ {<}I_s$$

$$\frac{\dfrac{}{m \in \mathbf{nat} \vdash m \in \mathbf{nat}} \qquad \dfrac{}{m \in \mathbf{nat} \vdash \mathbf{0} < \mathbf{s}(\mathbf{0})}\ {<}I_0 \qquad \mathcal{D}}{m \in \mathbf{nat} \vdash m < \mathbf{s}(m)}\ \mathbf{nat}E'$$

The property $A(x)$ appearing in the induction principle is $A(x) = x < \mathbf{s}(x)$. So the final conclusion is $A(m) = m < \mathbf{s}(m)$. In the second premise we have to prove $A(\mathbf{0}) = \mathbf{0} < \mathbf{s}(\mathbf{0})$ which follows directly by an introduction rule.

Despite the presence of the induction rule, there are other properties we cannot yet prove easily since the logic does not have quantifiers. An example is the decidability of equality: For any natural numbers m and n, either $m =_N n$ or $\neg(m =_N n)$. This is an example of the practical limitations of *quantifier-free induction*, that is, induction where the induction predicate does not contain any quantifiers.

The topic of this chapter is the interpretation of constructive proofs as programs. So what is the computational meaning of induction? It actually corresponds very closely to primitive recursion.

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash M : A(\mathbf{0}) \qquad \Gamma, x \in \mathbf{nat}, u(x){:}A(x) \vdash N : A(\mathbf{s}(x))}{\Gamma \vdash \mathbf{ind}\ t\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N : A(t)}\ \mathbf{nat}E'$$

Here, $u(x)$ is just the notation for a variable which may occur in N. Note that u cannot occur in M or in N in any other form. The reduction rules are precisely the same as for primitive recursion.

$$\begin{array}{rcl} (\mathbf{ind}\ \mathbf{0}\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N) & \Longrightarrow & M \\ (\mathbf{ind}\ \mathbf{s}(n)\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N) & \Longrightarrow & \\ [(\mathbf{ind}\ n\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N)/u(n)]\ [n/x]N & & \end{array}$$

We see that primitive recursion and induction are almost identical. The only difference is that primitive recursion returns an element of a type, while induction generates a proof of a proposition. Thus one could say that they are related by an extension of the Curry-Howard correspondence. However, not every type τ can be naturally interpreted as a proposition (which proposition, for example, is expressed by **nat**?), so we no longer speak of an isomorphism.

We close this section by the version of the rules for the basic relations between natural numbers that carry proof terms. This annotation of the rules is straightforward.

$$\frac{\Gamma \vdash n \in \mathbf{nat} \quad \Gamma \vdash m \in \mathbf{nat}}{\Gamma \vdash n < m \; prop} <F$$

$$\frac{}{\Gamma \vdash \mathbf{lt}_0 : \mathbf{0} < \mathbf{s}(n)} <I_0 \qquad \frac{\Gamma \vdash M : m < n}{\Gamma \vdash \mathbf{lt}_s(M) : \mathbf{s}(m) < \mathbf{s}(n)} <I_s$$

$$\frac{\Gamma \vdash M : m < \mathbf{0}}{\Gamma \vdash \mathbf{ltE}_0(M) : C} <E_0$$

$$\textit{no rule for } \mathbf{0} < \mathbf{s}(n') \qquad \frac{\Gamma \vdash M : \mathbf{s}(m') < \mathbf{s}(n')}{\Gamma \vdash \mathbf{ltE}_s(M) : m' < n'} <E_s$$

$$\frac{\Gamma \vdash m \in \mathbf{nat} \qquad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash m =_{\mathbf{N}} n \; prop} =_{\mathbf{N}} F$$

$$\frac{}{\Gamma \vdash \mathbf{eq}_0 : \mathbf{0} =_{\mathbf{N}} \mathbf{0}} =_{\mathbf{N}} I_0 \qquad \frac{\Gamma \vdash M : m =_{\mathbf{N}} n}{\Gamma \vdash \mathbf{eq}_s(M) : \mathbf{s}(m) =_{\mathbf{N}} \mathbf{s}(n)} =_{\mathbf{N}} I_s$$

$$\textit{no } =_{\mathbf{N}} E_{00} \textit{ elimination rule} \qquad \frac{\Gamma \vdash M : \mathbf{0} =_{\mathbf{N}} \mathbf{s}(n)}{\Gamma \vdash \mathbf{eqE}_{0s}(M) : C} =_{\mathbf{N}} E_{0s}$$

$$\frac{\Gamma \vdash M : \mathbf{s}(m) =_{\mathbf{N}} \mathbf{0}}{\Gamma \vdash \mathbf{eqE}_{s0}(M) : C} =_{\mathbf{N}} E_{s0} \qquad \frac{\Gamma \vdash M : \mathbf{s}(m) =_{\mathbf{N}} \mathbf{s}(n)}{\Gamma \vdash \mathbf{eqE}_{ss}(M) : m =_{\mathbf{N}} n} =_{\mathbf{N}} E_{ss}$$

Bibliography

[CGP99] E.M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.