Now this can be proven by a straightforward structural induction over l. It most natural to pick l as the induction variable here, since this allows reduction on the right-hand side as well as the left-hand side. In general, it a good heuristic to pick variables that permit reduction when instantiated.

Proof: By structural induction on l.

Case: $l = \mathbf{nil}$. Then we get

$$\begin{array}{rl} \text{left-hand side:} & \mathit{rev}\ \mathbf{nil}\ (\mathit{app}\ m\ k) \Longrightarrow \mathit{app}\ m\ k \\ \text{right-hand side:} & \mathit{app}\ (\mathit{rev}\ \mathbf{nil}\ m)\ k \Longrightarrow \mathit{app}\ m\ k \end{array}$$

so the equality follows by computation and reflexivity of equality.

Case: $l = x{::}l'$. It is often useful to write out the general form of the induction hypothesis before starting the proof in the induction step.

$$\forall m{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \mathit{rev}\ l'\ (\mathit{app}\ m\ k) =_L \mathit{app}\ (\mathit{rev}\ l'\ m)\ k$$

As we will see, the quantifiers over m and k are critical here. Now we follow the general strategy to reduce the left-hand side and the right-hand side to see if we can close the gap by using the induction hypothesis.

$$\begin{array}{lll} \text{lhs:} & \mathit{rev}\ (x :: l')\ (\mathit{app}\ m\ k) & \\ & \Longrightarrow \mathit{rev}\ l'\ (x :: (\mathit{app}\ m\ k)) & \\ \text{rhs:} & \mathit{app}\ (\mathit{rev}\ (x :: l')\ m)\ k & \\ & \Longrightarrow \mathit{app}\ (\mathit{rev}\ l'\ (x :: m))\ k & \\ & =_L \mathit{rev}\ l'\ (\mathit{app}\ (x :: m)\ k) & \text{by ind. hyp} \\ & \Longrightarrow \mathit{rev}\ l'\ (x :: (\mathit{app}\ m\ k)) & \end{array}$$

So by computation and the induction hypothesis the left-hand side and the right-hand side are equal. Note that the universal quantifier on m in the induction hypothesis needed to be instantiated by $x :: m$. This is a frequent pattern when accumulator variables are involved.

□

Returning to our original question, we generalize the term on the left-hand side, *reverse* $(\mathit{app}\ l\ k)$, to $\mathit{rev}\ (\mathit{app}\ l\ k)\ m$. The appropriate generalization of the right-hand side yields

$$\forall l{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \forall m{\in}\tau\ \mathbf{list}.\ \mathit{rev}\ (\mathit{app}\ l\ k)\ m =_L \mathit{rev}\ k\ (\mathit{rev}\ l\ m)$$

In this general form we can easily prove it by induction over l.

Proof: By induction over l.

Case: $l = \mathbf{nil}$. Then

$$\begin{array}{rl} \text{lhs:} & rev\ (app\ \mathbf{nil}\ k)\ m \Longrightarrow rev\ k\ m \\ \text{rhs:} & rev\ k\ (rev\ \mathbf{nil}\ m) \Longrightarrow rev\ k\ m \end{array}$$

So the left- and right-hand side are equal by computation.

Case: $l = x :: l'$. Again, we write out the induction hypothesis:

$$\forall k{\in}\tau\ \mathbf{list}.\ \forall m{\in}\tau\ \mathbf{list}.\ \forall rev\ (app\ l'\ k)\ m =_L rev\ k\ (rev\ l'\ m)$$

Then

$$\begin{array}{rl} \text{lhs} & rev\ (app\ (x :: l')\ k)\ m \\ & \Longrightarrow rev\ (x :: (app\ l'\ k))\ m \\ & \Longrightarrow rev\ (app\ l'\ k)\ (x :: m) \\ \text{rhs} & rev\ k\ (rev\ (x :: l')\ m) \\ & \Longrightarrow rev\ k\ (rev\ l'\ (x :: m)) \end{array}$$

So the left- and right-hand sides are equal by computation and the induction hypothesis. Again, we needed to use $x :: m$ for m in the induction hypothesis.

□

By using these two properties together we can now show that this implies the original theorem directly.

$$\forall l{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ reverse\ (app\ l\ k) =_L app\ (reverse\ k)\ (reverse\ l)$$

Proof: Direct, by computation and previous lemmas.

$$\begin{array}{rll} \text{lhs} & reverse\ (app\ l\ k) & \\ & \Longrightarrow rev\ (app\ l\ k)\ \mathbf{nil} & \\ & =_L rev\ k\ (rev\ l\ \mathbf{nil}) & \text{by lemma} \\ \text{rhs} & app\ (reverse\ k)\ (reverse\ l) & \\ & \Longrightarrow app\ (rev\ k\ \mathbf{nil})\ (rev\ l\ \mathbf{nil}) & \\ & =_L rev\ k\ (app\ \mathbf{nil}\ (rev\ l\ \mathbf{nil}) & \text{by lemma} \\ & =_L rev\ k\ (rev\ l\ \mathbf{nil}) & \end{array}$$

So the left- and right-hand sides are equal by computation and the two preceding lemmas. □

4.6 Reasoning about Data Representations

So far, our data types have been "freely generated" from a set of constructors. Equality on such types is structural. This has been true for natural numbers, lists, and booleans. In practice, there are many data representation which does not have this property. In this section we will examine two examples of this form.

The first is a representation of natural numbers in *binary* form, that is, as bit string consisting of zeroes and ones. This representation is of course prevalent in hardware and also much more compact than the unary numbers we have considered so far. The length of the representation of n is logarithmic in n. Thus, almost all work both on practical arithmetic and complexity theory uses binary representations. The main reason to consider unary representations in our context is the induction principle, and the connection between induction and primitive recursion.

We define the binary numbers with three constructors. We have the empty bit string ϵ, the operation of appending a $\mathbf{0}$ at the end, and the operation of appending a $\mathbf{1}$ at the end. We write the latter two in postfix notation, following the usual presentation of numbers as sequences of bits.

$$\frac{}{\Gamma \vdash \mathbf{bin}\ \mathit{type}}\ \mathbf{bin}F$$

$$\frac{}{\Gamma \vdash \epsilon \in \mathbf{bin}}\ \mathbf{bin}I_\epsilon \qquad \frac{\Gamma \vdash b \in \mathbf{bin}}{\Gamma \vdash b\,\mathbf{0} \in \mathbf{bin}}\ \mathbf{bin}I_0 \qquad \frac{\Gamma \vdash b \in \mathbf{bin}}{\Gamma \vdash b\,\mathbf{1} \in \mathbf{bin}}\ \mathbf{bin}I_1$$

The schema of primitive recursion has the following form

$$\begin{array}{lcll} f & \epsilon & = & t_\epsilon \\ f & (b\,\mathbf{0}) & = & t_0(b, f(b)) \\ f & (b\,\mathbf{1}) & = & t_1(b, f(b)) \end{array}$$

Note that f, the recursive function, can occur only applied to b in the last two cases and not at all in the first case. It should be clear by now how to formulate the corresponding **rec** term and proof term construct. The induction principle is also straightforward.

To prove $A(b)$ true for an arbitrary bit string b, prove

Base Case: $A(\epsilon)$ *true.*

Step Case 0: $A(b'\,\mathbf{0})$ *true assuming* $A(b')$ *true for an arbitrary* b'.

Step Case 1: $A(b'\,\mathbf{1})$ *true assuming* $A(b')$ *true for an arbitrary* b'.

In order to describe formally how bitstring represent natural numbers, recall the function on natural numbers doubling its argument, specified as follows:

$$\begin{array}{lcll} & \mathit{double} & \in & \mathbf{nat} \to \mathbf{nat} \\ \mathit{double} & \mathbf{0} & = & \mathbf{0} \\ \mathit{double} & (\mathbf{s}(x)) & = & \mathbf{s}(\mathbf{s}(\mathit{double}\ x)) \end{array}$$

Then we specify

$$\begin{array}{lcll} & \mathit{tonat} & \in & \mathbf{bin} \to \mathbf{nat} \\ \mathit{tonat} & \epsilon & = & \mathbf{0} \\ \mathit{tonat} & (b\,\mathbf{0}) & = & \mathit{double}\ (\mathit{tonat}\ b) \\ \mathit{tonat} & (b\,\mathbf{1}) & = & \mathbf{s}(\mathit{double}\ (\mathit{tonat}\ b)) \end{array}$$

Note that this satisfies the schema of primitive recursion. Now we can see why we think of binary numbers as satisying some non-structural equality: every natural number has an infinite number of bit strings as representations, because we can always add leading zeroes without changing the result of *tonat*. For example,

$$\mathit{tonat}(\epsilon\,\mathbf{1}) =_N \mathit{tonat}(\epsilon\,\mathbf{0}\,\mathbf{1}) =_N \mathit{tonat}(\epsilon\,\mathbf{0}\,\mathbf{0}\,\mathbf{1}) =_N \mathbf{s}(\mathbf{0})$$

This has several consequences. If we think of bit strings only as a means to represent natural numbers, we would define equality such that $\epsilon =_B \epsilon\,\mathbf{0}$. Secondly, we can define functions which are ill-defined as far as their interpretation as natural numbers is concerned. For example,

$$\begin{array}{llcl} \mathit{flip} & \epsilon & = & \epsilon \\ \mathit{flip} & (b\,\mathbf{0}) & = & b\,\mathbf{1} \\ \mathit{flip} & (b\,\mathbf{1}) & = & b\,\mathbf{0} \end{array}$$

may make sense intuitively, but it maps ϵ and $\epsilon\mathbf{0}$ to different results and thus does not respect the intended equality.

A general mechanism to deal with such problems is to define *quotient types*. This is somewhat more complicated than needed in most instances, so we will stick to the simpler idea of just verifying that functions implement the intended operations on natural numbers.

A simple example is the increment function *inc* on binary numbers. Assume a bit string b represents a natural number n. When we can show that $\mathit{inc}(b)$ always represents $\mathbf{s}(n)$ we say that *inc implements* $\mathbf{s}$. In general, a function f *implements* g if $f(b)$ represents $g(n)$ whenever b represents n. Representation is defined via the function *tonat*, so by definition f implements g if we can prove that

$$\forall b{\in}\mathbf{bin}.\ \mathit{tonat}(f\ b) =_N g(\mathit{tonat}\ b)$$

In our case:

$$\forall b{\in}\mathbf{bin}.\ \mathit{tonat}(\mathit{inc}\ b) =_N \mathbf{s}(\mathit{tonat}\ b)$$

The increment function is primitive recursive and defined as follows:

$$\begin{array}{llcl} & \mathit{inc} & \in & \mathbf{bin} \to \mathbf{bin} \\ \mathit{inc} & \epsilon & = & \epsilon\,\mathbf{1} \\ \mathit{inc} & (b\,\mathbf{0}) & = & b\,\mathbf{1} \\ \mathit{inc} & (b\,\mathbf{1}) & = & (\mathit{inc}\ b)\,\mathbf{0} \end{array}$$

Now we can prove that *inc* correctly implements the successor function.

$$\forall b{\in}\mathbf{bin}.\ \mathit{tonat}(\mathit{inc}\ b) =_N \mathbf{s}(\mathit{tonat}\ b)$$

Proof: By structural induction on b.

Case: $b = \epsilon$.

$$\begin{array}{rl}
\text{lhs:} & \mathit{tonat}(\mathit{inc}\ \epsilon) \\
& \Longrightarrow \mathit{tonat}(\epsilon\,\mathbf{1}) \\
& \Longrightarrow \mathbf{s}(\mathit{double}(\mathit{tonat}\ \epsilon)) \\
& \Longrightarrow \mathbf{s}(\mathbf{0}) \\
\text{rhs:} & \mathbf{s}(\mathit{tonat}\ \epsilon) \Longrightarrow \mathbf{s}(\mathbf{0})
\end{array}$$

Case: $b = b'\,\mathbf{0}$. We simply calculate left- and right-hand side without appeal to the induction hypothesis.

$$\begin{array}{rl}
\text{lhs:} & \mathit{tonat}(\mathit{inc}(b'\,\mathbf{0})) \\
& \Longrightarrow \mathit{tonat}(b'\,\mathbf{1}) \\
& \Longrightarrow \mathbf{s}(\mathit{double}(\mathit{tonat}\ b')) \\
\text{rhs:} & \mathbf{s}(\mathit{tonat}(b'\,\mathbf{0})) \\
& \Longrightarrow \mathbf{s}(\mathit{double}(\mathit{tonat}\ b'))
\end{array}$$

Case: $b = b'\,\mathbf{1}$. In this case we need the induction hypothesis.

$$\mathit{tonat}(\mathit{inc}\ b') =_N \mathbf{s}(\mathit{tonat}\ b')$$

Then we compute as usual, starting from the left- and right-hand sides.

$$\begin{array}{rll}
\text{lhs:} & \mathit{tonat}(\mathit{inc}(b'\,\mathbf{1})) & \\
& \Longrightarrow \mathit{tonat}((\mathit{inc}\ b')\,\mathbf{0}) & \\
& \Longrightarrow \mathit{double}(\mathit{tonat}(\mathit{inc}\ b')) & \\
& \Longrightarrow \mathit{double}(\mathbf{s}(\mathit{tonat}\ b')) & \text{by ind. hyp.} \\
& \Longrightarrow \mathbf{s}(\mathbf{s}(\mathit{double}(\mathit{tonat}\ b'))) & \\
\text{rhs:} & \mathbf{s}(\mathit{tonat}(b'\,\mathbf{1})) & \\
& \Longrightarrow \mathbf{s}(\mathbf{s}(\mathit{double}(\mathit{tonat}\ b'))) &
\end{array}$$

□

The second case of the proof looks straightforward, but we have swept an important step under the rug. The induction hypothesis had the form $s =_N t$. We used it to conclude $\mathit{double}(s) =_N \mathit{double}(t)$. This is a case of a general substitution principle for equality. However, our notion of equality on natural numbers is defined by introduction and elimination rules, so we need to justify this principle. In general, an application of substitutivity of equality can have one of the two forms

$$\frac{\Gamma \vdash m =_N n\ \mathit{true} \qquad \Gamma \vdash A(m)\ \mathit{true}}{\Gamma \vdash A(n)\ \mathit{true}}\,\mathit{subst}$$

$$\frac{\Gamma \vdash m =_N n\ \mathit{true} \qquad \Gamma \vdash A(n)\ \mathit{true}}{\Gamma \vdash A(m)\ \mathit{true}}\,\mathit{subst}'$$

The second one is easy to justify from the first one by symmetry of equality.

These are examples of *admissible rules of inference.* We cannot derive them directly from the elimination rules for equality, but every instance of them is correct. In general, we say that an inference rule is *admissible* if every instance of the rule is valid.

Theorem: The rule *subst* is admissible.

Proof: By induction on m.

Case: $m = \mathbf{0}$. Then we distinguish cases on n.

Case: $n = \mathbf{0}$. Then $A(m) = A(0) = A(n)$ and the right premise and conclusion are identical.

Case: $n = \mathbf{s}(n')$. Then the right premise is not even needed to obtain the conclusion.

$$\frac{\Gamma \vdash \mathbf{0} =_{N} \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash A(\mathbf{s}(n'))\ \mathit{true}} =_{N} E_{0s}$$

Case: $m = \mathbf{s}(m')$. Then we distinguish cases on n.

Case: $n = \mathbf{0}$. Again, the right premise is not needed to justify the conclusion.

$$\frac{\Gamma \vdash \mathbf{s}(m') =_{N} \mathbf{0}\ \mathit{true}}{\Gamma \vdash A(\mathbf{0})\ \mathit{true}} =_{N} E_{s0}$$

Case: $n = \mathbf{s}(n')$. Then we derive the rule as follows.

$$\frac{\dfrac{\Gamma \vdash \mathbf{s}(m') =_{N} \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash m' =_{N} n'\ \mathit{true}} =_{N} E_{ss} \qquad \Gamma \vdash A(\mathbf{s}(m'))\ \mathit{true}}{\Gamma \vdash A(\mathbf{s}(n'))} \ i.h.$$

Here, a derivation of the conclusion exists by induction hypothesis on m'. Critical is to use the induction hypothesis for $B(m') = A(\mathbf{s}(m'))$ which yields the desired $B(n') = A(\mathbf{s}(n'))$ in the conclusion.

□

In this case, we must formulate the desired principle as a rule of inference. We can write it out as a parametric proposition,

$$\forall m{\in}\mathbf{nat}.\ \forall n{\in}\mathbf{nat}.\ m =_{N} n \supset A(m) \supset A(n)$$

but this can not be proven parametrically in A. The problem is that we need to use the induction hypothesis with a predicate different from A, as the last case in our proof of admissibility shows. And quantification over A, as in

$$\forall m{\in}\mathbf{nat}.\ \forall n{\in}\mathbf{nat}.\ m =_{N} n \supset \forall A{:}\mathbf{nat} \rightarrow \mathit{prop}.\ A(m) \supset A(n)$$

is outside of our language. In fact, quantification over arbitrary propositions or predicates can not be explained satisfactorily using our approach, since the domain of quantification (such at $\mathbf{nat} \to \mathit{prop}$ in the example), includes the new kind of proposition we are just defining. This is an instance of *impredicativity* which is rejected in constructive type theory in the style of Martin-Löf. The rules for quantification over propositions would be something like

$$\frac{\Gamma, p\ \mathit{prop} \vdash A(p)\ \mathit{prop}}{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ A(p)\ \mathit{prop}}\ \forall_2 F^p$$

$$\frac{\Gamma, p\ \mathit{prop} \vdash A(p)\ \mathit{true}}{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ A(p)\ \mathit{true}}\ \forall_2 I^p \qquad \frac{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ A(p)\ \mathit{true} \qquad \Gamma \vdash B\ \mathit{prop}}{\Gamma \vdash A(B)\ \mathit{true}}\ \forall_2 E$$

The problem is that $A(p)$ is not really a subformula of $\forall_2 p{:}\mathit{prop}.\ A(p)$. For example, we can instantiate a proposition with itself!

$$\frac{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ p \supset p\ \mathit{true} \qquad \Gamma \vdash \forall_2 p{:}\mathit{prop}.\ p \supset p\ \mathit{prop}}{\Gamma \vdash (\forall_2 p{:}\mathit{prop}.\ p \supset p) \supset (\forall_2 p{:}\mathit{prop}.\ p \supset p)\ \mathit{true}}\ \forall_2 E$$

Nonetheless, it is possible to allow this kind of quantification in constructive or classical logic, in which case we obtain *higher-order logic*. Another solution is to introduce *universes*. In essence, we do not just have one kind of proposition, by a whole hierarchy of propositions, where higher levels may include quantification over propositions at a lower level. We will not take this extra step here and instead simply use admissible rules of inference, as in the case of substitutivity above.

Returning to data representation, some functions are easy to implement. For example,

$$\begin{array}{rcl} \mathit{shiftl} & \in & \mathbf{bin} \to \mathbf{bin} \\ \mathit{shiftl}\ \ b & = & b\,\mathbf{0} \end{array}$$

implements the double function.

$$\forall b{\in}\mathbf{bin}.\ \mathit{tonat}(\mathit{shiftl}\ b) =_N \mathit{double}(\mathit{tonat}\ b)$$

Proof: By computation.

$$\mathit{tonat}(\mathit{shiftl}\ b) \Longrightarrow \mathit{tonat}(b\,\mathbf{0}) \Longrightarrow \mathit{double}(\mathit{tonat}\ b)$$

□

This trival example illustrates why it is convenient to allow multiple representations of natural numbers. According to the definition above, we have $\mathit{shiftl}\ \epsilon \Longrightarrow \epsilon\,\mathbf{0}$. The result has leading zeroes. If we wanted to keep representations in a standard form without leading zeroes, doubling would have to have a more complicated definition. The alternative approach to work only with standard forms in the representation is related to the issue of data structure invariants, which will be discussed in the next section.

In general, proving the representation theorem for some functions may require significant knowledge in the theory under consideration. As an example, we consider addition on binary numbers.

$$
\begin{array}{lrrcl}
 & & \mathit{add} & \in & \mathbf{bin} \rightarrow \mathbf{bin} \rightarrow \mathbf{bin} \\
\mathit{add} & \epsilon & c & = & c \\
\mathit{add} & (b\,\mathbf{0}) & \epsilon & = & b\,\mathbf{0} \\
\mathit{add} & (b\,\mathbf{0}) & (c\,\mathbf{0}) & = & (\mathit{add}\ b\ c)\,\mathbf{0} \\
\mathit{add} & (b\,\mathbf{0}) & (c\,\mathbf{1}) & = & (\mathit{add}\ b\ c)\,\mathbf{1} \\
\mathit{add} & (b\,\mathbf{1}) & \epsilon & = & b\,\mathbf{1} \\
\mathit{add} & (b\,\mathbf{1}) & (c\,\mathbf{0}) & = & (\mathit{add}\ b\ c)\,\mathbf{1} \\
\mathit{add} & (b\,\mathbf{1}) & (c\,\mathbf{1}) & = & (\mathit{inc}\ (\mathit{add}\ b\ c))\,\mathbf{0}
\end{array}
$$

This specification is primitive recursive: all recursive calls to add are on b. The representation theorem states

$$\forall b{\in}\mathbf{bin}.\ \forall c{\in}\mathbf{bin}.\ \mathit{tonat}(\mathit{add}\ b\ c) =_{N} \mathit{plus}\ (\mathit{tonat}\ b)\ (\mathit{tonat}\ c)$$

The proof by induction on b of this property is left as an exercise to the reader. One should be careful to extract the needed properties of the natural numbers and addition and prove them separately as lemmas.