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.

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

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.

4.7 Complete Induction

In the previous section we have seen an example of a correct rule of inference which was not derivable, only admissible. This was because our logic was not expressive enough to capture this inference rule as a proposition. In this section we investigate a related question: is the logic expressive enough so we can derive different induction principles?

The example we pick is the principle of *complete induction* also known as *course-of-values induction*. On natural numbers, this allows us to use the induction hypothesis on any number smaller than the induction induction variable. The principle of mathematical induction considered so far allows only the immediate predecessor. Corresponding principles exist for structural inductions. For examples, complete induction for lists allows us to apply the induction hypothesis on any tail of the original list.

Complete induction is quite useful in practice. As an example we consider the integer logarithm function. First, recall the specification of *half*.

$$
\begin{array}{lrcl}
 & \mathit{half} & \in & \mathbf{nat} \to \mathbf{nat} \\
\mathit{half} & \mathbf{0} & = & \mathbf{0} \\
\mathit{half} & (\mathbf{s}(\mathbf{0})) & = & \mathbf{0} \\
\mathit{half} & (\mathbf{s}(\mathbf{s}(n))) & = & \mathbf{s}(\mathit{half}(n))
\end{array}
$$

This function is not immediately primitive recursive, but it follows the schema of course-of-values recursion. This is because the recursive call to $\mathit{half}(n +$

2) is on n and $n < n + 2$. We have seen how this can be transformed into a primitive recursion using pairs. In a sense, we show in this section that *every* function specified using course-of-values recursion can be implemented by primitive recursion. Since we prove this constructively, we actually have an effective method to implement course-of-values recursion by primitive recursion.

Next we specify the function $lg(n)$ which calculates the number of bits in the binary representation of n. Mathematically, we have $lg(n) = \lfloor log_2(n+1) \rfloor$.

$$\begin{array}{lrcl} & lg & \in & \mathbf{nat} \to \mathbf{nat} \\ lg & \mathbf{0} & = & \mathbf{0} \\ lg & (\mathbf{s}(n)) & = & \mathbf{s}(lg(half(\mathbf{s}(n)))) \end{array}$$

This specifies a terminating function because $half(\mathbf{s}(n)) < \mathbf{s}(n)$. We now introduce the principal of complete induction and then verify the observation that lg is a terminating function.

> **Principle of Complete Induction.** In order to prove $A(n)$, assume $\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)$ and prove $A(x)$ for arbitrary x.

In order to simplify the discussion below, we say the property A is *complete* if $\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)) \supset A(x)$ is true.

Why is this induction principle valid? The idea is as follows: assume A is complete. We want to show that $\forall n{\in}\mathbf{nat}.\ A(n)$ holds. Why does $A(0)$ hold? If A is complete, then $A(0)$ must be true because there is no $z < 0$. Now, inductively, if $A(0), A(1), \ldots, A(n)$ are all true, then $A(\mathbf{s}(n))$ must also be true, because $A(z)$ for every $z < \mathbf{s}(n)$ and hence $A(n)$ by completeness.

More explicitly, we can prove the principle of complete induction correct as follows.

$$(\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)) \supset A(x)) \supset \forall n{\in}\mathbf{nat}.\ A(n)$$

However, a direct proof attempt of this theorem fails—the induction hypothesis needs to be generalized. The structure of the brief informal argument tells us what it must be.

> **Proof:** Assume A is complete, that is
>
> $$(\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)) \supset A(x)).$$
>
> We show that
>
> $$\forall n{\in}\mathbf{nat}.\ \forall m{\in}\mathbf{nat}.\ m < n \supset A(m)$$
>
> by induction on n. From this the theorem follows immediately. Now to the proof of the generalized theorem.
>
> **Case:** $n = \mathbf{0}$. We have to show $\forall m{\in}\mathbf{nat}.\ m < \mathbf{0}.\ A(m)$. So let m be given and assume $m < \mathbf{0}$. But this is contradictory, so we conclude $A(m)$ by rule $<E_0$.

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

$$\forall m{\in}\mathbf{nat}.\ m < n'.\ A(m).$$

We have to show:

$$\forall m{\in}\mathbf{nat}.\ m < \mathbf{s}(n').\ A(m).$$

So let m be given and assume $m < \mathbf{s}(n')$. Then we distinguish two cases: $m =_N n'$ or $m < n'$. It is a straightforward lemma (which have not proven), that $m < \mathbf{s}(n') \supset (m =_N n' \vee m < n')$.

Subcase: $m =_N n'$. From the completeness of A, using n' for x, we get

$$(\forall z{\in}\mathbf{nat}.\ z < n' \supset A(z)) \supset A(n').$$

But, by renaming z to m the left-hand side of this implication is the induction hypothesis and we conclude $A(n')$ and therefore $A(m)$ by substitution from $m =_N n'$.

Subcase: $m < n'$. Then $A(m)$ follows directly from the induction hypothesis.

□

Now we can use this, for example, to show that the *lg* function is total. For this we formalize the specification from above as a proposition. So assume $lg \in \mathbf{nat} \to \mathbf{nat}$, and assume

$$(lg\ \mathbf{0} =_N \mathbf{0}) \wedge$$
$$(\forall n{\in}\mathbf{nat}.\ lg\ (\mathbf{s}(n)) =_N \mathbf{s}(lg(\mathit{half}(\mathbf{s}(n)))))$$

We prove

$$\forall x{\in}\mathbf{nat}.\ \exists y{\in}\mathbf{nat}.\ lg(x) =_N y$$

This expresses that *lg* describes a total function. In fact, from this constructive proof we can eventually *extract* a primitive recursive implementation of *lg*!

Proof: By complete induction on x. Note that in this proof the property

$$A(x) = (\exists y.\ lg(x) =_N y)$$

Assume the complete induction hypothesis:

$$\forall z.\ z < x \supset \exists y.\ lg(z) =_N y$$

Following the structure of the specification, we distinguish two cases: $x = \mathbf{0}$ and $x = \mathbf{s}(x')$.

Case: $x = \mathbf{0}$. Then $y = \mathbf{0}$ satisfies the specification since $lg(\mathbf{0}) =_N \mathbf{0}$.

Case: $x = \mathbf{s}(x')$. Then $\mathit{half}(\mathbf{s}(x')) < \mathbf{s}(x')$ (by an unproven lemma) and we can use the induction hypothesis to obtain a y' such that $\mathit{lg}(\mathit{half}(\mathbf{s}(x'))) =_N y'$. Then $y = \mathbf{s}(y')$ satisfies

$$\mathit{lg}(\mathbf{s}(x')) =_N \mathbf{s}(\mathit{lg}(\mathit{half}(\mathbf{s}(x')))) =_N \mathbf{s}(y')$$

by the specification of lg and transitivity of equality.

□

Next we examine the computational contents of these proofs. First, the correctness of the principle of complete induction. For simplicity, we assume an error element $\mathit{error} \in \mathbf{0}$. Then we hide information in the statement of completeness in the following manner:

$$\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ [z < x] \supset A(z)) \supset A(x)$$

If we assume that a type τ represents the computational contents of A, then this corresponds to

$$c \in \mathbf{nat} \to (\mathbf{nat} \to \tau) \to \tau$$

In the proof of complete induction, we assume that A is complete. Computationally, this means we assume a function c of this type. In the inductive part of the proof we show

$$\forall n{\in}\mathbf{nat}.\ \forall m{\in}\mathbf{nat}.\ [m < n] \supset A(m)$$

The the function h extracted from this proof satisfies

$$\begin{array}{rrcll} & & h & \in & \mathbf{nat} \to \mathbf{nat} \to \tau & \\ h & \mathbf{0} & m & = & \mathbf{abort}(\mathit{error}) & \\ h & (\mathbf{s}(n')) & m & = & c\ m\ (\lambda m'.\ h(n')\ m') & \text{for } m =_N n' \\ h & (\mathbf{s}(n')) & m & = & h(n')\ m & \text{for } m < n' \end{array}$$

Note that h is clearly primitive recursive in its first argument. In this specification the nature of the proof and the cases it distinguishes are clearly reflected. The overall specification

$$\forall n{\in}\mathbf{nat}.\ A(n)$$

is contracted to a function f where

$$\begin{array}{rrcl} & f & : & \mathbf{nat} \to \tau \\ f & n & = & h\ (\mathbf{s}(n))\ n \end{array}$$

which is not itself recursive, but just calls h.

Assume a function f is defined by the schema of complete recursion and we want to compute $f(n)$. We compute it by primitive recursion on h, starting with $h\ (\mathbf{s}(n))\ n$. The first argument to h is merely a counter. We start at $\mathbf{s}(n)$ and count it down all the way to $\mathbf{s}(\mathbf{0})$. This is what makes the definition of h primitive recursive.

Meanwhile, in the second argument to h (which is always smaller than the first), we compute as prescribed by f. Assume $f(n')$ calls itself recursively on $g(n') < n'$. Then we compute $h\ (\mathbf{s}(n'))\ n'$ by computing $h\ n'\ (g(n'))$, which is a legal recursive call for h. The situation is complicated by the fact that f might call itself recursively several times on different arguments, so we may need to call h recursively several times. Each time, however, the first argument will be decreased, making the recursion legal.

As an example, consider the specification of lg that satisfies the schema of complete recursion since $\mathit{half}(\mathbf{s}(n)) < \mathbf{s}(n)$.

$$\begin{array}{l}(\mathit{lg}\ \mathbf{0} =_N \mathbf{0}) \wedge \\ (\forall n{\in}\mathbf{nat}.\ \mathit{lg}\ (\mathbf{s}(n)) =_N \mathbf{s}(\mathit{lg}(\mathit{half}(\mathbf{s}(n)))))\end{array}$$

The function c that is extracted from the proof of

$$\forall x{\in}\mathbf{nat}.\ \exists y{\in}\mathbf{nat}.\ [\mathit{lg}(x) =_N y]$$

assuming completeness is

$$\begin{array}{llcll} & & c & \in & \mathbf{nat} \to (\mathbf{nat} \to \mathbf{nat}) \to \mathbf{nat} \\ c & \mathbf{0} & r & = & \mathbf{0} \\ c & (\mathbf{s}(n')) & r & = & \mathbf{s}(r\ (\mathit{half}\ (\mathbf{s}(n')))) \end{array}$$

Note that the second argument to c called r represents the induction hypothesis. c itself is not recursive since we only *assumed* the principle of complete induction. To obtain an implementation of lg we must use the proof of the principle of complete induction. Next, the helper function h is

$$\begin{array}{llcll} & & h & \in & \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} & \\ h & \mathbf{0} & m & = & \mathbf{abort}(\mathit{error}) & \\ h & (\mathbf{s}(n')) & m & = & c\ m\ (\lambda m'.\ h(n')\ m') & \text{for } m =_N n' \\ h & (\mathbf{s}(n')) & m & = & h(n')\ m & \text{for } m < n' \end{array}$$

We can expand the definition of c on the right-hand side for the special case of the logarithm and obtain:

$$\begin{array}{llcll} h & \mathbf{0} & m & = & \mathbf{abort}(\mathit{error}) \\ h & (\mathbf{s}(\mathbf{0})) & \mathbf{0} & = & \mathbf{0} \\ h & (\mathbf{s}(\mathbf{s}(n''))) & (\mathbf{s}(n'')) & = & \mathbf{s}(h(\mathbf{s}(n''))\ (\mathit{half}\ (\mathbf{s}(n')))) \\ h & (\mathbf{s}(n')) & m & = & h(n')\ m \qquad \text{for } m < n' \end{array}$$

and

$$\begin{array}{rcl} \mathit{lg} & : & \mathbf{nat} \to \mathbf{nat} \\ \mathit{lg}\ n & : & h\ (\mathbf{s}(n))\ n \end{array}$$

which can easily be seen as a primitive recursive definition of lg since equality and less-than are decidable and we can eliminate the dependency on *error* by returning an arbitrary number in the (impossible) first case in the definition of h.