$$\begin{array}{lcll} ev_2 & n & = & \mathbf{inl}\,\langle\rangle \quad \text{if } n \text{ is even} \\ ev_2 & n & = & \mathbf{inr}\,\langle\rangle \quad \text{if } n \text{ is odd} \end{array}$$

As a third modification, assume we intend to apply ev to even numbers n to obtain $n/2$; if n is odd, we just want an indication that it was not even. The annotation of the type is straightforward.

$$ev_3 : \forall x{\in}\mathbf{nat}.\ (\exists y{\in}\mathbf{nat}.\ [y + y =_N x]) \vee [\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N x]$$

Applying our annotation algorithm to the proof term leads to the following.

$$\begin{array}{l} ev_3 = \lambda x.\ \mathbf{rec}\ x \\ \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, [\mathbf{eq}_0]\rangle \\ \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\ \qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\ \langle c, [p]\rangle = u\ \mathbf{in}\ \mathbf{inr}[\langle c, \mathbf{eq}_s(p)\rangle] \\ \qquad\qquad\qquad\ \mid \mathbf{inr}[w] \Rightarrow \mathbf{let}\ [\langle d, q\rangle = w]\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), [r(x', d, q)]\rangle \end{array}$$

But this version of ev does not satisfy our restriction: in the last line, the hidden variable $[d]$ occurs outside of brackets. Indeed, if we apply our technique of erasing computationally irrelevant subterms we obtain

$$\begin{array}{l} ev_3\ :\ \mathbf{nat} \rightarrow (\mathbf{nat} + \mathbf{1}) \\ ev_3 = \lambda x.\ \mathbf{rec}\ x \\ \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}(\mathbf{0}) \\ \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\ \qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{inr}\,\langle\rangle \\ \qquad\qquad\qquad\ \mid \mathbf{inr}(\_) \Rightarrow \mathbf{inl}(\mathbf{s}(d)) \end{array}$$

where d is required, but not generated by the recursive call. Intuitively, the information flow in the program is such that, in order to compute $n/2$ for even n, we *must* compute $(n-1)/2$ for odd n.

The particular proof we had did not allow the particular bracket annotation we proposed. However, we can give a different proof, which permits this annotation. In this example, it is easier to just write the function with the desired specification directly, using the function ev_1 which preserved the information for the case of an odd number.

$$\begin{array}{l} ev_3\ :\ \mathbf{nat} \rightarrow (\mathbf{nat} + \mathbf{1}) \\ ev_3\, n = \mathbf{inl}(n/2) \qquad\qquad \text{if } n \text{ is even} \\ ev_3\, n = \mathbf{inr}\,\langle\rangle \qquad\qquad \text{if } n \text{ is odd} \end{array}$$

$$\begin{array}{l} ev_3 = \lambda x.\ \mathbf{case}\ ev_1(x) \\ \qquad\qquad \mathbf{of}\ \mathbf{inl}(c) \Rightarrow \mathbf{inl}(c) \\ \qquad\qquad\ \mid \mathbf{inr}(d) \Rightarrow \mathbf{inr}\,\langle\rangle \end{array}$$

To complete this section, we return to our example of the predecessor specification and proof.

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

If we hide all proof objects we obtain:

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

Note that this function does *not* satisfy our restriction: the hidden variable u occurs outside a bracket in the case for $f(\mathbf{0})$. This is because we cannot bracket any subterm of

$$\mathbf{abort}\,(u\ \mathbf{eq}_0) : \exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y) =_N \mathbf{0}]$$

We conclude that our proof of pred' does not lend itself to the particular given annotation. However, we can give a different proof where we supply an arbitrary witness c for y in case x is $\mathbf{0}$ and prove that it satisfies $\mathbf{s}(y) =_N \mathbf{0}$ by $\bot E$ as before. We chose $c = \mathbf{0}$.

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

Now annotation and extraction succeeds, yielding pred. Of course, any natural number would do for the result of $\mathit{pred}(\mathbf{0})$

$$\begin{array}{lcl} \mathit{pred}'_2 & : & \forall x{\in}\mathbf{nat}.\, [\neg x =_N \mathbf{0}] \supset \exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y) =_N x] \\ \mathit{pred}'_2 & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda[u].\, \langle \mathbf{0}, [\mathbf{abort}\,(u\ \mathbf{eq}_0)\rangle]) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda[u].\, \langle x', [\mathit{refl}(\mathbf{s}(x'))]\rangle) \\ \mathit{pred} & : & \mathbf{nat} \to \mathbf{nat} \\ \mathit{pred} & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow x' \end{array}$$

The reader may test his understanding of the erasure process by transforming pred'_2 from above step by step into pred. It requires some of the simplifications on function types.

1.5 Structural Induction

We now leave arithmetic, that is, the theory of natural numbers, and discuss more general data types. We first return to lists, whose elements are drawn

from arbitray types. The reader may wish to remind himself of the basic computation constructs given in Section **??**. We recall here only that there are two introduction rules for lists:

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

In the induction principle, correspondingly, we have to account for two cases. We first state it informally.

To prove $A(l)$ true for an arbitrary list l, prove

1. *$A(\mathbf{nil})$ true and*
2. *$A(x :: l')$ true for an arbitrary x and l', under the assumption $A(l')$ true.*

The first is the base case, the second the induction step. When we write this as a formal inference rules, we obtain the analogue of primitive recursion.

$$\frac{\Gamma \vdash l \in \tau\,\mathbf{list} \quad \Gamma \vdash A(\mathbf{nil})\ true \quad \Gamma, x{\in}\tau, l'{\in}\tau\,\mathbf{list}, A(l')\ true \vdash A(x :: l')\ true}{\Gamma \vdash A(l)\ true}\,\mathbf{list}E$$

This principle is called *structural induction over lists*. Our first theorem about lists will be a simple property of the append function. In order to formulate this property, we need equality over lists. It is defined in analogy with the propositional equality between natural numbers, based on the structure of lists.

$$\frac{\Gamma \vdash l \in \tau\,\mathbf{list} \qquad \Gamma \vdash k \in \tau\,\mathbf{list}}{\Gamma \vdash l =_L k\ prop} =_L F$$

$$\frac{}{\Gamma \vdash \mathbf{nil} =_L \mathbf{nil}\ true} =_L I_n \qquad \frac{\Gamma \vdash l =_L k\ true}{\Gamma \vdash x :: l =_L x :: k\ true} =_L I_c$$

The second introduction rules requires the heads of the two lists to be identical. We can not require them to be equal, because they are of unknown type τ and we do not have a generic equality proposition that works for arbitrary types. However, in this section, we are interested in proving generic properties of lists, rather than, say, properties of integer lists. For this purpose, the introduction rule above, and the three elimination rules below are sufficient.

$$\frac{\Gamma \vdash x :: l =_L y :: k\ true}{\Gamma \vdash l =_L k\ true} =_L E_{cc}$$

$$\frac{\Gamma \vdash \mathbf{nil} =_L y :: k\ true}{\Gamma \vdash C\ true} =_L E_{nc} \qquad \frac{\Gamma \vdash x :: l =_L \mathbf{nil}\ true}{\Gamma \vdash C\ true} =_L E_{cn}$$

Note that the first elimination rule is incomplete in the sense that we also know that x must be identical to y, but we cannot obtain this information by the rule. A solution to this problem is beyond the scope of these notes.

It is straightforward to show that equality is reflexive, symmetric and transitive, and we will use these properties freely below.

Next we give a definition of a function to append two lists which is a slightly modified version from that in Section **??**.

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

In the notation of primitive recursion:

$$\begin{array}{lcl} \mathit{app} & \in & \tau\ \mathbf{list} \to \tau\ \mathbf{list} \to \tau\ \mathbf{list} \\ \mathit{app} & = & \lambda l.\ \mathbf{rec}\ l \\ & & \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ & & \quad \mid f(x :: l') \Rightarrow \lambda k.\ x :: (f(l')\ k) \end{array}$$

We now prove

$$\forall l{\in}\tau\ \mathbf{list}.\ \mathit{app}\ l\ \mathbf{nil} =_L l$$

Proof: By induction on the structure of l.

Case: $l = \mathbf{nil}$. Then $\mathit{app}\ \mathbf{nil}\ \mathbf{nil} =_L \mathbf{nil}$ since

$$\begin{array}{l} \mathit{app}\ \mathbf{nil}\ \mathbf{nil} \\ \Longrightarrow (\mathbf{rec}\ \mathbf{nil} \\ \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\ k)\ \mathbf{nil} \\ \Longrightarrow (\lambda k.\ k)\ \mathbf{nil} \\ \Longrightarrow \mathbf{nil} \end{array}$$

Case: $l = x :: l'$. Then $\mathit{app}\ l'\ \mathbf{nil} =_L l'$ by induction hypothesis. Therefore

$$x :: (\mathit{app}\ l'\ \mathbf{nil}) =_L x :: l'$$

by rule $=_L I_c$. We have to show

$$\mathit{app}\ (x :: l')\ \mathbf{nil} =_L x :: l'.$$

This follows entirely by computation. Starting from the term in the conclusion:

$$\begin{array}{l} \mathit{app}\ (x :: l')\ \mathbf{nil} \\ \Longrightarrow (\mathbf{rec}\ x :: l' \\ \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\ k)\ \mathbf{nil} \\ \Longrightarrow (\lambda k.\ x :: (\mathbf{rec}\ l' \\ \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\ k)\ k)\ \mathbf{nil} \\ \Longrightarrow x :: (\mathbf{rec}\ l' \\ \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\ k)\ \mathbf{nil} \end{array}$$

We arrive at the same term if we start from the induction hypothesis.

$$\begin{array}{l} x :: (\mathit{app}\ l'\ \mathbf{nil}) \\ \Longrightarrow x :: (\mathbf{rec}\ l' \\ \qquad\qquad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \qquad\qquad\ \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\ k)\ \mathbf{nil} \end{array}$$

Recall that computation is allowed in both directions (see Section 1.3), thereby closing the gap between the induction hypothesis and the conclusion. □

For the next theorem, we recall the specification of the reverse function on lists from Section **??**, using an auxiliary function *rev* with an accumulator argument a.

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

The property we will prove is the interaction between *reverse* and *app*.

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

Based on general heuristics, an induction on l is indicated, since it allows us to reduce in the left-hand side. However, such a proof attempt will fail. The reason is that *reverse* is not itself recursive, but defined in terms of *rev*. In such a situation, generalizing the induction hypothesis to express a corresponding property of the recursive function is almost always indicated.

It is often quite difficult to find an appropriate generalization of the induction hypothesis. It is useful to analyse the properties of *rev* in terms of *reverse* and *app*. We generalize from an example

$$\mathit{rev}\ (1 :: 2 :: 3 :: \mathbf{nil})\ (4 :: 5 :: \mathbf{nil}) \Longrightarrow 3 :: 2 :: 1 :: 4 :: 5 :: \mathbf{nil}$$

to conjecture that $\mathit{rev}\ l\ k =_L \mathit{app}\ (\mathit{reverse}\ l)\ k$ (omitting the quanitifers on l and k for the sake of brevity). We may or may not need this property, but it will help us to develop conjectures about the interaction between *rev* and *app*. Once again, the problem with this property is that the right-hand side mentions *reverse* and is not expressed in terms of *rev*. If we substitute the right-hand side will be

$$\mathit{rev}\ l\ k =_L \mathit{app}\ (\mathit{rev}\ l\ \mathbf{nil})\ k$$

Again this does not appear general enough, because of the occurrence of **nil**. If we replace this my a new term m, we also need to modify the left-hand side. The right generalization is suggested by our observation about the interaction of *reverse*, *app* and *rev*. We obtain

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

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* (*app* $l\ k$), to *rev* (*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

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

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

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

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

Then

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

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.

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

Proof: Direct, by computation and previous lemmas.

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

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

1.6 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.