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

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

4.4 Contracting Proofs to Programs

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

Thus we would like to contract proofs to programs, ignoring those parts of a proof term that are not of interest. Of course, what is and what is not of interest depends on the application. To illustrate this point and the process of erasing parts of a proof term, we consider the example of even and odd numbers. We define the addition function (in slight variation to the definition in Section 3.5) and the predicates *even* and *odd*.

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

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

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

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

Now back to the main theorem.

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

Proof: The proof is by induction on x.

Case: $x = \mathbf{0}$. Then x is even, because $\mathbf{0} + \mathbf{0} =_N \mathbf{0}$ by computation and $=_N I_0$. The computation needed here is $\mathbf{0} + \mathbf{0} \Longrightarrow \mathbf{0}$

Case: $x = \mathbf{s}(x')$. By induction hypothesis we know that x' is either even or odd. We distinguish these two subcases.

Subcase: x' is even, that is, $\exists y{\in}\mathbf{nat}.\ y + y =_N x'$. Let's call this element c. Then $c{+}c =_N x'$ and hence $\mathbf{s}(c{+}c) =_N \mathbf{s}(x')$ by rule $=_N I_s$. Therefore $\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N \mathbf{s}(x')$ and x is odd.

Subcase: x' is odd, that is, $\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N x'$. Let's call this element d. Then $\mathbf{s}(d + d) =_N x'$ and $\mathbf{s}(\mathbf{s}(d + d)) =_N \mathbf{s}(x')$ by rule $=_N I_s$. Now, we compute $\mathbf{s}(\mathbf{s}(d + d)) \Longrightarrow \mathbf{s}(\mathbf{s}(d) + d)$ and apply lemma *lmps* to conclude $\mathbf{s}(d){+}\mathbf{s}(d) =_N \mathbf{s}(\mathbf{s}(d{+}d))$. By transitivity, therefore, $\mathbf{s}(d){+}\mathbf{s}(d) =_N \mathbf{s}(x')$. Therefore $\exists y{\in}\mathbf{nat}.\ y + y =_N \mathbf{s}(x')$ and x is even.

□

The proof term corresponding to this informal proof is mostly straightforward.

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

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

Here, $r(x', d, q)$ is a proof term verifying that $\mathbf{s}(d) + \mathbf{s}(d) =_N \mathbf{s}(x')$. It uses transitivity of equality and the lemma *lmps*. Its precise form is not important for the discussion in this section—we give it here only for completeness.

$$\begin{array}{lcl} r(x', d, q) & : & \mathbf{s}(d) + \mathbf{s}(d) =_N \mathbf{s}(x') \\ r(x', d, q) & = & \mathit{trans}\,(\mathbf{s}(d) + \mathbf{s}(d))\,(\mathbf{s}(\mathbf{s}(d) + d))\,(\mathbf{s}(x'))\,(\mathit{lmps}\,(\mathbf{s}(d))\,d)\,q \end{array}$$

Next we consider various versions of this specification and its implementation, erasing "uninteresting" subterms. For the first version, we would like to obtain the witnesses $y{\in}\mathbf{nat}$ in each case, but we do not want to carry the proof that $y + y =_N x$. We indicate this by bracketing the corresponding part of the proposition.

$$ev_1\ :\ \forall x{\in}\mathbf{nat}.\ (\exists y{\in}\mathbf{nat}.\ [y + y =_N x]) \vee (\exists y{\in}\mathbf{nat}.\ [\mathbf{s}(y + y) =_N x])$$

We then bracket the corresponding parts of the proof term. Roughly, every subterm whose type has the form $[A]$ should be bracketed, including variables whose type has this form. The intent is that these subterms will be completely erased before the program is run. In the case of the annotation above we obtain:

$$
\begin{array}{l}
ev_1 \;:\; \forall x{\in}\mathbf{nat}.\, (\exists y{\in}\mathbf{nat}.\, [y + y =_N x]) \vee (\exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y + y) =_N x]) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, [\mathbf{eq}_0]\rangle \\
\qquad\ \mid\ f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\ \langle c, [p]\rangle = u\ \mathbf{in}\ \mathbf{inr}\langle c, [\mathbf{eq}_s(p)]\rangle \\
\qquad\qquad\qquad\ \mid\ \mathbf{inr}(w) \Rightarrow \mathbf{let}\ \langle d, [q]\rangle = w\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), [r(x', d, q)]\rangle
\end{array}
$$

Not every possible bracket annotation of a term is correct. A formal treatment of which bracket annotations are valid is beyond the scope of these notes. However, the main rule is easy to state informally:

Bracketed variables $[x]$ *may occur only inside brackets* $[\ldots x \ldots]$.

This is because bracketed variables are erased before execution of the program. Therefore, an occurrence of a bracketed variable in a term that is *not* erased would lead to a runtime error, since the corresponding value would not be available. We refer to variables of this form as *hidden variables*.

In the example above, $[p]$ and $[q]$ are the only hidden variables. Our restriction is satisfied: p occurs only in $[\mathbf{eq}_s(p)]$ and q only in $[r(x', d, q)]$.

The actual erasure can be seen as proceding in three steps. In the first step, we replace every bracketed proposition $[A]$ by $\top$ and every subterm $[M]$ by its proof term $\langle\,\rangle$. Furthermore, every bracketed variable $[u]$ is replaced by an anonymous variable $\_$, since this variable is not supposed to occur after erasure. We obtain:

$$
\begin{array}{l}
ev_1 \;:\; \forall x{\in}\mathbf{nat}.\, (\exists y{\in}\mathbf{nat}.\, \top) \vee (\exists y{\in}\mathbf{nat}.\, \top) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, \langle\,\rangle\rangle \\
\qquad\ \mid\ f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\ \langle c, \_\rangle = u\ \mathbf{in}\ \mathbf{inr}\langle c, \langle\,\rangle\rangle \\
\qquad\qquad\qquad\ \mid\ \mathbf{inr}(w) \Rightarrow \mathbf{let}\ \langle d, \_\rangle = w\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), \langle\,\rangle\rangle
\end{array}
$$

In the second step we perform simplifications to obtain a function purely operating on data types. For this we have to recall that, under the Curry-Howard isomorphism, for example $\top$ is interpreted as the unit type $\mathbf{1}$, and that disjunction $A \vee B$ is interpreted as a disjoint sum type $\tau + \sigma$.

What happens to universal and existential quantification? Recall that the proof term for $\forall x{\in}\mathbf{nat}.\, A(x)$ is a function which maps every natural number n to a proof term for $A(n)$. When we erase all proof terms, n cannot actually occur in the result of erasing $A(n)$ and the result has the form $\mathbf{nat} \to \tau$, where τ is the erasure of $A(x)$.

Similarly, a proof term for $\exists x{\in}\mathbf{nat}.\, A(x)$ consists of a pair $\langle n, M\rangle$, where n is a natural number (the witness) and M is a proof term for $A(n)$. When we

turn propositions into types by erase, we obtain $\mathbf{nat} \times \tau$, where τ is the erasure of $A(n)$.

Applying this translation operation to our proof, we obtain:

$$\begin{array}{l} ev_1 \ : \ \mathbf{nat} \to (\mathbf{nat} \times \mathbf{1}) + (\mathbf{nat} \times \mathbf{1}) \\ ev_1 = \lambda x.\ \mathbf{rec}\ x \\ \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, \langle\,\rangle\rangle \\ \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\ \qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{inr}\langle \mathbf{fst}(u), \langle\,\rangle\rangle \\ \qquad\qquad\qquad\qquad\ \mid \mathbf{inr}(w) \Rightarrow \mathbf{inl}\langle \mathbf{s}(\mathbf{fst}(w)), \langle\,\rangle\rangle \end{array}$$

Note that our proof term changes in only two places, because the elimination rules for existentials and pairs do not match up. In all other cases we have overloaded our notation, precisely in anticipation of this correspondence. For existentials, we replace

$$\mathbf{let}\,\langle x, u\rangle = M\ \mathbf{in}\ N$$

by

$$[\mathbf{fst}(M)/x][\mathbf{snd}(M)/u]N$$

Finally, we apply some optimizations by eliminating unnecessary constructions involving the unit type. We take advantage of isomorphisms such as

$$\begin{array}{rcl} \tau \times \mathbf{1} & \mapsto & \tau \\ \mathbf{1} \times \tau & \mapsto & \tau \\ \\ \mathbf{1} \to \tau & \mapsto & \tau \\ \tau \to \mathbf{1} & \mapsto & \mathbf{1} \end{array}$$

Note that $\mathbf{1} + \mathbf{1}$ can *not* be simplified: it is a type with two elements, $\mathbf{inl}\,\langle\,\rangle$ and $\mathbf{inr}\,\langle\,\rangle$.

An optimization in a type must go along with a corresponding optimization in a term so that it remains well-typed. This is accomplished by the following simplification rules.

$$\begin{array}{rrcl} & \langle t, s\rangle \in \tau \times \mathbf{1} & \mapsto & t \\ \text{for } t \in \tau \times \mathbf{1}, & \mathbf{fst}(t) \in \tau & \mapsto & t \\ \text{for } t \in \tau \times \mathbf{1}, & \mathbf{snd}(t) \in \mathbf{1} & \mapsto & \langle\,\rangle \\ \\ & \langle s, t\rangle \in \mathbf{1} \times \tau & \mapsto & t \\ \text{for } t \in \mathbf{1} \times \tau, & \mathbf{snd}(t) \in \tau & \mapsto & t \\ \text{for } t \in \mathbf{1} \times \tau, & \mathbf{fst}(t) \in \mathbf{1} & \mapsto & \langle\,\rangle \\ \\ & (\lambda x{\in}\mathbf{1}.\ t) \in \mathbf{1} \to \tau & \mapsto & t \\ \text{for } t \in \mathbf{1} \to \tau, & t\,s \in \tau & \mapsto & t \\ \\ & (\lambda x{\in}\tau.\ t) \in \tau \to \mathbf{1} & \mapsto & \langle\,\rangle \\ \text{for } t \in \tau \to \mathbf{1}, & t\,s \in \mathbf{1} & \mapsto & \langle\,\rangle \end{array}$$

When we apply these transformation to our running example, we obtain

$$
\begin{array}{l}
ev_1 \;:\; \mathbf{nat} \rightarrow (\mathbf{nat} + \mathbf{nat}) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}(\mathbf{0}) \\
\qquad\qquad\ \mid\ f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{inr}(u) \\
\qquad\qquad\qquad\qquad\qquad\ \mid\ \mathbf{inr}(w) \Rightarrow \mathbf{inl}(\mathbf{s}(w))
\end{array}
$$

We can see that this function satisfies the following specification:

$$
\begin{array}{llll}
ev_1 & n & = & \mathbf{inl}(n/2) \qquad \text{if } n \text{ is even} \\
ev_1 & n & = & \mathbf{inr}((n-1)/2) \qquad \text{if } n \text{ is odd}
\end{array}
$$

So $ev_1(n)$ returns the floor of $n/2$, plus a tag which tells us if n was even or odd.

The whole process by which we arrived at this function, starting from the bracket annotation of the original specification can be done automatically by a compiler. A similar process is used, for example, in the Coq system to extract efficient ML functions from constructive proofs in type theory.

Returning to the original specification, assume we want to return only an indication whether the argument is even or odd, but not the result of dividing it by two. In that case, we bracket both existential quantifiers, in effect erasing the witness in addition to the proof term.

$$
\begin{array}{l}
ev_2 \;:\; \forall x{\in}\mathbf{nat}.\, [\exists y{\in}\mathbf{nat}.\, y + y =_N x] \vee [\exists y{\in}\mathbf{nat}.\, \mathbf{s}(y + y) =_N x] \\
ev_2 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}[\langle \mathbf{0}, \mathbf{eq}_{\mathbf{0}} \rangle] \\
\qquad\qquad\ \mid\ f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}[u] \Rightarrow \mathbf{let}\, [\langle c, p \rangle = u]\ \mathbf{in}\ \mathbf{inr}[\langle c, \mathbf{eq}_{\mathbf{s}}(p) \rangle] \\
\qquad\qquad\qquad\qquad\qquad\ \mid\ \mathbf{inr}[w] \Rightarrow \mathbf{let}\, [\langle d, q \rangle = w]\ \mathbf{in}\ \mathbf{inl}[\langle \mathbf{s}(d), r(x', d, q) \rangle]
\end{array}
$$

Fortunately, our restriction is once again satisfied: bracketed variables (this time, u, c, p, w, d, q) appear only within brackets. The occurrences of c and p in the **let**-expression should be considered bracketed, because u and therefore c and p will not be carried when the program is executed. A similar remark applies to w, d. We now skip several steps, which the reader may want to reconstruct, to arrive at

$$
\begin{array}{l}
ev_2 \;:\; \mathbf{nat} \rightarrow (\mathbf{1} + \mathbf{1}) \\
ev_2 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\, \langle\,\rangle \\
\qquad\qquad\ \mid\ f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}\_ \Rightarrow \mathbf{inr}\, \langle\,\rangle \\
\qquad\qquad\qquad\qquad\qquad\ \mid\ \mathbf{inr}\_ \Rightarrow \mathbf{inl}\, \langle\,\rangle
\end{array}
$$

Note that this function simply alternates between $\mathbf{inl}\,\langle\,\rangle$ and $\mathbf{inr}\,\langle\,\rangle$ in each recursive call, thereby keeping track if the number is even or odd. It satisfies

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

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

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

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.

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

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

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.