In this section, we will compare the Coq and Minlog formalizations of the normalization proof with a formalization in the theorem prover Isabelle/HOL. We start by giving the definition of types and terms:

```plaintext
datatype type = Iota | Arrow type type (infix \rightarrow 80)

datatype trm = Var name (\' - 90) | App trm trm (infix \cdot 80) | Abs name trm (\lambda [\cdot [70,70] 80])
```

The judgement stating that \( t \) has type \( \tau \) in context \( \rho \) is denoted by \( \rho_s \vdash t : \tau \).

The definition of the typing judgement is as usual. In contrast to Minlog, Isabelle/HOL allows the definition of predicates by recursion over datatypes (also called “strong elimination” in Coq jargon). Thus, we can simply define \( SC \) by recursion over the datatype \( type \):

```plaintext
consts SC :: type \Rightarrow trm \Rightarrow bool
primrec
  SC-Atom: SC Iota r = SN r
  SC-Fun: SC (\rho \rightarrow \sigma) r = (\forall s. (SC \rho s) \rightarrow (SC \sigma (r \cdot s)))
```

However, when it comes to extracting a program from a proof involving \( SC \), we face a problem: Since predicates in a proof become types in the extracted program, predicates such as \( SC \) defined by recursion on datatypes give rise to programs using dependent types. Such programs can neither be expressed inside Isabelle/HOL, nor can they easily be translated to functional programming languages such as ML. In order to get a program which is typable in a functional programming language without dependent types, we realize the formula \( SC \tau t \) by a datatype \( D \) with two constructors \( Term \) and \( Func \). The former corresponds to the case where \( \tau \) is a base type, whereas the latter corresponds to the case where \( \tau \) is a function type. In ML, one would define the datatype \( D \) as follows:
This datatype is beyond the scope of Isabelle/HOL, since $D$ occurs negatively in the argument type of $Func$. It is interesting to note that in the logic HOLCF, which is a conservative extension of HOL with LCF, one can actually define the above datatype, provided that the type of partial continuous functions is used instead of the type of total functions. For the moment, we just assert the existence of type $D$, together with suitable constructors:

```plaintext
typedecl D
consts
  Term :: (nat ⇒ trm) ⇒ D
  Func :: (trm ⇒ D ⇒ D) ⇒ D
  destTerm :: D ⇒ nat ⇒ trm
  destFunc :: D ⇒ trm ⇒ D ⇒ D
```

Here, $destTerm$ and $destFunc$ are destructors corresponding to the constructors of the datatype $D$. These can be implemented using pattern matching.

We can now assign realizing terms to the two characteristic equations for $SC$. Since we can view an equation between propositions as a conjunction of two implications, the equations for $SC$ are realized by pairs, of which the first component is a destructor, and the second component is a constructor:

realizers

$SC$-Atom: $\lambda t. (destTerm, Term)$
$SC$-Fun: $\lambda q \sigma t. (destFunc, Func)$

The proof of strong normalization is composed of the following parts:

```
lemma One : $\forall r. (SC q r \rightarrow SN r) \land (SA r \rightarrow SC q r)$
lemma Two : $\forall r r'. SC q r' \rightarrow H r r' \rightarrow SC q r$
lemma Three : $\forall gs \varphi q. gs \vdash r : q \rightarrow (\forall z. SC (qs z) (\varphi z)) \rightarrow SC q (r \cdot \varphi)
lemma Norm: $\forall gs q r. gs \vdash r : q \rightarrow (\forall k. F r k \rightarrow (\exists s. N r s))$
```

The computationally relevant predicates $SN$ and $SA$ are defined by

$SN r \equiv \forall k. F r k \rightarrow (\exists s. N r s)$
$SA r \equiv \forall k. F r k \rightarrow (\exists s. A r s)$

The programs extracted from the above theorems have the types

```
One: type ⇒ trm ⇒ (D ⇒ nat ⇒ trm) × ((nat ⇒ trm) ⇒ D)
Two: type ⇒ trm ⇒ trm ⇒ D ⇒ D
Three: trm ⇒ (name ⇒ type) ⇒ (name ⇒ trm) ⇒
       type ⇒ (name ⇒ D) ⇒ D
Norm: (name ⇒ type) ⇒ type ⇒ trm ⇒ nat ⇒ trm
```

They are defined as follows:
One ≡
\text{type-rec} (\lambda x. (\text{destTerm}, \text{Term}))
(\lambda x xa H Ha xb.
(\lambda Hb x.
\lambda[x, x]. \text{fst} (H (xa - \star_{\#} x))
(destFunc Hb (\star_{\#} x) (snd (H (\star_{\#} x)) (\lambda xa. \star_{\#} x)))
(Suc x),
\lambda Hb. \text{Func} (\lambda s Hc. \text{snd} (Ha (xb - \#_{\#} x)) (\lambda x. Hb x - \text{fst} (H s) Hc x)))))

Two ≡
\text{type-rec} (\lambda x xa H. \text{Term} (\text{destTerm} H))
(\lambda \text{type1 type2 H Ha r r' Hb}.
\text{Func} (\lambda s H. \text{Ha} (r - s) (r' - s) (\text{destFunc} Hb s H)))

Three ≡
\text{trm-rec} (\lambda name x xa xb H.
(\lambda x xa H Ha xb xc xd Hb.
\text{destFunc} (H xb xc (\text{app-type-elim} xb x xa xd \rightarrow xd) Hb) (xa \cdot xc)
(Ha xb xc (\text{app-type-elim} xb x xa xd) Hb))
(\lambda name trm H gs \vartheta \varrho.
let (x, y) = \text{abs-type-elim} \varrho
in \text{Func}
(\lambda s Hb.
\text{Two} y ((\lambda [name, trm \cdot \vartheta] - s) (\text{trm} \cdot \text{update} \vartheta name s)
(H (\text{update} gs name x) (\text{update} \vartheta name s) y)
(\lambda z. \text{case name-eq-dec name z of Left } \Rightarrow Hb
| \text{Right } \Rightarrow Ha z)))))

Norm ≡
\lambda \varrho s \vartheta r.
\text{fst} (\text{One} \varrho r)
(\text{Three} \varrho s \vartheta \text{subst-id} \varrho (\lambda x. \text{snd} (\text{One} (\varrho s x) (\star_{\#} x)) (\lambda xa. \star_{\#} x)))

Due to the lack of non-computational quantifiers in Isabelle, the above programs contain typing information for the term to be normalized, which is unnecessary for the computation. In particular, the extracted programs use auxiliary functions corresponding to the elimination rules

\begin{align*}
\text{var-type-elim:} & \quad gs \vdash \hat{x} : q \implies q = gs x \\
\text{abs-type-elim:} & \quad \Gamma \vdash \lambda[x]. t : q \implies \\
& \quad \exists \sigma \sigma', q = \sigma \rightarrow \sigma' \land \text{update} \quad \Gamma x \sigma \vdash t : \sigma' \\
\text{app-type-elim:} & \quad \Gamma \vdash r - s : q \implies \exists \sigma. \quad \Gamma \vdash r : \sigma \rightarrow q \land \Gamma \vdash s : \sigma
\end{align*}

for the typing judgement. It turns out that all typing information can be omitted from function Three. This may seem a bit surprising, since Three calls function Two, which is defined by recursion on types. However, a closer look reveals that Two is actually just a complicated formulation of the identity function and can therefore be omitted. Unfortunately, Isabelle’s program extraction framework cannot detect this automatically.