Tait

Pierre Letouzey and Stefan Berghofer

August 16, 2005

Contents

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:

datatype *type* =
  Iota
| *Arrow type type* (**infix** $\rightharpoonup$ *80*)

datatype *trm* =
  Var name (´- *90*)
| *App trm trm* (**infix** ¨ *80*)
| *Abs name trm* (λ[-].- [*70*,*70*] *80*)

The judgement stating that t has type τ in context ϱs is denoted by $\varrho s \vdash t : \tau$. 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*:

consts *SC* :: *type* ⇒ *trm* ⇒ *bool*
primrec
SC-Atom: *SC Iota r* = *SN r*
SC-Fun: *SC* (*ϱ*⇀*σ*) *r* = (∀ *s*. (*SC ϱ s*) ⟶ (*SC σ* (*r*¨*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 τ is a base type, whereas the latter corresponds to the case where τ is a function type. In ML, one would define the datatype D as follows:

```
datatype D = Term of nat -> trm | Func of trm -> D -> D
```

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:

typedecl D

consts
$Term :: (nat \Rightarrow trm) \Rightarrow D$
$Func :: (trm \Rightarrow D \Rightarrow D) \Rightarrow D$
$destTerm :: D \Rightarrow nat \Rightarrow trm$
$destFunc :: D \Rightarrow trm \Rightarrow D \Rightarrow D$

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: $\lambda\ \varrho\ \sigma\ t.\ (destFunc,\ Func)$

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

lemma *One* : $\forall r.\ (SC\ \varrho\ r \longrightarrow SN\ r) \wedge (SA\ r \longrightarrow SC\ \varrho\ r)$
lemma *Two* : $\forall r\ r'.\ SC\ \varrho\ r' \longrightarrow H\ r\ r' \longrightarrow SC\ \varrho\ r$
lemma *Three* : $\forall \varrho s\ \vartheta\ \varrho.\ \varrho s \vdash r : \varrho \longrightarrow (\forall z.\ SC\ (\varrho s\ z)\ (\vartheta\ z)) \longrightarrow SC\ \varrho\ (r \cdot \vartheta)$
lemma *Norm*: $\forall\ \varrho s\ \varrho\ r.\ \varrho s \vdash r : \varrho \longrightarrow (\forall k.\ F\ r\ k \longrightarrow (\exists s.\ N\ r\ s))$

The computationally relevant predicates SN and SA are defined by

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

The programs extracted from the above theorems have the types

$$\begin{array}{ll}
One: & type \Rightarrow trm \Rightarrow (D \Rightarrow nat \Rightarrow trm) \times ((nat \Rightarrow trm) \Rightarrow D) \\
Two: & type \Rightarrow trm \Rightarrow trm \Rightarrow D \Rightarrow D \\
Three: & trm \Rightarrow (name \Rightarrow type) \Rightarrow (name \Rightarrow trm) \Rightarrow \\
 & type \Rightarrow (name \Rightarrow D) \Rightarrow D \\
Norm: & (name \Rightarrow type) \Rightarrow type \Rightarrow trm \Rightarrow nat \Rightarrow trm
\end{array}$$

They are defined as follows:

```
One ≡
type-rec (λx. (destTerm, Term))
 (λx xa H Ha xb.
    (λHb x.
       λ[♯ x].fst (Ha (xb ¨ ´♯ x))
              (destFunc Hb (´♯ x) (snd (H (´♯ x)) (λxa. ´♯ x)))
              (Suc x) ,
     λHb. Func (λs Hc. snd (Ha (xb ¨ s)) (λx. Hb x ¨ fst (H s) Hc x))))

Two ≡
type-rec (λx xa H. Term (destTerm H))
 (λtype1 type2 H Ha r r' Hb.
    Func (λs H. Ha (r ¨ s) (r' ¨ s) (destFunc Hb s H)))

Three ≡
trm-rec (λname x xa xb H. H name)
 (λx xa H Ha xb xc xd Hb.
    destFunc (H xb xc (app-type-elim xb x xa xd ⇁ xd) Hb) (xa • xc)
     (Ha xb xc (app-type-elim xb x xa xd) Hb))
 (λname trm H ϱs ϑ ϱ Ha.
    let (x, y) = abs-type-elim ϱ
    in Func
       (λs Hb.
          Two y ((λ[name].trm • ϑ) ¨ s) (trm • update ϑ name s)
           (H (update ϱs name x) (update ϑ name s) y
             (λz. case name-eq-dec name z of Left ⇒ Hb
                  | Right ⇒ Ha z))))

Norm ≡
λϱs ϱ r.
  fst (One ϱ r)
   (Three r ϱs subst-id ϱ (λx. snd (One (ϱs x) (´x)) (λxa. ´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

$$
\begin{array}{ll}
\textit{var-type-elim}: & \varrho s \vdash {}´x : \varrho \Longrightarrow \varrho = \varrho s\ x \\
\textit{abs-type-elim}: & \Gamma \vdash\ \lambda[x].t\ : \varrho \Longrightarrow \\
 & \quad \exists \sigma\ \sigma'.\ \varrho = \sigma \rightharpoonup \sigma' \wedge\ \ \textit{update}\ \Gamma\ x\ \sigma \vdash t : \sigma' \\
\textit{app-type-elim}: & \Gamma \vdash r\ ¨\ s : \varrho \Longrightarrow \exists \sigma.\ \ \Gamma \vdash r : \sigma \rightharpoonup \varrho \wedge\ \ \Gamma \vdash s : \sigma
\end{array}
$$

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.