Dependently Typed Conversions

Ramana Kumar

November 10, 2011

In the context of HOL-like theorem provers, a *conversion* is an ML function of type `term -> thm` that takes a term t and produces a theorem $\vdash t = t'$ for some term t'. The ML type system only guarantees that a conversion returns a theorem (or raises an exception). The minimum we would expect from a conversion is that it produces a theorem of the correct form (that is, an equality with the input term on the left).

In general, we might expect even more. For example, we might want a conversion intended to transform a term to negation normal form to take t to a theorem $\vdash t = t'$ such that $\texttt{nnf}(t')$, where `nnf` is a predicate testing whether a term is in negation normal form. The conversion might impose conditions on its input term, for example, that it is made only of first-order quantifiers and connectives, and is closed. Enhancing the opaque `term -> thm` type, we might better express the type of such a conversion as $\{t : \texttt{term} \mid \texttt{fof}(t)\} \to \{th : \texttt{thm} \mid \texttt{lhs}(\texttt{concl}(th)) = t \wedge \texttt{nnf}(\texttt{rhs}(\texttt{concl}(th)))\}$.

Such a type is dependent, because the type of theorems returned depends on the input term, which is a value. It is well-known that dependent types can be used to express rich information about programs. Why would we want this information in the type system for conversions? Because conversions are easier to use and reuse if you know what they are doing. Currently (that is, in HOL4 and other existing proof assistants), the only clues about what a conversion might do are in its name, comments surrounding its description, and the code itself.

The code would often require a significant time investment to read and understand. The name and comments are liable to be wrong for at least two reasons: code evolves faster than documentation, and people make mistakes in reasoning about what the code does. The advantage of putting more information in the type system is that it is always current and always correct.

An additional benefit is that a type-checked program with a rich type has essentially been proved correct. Optimizations and tricks in the code can be added safely (albeit maybe not easily) without compromising correctness.

Conversions are often strung together (with the `THENC` combinator). In this situation, knowing exactly what each one does is important for reasoning about the combined conversion. This can also open up opportunities for optimization, for example by skipping steps on terms that are already known to be in a certain good form.