Dedukti in a Nutshell

Ronan Saillard
MINES ParisTech
ronan.saillard@cri.ensmp.fr

*Dedukti* [1] is a proof checker based on rewriting and dependent types. It implements the \(\lambda\Pi\)-calculus modulo, a very expressive logical framework introduced by Cousineau and Dowek in [3]. The combination of rewriting and dependent types makes it a convenient tool for writing proof and programs.

For instance let us consider a *Dedukti* transcription of how the addition of Peano naturals is usually defined in Coq [2] or Agda [4]:

\[
\begin{align*}
\text{Nat} & : \text{Type}. \\
0 & : \text{Nat}. \\
S & : \text{Nat} \to \text{Nat}. \\
\text{plus} & : \text{Nat} \to \text{Nat} \to \text{Nat}. \\
[n : \text{Nat}] & \text{plus} 0 n \to n \\
[n1 : \text{Nat}, n2 : \text{Nat}] & \text{plus} (S n1) n2 \to S (\text{plus} n1 n2).
\end{align*}
\]

The definition of \text{plus} is asymmetric in its arguments. In particular, \text{0} is computationally left-neutral but only propositionally right-neutral (for all term \(n\) of type \text{Nat}, \text{plus} 0 \(n\) is syntactically convertible to \(n\) but \text{plus} \(n\) \text{0} is not).

This difference becomes crucial in presence of dependent types. Let us consider the definition of vectors defined as lists depending on their length:

\[
\begin{align*}
A & : \text{Type}. \\
\text{Vector} & : \text{Nat} \to \text{Type}. \\
\text{Nil} & : \text{Vector} \text{0}. \\
\text{Cons} & : n : \text{Nat} \to A \to \text{Vector} n \to \text{Vector} (S n). \\
\text{append} & : n1 : \text{Nat} \to n2 : \text{Nat} \to \\
& \quad l1 : \text{Vector} n1 \to l2 : \text{Vector} n2 \to \text{Vector} (\text{plus} n1 n2).
\end{align*}
\]

\[
\begin{align*}
[n : \text{Nat}, l : \text{Vector} n] & \text{append} \text{Nil} l \to l \\
[n1 : \text{Nat}, n2 : \text{Nat}, l1 : \text{Vector} n1, l2 : \text{Vector} n2, a : A] & \text{append} (S n1) n2 (\text{Cons} n1 a l1) l2 \to \text{Cons} (\text{plus} n1 n2) a (\text{append} n1 n2 l1 l2).
\end{align*}
\]

For all terms \(n\) and \(l\) of types \text{Nat} and \text{Vector} \(n\), \text{append} \(1\) \text{Nil} is not convertible to \(1\) and these two terms don’t even have the same type; \text{append} \(1\) \text{Nil} has type \text{Vector} (\text{plus} \text{0} \(0\)) which is not convertible to \text{Vector} \(n\) so \text{append} \(1\) \text{Nil} is not even propositionally equal to \(1\).

In *Dedukti*, we can add rewrite-rules to get a symmetric version of \text{plus}.

To our definition of \text{plus}, we can add these rules:

\[
\begin{align*}
[n : \text{Nat}] & \text{plus} n 0 \to n \\
[n1 : \text{Nat}, n2 : \text{Nat}] & \text{plus} n1 (S n2) \to S (\text{plus} n1 n2).
\end{align*}
\]
This way, \texttt{append 1 Nil} becomes propositionally equal to \texttt{1} and we can even add the rewrite-rule:

\[
[n : \text{Nat}, l : \text{Vector n}] \text{append } l \text{ Nil } \rightarrow l.
\]

This technique is handy but comes at a price: whenever a rewrite rule is added, we have to make sure that the system remains confluent and strongly normalizing. This property needs to be verified either by the system or by the user. In our example, we introduced critical pairs. However we can show that they are joinable and, combined with the strong normalization, this implies the confluence of the system.

This talk will introduce \textit{Dedukti} through a series of examples showing how rewrite rules can be conveniently used to write programs in a dependently typed framework. We will also present efficient encodings of different logics into the $\lambda\Pi$-calculus modulo and show how to check theorems in these logics with \textit{Dedukti}.

\textbf{Bibliographie}


