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]:

```
Nat : Type.
0 : Nat.
S : Nat -> Nat.

plus : Nat -> Nat -> Nat.
[n : Nat] plus 0 n --> n
[n1 : Nat, n2 : Nat] plus (S n1) n2 --> S (plus n1 n2).
```

The definition of `plus` is asymmetric in its arguments. In particular, `0` is computationally left-neutral but only propositionally right-neutral (for all term `n` of type `Nat`, `plus 0 n` is syntactically convertible to `n` but `plus n 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:

```
A : Type.
Vector : Nat -> Type.
Nil : Vector 0.
Cons : n : Nat -> A -> Vector n -> Vector (S n).

append : n1 : Nat -> n2 : Nat ->
         l1 : Vector n1 -> l2 : Vector n2 -> Vector (plus n1 n2).

[n : Nat, l : Vector n] append Nil l --> l
[n1 : Nat, n2 : Nat, l1 : Vector n1, l2 : Vector n2, a : A]
  append (S n1) n2 (Cons n1 a l1) l2 --> Cons (plus n1 n2) a (append n1 n2 l1 l2).
```

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

In *Dedukti*, we can add rewrite-rules to get a symmetric version of `plus`.

To our definition of `plus`, we can add these rules:

```
[n : Nat] plus n 0 --> n
[n1 : Nat, n2 : Nat] plus n1 (S n2) --> S (plus n1 n2).
```

This way, `append l Nil` becomes propositionally equal to `l` and we can even add the rewrite-rule:

```
[n : Nat, l : Vector n] append l Nil --> 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 *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 *Dedukti*.

Bibliographie

[1] Mathieu Boespflug, Quentin Carbonneaux, Olivier Hermant, and Ronan Saillard. Dedukti : `http://dedukti.gforge.inria.fr`.

[2] The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

[3] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[4] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266, 2008.