Dedukti: a universal proof checker

Ronan Saillard

INRIA Deducteam

MINES ParisTech

April 11, 2013

Table of Contents

Dedukti

Overview

What is Dedukti?

Dedukti is:

- A type-checker for the $\lambda\Pi$-calculus modulo
- A theory-independent proof-checker
- A logical framework with rewrite rules
- A framework to study interoperability
- A type-checker with new implementation techniques

Contributors:

- Mathieu Boespflug
- Quentin Carbonneaux
- Olivier Hermant, Ronan Saillard

Dedukti

Motivations

(RE-)CHECKING PROOFS

RE-CHECKING

We want to be able to recheck proofs from proof assistants (Coq, HOL, etc).

Why ?

- Extra confidence.
- de Bruijn Criterion

  A proof assistant satisfies the D.B. criterion if it generates proof objects that can be checked independently of the system that created it using a simple program that a skeptical user can write him/herself. (H. Geuvers)
- (More efficient proof checker).

CHECKING PROOF FROM UNTRUSTED SOFTWARES

Eg. we plan to look at traces generated by SMT solvers.

Interoperability

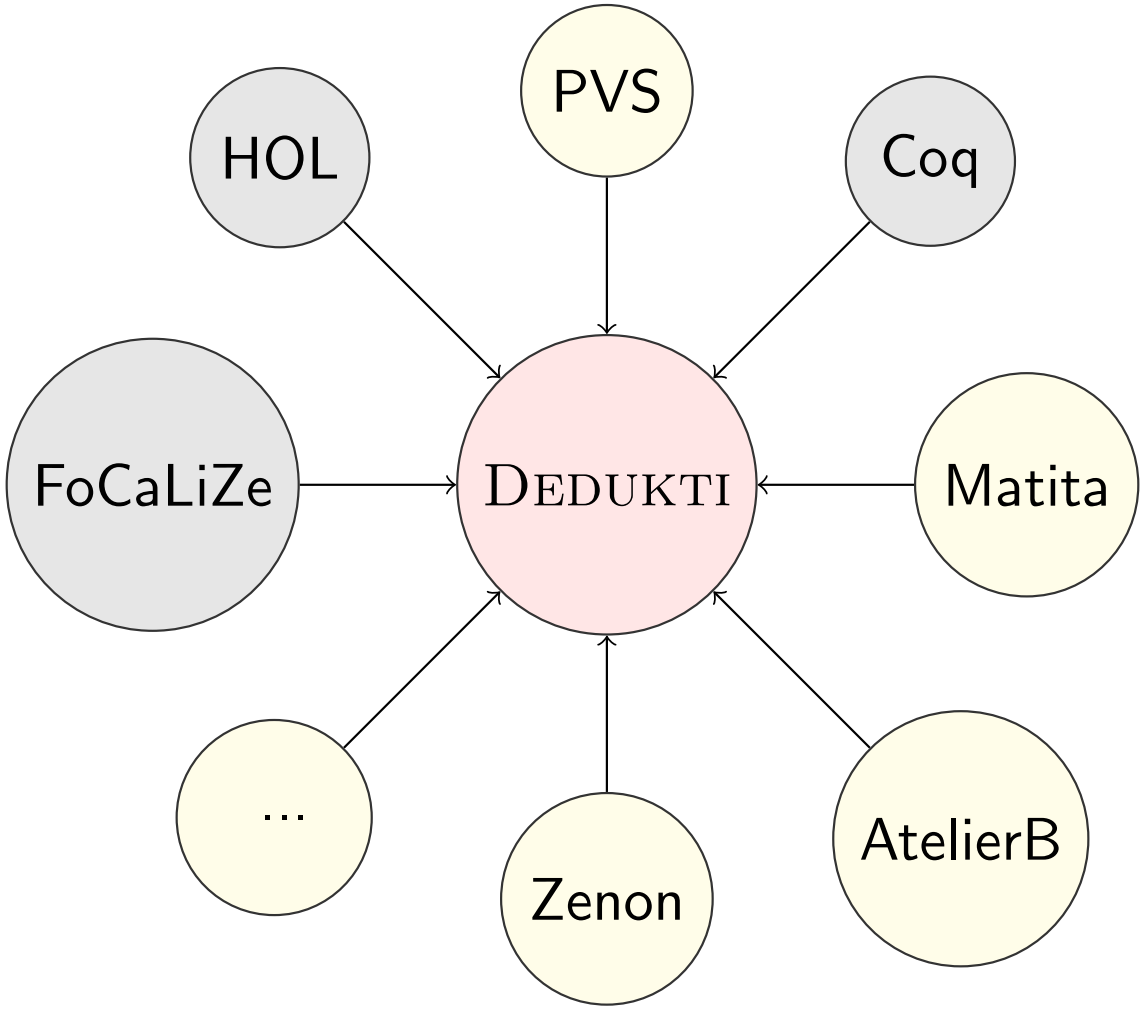
long term goal

- Prove A in **Coq**
- Prove $A \Rightarrow B$ in **HOL**
- Deduce B in **Dedukti**

Dedukti

Companion Tools

Companion tools



Theoretical Foundations

$\lambda\Pi$-calculus modulo

The λΠ-calculus modulo

λΠ-calculus modulo := simply typed λ-calculus + dependent types + rewrite rules

Dependent types

Listn : nat → Type.
List42 : Type := Listn 42.

Rewrite Rules

r: $head\ (hd :: tl) \longrightarrow hd$
new reduction relation: $\rightarrow_{\beta r}$

The λΠ-calculus modulo as a logical framework

The λΠ-calculus modulo is very expressive:
it can encode all the Functional Pure Type Systems [Cousineau & Dowek, 2007]

Simply Typed Lambda Calculus

$$\frac{\Gamma \text{ wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \text{ (Variable)}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \text{ (Application)}$$

$$\frac{\Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B} \text{ (Abstraction)}$$

Figure: Typing rules for the **simply typed λ-calculus**

$\lambda\Pi$-calculus

$$\frac{}{\emptyset \text{ wf}}\ (\text{Empty}) \qquad \frac{\Gamma \ wf \qquad \Gamma \vdash A : s}{\Gamma(x : A) \text{ wf}}\ (\text{Declaration})$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}\ (\text{Type}) \qquad \frac{\Gamma \text{ wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\text{Variable})$$

$$\frac{\Gamma \vdash t : \Pi x^A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B[x \leftarrow u]}\ (\text{Application})$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma(x : A) \vdash B : s}{\Gamma \vdash \Pi x^A.B : s}\ \textbf{(Product)}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : s \qquad \Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}\ (\text{Abstraction})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash t : B}\ \textbf{(Conversion)}$$

FIGURE: Typing rules for the **$\lambda\Pi$-calculus**

$\lambda\Pi$-calculus modulo

$$\frac{}{\emptyset \text{ wf}}\ (\text{Empty}) \qquad \frac{\Gamma \ wf \qquad \Gamma \vdash A : s}{\Gamma(x : A) \text{ wf}}\ (\text{Declaration})$$

$$\frac{\Gamma\Delta \vdash l : t \qquad \Gamma\Delta \vdash r : t}{\Gamma(\Delta : l \to r) \textbf{ wf}}\ \textbf{(Rewrite)}$$

$$\frac{\Gamma \text{ wf}}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}\ (\text{Type}) \qquad \frac{\Gamma \text{ wf} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\text{Variable})$$

$$\frac{\Gamma \vdash t : \Pi x^A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B[x \leftarrow u]}\ (\text{Application})$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma(x : A) \vdash B : s}{\Gamma \vdash \Pi x^A.B : s}\ (\text{Product})$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : s \qquad \Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}\ (\text{Abstraction})$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_\Gamma B}{\Gamma \vdash t : B}\ (\text{Conversion})$$

FIGURE: Typing rules for the **$\lambda\Pi$-calculus modulo**

An example

Concatenation of dependent lists

An example

```
A   : Type.
Nat: Type.
Z   : Nat.
S   : Nat -> Nat.

plus: Nat -> Nat -> Nat.
[m:Nat]        plus  Z    m --> m
[n:Nat,m:Nat] plus (S n) m --> S (plus n m).

Listn : Nat -> Type.
nil   : Listn Z.
cons  : n:Nat -> A -> Listn n -> Listn (S n).

append: n:Nat -> Listn n -> m:Nat -> Listn m -> Listn (plus n m).
[n:Nat,l2:Listn n]      append Z nil n l2 --> l2
[n:Nat,l1:Listn n,m:Nat,l2:Listn m,a:A]
     append {S n} (cons n a l1) m l2 --> cons (plus n m) a (append n l1 m l2).
```

Implementation

The big picture

Dedukti: goals and weapons

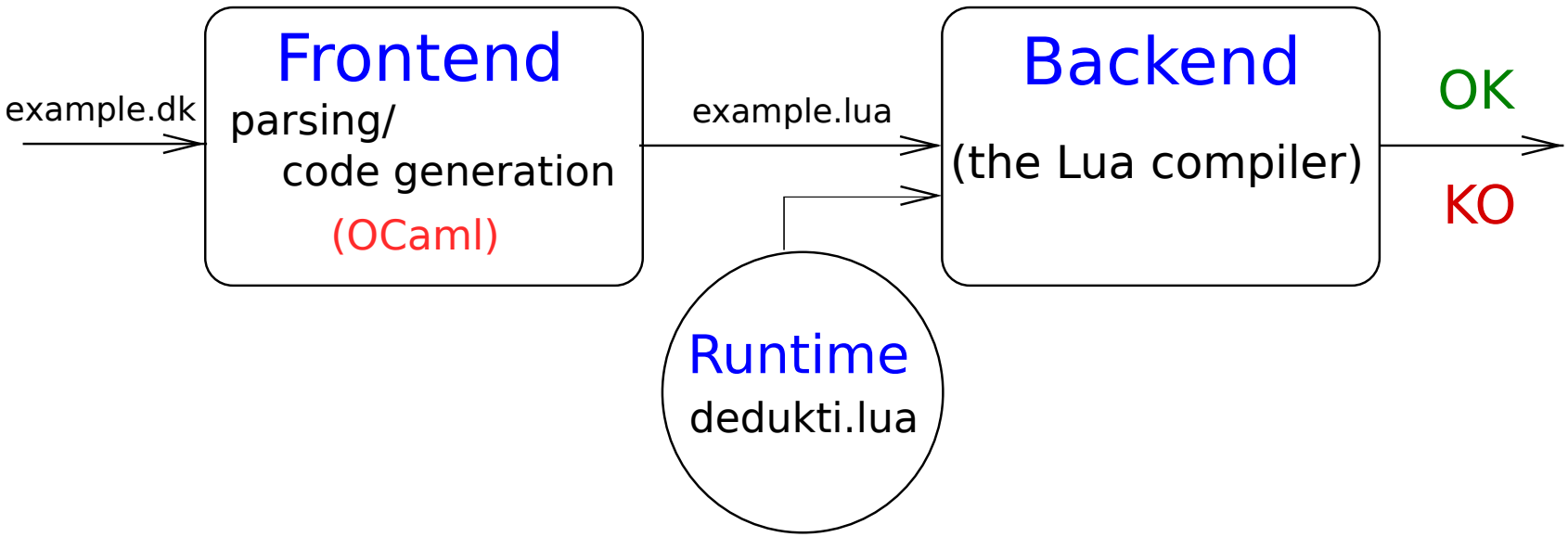
Goals:

- Fast type checking.
- Versatility: efficient for any (embedded) logic.

Philosophy, be lazy:

- Do not reimplement long-standing features.
- Reuse existing tools.

The Big Picture



- Dedukti is a proof-checker **generator**

Advantages

Use of the target language capability:

- **Higher Order Abstract Syntax** (HOAS).
- **Normalisation by Evaluation** (NbE).

Dedukti	Lua
$(\lambda x.M)N$	(function (x) M end)(N)

$head\ (hd :: tl) \longrightarrow hd$
$head\ nil \longrightarrow default$
	function head (x)
	  if x.id == Cons and then
	  return x.args[1]
	  elseif x.id = Nil
	  return default
	  end
	end

Implementation

The type-checking algorithm

Bidirectional/Context-Free Type Checking

Bidirectional type checking [Coquand, 1996]:

- Mix of type-checking and type-inference.
- Smaller terms (Curry-style).

Context-Free type checking [Boespflug, 2011]:

- No search in contexts.
- Type annotation for free variables.

The Resulting System

$\boxed{\vdash M \Rightarrow A}$ the term M synthesizes type A

$$(Type)\ \frac{}{\vdash Type \Rightarrow Kind} \qquad (Var)\ \frac{}{\vdash x^A \Rightarrow A}$$

$$(App)\ \frac{\vdash M \Rightarrow C \qquad C \rightarrow_w^* \Pi x^A.B \qquad \vdash N \Leftarrow A}{\vdash M\ N \Rightarrow B[x \leftarrow N]}$$

$\boxed{\vdash M \Leftarrow A}$ the terms M verifies type A

$$(Abs)\ \frac{C \rightarrow_w^* \Pi x^A.B \qquad \vdash M[x \leftarrow y^A] \Leftarrow B[x \leftarrow y^A]}{\vdash \lambda x.M \Leftarrow C}\ (\text{y fresh})$$

$$(Prod)\ \frac{\vdash A \Leftarrow Type \qquad \vdash B[x \leftarrow y^A] \Leftarrow s}{\vdash \Pi x^A.B \Leftarrow s}\ (\text{y fresh})\ (s \in \{Type, Kind\})$$

$$(conv)\ \frac{\vdash N \Rightarrow B \qquad A \equiv_{\beta \mathbf{R}} B}{\vdash N \Leftarrow A}$$

Figure: An implementation of the $\lambda\Pi$-calculus modulo

Implementation
Versatility

Versatile Type-Checking

The conversion test: how to normalize ?

- Proofs terms with few reductions (ex: from HOL).
  - Best technique: **interpretation** of the λ-terms.
- Proofs terms with a long reduction sequence (ex: proof by reflection (Coq)).
  - Best technique: **compilation** and execution of the λ-terms.

How to choose the correct strategy?

The JIT compromise

- **compile** the computational parts, **interpret** the rest
- **choice delegated** to a cutting edge JIT: luajit

File	Time to process
`fact.hs`	0.7 sec + 0.04 sec
`fact.lua`	0.7 sec
`fact.vo`	3.3 sec
`Coq_Init_Logic.hs`	45 sec + 0.4 sec
`Coq_Init_Logic.lua`	0.4 sec
`Coq_Init_Logic.vo`	0.14 sec

Figure: Compilation vs JIT vs Interpretation

`Coq_Init_Logic` is a module in Coq's prelude, `fact` typechecks the identity with the type $vec\ 8! \to vec\ 8!$.

Conclusion
Further work

Further work

Dedukti

non-linear pattern matching, reducing the size of the generated code

Embedding tools

PVS, AtelierB, Matita, Zenon, ...

Termination

A confluence/termination checker.

DEMO