DEDUKTI: A Universal Proof Checker

Mathieu Boespflug Quentin Carbonneaux Olivier Hermant
Ronan Saillard

MINES Paristech

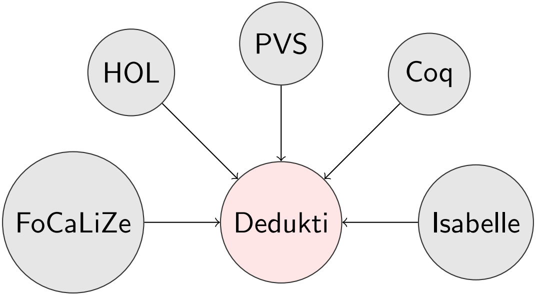
McGill University

INRIA

Journée LAC/LTP, Orléans

October 26, 2012

Dedukti as a universal backend



Dedukti: Core

- type theory: the $\lambda\Pi$-calculus (dependent types):

$$list \; : \; nat \to \mathrm{Type}$$

- enriched with rewrite rules (unlike CoC or CIC)

$$\begin{aligned} head\ (S\ n)\ (hd :: tl) &\longrightarrow hd \\ T_1 &\longrightarrow T_2 \end{aligned}$$

- used to weaken the conversion rule
- can encode all the Functional PTS [Cousineau & Dowek, 2007]

Typing rules

$$s \in \{\text{Type}, \text{Kind}\}$$

$$(sort)\ \frac{\Gamma\ \text{Well-Formed}}{\Gamma \vdash \text{Type} : \text{Kind}} \qquad (var)\ \frac{\Gamma\ \text{Well-Formed} \qquad x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

$$(prod)\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash \Pi x{:}A.\ B : s}$$

$$(abs)\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x{:}A \vdash B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}$$

$$(app)\ \frac{\Gamma \vdash M : \Pi x{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

$$(conv)\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A \equiv_{\beta\mathcal{R}} B$$

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

Dedukti: goals and weapons

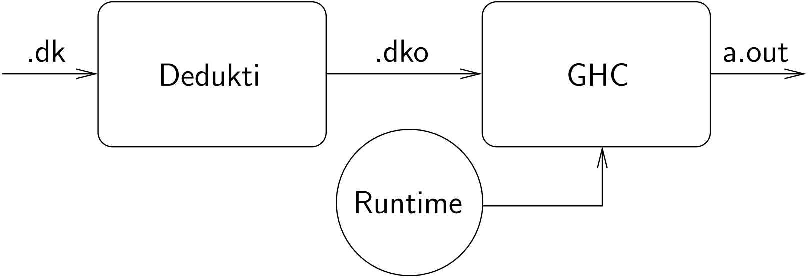
Goals:

- fast type checking
- versatility: check reasonabily fast any type of proofs

Philosophy, be lazy (**reuse** existing features):

- do not reimplement longstanding features
- existing efficient compilers
- reduce the trusted base

The big picture



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

Reuse and efficiency

Rely on the host language's features:

- substitutions: **HOAS** (Higher-Order Abstract Syntax)

Reuse and efficiency

Rely on the host language's features:

- substitutions: **HOAS** (Higher-Order Abstract Syntax)
- from $\Gamma \vdash t : T$ to $\vdash t : T$, **context-free** type checking: no search in contexts.
- **bidirectional** type checking: smaller terms (Curry-style)

$$(abs^b)\ \frac{C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash \{[\mathbf{y} : \mathbf{A}]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.\ M \Leftarrow C}$$

Reuse and efficiency

Rely on the host language's features:

- substitutions: **HOAS** (Higher-Order Abstract Syntax)
- from $\Gamma \vdash t : T$ to $\vdash t : T$, **context-free** type checking: no search in contexts.
- **bidirectional** type checking: smaller terms (Curry-style)

$$(abs^b)\ \frac{C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash \{[\mathbf{y} : \mathbf{A}]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.\ M \Leftarrow C}$$

- **conversion** test: **normalization by evaluation**
- **versatility**:
  - sometimes very few computation (typically, Isabelle/HOL)
  - sometimes many computation (typically, proofs by reflection)

The JIT compromise

- **compile** the computational parts, **interpret** the rest
- **delegate** the choice to a cutting edge JIT: luajit
- Lua is not statically typed, no global scoping analysis: no needless overhead

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 $vec\ 8! \rightarrow vec\ 8!$.

Dedukti: an Example

Three ingredients:

- a dependent type, `list` ($nat \to Type$), with constructors `Nil` ($list(0)$) and `Cons` ($\Pi n : nat.(nat \to list(n) \to list(n+1))$).

A 3 elements list

The list $[0;1;2]$ ($list(3)$) is written:

```
Cons 2 0 (Cons 1 1 (Cons 0 2 Nil))
```

- a naïve computation of lists' length by case analysis, the function `length` ($\Pi n : nat.(list(n) - > nat)$):

```
length 0 Nil = 0
length S(n) (Cons n e l) = 1 + (length n l)
```

- a theorem stating

Theorem

For all n and l of type $list(n)$, we have $length\ n\ l \quad = \quad n$

Implementation: typechecking *vs.* rewriting

Rules are:

- first **typechecked**
- second used to **rewrite** while typechecking other rules

Those are two separate concerns:

- need for a **static** representation
- need for a **dynamic** representation

$$(abs^b)\ \frac{C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash \{[y : A]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.\ M \Leftarrow C}$$

Also:

- **spine** representation of terms: better performance in rewriting
- **dot patterns**: avoid **non-linerar** left patterns: by experience necessary for (dependent) typechecking, not for computing.

Two interpretations

The static version of terms in HOAS ($\ulcorner . \urcorner$).

```
data Term =
      Lam (Term → Term)
    | App Term Term
    | B Term
```

$$\ulcorner x \urcorner = \mathsf{B}\ x$$
$$\ulcorner \lambda x.\ t \urcorner = \mathsf{Lam}\ (\lambda x. \ulcorner t \urcorner)$$
$$\ulcorner a\ b \urcorner = \mathsf{App}\ \ulcorner a \urcorner\ \ulcorner b \urcorner$$

Two interpretations

The static version of terms in HOAS ($\ulcorner . \urcorner$).

```
data Term =
    Lam (Term → Term)
  | App Term Term
  | B Term
```

With this meta-circular interpreter:

$$\ulcorner x \urcorner = \mathsf{B}\ x$$
$$\ulcorner \lambda x.\ t \urcorner = \mathsf{Lam}\ (\lambda x. \ulcorner t \urcorner)$$
$$\ulcorner a\ b \urcorner = \mathsf{App}\ \ulcorner a \urcorner\ \ulcorner b \urcorner$$

$$\mathsf{eval}\ (\mathsf{B}\ x) = x$$
$$\mathsf{eval}\ (\mathsf{Lam}\ f) = \lambda x. \mathsf{eval}\ (f\ x)$$
$$\mathsf{eval}\ (\mathsf{App}\ a\ b) = (\mathsf{eval}\ a)(\mathsf{eval}\ b)$$

How to peel the result of the evaluation?

Two interpretations

$$\begin{aligned} \text{eval' } (\text{B } x) &= x \\ \text{eval' } (\text{Lam } f) &= \text{L } (\lambda x.\text{eval' } (f\ x)) \\ \text{eval' } (\text{App } a\ b) &= \text{app } (\text{eval' } a)\ (\text{eval' } b) \end{aligned}$$

$$\begin{aligned} \text{app } (\text{L } f)\ x &= f\ x \\ \text{app } a\ b &= \text{A } a\ b \end{aligned}$$

Two interpretations

$$\text{eval'}\ (\text{B}\ x) = x$$
$$\text{eval'}\ (\text{Lam}\ f) = \text{L}\ (\lambda x.\text{eval'}\ (f\ x))$$
$$\text{eval'}\ (\text{App}\ a\ b) = \text{app}\ (\text{eval'}\ a)\ (\text{eval'}\ b)$$

$$\text{app}\ (\text{L}\ f)\ x = f\ x$$
$$\text{app}\ a\ b = \text{A}\ a\ b$$

The dynamic version of terms ($[\![.]\!]$).

$$[\![.]\!] = \text{eval'} \circ \ulcorner . \urcorner$$

$$[\![x]\!] = x$$
$$[\![\lambda x.\ t]\!] = \text{L}\ (\lambda x.[\![t]\!])$$
$$[\![a\ b]\!] = \text{app}\ [\![a]\!]\ [\![b]\!]$$

Two interpretations

$$\begin{aligned}\text{eval' } (\text{B } x) &= x\\ \text{eval' } (\text{Lam } f) &= \text{L } (\lambda x.\text{eval' } (f\ x))\\ \text{eval' } (\text{App } a\ b) &= \text{app } (\text{eval' } a)\ (\text{eval' } b)\end{aligned}$$

$$\begin{aligned}\text{app } (\text{L } f)\ x &= f\ x\\ \text{app } a\ b &= \text{A } a\ b\end{aligned}$$

$$\begin{aligned}\ulcorner x \urcorner &= \text{B } x\\ \ulcorner \lambda x.\ t \urcorner &= \text{Lam } (\lambda x.\ulcorner t \urcorner)\\ \ulcorner a\ b \urcorner &= \text{App } \ulcorner a \urcorner\ \ulcorner b \urcorner\end{aligned}$$

$$\begin{aligned}[\![x]\!] &= x\\ [\![\lambda x.\ t]\!] &= \text{L } (\lambda x.[\![t]\!])\\ [\![a\ b]\!] &= \text{app } [\![a]\!]\ [\![b]\!]\end{aligned}$$

Dedukti: ecosystem and contributors

At he heart of **Deducteam** - INRIA exploratory action:

- **Front-end**:
  - implementation in Haskell (M. Boespflug)
  - (much) faster implementation in C (Q. Carbonneaux)
  - experiment in OCaml (work in progress by R. Saillard)
- **Back-end**:
  - Haskell (M. Boespflug)
  - Lua (Q. Carbonneaux)

Together with **embedding tools**:

- **CoqInE** (M. Boespflug, G. Burel, Q. Carbonneaux)
- **HOLiDe** (A. Assaf)
- **Semantics of FoCaLiZe** in Dedukti (R. Cauderlier, C. Dubois)
- **PVS** (work in progress by A. Assaf)