DEDUKTI: A Universal Proof Checker

Mathieu Boespflug[1] Quentin Carbonneaux[2]
Olivier Hermant[3]

[1]McGill University

[2]INRIA and ENPC

[3]INRIA and ISEP

PxTP 2012

Contents

Introduction

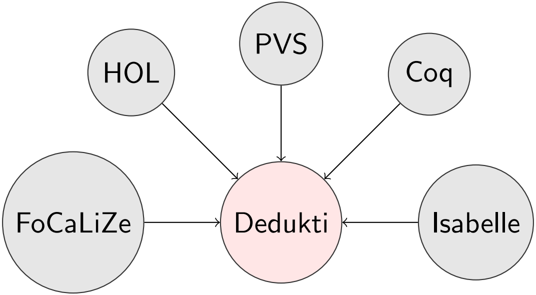
The $\lambda\Pi$-calculus modulo

The Dedukti proof checker

Better performance using JIT compilation

Conclusion

Dedukti as a universal backend



The $\lambda\Pi$-calculus at the core

A calculus with dependent types: *array* : *nat* $\rightarrow$ Type.

In a Curry-de Bruijn-Howard style, the $\lambda\Pi$-calculus is a language representing proofs of minimal predicate logic.

At least two choices to increase expressiveness:

1. enrich the $\lambda\Pi$-calculus by adding more deduction rules (*e.g.* CIC);
2. liberalize the conversion rule ($\lambda\Pi$-calculus modulo).

The $\lambda\Pi$-calculus modulo

$$\begin{aligned}\mathrm{Var} &\ni x, y, z \\ \mathrm{Term} &\ni t, A, B ::= x \mid \lambda x{:}A.\ M \mid \Pi x{:}A.\ B \mid M\ N \mid \mathsf{Type} \mid \mathsf{Kind}\end{aligned}$$

Figure: Grammar of the $\lambda\Pi$-calculus modulo

Typing rules: Abstractions

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

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

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

Typing rules: Dependent application

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

Typing rules: Conversion modulo

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

A Dedukti signature

$$\forall y, 0 + y = y$$
$$\forall x, \forall y, \mathsf{S}\ x + y = \mathsf{S}\ (x + y).$$

```
nat : Type.
Z : nat.
S : nat → nat.
plus : nat → nat → nat.

[y:nat]        plus Z     y ↪ y
[x:nat, y:nat] plus (S x) y ↪ S (plus x y).
```

A sample derivation

In the following context:

$$\begin{aligned}\Gamma := \;& nat : \mathsf{Type},\;\; vec : nat \to \mathsf{Type},\\ & cat : \Pi n{:}nat.\, \Pi m{:}nat.\, vec\;\, n \to vec\;\, m \to vec\;\, (n+m)\\ & n : nat,\;\; v : vec\;\, n.\end{aligned}$$

we have

$$(apps)\,\dfrac{\Gamma \vdash cat : \ldots \qquad \Gamma \vdash n : nat \qquad \Gamma \vdash v : vec\;\, n}{(conv)\,\dfrac{\Gamma \vdash cat\;\, n\;\, n\;\, v\;\, v : vec\;\, (n+n)}{\Gamma \vdash cat\;\, n\;\, n\;\, v\;\, v : vec\;\, (2*n)}}$$

Dedukti's goals

- Fast type checking of an extensible λ-calculus.
- Use compilation techniques.
  - Plenty of efficient compilers available;
  - reuse them off the shelf (separate concerns).
- Lightest possible runtime system.

Two choices are possible:

Dedukti's goals

- Fast type checking of an extensible λ-calculus.
- Use compilation techniques.
  - Plenty of efficient compilers available;
  - reuse them off the shelf (separate concerns).
- Lightest possible runtime system.

Two choices are possible:

- generate a specific type checker for each theory (LFSC);

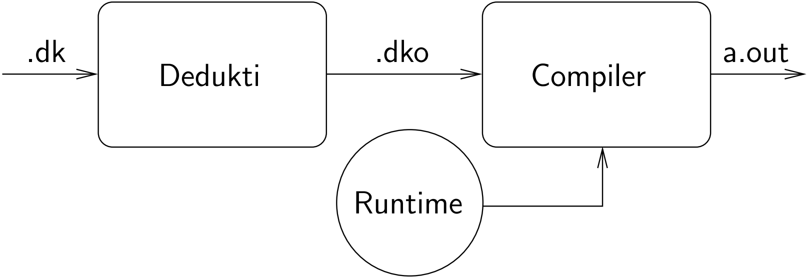
Dedukti's goals

- Fast type checking of an extensible λ-calculus.
- Use compilation techniques.
  - Plenty of efficient compilers available;
  - reuse them off the shelf (separate concerns).
- Lightest possible runtime system.

Two choices are possible:

- generate a specific type checker for each theory (LFSC);
- generate a specific type checker for a set of terms (Dedukti).

The big picture



Two interpretations

We fully embed the type checking logic in the target language.

Generated data/code must fit two purposes:

1. Type checking (*static* representation).
2. Normalizing (*dynamic* representation).

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 interpreter:

$$\ulcorner x \urcorner = \mathsf{B}\ x \qquad\qquad \mathsf{eval}\ (\mathsf{B}\ x) = x$$

$$\ulcorner \lambda x.\ t \urcorner = \mathsf{Lam}\ (\lambda x. \ulcorner t \urcorner) \qquad\qquad \mathsf{eval}\ (\mathsf{Lam}\ f) = \lambda x. \mathsf{eval}\ (f\ x)$$

$$\ulcorner a\ b \urcorner = \mathsf{App}\ \ulcorner a \urcorner\ \ulcorner b \urcorner \qquad\qquad \mathsf{eval}\ (\mathsf{App}\ a\ b) = (\mathsf{eval}\ a)(\mathsf{eval}\ b)$$

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

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

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

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

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

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

Two interpretations

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

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

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

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

Context free type checking

de Bruijn's criterion

We must have the simplest possible runtime.

As a solution, we rely on the host language's features.

Judgements become closures: we move from $\Gamma \vdash t{:}T$ to $\vdash t{:}T$; substitutions are performed using HOAS.

$$\mathrm{Term} \ni t, A, B ::= x \mid [y : T] \mid \lambda x.\ M \mid \Pi x{:}A.\ B \mid M\ N \mid \mathsf{Type} \mid \mathsf{Kind}$$

Context free type checking

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

Context free type checking

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

Which maps trivially to this Haskell snippet:

```
check n (Lam f) (Pi a t) =
    check (n + 1) (f box) (t var)
  where box = Box n a
        var = Var n
```

Dedukti on a simple example

Module	Dedukti	Compilation and execution
`Coq.Init.Logic`	50 sec	1 min 13 sec + 0.261 sec

Dedukti on a simple example

Module	Dedukti	Compilation and execution
`Coq.Init.Logic`	50 sec	1 min 13 sec + 0.261 sec

Module	Chicken	
`Coq.Init.Logic`	0.170 sec	

A complete rewrite

DEDUKTI was freshly (6 weeks ago) rewritten in C.

Simple observation: the translator is a syntactic map.

This allows a new design:

A complete rewrite

DEDUKTI was freshly (6 weeks ago) rewritten in C.

Simple observation: the translator is a syntactic map.

This allows a new design:

1. the translator can be an online program (work in a stream friendly way);

A complete rewrite

Dedukti was freshly (6 weeks ago) rewritten in C.

Simple observation: the translator is a syntactic map.

This allows a new design:

1. the translator can be an online program (work in a stream friendly way);
2. the internal state of the translator is tiny, hence no garbage collection is needed.

A complete rewrite

Dedukti now switches from Haskell to Lua.

- Lua is a minimal programming language.
- Lua enjoys a very fast cutting edge JIT (luajit).
- Lua is not statically typed, not scoped.

A huge performance gap

File	Dedukti before	Dedukti after
`bool_steps.dk`	$>$ 5 min	6 sec
`Coq_Init_Logic.dk`	50 sec	0.08 sec

Figure: Speed of the first translation

Because memory management is handmade, several gigabytes are saved during the processing of big files.

The JIT compromise

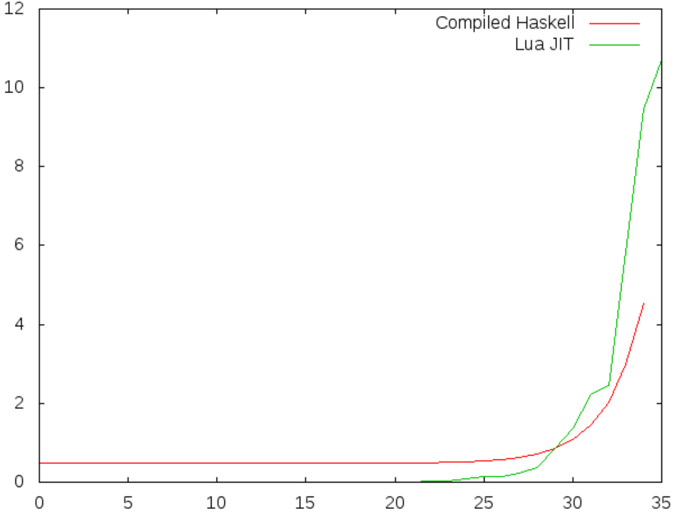


Figure: Compilation vs JIT

Conclusion

- Dedukti is
  - 1285 lines of C (+ 451 lines of comments);
  - blazingly fast on resonably sized examples;
  - not worse than a trivial implementation;
  - generating Lua code.
- Using a JIT allows a a smoother behavior of type checking times.
- Next steps: improve our control on generated code, cope with luajit's limits.