Yet Another Complete Rewrite of Dedukti

Ronan Saillard

Deducteam INRIA

MINES ParisTech

May 26, 2014
Table of Contents

Introduction

Yet Another Complete Rewrite of Dedukti
  From Lua to OCaml
  Reduction Algorithm
  Benchmarks

Dot Patterns

Future Work
Dedukti is a type-checker for the $\lambda\Pi$-calculus modulo.

The $\lambda\Pi$-calculus modulo is an extension of the $\lambda$-calculus with dependent types ($\lambda\Pi$-calculus) with rewrite rules.
**Reminder: Dedukti’s (past) architecture**

- **Frontend**
  - parsing/
  - code generation
  - (OCaml)

- **Backend**
  - (the Lua compiler)

- **Runtime**
  - dedukti.lua

- **Example Files**
  - example.dk
  - example.lua

- **Status**
  - OK
  - KO

- **Dedukti** is a type-checker **generator**.
Yet Another Dedukti
From Lua to OCaml
Why a New Version?

Dedukti in OCaml/Lua

- Does **not scale well**: Lua is quickly unable to interpret big generated files.
- It has roughly the same **performance** than Camelide (but with much more implementation effort).
- Error reporting is problematic.

AND ALSO

- **Lua** more suited to developing small scripts than complex algorithms (imperative, untyped, only array as data structure).
- No performance **comparison** of Dedukti and its two-phases architecture with a more standard approach.
Comparison of Versions

Old Dedukti

- Two-steps architecture.
- (Context-free) Normalization by Evaluation (NbE).
- 950 lines of OCaml and 380 lines of Lua.

New Dedukti

- More standard approach.
- Reduction Machine inspired by Matita’s (*).
- ~1000 lines of OCaml.
- (No more code generation).
- (No more NbE).

Yet Another Dedukti Reduction Algorithm
The Reduction Machine (1)

type cbn_state = int (*size of context*)
   *(term Lazy.t) list (*context*)
   * term (*term to reduce*)
   * cbn_state list (*stack*)

(* Head Normal Form Reduction *)

let rec cbn_reduce (config:cbn_state) : cbn_state =
   match config with
   | (k, e, DB n, s) when n<k ->
     cbn_reduce (0, [], Lazy.force(List.nth e n), s)
   | (k, e, App(he::tl), s) ->
     let tl' = List.map(fun t -> (k,e,t,[])) tl in
     cbn_reduce (k, e, he, tl' @ s)
   | (k, e, Lam(_,t), p::s) ->
     cbn_reduce (k+1, (lazy (cbn_term_of_state p)::e, t, s))
   | (_, _, Const(m,v), s) ->
     let (s1, s2) = split_args(m,v) s in
     (match rewrite(get_gdt(m,v)) s1 with
     | None -> config
     | Some(k',e',t) -> cbn_reduce(k',e',t,s2))
   | (_, _, _, _, _) -> config
and rewrite (args:cbn_state array) (g:gdt) =
match g with

  | Leaf right    ->
    Some ( Array.length args ,
           List.map (fun a -> lazy (cbn_term_of_state a)) (Array.to_list args) ,
           right )

  | Switch (i, cases, def_opt) ->
    (match cbn_reduce (args.(i)) with
     | ( , , , Const (m,v), s ) ->
       (match safe_find m v cases , def_opt with
        | Some tr , _   -> rewrite (mk_new_args i args s) tr
        | None , Some def -> rewrite args def
        | _ , _ , _ , _ , s ) ->
        (match def_opt with
         | Some def  -> rewrite args def
         | None     -> None
        )
     )

  | ( , , , , s ) ->
    (match def_opt with
     | Some def  -> rewrite args def
     | None     -> None
    )

)
Yet Another Dedukti
Benchmarks
**Benchmarks: Overview**

**Encoding of OpenTheory generated by Holide**

- **$\lambda\Pi$-calculus**: comparison between Coq, Twelf, Camelide, Dedukti (OCaml/Lua) and Dedukti (OCaml).
- **$\lambda\Pi$-calculus modulo**: comparison between Camelide, Dedukti (OCaml/Lua) and Dedukti (OCaml).

**Church Integers**

*$\lambda\Pi$-calculus*: Conversion between complex expressions involving addition and multiplication on Church integers.

**Arithmetic Using Rewrite Rules**

*$\lambda\Pi$-calculus modulo*: Computation of arithmetic expressions (defined as rewrite rules) on unary integers. Comparison with the Maude System.
# Benchmarks: λΠ-calculus

## OpenTheory

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>new DK</th>
<th>old DK</th>
<th>old DK(*)</th>
<th>Camelide</th>
<th>Coq</th>
<th>Twelf</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiom-infinity.dk</td>
<td>0.7M</td>
<td>&lt; 1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>&lt; 1</td>
<td>1</td>
</tr>
<tr>
<td>natural-...-def.dk</td>
<td>4.7M</td>
<td>1</td>
<td>12</td>
<td>7</td>
<td>4</td>
<td>&lt; 1</td>
<td>3</td>
</tr>
<tr>
<td>list-filter-thm.dk</td>
<td>8.5M</td>
<td>3</td>
<td>24</td>
<td>13</td>
<td>13</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>pair-thm.dk</td>
<td>11M</td>
<td>3</td>
<td>36</td>
<td>FAIL</td>
<td>22</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>relation-...-thm.dk</td>
<td>22M</td>
<td>7</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>natural-exp-thm.dk</td>
<td>55M</td>
<td>10</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td>list-def.dk</td>
<td>84M</td>
<td>19</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>21</td>
<td>46</td>
</tr>
<tr>
<td>set-thm.dk</td>
<td>97M</td>
<td>28</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>&gt; 60</td>
<td>52</td>
</tr>
<tr>
<td>relation-...-thm.dk</td>
<td>122M</td>
<td>45</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>40</td>
<td>&gt; 60</td>
</tr>
<tr>
<td>real-def.dk</td>
<td>259M</td>
<td>50</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>&gt; 60</td>
<td>&gt; 60</td>
</tr>
<tr>
<td>All files (88)</td>
<td>1.4G</td>
<td>6mn35</td>
<td>&gt; 45mn</td>
<td>FAIL</td>
<td>&gt; 45mn</td>
<td>7mn50</td>
<td>8mn43</td>
</tr>
</tbody>
</table>

## Church Integers

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>new DK</th>
<th>old DK</th>
<th>old DK(*)</th>
<th>Camelide</th>
<th>Coq</th>
<th>Twelf</th>
</tr>
</thead>
<tbody>
<tr>
<td>church16.dk</td>
<td>2K</td>
<td>1</td>
<td>FAIL</td>
<td>FAIL</td>
<td>&lt; 1</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>church20.dk</td>
<td>2K</td>
<td>23</td>
<td>FAIL</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

(* ) = with LuaJIT
**Benchmarks: \(\lambda\Pi\)-calculus modulo (1)**

---

### OpenTheory

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>new DK</th>
<th>old DK</th>
<th>old DK (*)</th>
<th>Camelide</th>
<th>new DK ((\lambda\Pi))</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiom-infinity.dk</td>
<td>0.7M</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>1</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>natural-order-def.dk</td>
<td>4.7M</td>
<td>&lt; 1</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>list-filter-thm.dk</td>
<td>8.5M</td>
<td>&lt; 1</td>
<td>23</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>pair-thm.dk</td>
<td>11M</td>
<td>&lt; 1</td>
<td>25</td>
<td>11</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>relation-wellfounded-thm.dk</td>
<td>22M</td>
<td>&lt; 1</td>
<td>&gt; 60</td>
<td>35</td>
<td>20</td>
<td>7</td>
</tr>
<tr>
<td>natural-exp-thm.dk</td>
<td>55M</td>
<td>1</td>
<td>22</td>
<td>8</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>list-def.dk</td>
<td>84M</td>
<td>1</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>19</td>
</tr>
<tr>
<td>set-thm.dk</td>
<td>97M</td>
<td>2</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>28</td>
</tr>
<tr>
<td>relation-natural-thm.dk</td>
<td>122M</td>
<td>2</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>45</td>
</tr>
<tr>
<td>real-def.dk</td>
<td>259M</td>
<td>3</td>
<td>&gt; 60</td>
<td>FAIL</td>
<td>&gt; 60</td>
<td>50</td>
</tr>
<tr>
<td><strong>All files (88)</strong></td>
<td><strong>1.4G</strong></td>
<td><strong>17</strong></td>
<td><strong>&gt; 45mn</strong></td>
<td><strong>FAIL</strong></td>
<td><strong>15mn</strong></td>
<td><strong>6mn35</strong></td>
</tr>
</tbody>
</table>

(*) = with LuaJIT
### Arithmetic with Rewrite Rules

<table>
<thead>
<tr>
<th>Expression</th>
<th>new DK</th>
<th>old DK</th>
<th>Maude</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>31</td>
<td>FAIL</td>
<td>6</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>267 (4mn27)</td>
<td>FAIL</td>
<td>45</td>
</tr>
<tr>
<td>$3^6$</td>
<td>5</td>
<td>FAIL</td>
<td>1</td>
</tr>
<tr>
<td>$3^7$</td>
<td>174 (2mn54)</td>
<td>FAIL</td>
<td>28</td>
</tr>
<tr>
<td>$5 \times 4^5$</td>
<td>56</td>
<td>FAIL</td>
<td>53</td>
</tr>
<tr>
<td>$(10 \times 10) \times (10 \times 10)$</td>
<td>120 (2mn)</td>
<td>FAIL</td>
<td>218 (3mn38)</td>
</tr>
<tr>
<td>$10 \times (10 \times (10 \times 10))$</td>
<td>4</td>
<td>FAIL</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>FAIL</td>
<td>16</td>
</tr>
</tbody>
</table>
New Dedukti

The new version of Dedukti is:

- Simpler.
- Smaller.
- Faster.
- More user-friendly (Error messages).

The new version of Dedukti will be:

- Easier to maintain.
- Easier to improve/extend.

Thanks to Raphaël Cauderlier, Dedukti now has:

- A nice tutorial.
- An Emacs mode.
Dot Patterns
Dot Patterns were introduced in Agda to deal with non-linear patterns arising from the use of dependent types. This technique was also used in previous versions of Dedukti. We don’t need them anymore since non-linear pattern matching is now implemented. In fact they are unsound in Dedukti!
Example

(;; Lists parametrized by their size ;;)
Listn : Nat -> Type.
nil : Listn zero.

(;; Concatenation of lists ;;)
append : n:Nat -> Listn n -> m:Nat -> Listn m -> Listn (plus n m).
[n:Nat,l2:Listn n] append zero nil n 12 --> 12
[n:Nat,l1:Listn n,m:Nat,l2:Listn m,a:A]
append (succ n) (cons n a l1) m 12 --> cons (plus n m) a (append n l1 m 12).

(;; This is non-linear in n! ;;)

(;; Second solution ;;)
append2 : n:Nat -> Listn n -> m:Nat -> Listn m -> Listn (plus n m).
[n:Nat,l1:Listn n,m:Nat,l2:Listn m,a:A]
append2 {succ n} (cons n a l1) m 12 --> cons (plus n m) a (append2 n l1 m 12).

(;; Dedukti checks that the term between brackets can be reconstruct from typing.
Here it is necessarily (succ n).
And the rewrite rule added is the linear one ;;)
PROBLEM

\[\begin{align*}
\text{List: Type. } & \quad X: \text{Nat. } \quad N: \text{Nat. } \quad M: \text{Nat.} \\
[ ] & \quad \text{Listn } X \rightarrow \text{List.} \\
[ ] & \quad \text{Listn } (\text{succ } N) \rightarrow \text{List.}
\end{align*}\]

\[
; \quad \text{Listn } X \simeq \text{Listn } (\text{succ } N)
\]

Thus

\[
\begin{align*}
\text{append2 } X & \quad (\text{cons } N \ a \ l1) \ M \ l2 : \text{Listn } (\text{plus } X \ M) \\
& \rightarrow \\
\text{cons } (\text{plus } N \ M) & \quad a \ (\text{append2 } N \ l1 \ M \ l2) : \text{Listn } (\text{succ } (\text{plus } N \ M))
\end{align*}
\]

but \text{Listn } (\text{plus } X \ M) \not\simeq \text{Listn } (\text{succ } (\text{plus } N \ M))

;
Where is the bug?

**Unification Algorithm**

\[ Listn \ k \equiv Listn \ (\text{succ} \ n) \quad \Rightarrow \quad k \equiv (\text{succ} \ n). \]

**But**

We cannot assume that Listn is injective since one can later rewrite it.
But without this rule unification becomes useless.

**Conclusion**

- We cannot use dot patterns.
- We need (and have) non-linear patterns.
**Example**

```plaintext
type : srt -> Type.
term : s : srt -> A : type s -> Type.
sort : s : srt -> type (t s).

Ty: srt.
Ki: srt.
[ ] t Ty --> Ki.

[s : srt] term {t s} (sort s) --> type s.

(;
   Without brackets this pattern won't match anything
   because (t s) is not normal for a given s.
   Ex: term (t Ty) (sort Ty) --> term Ki (sort Ty) -/> type Ty.
;)
```
Conditional Rewriting

Solution
Let us change the meaning of ‘{ }’!
Now
\[ [s : srt] \text{term } \{ t \ s \} \ (\text{sort } s) \hookrightarrow \text{type } s. \]
stands for
\[ [s : srt, k : \text{sort}] \text{term } k \ (\text{sort } s) \hookrightarrow \text{type } s \text{ when } k \equiv t \ s. \]
We call this feature conditional rewriting.
**Summary**

**Dot Patterns** are now replaced by

- **Non-linear** patterns.
- **Conditional** patterns.

Of course these features have more applications than just replacing dot patterns.
Future Work
What’s next
What’s Next?

Future Work:

- Non-linear pattern matching (Done).
- Conditional pattern matching (Experimental).
- Pattern ‘à la Miller’.
- Confluence checking.
- Termination checking?
- Twelf-Style type reconstruction?
Thanks for your Attention!

Any Questions?
Yet Another Complete Rewrite of Dedukti

Ronan Saillard

Deducteam INRIA

MINES ParisTech

May 26, 2014
### \( \lambda \Pi \)-calculus modulo

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Empty)</td>
<td>( \emptyset _wf )</td>
<td>( \emptyset \wf )</td>
</tr>
<tr>
<td>(Dec)</td>
<td>( \Gamma \wf \ \Gamma \vdash A : s \ \ x \notin \Gamma )</td>
<td>( \Gamma(x : A) \wf )</td>
</tr>
<tr>
<td>(Rewrite)</td>
<td>( \Gamma \wf \ \Gamma \Delta \vdash l : T \ \ \Gamma \Delta \vdash r : T )</td>
<td>( \Gamma(\Delta / l \leftrightarrow r) \wf )</td>
</tr>
<tr>
<td>(Type)</td>
<td>( \Gamma \wf )</td>
<td>( \Gamma \vdash Type : \text{Kind} )</td>
</tr>
<tr>
<td>(Var)</td>
<td>( \Gamma \wf \ \ (x : A) \in \Gamma )</td>
<td>( \Gamma \vdash x : A )</td>
</tr>
<tr>
<td>(App)</td>
<td>( \Gamma \vdash t : \Pi x^A.B \ \ \Gamma \vdash u : A )</td>
<td>( \Gamma \vdash tu : B[x/u] )</td>
</tr>
<tr>
<td>(Conv)</td>
<td>( \Gamma \vdash B : s \neq \text{Kind} )</td>
<td>( \Gamma \vdash t : B )</td>
</tr>
<tr>
<td>(Abs)</td>
<td>( \Gamma \vdash A : \text{Type} )</td>
<td>( \Gamma(x : A) \vdash t : B )</td>
</tr>
<tr>
<td>(Prod)</td>
<td>( \Gamma \vdash A : \text{Type} )</td>
<td>( \Gamma(x : A) \vdash B : s )</td>
</tr>
</tbody>
</table>

### Type System

- **Type Formation**: \( \Gamma \vdash Type : \text{Kind} \)
- **Variable Introduction**: \( \Gamma \wf \ \ (x : A) \in \Gamma \) \( \Gamma \vdash x : A \)
- **Application**: \( \Gamma \vdash t : \Pi x^A.B \ \ \Gamma \vdash u : A \) \( \Gamma \vdash tu : B[x/u] \)
- **Conversion**: \( \Gamma \vdash B : s \neq \text{Kind} \) \( \Gamma \vdash t : B \)
- **Abstraction**: \( \Gamma \vdash A : \text{Type} \) \( \Gamma(x : A) \vdash t : B \)
- **Product Formation**: \( \Gamma \vdash A : \text{Type} \) \( \Gamma(x : A) \vdash B : s \) \( \Gamma \vdash \Pi x^A.B : s \)