Yet Another Complete Rewrite of Dedukti

Ronan Saillard

Deducteam INRIA

MINES ParisTech

May 26, 2014

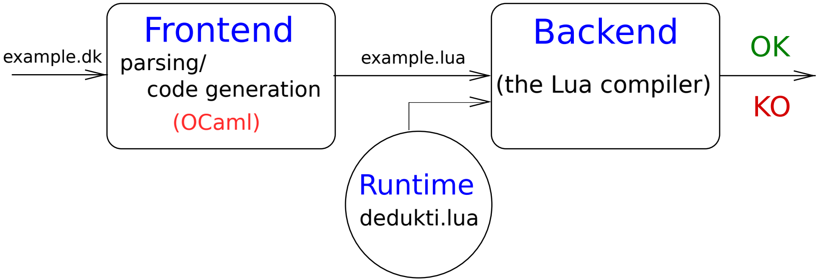
Table of Contents

Remainder: Dedukti

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

The **$\lambda\Pi$-calculus modulo** is an extension of the λ-calculus with **dependent types** ($\lambda\Pi$-calculus) with **rewrite rules**.

Reminder: Dedukti's (past) architecture



- 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).

(*) Asperti, Ricciotti, Sacerdoti Coen and Tassi, **A Compact Kernel for the Calculus of Inductive Constructions**. In Sadhana, 2009.

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
```

The Reduction Machine (2)

```
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
                | _ , _                -> 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

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

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

Church Integers

File	Size	new DK	old DK	old DK(*)	Camelide	Coq	Twelf
church16.dk	2K	1	FAIL	FAIL	< 1	9	2
church20.dk	2K	**23**	FAIL	FAIL	FAIL	> 60	26

(*) = with LuaJIT

Benchmarks: λΠ-calculus modulo (1)

OpenTheory

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

(*) = with LuaJIT

Benchmarks: λΠ-calculus modulo (2)

Arithmetic with Rewrite Rules

Expression	new DK	old DK	Maude
2^{10}	31	FAIL	**6**
2^{11}	267 (4mn27)	FAIL	**45**
3^6	5	FAIL	**1**
3^7	174 (2mn54)	FAIL	**28**
$5 * 4^5$	56	FAIL	**53**
$10 * 4^5$	**120 (2mn)**	FAIL	218 (3mn38)
$(10 * 10) * (10 * 10)$	**4**	FAIL	54
$10 * (10 * (10 * 10))$	**1**	FAIL	16

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

- **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.
cons  : n:Nat -> A -> Listn n -> Listn (succ n).

(; 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 l2 --> l2
[n:Nat,l1:Listn n,m:Nat,l2:Listn m,a:A]
 append (succ n) (cons n a l1) m l2 --> cons (plus n m) a (append n l1 m l2).
(; This is non-linear in n! ;)

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

(; 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

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

List:Type.      X:Nat.  N:Nat.  M:Nat.

[ ] Listn X --> List.
[ ] Listn (succ N) --> List.

(;      Listn X ~= Listn (succ N)

        Thus

        append2 X (cons N a l1) M l2 : Listn (plus X M)
        -->
        cons (plus N M) a (append2 N l1 M l2) : Listn (succ (plus N M))

        but Listn (plus X M) !~ Listn (succ (plus N M))
;)
```

Where is the bug?

Unification Algorithm

$Listn\ k \equiv Listn\ (succ\ n) \implies k \equiv (succ\ n)$.

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

```
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

$[s : srt]\ term\ \{\mathbf{t}\ \mathbf{s}\}\ (sort\ s) \hookrightarrow type\ s.$

stands for

$[s : srt, k : sort]\ term\ \mathbf{k}\ (sort\ s) \hookrightarrow type\ s$ **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

λΠ-calculus modulo

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

$$\text{(Rewrite)}\ \frac{\Gamma\ \mathbf{wf} \qquad \Gamma\Delta \vdash l : T \qquad \Gamma\Delta \vdash r : T}{\Gamma([\Delta]l \hookrightarrow r)\ \mathbf{wf}}$$

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

$$\text{(App)}\ \frac{\Gamma \vdash t : \Pi x^A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B[x/u]} \qquad \text{(Conv)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}$$

$$\text{(Abs)}\ \frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma(x : A) \vdash t : B \qquad B \neq \mathbf{Kind}}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}$$

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