Focalizing on Dedukti

CNAM, Laboratoire Cedric
Inria, Deducteam
March 25, 2015

Critical Software

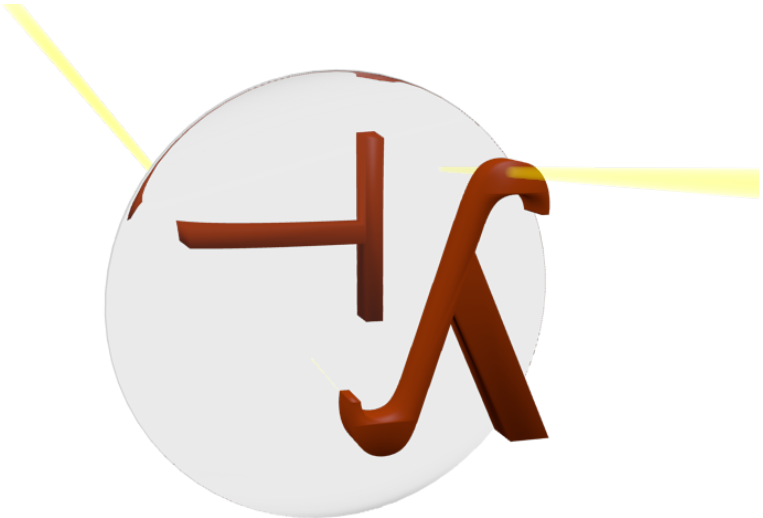
- Software is everywhere
- Lives depend on software
- Formal methods to the rescue!

Interoperability

- Proof development is expensive
- Proof systems are specialized
- They implement different logics
- We need a common formalism: Dedukti

Logic vs. Object-Oriented Programming

- Proof development is programming
- OOP is very popular
- FoCaLiZe



$\lambda\Pi$-calcul modulo

- Dedukti is a type-checker for the $\lambda\Pi$-calculus modulo
- $\lambda\Pi$-calculus = λ-calculus + dependent types
- $\lambda\Pi$-calculus modulo = $\lambda\Pi$-calculus + rewriting
- $\lambda\Pi$-calculus modulo can express FPTS and CIC

Rewriting

```
Nat : Type.
0 : Nat.
S : Nat -> Nat.

plus : Nat -> Nat -> Nat.
[m : Nat] plus m 0 --> m
[n : Nat] plus 0 n --> n
[m : Nat, n : Nat] plus (S m) n --> S (plus m n)
[m : Nat, n : Nat] plus m (S n) --> S (plus m n).
```

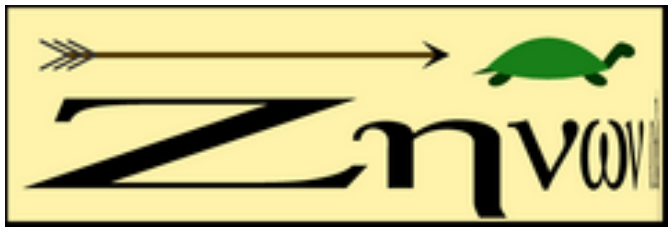
Dependent types

```
Elt : Type.
Vector : Nat -> Type.
nil : Vector 0.
cons : n : Nat ->
       Elt -> Vector n ->
       Vector (S n).

append : m : Nat -> n : Nat ->
         Vector m -> Vector n ->
         Vector (plus m n).
[ m : Nat, v : Vector m ] append m 0 v nil --> v
[ n : Nat, w : Vector n ] append 0 n nil w --> w
[ m : Nat, a : Elt, v : Vector m,
  n : Nat, w : Vector n ]
    append (S m) n (cons m a v) w
      -->
    cons (plus m n) a (append m n v w).
```

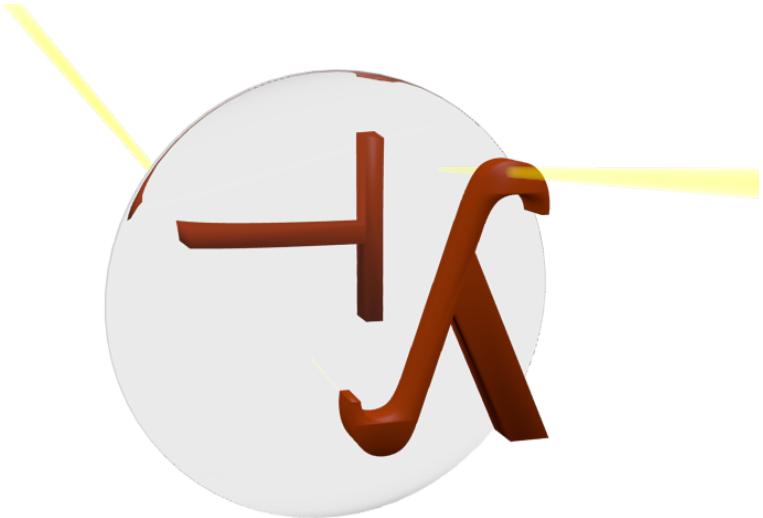
Zenon

- Automatic prover for first-order logic
- Tableaux method
- Used by FoCaLiZe
- Extensions
- Coq Input/Output



FoCaLiZe

- Framework for developing certified programs
- Two outputs : OCaml and Coq
- Computational language = ML
- Specification langage = FOL
- Proofs found by Zenon and checked by Coq
- OO mecanisms statically resolved



An abstract species

```
species Setoid =
  signature eq : Self -> Self -> bool;
  property eq_refl :
    all x : Self, eq(x, x);
  property eq_sym :
    all x y : Self, eq(x, y) -> eq(y, x);
  property eq_trans :
    all x y z : Self,
      eq(x, y) -> eq(y, z) -> eq(x, z);
end;;
```

Inheritance

```
species Order =
  inherit Setoid;

  signature leq : Self -> Self -> bool;
  let eq(x, y) = leq(x, y) && leq(y, x);

  property leq_refl : all x : Self, leq(x, x);
  property leq_trans : all x y z : Self, leq(x, y) -> leq(y, z) -> leq(x, z);

  theorem leq_antisym : all x y : Self, leq(x, y) -> leq(y, x) -> eq(x, y)
    proof = by definition of eq;
  theorem eq_implies_leq : all x y : Self, eq(x, y) -> leq(x, y)
    proof = by definition of eq;

  proof of eq_refl = by property leq_antisym, leq_refl;
  proof of eq_sym = by definition of eq;
  proof of eq_trans = by property eq_implies_leq, eq_sym, leq_trans, leq_antisym;
end;;
```

A complete species

```
species BoolOrder_slow =
  inherit Order;
  representation = bool;
  let leq(x, y) = if x then y else true;
  proof of leq_refl = by definition of leq;
  proof of leq_trans = by definition of leq;
end;;

collection BoolOrderCollection_slow =
  implement BoolOrder_slow;
end;;
```

Redefinition

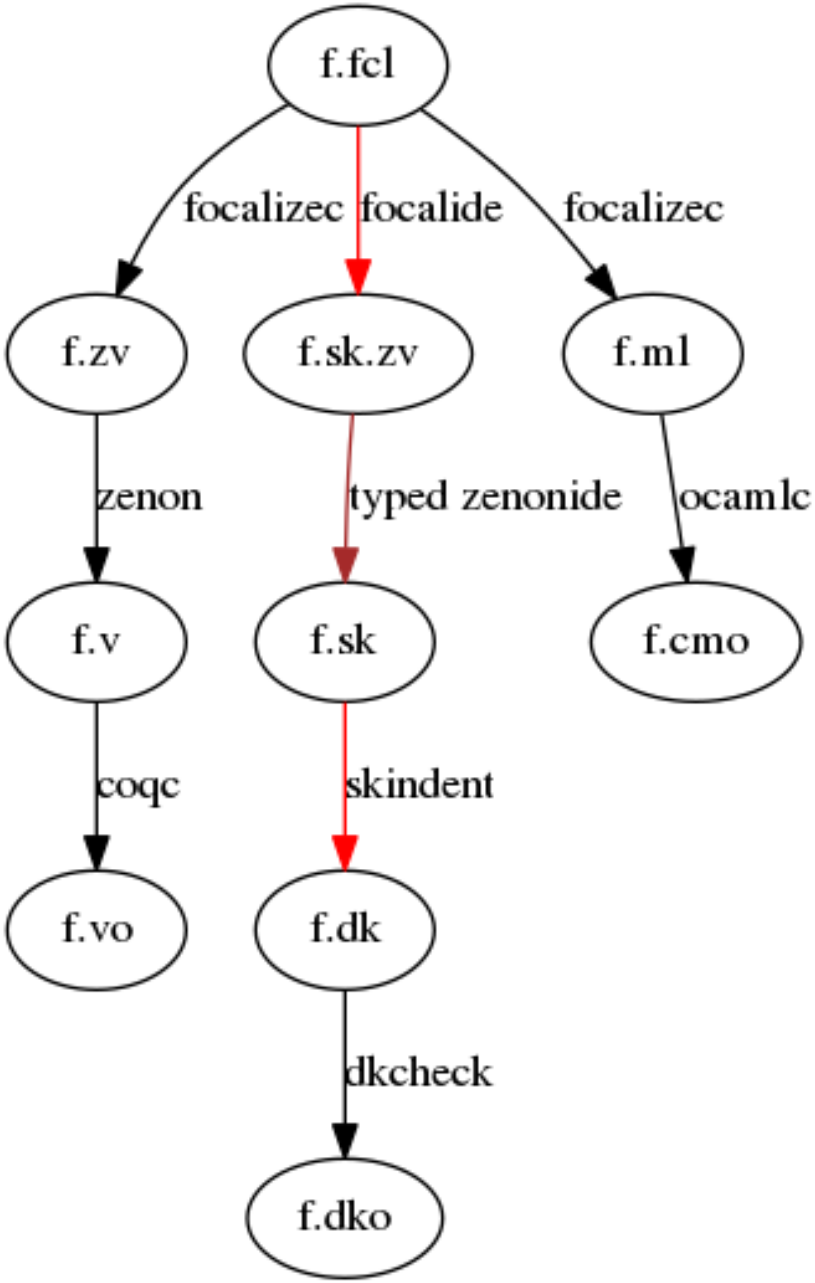
```
species BoolOrder_fast =
  inherit BoolOrder_slow;
  let eq(x, y) = if x then y else ~~(y);
  proof of leq_antisym =
    by definition of leq, eq;
  proof of eq_sym =
    by definition of eq;
  proof of eq_implies_leq =
    by definition of leq, eq;
end;;

collection BoolOrderCollection_fast =
  implement BoolOrder_fast;
end;;
```

Focalide  

- A Dedukti output for FoCaLiZe compiler
- Adapted from Coq output

FoCaLiZe Compilation scheme



Translation of the logical part

- Zenonide (Gilbert) : Dedukti output for Zenon
- Dedukti input for Zenon
- Typing in Zenon (Bury, Cauderlier, Halmagrand)
- Compatibility of extensions with Dedukti output and typing

Translation of the computational part

- Primitive FoCaLiZe types (`bool`, `int`, `string` and `list`): Dklib (1700 lines)
- Pattern-matching: compilation to destructors

Naïve translation of pattern matching

```
let f (x) =                  |  f : int -> int.
  match x with               |
  | 0 -> 0                   |  []                  f 0 --> 0
  | n -> n-1;                |  [ n : int ] f n --> - n 1.
```

Naïve translation of pattern matching

```
let f (x) =                 |  f : int -> int.
  match x with              |
  | 0 -> 0                  |  []              f 0 --> 0
  | n -> n-1;               |  [ n : int ] f n --> - n 1.
```

- Not confluent!

  `f 0` can reduce to both `0` and `-1`.

Naïve translation of pattern matching

```
let f (x) =                 |  f : int -> int.
  match x with              |
  | 0 -> 0                  |  []              f 0 --> 0
  | n -> n-1;               |  [ n : int ] f n --> - n 1.
```

- Not confluent!

  `f 0` can reduce to both `0` and `-1`.

- Not consistent!

  We can prove `0 = -1`:

```
= : int -> int -> Type.
refl : n : int -> = n n.

lemma (n : int) : = (f n) (- n 1)
  := refl (f n).
thm : = 0 -1 := lemma 0.
```

Using destructors

```
let f (x) =
  match x with
  | 0 -> 0
  | n -> n-1;
```

```
f : int -> int.

[] f x
    -->
   if_is_0 x
      0
      ((n : int => - n 1) x).
```

```
let f (x) =
  match x with
  | 0 -> 0
  | S(n) -> n;
```

```
f : nat -> nat.

[] f x
    -->
   if_is_0 x
      0
      (if_is_S x
         (n : nat => n)
         fail).
```

Recursive functions (Not yet implemented)

```
signature fact : int -> int -> int;

let rec fact (n) =
  if leq(n, 1) then
    1
  else
    n * fact (n-1);
```

- How to preserve termination in Dedukti?

Recursive function in standard library

```
let eq (x, y) =
  if is_empty(x) then
    is_empty(y)
  else
    if is_empty(y) then
      false
    else
      eq_element(head(x), head(y)) &&
        eq(tail(x), tail(y))
```

Recursive function in standard library

```
let eq (x, y) =
  if is_empty(x) then
    is_empty(y)
  else
    if is_empty(y) then
      false
    else
      eq_element(head(x), head(y)) &&
        eq(tail(x), tail(y))
```

- not terminating under available assumptions on `tail`

Recursive function in standard library

```
let eq (x, y) =
  if is_empty(x) then
    is_empty(y)
  else
    if is_empty(y) then
      false
    else
      eq_element(head(x), head(y)) &&
        eq(tail(x), tail(y))
```

- not terminating under available assumptions on `tail`
- termination assumed in Coq backend

FoCaLiZe compilation passes

- Lexing
- Parsing
- Scoping
- Typing
- Compilation of OO mechanisms
- Code generation

Lifting

```
species EqDiff =
  signature eq : Self -> Self -> bool;
  property eq_refl: all x : Self, eq(x, x);

  let diff(x, y) = ~~eq(x, y);
  theorem diff_irrefl: all x : Self, ~diff(x, x)
    proof = ...;
end;;
```

- method `diff` depends on undefined methods `Self` and `eq`
- translated using λ-lifting

```
diff
  (Self : Type)
  (eq : Self -> Self -> bool)
  (x : Self)
  (y : Self) : bool
:=
  not (eq x y).
```

Compilation of object mechanisms

FoCaLiZe	OCaml/Coq
method	λ-lifted function
species	module
complete species	record
collection	module

- Modules and records don't exist in Dedukti
  - Records added as syntactic sugar in Sukerujo

Results: Summary

- `https://www.rocq.inria.fr/deducteam/Focalide/index.html`
- 95% of FoCaLiZe standard library
- Time compared to Coq (s)

	Dedukti	Coq
Compilation	0.7	0.7
Proof search	14.3	12.1
Proof checking	1.0	16.7

Results: Focalide Coverage

File	Ignored (%)
`basics`	0
`sets`	0
`products`	0
`lattices`	0
`orders`	0.8
`strict_orders`	13.5
`orders_and_lattices`	10.0
`sums`	1.9
`quotients`	7.1
`fix`	13.3
Total	4.4

Results: Size (kB)

File	FoCaLiZe	Dedukti	Coq
`basics`	4.6	2.5	3.3
`sets`	1.8	2.5	4.2
`products`	2.9	7.1	13.5
`lattices`	3.8	10.9	20.1
`orders`	1.7	5.1	10.3
`strict_orders`	1.6	2.4	9.1
`orders_and_lattices`	2.8	8.4	27.8
`sums`	4.2	14.2	20.6
`quotients`	2.3	5.5	9.7
`fix`	5.2	3.6	16.4
Total	31.0	62.0	135.8

Results: Translation Time (ms)

File	Focalide	FoCaLiZe to Coq
`basics`	60	60
`sets`	350	280
`products`	1420	930
`lattices`	5490	2980
`orders`	930	620
`strict_orders`	360	600
`orders_and_lattices`	3730	4150
`sums`	1620	1380
`quotients`	420	560
`fix`	1070	1200
Total	15.5 s	12.8 s

Results: Checking Time (ms)

File	Dedukti	Coq
`basics`	60	1340
`sets`	30	1210
`products`	180	1590
`lattices`	130	1990
`orders`	70	1460
`strict_orders`	30	1410
`orders_and_lattices`	130	2260
`sums`	350	2120
`quotients`	150	1350
`fix`	70	2000
Total	1.19 s	16.7 s

Conclusion and perspectives

- Focalide works
- Recursive functions
  - Last important feature of FoCaLiZe
- Zenon modulo
  - Faster
  - Hopefully better suited for proofs on recursive functions
  - Already compatible with types
- Basis for an interoperability framework
  (first experiment with Ali)