[1]INRIA, [2]Ecole Polytechnique, [3]ENSIIE/Cedric

Embedding logics in Dedukti

Ali Assaf[12], Guillaume Burel[3]

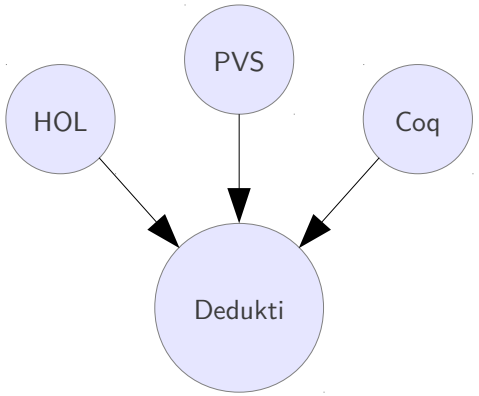
April 12, 2013

Outline

- Introduction
- Embedding pure type systems
- Holide: HOL in Dedukti
- Coqine: Coq in Dedukti
- Conclusion

Outline

Universal proof checker



Source: HOL, Coq, ...

- Pure type systems
- Reconstruction, proof search, ...

Target: Dedukti

- $\lambda\Pi$-calculus modulo (see previous talk)
- Proof checking (no reconstruction, no proof search, ...)

Towards interoperability

Prove $A \Rightarrow B$ in system 1. Prove A in system 2.

- Can we deduce B?

$$\frac{\|A \Rightarrow B\|_1 \qquad \|A\|_2}{\|B\|_?} \ ?$$

Shallow embeddings: $\|A \Rightarrow B\|_1 = \|A\|_1 \to \|B\|_1$

- Only need to relate $\|A\|_1$ with $\|A\|_2$

Shallow vs. deep embeddings

Reuse features of the target language

- Variables, binders, ...
- Rewriting, computation, ...

Advantages:

- Lighter translations
- Make interoperability easier

Outline

Pure type systems

Signature $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$

- $\mathcal{S}$ a set of *sorts*
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ a set of *axioms*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ a set of *rules*

Syntax

sorts	s	$\in$	$\mathcal{S}$
terms	t, u, A, B	$::=$	$x \mid s \mid \lambda x : A.\, t \mid \Pi x : A.\, B \mid t\, u$
contexts	Γ	$::=$	$\emptyset \mid \Gamma, x : A$

Typing rules

$$\frac{\Gamma \text{ well-formed} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \text{ well-formed} \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash t : B \qquad \Gamma \vdash (\Pi x : A.\, B) : s_3}{\Gamma \vdash (\lambda x : A.\, t) : (\Pi x : A.\, B)}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (\Pi x : A.\, B) : s_3}$$

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

$$\frac{}{\emptyset \text{ well-formed}} \qquad \frac{\Gamma \text{ well-formed} \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \text{ well-formed}}$$

Example

Example

The calculus of constructions (CC) is the PTS defined by the signature:

$$\begin{array}{lcl} \mathcal{S} & = & \mathsf{type}, \mathsf{kind} \\ \mathcal{A} & = & (\mathsf{type} : \mathsf{kind}) \\ \mathcal{R} & = & (\mathsf{type}, \mathsf{type}, \mathsf{type}), (\mathsf{kind}, \mathsf{type}, \mathsf{type}), \\ & & (\mathsf{type}, \mathsf{kind}, \mathsf{kind}), (\mathsf{kind}, \mathsf{kind}, \mathsf{kind}) \end{array}$$

The polymorphic identity function $id = (\lambda A : \mathsf{type}.\, \lambda x : A.\, x)$ is well-typed in CC:

$$\vdash id : (\Pi A : \mathsf{type}.\, A \to A)$$

Translations

- $\lambda\Pi$-calculus: types may only depend on terms
- PTS: terms and types may depend on types (polymorphism, type operators)

Two translations:

- $|t|$ as a **term**
- $\|t\|$ as a **type**

Two Translations

As a **type**:

- $\|s\| = s$, the universe of terms of type s
- $\|\Pi x : A.\, B\| = \Pi x : \|A\| .\, \|B\|$

As a **term**:

- $|s| = \dot{s}$, a constant
- $|\Pi x : A.\, B| = \dot{\pi}\ |A|\ (\lambda x : \|A\| .\, |B|)$

Decoding function

- A term A can be used as a type
- Decoding function ε

$$\|A\| = \varepsilon\,|A|$$

- Constraints:

$$\begin{array}{rcl} \|s\| & = & s \\ \|\Pi x : A.\, B\| & = & \Pi x : \|A\|\,.\ \|B\| \end{array}$$

Decoding function

- A term A can be used as a type
- Decoding function ε

$$\|A\| = \varepsilon\,|A|$$

- Constraints:

$$\begin{array}{rcl} \varepsilon\,|s| & = & s \\ \varepsilon\,|\Pi x : A.\, B| & = & \Pi x : \varepsilon\,|A|\,.\,\varepsilon\,|B| \end{array}$$

Decoding function

- A term A can be used as a type
- Decoding function ε

$$\|A\| = \varepsilon\,|A|$$

- Constraints:

$$\begin{array}{rcl} \varepsilon\,\dot{s} & = & s \\ \varepsilon\,(\dot{\pi}\ |A|\ (\lambda x : \|A\|\,.\ |B|)) & = & \Pi x : \varepsilon\,|A|\,.\,\varepsilon\,|B| \end{array}$$

Decoding function

- A term A can be used as a type
- Decoding function ε

$$\|A\| = \varepsilon\,|A|$$

- Constraints:

$$\begin{array}{rcl} \varepsilon\,\dot{s} & = & s \\ \varepsilon\,(\dot{\pi}\,A\,B) & = & \Pi x{:}\varepsilon\,A.\,\varepsilon\,(B\,x) \end{array}$$

Decoding function

- A term A can be used as a type
- Decoding function ε

$$\|A\| = \varepsilon\,|A|$$

- Constraints:

$$\begin{array}{rcl} \varepsilon\,\dot{s} & \longrightarrow & s \\ \varepsilon\,(\dot{\pi}\,A\,B) & \longrightarrow & \Pi x : \varepsilon\,A.\,\varepsilon\,(B\,x) \end{array}$$

The embedding

Constants

$$
\begin{array}{lcll}
s & : & \mathsf{Type} & \forall s \in \mathcal{S} \\
\varepsilon_s & : & s \to \mathsf{Type} & \forall s \in \mathcal{S} \\
\dot{s}_1 & : & s_2 & \forall (s_1 : s_2) \in \mathcal{A} \\
\dot{\pi}_{s_1,s_2,s_3} & : & \Pi A : s_1 . \, (\varepsilon_{s_1} A \to s_2) \to s_3 & \forall (s_1, s_2, s_3) \in \mathcal{R}
\end{array}
$$

Rewrite rules

$$
\begin{array}{lcll}
\varepsilon_{s_2} \, (\dot{s}_1) & \longrightarrow & s_1 & \forall (s_1 : s_2) \in \mathcal{A} \\
\varepsilon_{s_3} \, (\dot{\pi}_{s_1,s_2,s_3} \, A \, B) & \longrightarrow & \Pi x : \varepsilon_{s_1} A . \, \varepsilon_{s_2} (B \, x) & \forall (s_1, s_2, s_3) \in \mathcal{R}
\end{array}
$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\, \lambda x : A.\, x \quad : \quad (\Pi A : \mathsf{type}.\, A \to A)$$

Translation:

$$|\lambda A : \mathsf{type}.\, \lambda x : A.\, x| \quad : \quad \|\Pi A : \mathsf{type}.\, A \to A\|$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$|\lambda A : \mathsf{type}.\ \lambda x : A.\ x| \quad : \quad \|\Pi A : \mathsf{type}.\ A \to A\|$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\, \lambda x : A.\, x \quad : \quad (\Pi A : \mathsf{type}.\, A \to A)$$

Translation:

$$\lambda A : \|\mathsf{type}\|.\, \lambda x : \|A\|.\, x \quad : \quad \|\Pi A : \mathsf{type}.\, A \to A\|$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\, \lambda x : A.\, x \quad : \quad (\Pi A : \mathsf{type}.\, A \to A)$$

Translation:

$$\lambda A : \varepsilon\, |\mathsf{type}|.\, \lambda x : \varepsilon\, |A|.\, x \quad : \quad \|\Pi A : \mathsf{type}.\, A \to A\|$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$\lambda A : \mathsf{type}.\ \lambda x : \varepsilon\ A.\ x \quad : \quad \|\Pi A : \mathsf{type}.\ A \to A\|$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$\lambda A : \mathsf{type}.\ \lambda x : \varepsilon\ A.\ x \quad : \quad \varepsilon |\Pi A : \mathsf{type}.\ A \to A|$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$\lambda A : \mathsf{type}.\ \lambda x : \varepsilon\ A.\ x \quad : \quad \varepsilon(\dot{\pi}\ |\mathsf{type}|\ (\lambda A : \|\mathsf{type}\|.\ \ldots))$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$\lambda A : \mathsf{type}.\ \lambda x : \varepsilon\ A.\ x \quad : \quad \varepsilon(\dot{\pi}\ \dot{\mathsf{type}}\ (\lambda A : \mathsf{type}.\ \ldots))$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$\lambda A : \mathsf{type}.\ \lambda x : \varepsilon\ A.\ x \quad : \quad \Pi A : \varepsilon\ \mathsf{type}.\ (\lambda A : \mathsf{type}.\ \ldots) A$$

Example

Example

Polymorphic identity in CC

$$\lambda A : \mathsf{type}.\ \lambda x : A.\ x \quad : \quad (\Pi A : \mathsf{type}.\ A \to A)$$

Translation:

$$\lambda A : \mathsf{type}.\ \lambda x : \varepsilon\ A.\ x \quad : \quad \Pi A : \mathsf{type}.\ \varepsilon\ A \to \varepsilon\ A$$

Properties

Theorem
If $\Gamma \vdash t : A$ *then* $\|\Gamma\| \vdash |t| : \|A\|$. *(Soundness)*

Proof.
In [Cousineau and Dowek 2007]. □

Theorem
If $\|\Gamma\| \vdash t : \|A\|$ *then* $\Gamma \vdash t' : A$. *(Conservativity)*

Proof.
Coming soon!
Partial result in [Cousineau and Dowek 2007]. □

Outline

HOL

- A family of theorem provers
  - (HOL Light, HOL4, ProofPower, ...)
- Based on *higher order logic*
- Large formalizations
  - Flyspeck project (Kepler's conjecture)

Higher order logic

- Formulas are terms of the simply-typed λ-calculus

$$\begin{array}{llll} \text{types} & A, B & ::= & \mathsf{prop} \mid A \to B \\ \text{terms} & t, u & ::= & x \mid \lambda x : A.\, t \mid t\, u \mid \mathsf{c} \end{array}$$

- Can be seen as a PTS

$$\begin{array}{lll} \mathcal{S} & = & \mathsf{prop}, \mathsf{type}, \mathsf{kind} \\ \mathcal{A} & = & (\mathsf{prop} : \mathsf{type}), (\mathsf{type} : \mathsf{kind}) \\ \mathcal{R} & = & (\mathsf{prop}, \mathsf{prop}, \mathsf{prop}), (\mathsf{type}, \mathsf{prop}, \mathsf{prop}), \\ & & (\mathsf{type}, \mathsf{type}, \mathsf{type}), (\mathsf{kind}, \mathsf{kind}, \mathsf{kind}) \end{array}$$

Challenges

- Classical formulation
  - Equality as a primitive connective, no PTS structure
  - Axioms (β, extensionality, choice, ...)
- No proof inspection
  - LCF architecture: theorems as an abstract datatype
  - Proofs are not kept in memory
- Proof size
  - No computation in proofs (β, definitions, ...)
  - Huge proof trees
- Free variables
  - Implicit quantification
  - Name clashes

Proof retrieval

HOL kernel

- OCaml module defining an abstract datatype for theorems + methods for constructing theorems
- The HOL kernel must be modified!

OpenTheory project [Hurd 2011]

- A standard for exporting and exchanging HOL proofs
- A well-defined standard library

The OpenTheory article format

Instructions that are executed to reconstruct the theorems.

Example

Reflexivity on a variable x of type A:

$$\frac{}{\vdash x^A = x^A}\ \text{refl}$$

OpenTheory article file

```
"A"
varType
"x"
var
varTerm
refl
```

OCaml execution

```
let A = varType("A") in
let x = varTerm(var("x", A)) in
refl x
```

Proof size

Example

A proof of $t + t = u + u$:

```
let p = ... (* A very large proof of t = u *) in
appThm (appThm (refl f) p p)
```

$$\dfrac{\dfrac{\dfrac{}{(+) = (+)}\ \text{refl} \qquad \dfrac{\pi}{t = u}}{(+)\,t = (+)\,u}\ \text{appThm} \qquad \dfrac{\pi}{t = u}}{(+)\,t\,t = (+)\,u\,u}\ \text{appThm}$$

Need proof sharing!

Proof sharing

- Share common subproofs
- Already provided (partially) by OpenTheory
  - Going further: factorize every step

```
step1 : proof (t = u) := ...
step2 : proof ((+) = (+)) := refl q.
step3 : proof ((+) t = (+) u) := appThm step2 step1.
step4 : proof ((+) t t = (+) u u) := appThm step3 step1.
```

- Need lambda lifting
  - Abstract over free variables
  - α-renaming to avoid clashes

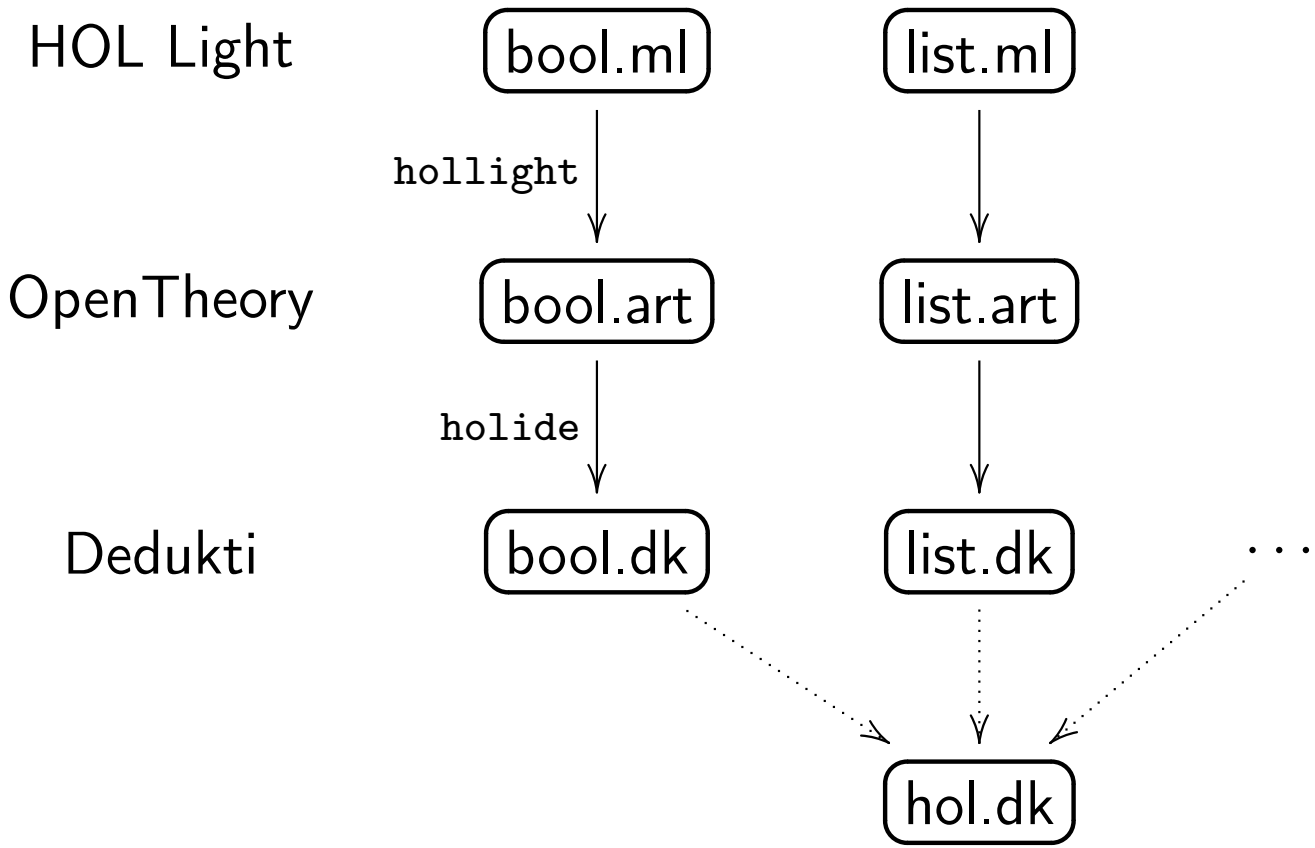
Proof sharing

- Share common subproofs
- Already provided (partially) by OpenTheory
  - Going further: factorize every step

```
step1 : proof (t = u) := ...
step2 : proof ((+) = (+)) := refl q.
step3 : proof ((+) t = (+) u) := appThm step2 step1.
step4 : proof ((+) t t = (+) u u) := appThm step3 step1.
```

- Need lambda lifting
  - Abstract over free variables
  - α-renaming to avoid clashes
- To-do: extend to terms

Translation process



Results

Package	Size (in kB)		Time (in s)		Percentage
	OpenTheory	Dedukti	Translation	Verification	Verified
unit	26	309	0	0	100%
function	89	1,301	1	3	100%
pair	195	4,943	4	15	100%
bool	305	4,258	2	7	100%
sum	502	20,988	17	99	100%
option	520	23,815	18	77	100%
relation	971	42,572	49	350	100%
list	1377	68,031	39	182	100%
real	1754	68,508	46	1	1%
natural	1952	130,111	38	496	100%
set	2329	90,819	65	431	100%
Total	10020	455,656	280	1,661	85%

Outline

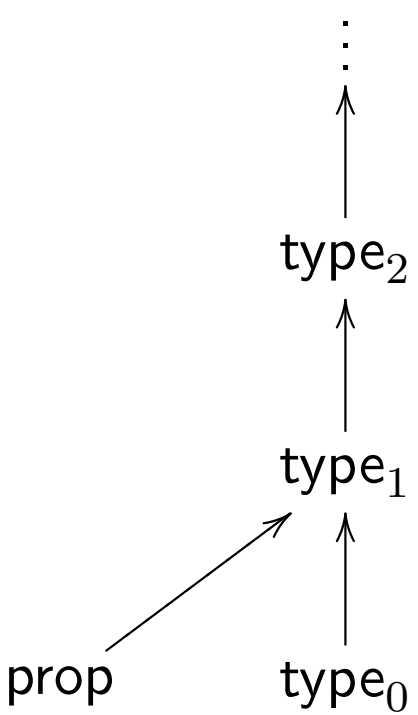
Coq

- Proof system based on the calculus of inductive constructions
  - Infinite hierarchy of universes type_i
  - Subtyping $\text{type}_i \subseteq \text{type}_{i+1}$
  - Floating universes
  - Inductive types
  - Co-inductive types
  - Modules
  - ...
- Large formalizations
  - 4 color theorem, Feit–Thompson theorem, ...

Challenges

- Rich and complex theory (CIC)
  - Much more than a PTS
  - Can we encode it in the $\lambda\Pi$-calculus modulo?
- Proof retrieval
  - No direct export functions
- Implementation jungle
  - No complete specification
  - Huge code base (100 000s lines of code)
  - Poorly documented

Infinite universe hierarchy



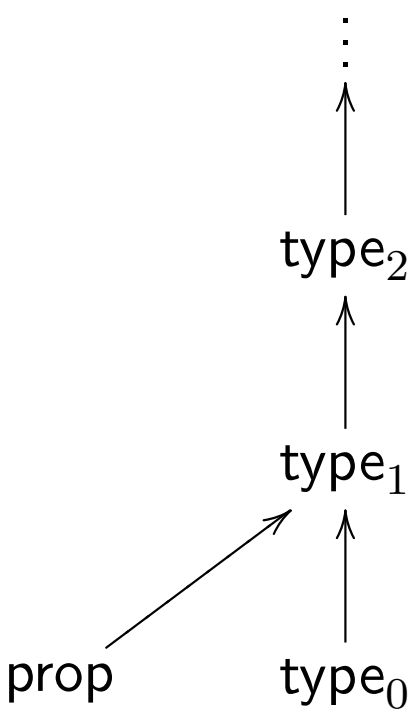

```
prop : Type.
type : Type.

eprop : prop → Type.
etype : type → Type.

dprop : type.
dtype : type.

etype dprop --→ prop.
etype dtype --→ type.
```

Infinite universe hierarchy




```
prop : Type.
type : nat → Type.

eprop : prop → Type.
etype : i :  nat → type i → Type.

dprop : type (s z).
dtype : i :  nat → type (s i).

etype (s z) dprop --→ prop.
etype (s i) (dtype i) --→ type i.
```

Subtyping

In Coq

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \subseteq B}{\Gamma \vdash t : B} \qquad \frac{A \equiv B}{A \subseteq B}$$

$$\frac{}{\mathsf{prop} \subseteq \mathsf{type}_i} \qquad \frac{i \leq j}{\mathsf{type}_i \subseteq \mathsf{type}_j} \qquad \frac{B \subseteq C}{\Pi x : A.\, B \subseteq \Pi x : A.\, C}$$

In Dedukti

- Confluence $\Longrightarrow$ uniqueness of types
- Uniqueness of types $\Longrightarrow$ no shallow encoding
- Use type casting

Type casting

- No implicit type casting

$$\frac{\Gamma \vdash t : \varepsilon\, A \qquad A \subseteq B}{\Gamma \vdash t : \varepsilon\, B}$$

- Cast function

$$\mathsf{cast} : A \subseteq B \to \varepsilon\, A \to \varepsilon\, B$$

- Insert when needed

$$\frac{\Gamma \vdash t : \varepsilon\, A \qquad \Gamma \vdash h : A \subseteq B}{\Gamma \vdash \mathsf{cast}\, h\, t : \varepsilon\, B}$$

- Need to define $A \subseteq B$

Subtyping relation

As an inductive relation

$$\begin{array}{lcl} \mathsf{sub\_refl} & : & A \subseteq A \\ \mathsf{sub\_prop} & : & \dot{\mathsf{prop}} \subseteq \dot{\mathsf{type}}_i \\ \mathsf{sub\_type} & : & \dot{\mathsf{type}}_i \subseteq \dot{\mathsf{type}}_{i+j} \\ \mathsf{sub\_prod} & : & (\Pi x : \varepsilon A.\, (B\, x) \subseteq (C\, x)) \to \dot{\pi}\, A\, B \subseteq \dot{\pi}\, A\, C \end{array}$$

- $\Longrightarrow$ large cast annotations ×

As a rewrite system

$$\begin{array}{lcl} \dot{\mathsf{prop}} \subseteq \dot{\mathsf{type}}_i & \dashrightarrow & \dot{\mathsf{prop}} \subseteq \dot{\mathsf{prop}} \\ \dot{\mathsf{type}}_i \subseteq \dot{\mathsf{type}}_{i+j} & \dashrightarrow & \dot{\mathsf{type}}_i \subseteq \dot{\mathsf{type}}_i \\ \dot{\pi}\, A\, B \subseteq \dot{\pi}\, A\, C & \dashrightarrow & \Pi x : \varepsilon A.\, (B\, x) \subseteq (C\, x) \end{array}$$

- $\Longrightarrow$ canonical proof $\mathsf{sub\_refl} : A \subseteq A$ ✓

Inductive types

In Coq

```
Inductive nat :  Type :=
| z :  nat
| s :  nat -> nat.
```

In Dedukti

```
nat : type.
z : etype nat.
s : etype nat → etype nat.
```

Pattern matching:

```
nat_case : P : (etype nat → type) →
           fz : etype (P z) →
           fs : (n : eset nat → etype (P (s n))) →
           n : etype nat → etype (P n).
[...] nat_case P fz fs z --→ fz.
[...] nat_case P fz fs (s x) --→ fs n.
```

Fixpoints

Translate fixpoints using rewriting rules

```
Fixpoint f x1 ...  xn struct xn := t.
```

```
[...] f x1 ... xn --> t. ×
```

Termination: CIC fixpoints are unfolded only if their structural argument starts with a constructor.

- Need to give the same semantics in the $\lambda\Pi$-calculus modulo

Fixing fixpoints

- Cannot use pattern matching because of dependent types

```
Fixpoint f (x y :  A) (p :  x = y) := t.
```

```
[...] f x y (eq_refl x) --> t. ×
```

Fixing fixpoints

- Cannot use pattern matching because of dependent types

```
Fixpoint f (x y :  A) (p :  x = y) := t.
```

```
[...] f x y (eq_refl x) --> t. ×
```

- Add guards to constructors

```
pre_nat : type.
z : etype pre_nat.
s : etype nat → etype pre_nat.
nat_constr : pre_nat → nat.
```

$$\begin{aligned} |\mathsf{z}| &= \mathsf{nat\_constr}\,\mathsf{z} \\ |\mathsf{s}\,n| &= \mathsf{nat\_constr}\,(\mathsf{s}\,|n|) \end{aligned}$$

```
[...] f x1 ... (nat_constr xn) --> t. ✓
```

  - Increased term size
  - no syntactic equivalence

Fixing fixpoints

- Cannot use pattern matching because of dependent types

```
Fixpoint f (x y :  A) (p :  x = y) := t.
```

```
[...] f x y (eq_refl x) --> t. ×
```

- Add guards to constructors

```
pre_nat : type.
z : etype pre_nat.
s : etype nat → etype pre_nat.
nat_constr : pre_nat → nat.
```

$$\begin{aligned} |\mathsf{z}| &= \mathsf{nat\_constr}\,\mathsf{z} \\ |\mathsf{s}\,n| &= \mathsf{nat\_constr}\,(\mathsf{s}\,|n|) \end{aligned}$$

```
[...] f x1 ... (nat_constr xn) --> t. ✓
```

  - Increased term size
  - no syntactic equivalence

Use recursors?

Modules

In Coq:

- Nested modules

```
Module A. ...  Module B. ...  End B. ...  End A.
```

- Functors

```
Module F (A : T). ...  End N.
```

- Module expressions

```
Module B := F(A).
```

In Dedukti:

- Only separate namespaces for files

Translating modules

Flatten nesting

- `A.B.C` translated as `A.B_C`

Module is expanded into its individual elements

```
Module A := B.
```

is translated into

```
A_x := B_x.
A_y := B_y.
...
```

Translating modules

Flatten nesting

- `A.B.C` translated as `A.B_C`

Module is expanded into its individual elements

```
Module A (C):= B.
```

is translated into

```
A_x C_z1 C_z2 ... := B_x.
A_y C_z1 C_Z2 ... := B_y.
...
```

- Increased term size

Proof retrieval

Coq proof scripts (`.v`) ×

- Notations
- Implicit arguments
- Tactics

Coq compiled file (`.vo`) ✓

- Low-level proof terms
- Binary dump of memory content

Using .vo files

Advantages:

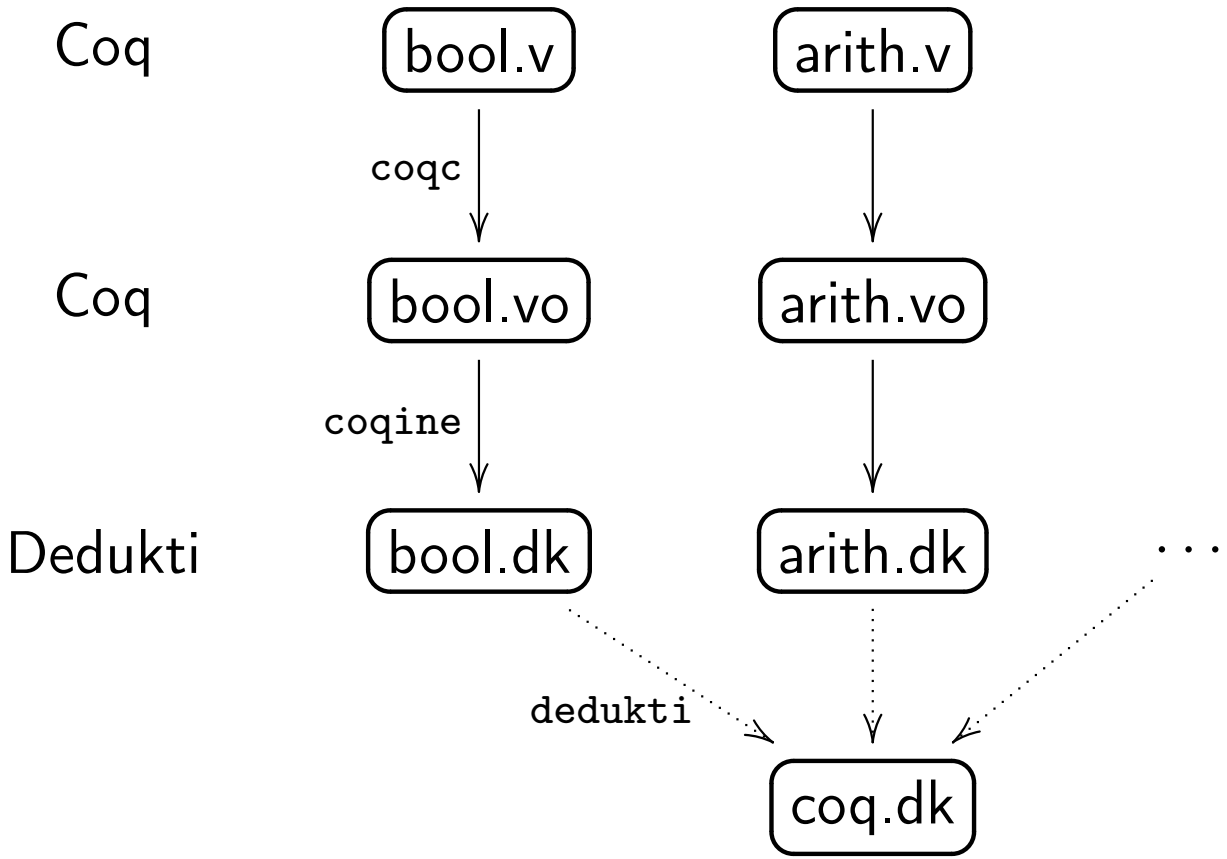
- Reconstructed proof terms
- Exactly the ones used by Coq
- Reuse Coq's code

Disadvantages:

- Rely on Coq's code
- Must use the same versions of both `ocaml` and `coqc`

Coqine is built on top of `coqchk`

Translation process



Results

- Working prototype
- Coq standard library:
  - Can translate 84%
  - Can type-check 20%
- File size:
  - `List.v`: 45 kB
  - `List.vo`: 260 kB
  - `List.dk`: 2.2MB

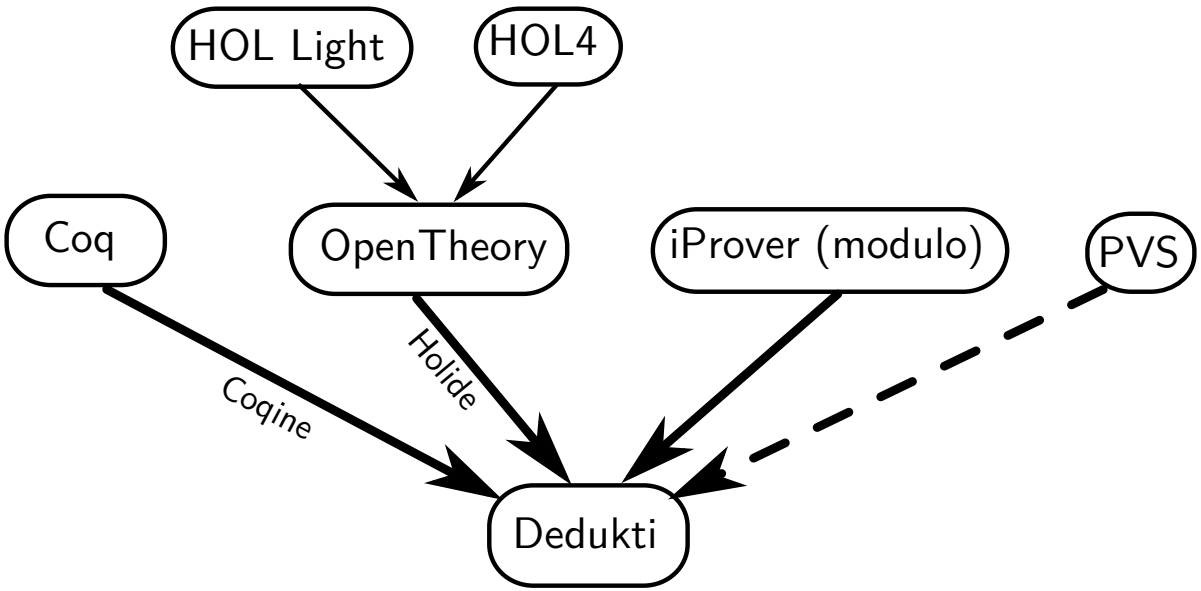
Future work

- Complete the current implementation
- Solve remaining features
  - Floating universes
    - Contradicts weakening
    - Non-modularity, side-effects
  - Co-inductive types
  - New coq features still coming!
- Long term: better specification for Coq

Outline

Conclusion

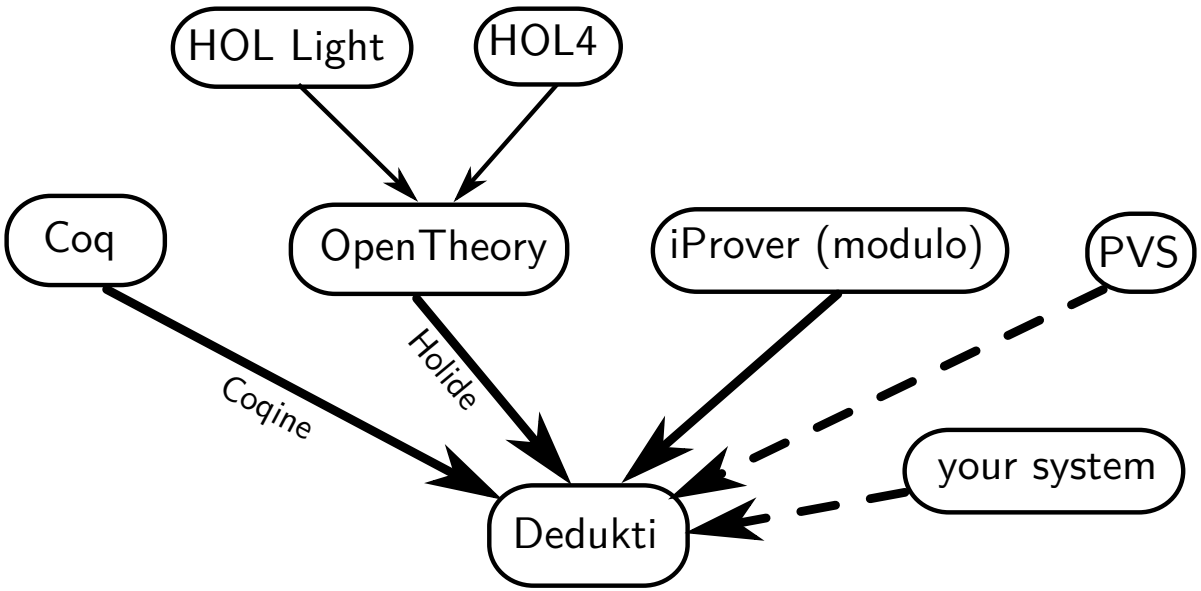
- Embedding various logics can be fun!
- Theoretical and practical challenges
- Need good specifications and exporting standards



Software available at
https://www.rocq.inria.fr/deducteam/software.html

Conclusion

- Embedding various logics can be fun!
- Theoretical and practical challenges
- Need good specifications and exporting standards



Software available at
https://www.rocq.inria.fr/deducteam/software.html