Importing HOL Light into Coq

Chantal Keller
keller@lix.polytechnique.fr

Benjamin Werner
benjamin.werner@inria.fr

INRIA – LIX – ÉNS de Lyon

July, 14th 2010

Presentation

Observation:

- many “languages” to write formal proofs

Dream?

- share theorems between formalisms
- prove theorems in different formalisms
- translate theorems from one formalism into another

Presentation

Observation:

- many "languages" to write formal proofs

Dream?

- share theorems between formalisms
- prove theorems in different formalisms
- translate theorems from one formalism into another automatically

Framework

Translation from HOL Light to Coq

Interests:

- use HOL Light's theorems in Coq (analysis)
- build a model of HOL Light in Coq
- compare the two logical frameworks

Objectives:

- re-check the theorems in Coq
- get intelligible theorems:
  - **forall** x x0 : **Prop**, (x ∧ x0) = (x0 ∧ x)
  - **forall** x x0 : hollight_Prop, hollight_eq (hollight_and x x0) (hollight_and x0 x)
- "reasonable" computing time and memory consumption

Outline

Computational reflection

Predicate `P : nat -> Prop`

1. Prove `P n` in a standard way (tactics...)
2. Prove `P n` using computation

Computational reflection

Predicate `P : nat -> Prop`

1. Prove `P n` in a standard way (tactics...)
2. Prove `P n` using computation

- Certificate:

```
certif: nat -> Type
```

- Checker:

```
f: forall n, certif n -> bool
```

such that

```
forall c n, f n (c n) = true -> P n
```

Computational reflection

Predicate `P : nat -> Prop`

1. Prove `P n` in a standard way (tactics...)
2. Prove `P n` using computation

- Certificate:
  ```
  certif: nat -> Type
  ```
- Checker:
  ```
  f: forall n, certif n -> bool
  ```
  such that
  ```
  forall c n, f n (c n) = true -> P n
  ```

Certificate: HOL Light proof term...

Embeddings

Different ways to embed a logical system A into another system B

Deep embedding

Different ways to embed a logical system A into another system B

- How to proceed:
  Data-type in B to represent the objects of A
- Example of the simply typed λ-calculus in Coq:

```
Inductive type :  Type :=
   | B : type
   | A : type -> type -> type.
Inductive term :  Type :=
   | Var:  string -> term
   | Lam:  string -> type -> term -> term
   | App:  term -> term -> term.
```

- Representation of $\lambda x : bool.x$:

```
Lam “x” B (Var “x”)
```

Shallow embedding

Different ways to embed a logical system A into another system B

- How to proceed:
  Objects of B to represent the objects of A
- Example of the simply typed λ-calculus in Coq:

```
Definition type := Type.
```

- Representation of $\lambda x : bool.x$:

```
fun x:  bool => x
```

Remarks

- Normalization by evaluation:
  deep $\rightarrow$ shallow $\rightarrow$ deep

Remarks

- Normalization by evaluation:
  deep $\rightarrow$ shallow $\rightarrow$ deep
- Ability to reason by induction over the structure of the objects of the embedded system
- Computational reflection
- Obtain Coq theorems

Remarks

- Normalization by evaluation:
  deep $\rightarrow$ shallow $\rightarrow$ deep
- Ability to reason by induction over the structure of the objects of the embedded system
- Computational reflection
- Obtain Coq theorems
- Here: deep $\rightarrow$ shallow (translation function)

Procedure

- Type translation
- Term translation according to the type

Type translation

```
Inductive type :  Type :=
   | B : type
   | A : type -> type -> type.
```

```
Fixpoint tr_type (t:  type) :  Type :=
   match t with
      | B => bool
      | A a b => (tr_type a) -> (tr_type b)
   end.
```

Term translation

```
Inductive term :  Type :=
   | Var:  string -> term
   | Lam:  string -> type -> term -> term
   | App:  term -> term -> term.
```

```
Fixpoint tr_term  I                    (t:  term)        :=
   match t with
      | Var x => I x
      | Lam x ty t => fun y => tr_term (add I x y) t
      | App t u => (tr_term I t) (tr_term I u)
   end.
```

Term translation

```
Inductive term :  Type :=
   | Var:  string -> term
   | Lam:  string -> type -> term -> term
   | App:  term -> term -> term.
```

```
Fixpoint tr_term  I                    (t:  term) :  ?  :=
   match t with
      | Var x => I x
      | Lam x ty t => fun y => tr_term (add I x y) t
      | App t u => (tr_term I t) (tr_term I u)
   end.
```

Term translation

```
Inductive term :  Type :=
   | Var:  string -> term
   | Lam:  string -> type -> term -> term
   | App:  term -> term -> term.
```

```
Fixpoint tr_term (I: string -> ?)  (t:  term) :  ?  :=
   match t with
      | Var x => I x
      | Lam x ty t => fun y => tr_term (add I x y) t
      | App t u => (tr_term I t) (tr_term I u)
   end.
```

Term translation

```
Inductive term :  Type :=
   | Var:  string -> term
   | Lam:  string -> type -> term -> term
   | App:  term -> term -> term.
```

```
Fixpoint tr_term (I: string -> ?)  (t:  term) :  ?  :=
   match t with
      | Var x => I x
      | Lam x ty t => fun y => tr_term (add I x y) t
      | App t u => (tr_term I t) (tr_term I u)
   end.
```

$\hookrightarrow$ consider only well typed terms

Term translation: general idea

- If `t` has type `T`, then `tr_term I t` $\in$ `tr_type T`

Term translation: general idea

- If `t` has type `T`, then `tr_term I t` $\in$ `tr_type T`
- Type inference:

```
Fixpoint infer (c:  string -> type) (t:  term) :  type
:=
   match t with
      | Var x => c x
      | Lam x ty t => A ty (infer (fun y =>
        if y == x then ty else c y) t)
      | App t u =>
      match infer c t, infer c u with
         | A T U, V => if T == V then U else fail
         | _, _ => fail
      end
   end.
```

Term translation: general idea

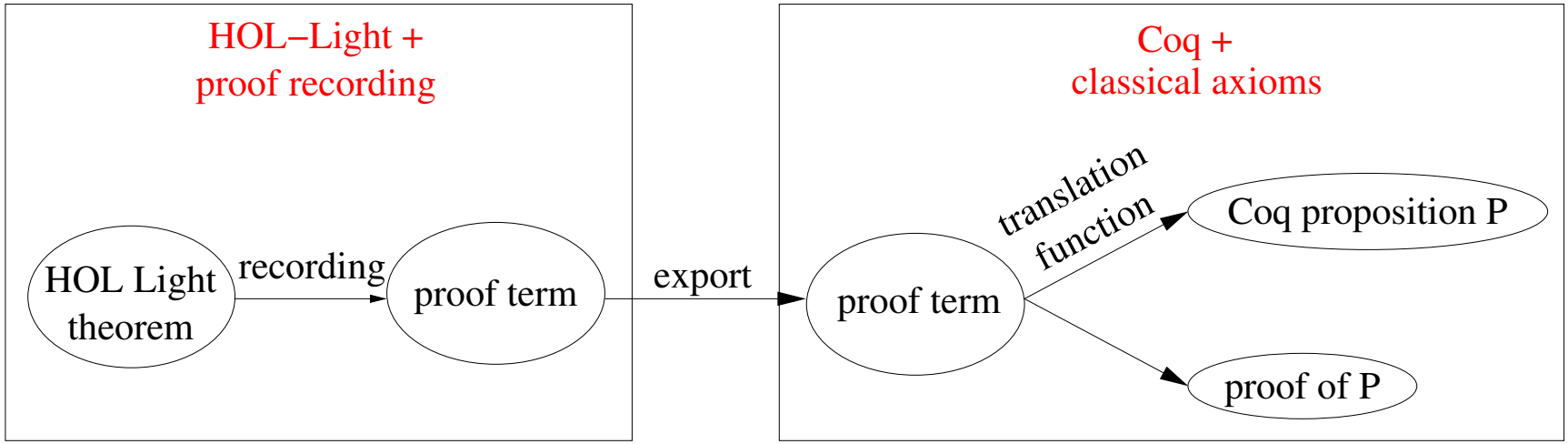
- If `t` has type `T`, then `tr_term I t` $\in$ `tr_type T`
- Type inference:

```
Fixpoint infer (c: string -> type) (t: term) : type
:= ...
```

- Refinement of the type inference function:

```
Fixpoint tr_term (c: string -> type)
(I: forall x, tr_type (c x)) (t: term) :
{ty: type & tr_type ty} := ...
```

Idea


HOL–Light +
proof recording
HOL Light
theorem
recording
proof term
export
Coq +
classical axioms
proof term
translation
function
Coq proposition P
proof of P

Outline

HOL Light

HOL Light:

- proof assistant written by John Harrison et al.
- in an OCaml top-level
- higher order classical logic
- automated tools and pre-proved theorems
- programmable without compromising soundness
- simpler logical kernel than HOL

System's safety

System's safety:

- no proof terms
- similar to LCF (thanks, Robin Milner)
- **abstract** type thm (theorems are built using only HOL Light's primitives)

Recording and exportation

Recording:

- the proof trees
- granularity

Exporting:

- the proof trees
- sharing

$\hookrightarrow$ tool by Steven Obua

Higher Order Logic Light

- Terms: simply-typed λ-calculus with equality
- Theorem:
  - a set of hypotheses and a conclusion (terms of type B)
  - built using derivation rules

Examples:

$$\frac{}{\vdash (\lambda x.t)\ x = t} \qquad \frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

where $\Leftrightarrow$ stands for $=_{\mathsf{bool}}$.

Certificate

$$\frac{}{\vdash (\lambda x.t)\ x = t} \qquad \frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

```
Inductive deriv:  set term -> term -> Prop :=
   | Dbeta:  forall x t T U,
     deriv empty (App (App (Eq T) (App (Lam x U t)
x)) t)
   | Deqmp:  forall h1 h2 a b,
     deriv h1 (App (App (Eq B) a) b) -> deriv h2 a ->
     deriv (h1 ∪ h2) b
```

Soundness

- Given: `t : B` and `c : deriv empty t`
- Property we want to prove: `tr_term ...t`
  (Coq version of the theorem)
- Checker: `sem_deriv` such that `sem_deriv t c = true -> tr_term ...t`

A smaller certificate

```
Inductive deriv:  set term -> term -> Prop :=
   | Dbeta:  forall x t T U,
      deriv empty (App (App (Eq T) (App (Lam x U t)
x)) t)
   | Deqmp:  forall h1 h2 a b,
      deriv h1 (App (App (Eq B) a) b) -> deriv h2 a ->
      deriv (h1 ∪ h2) b
```

A smaller certificate

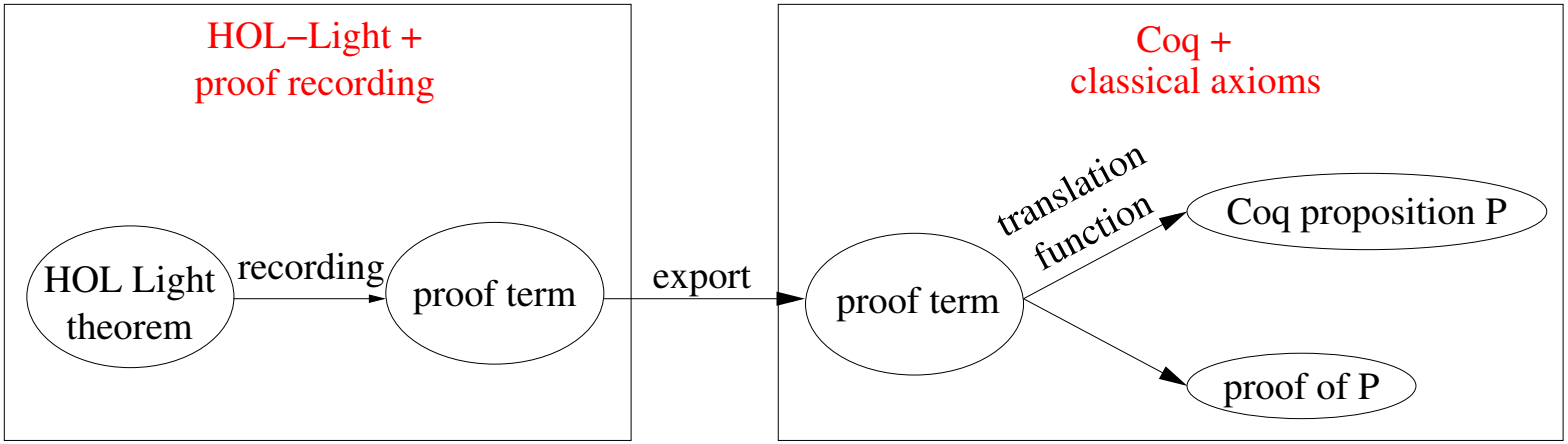
```
Inductive deriv:  set term -> term -> Prop :=
   | Dbeta:  forall x t T U,
     deriv empty (App (App (Eq T) (App (Lam x U t)
x)) t)
   | Deqmp:  forall h1 h2 a b,
     deriv h1 (App (App (Eq B) a) b) -> deriv h2 a ->
     deriv (h1 ∪ h2) b
```

⇑

```
Inductive proof:  Type :=
   | Pbeta:  string -> term -> type -> type -> proof
   | Peqmp:  proof -> proof -> proof
```

Outline

Qualitative



Intelligibility:

- CONJ_SYM : $\forall\ t_1\ t_2.\ t_1 \wedge t_2 \Leftrightarrow t_2 \wedge t_1$

  forall x x0 : **Prop**, (x ∧ x0) = (x0 ∧ x)
- arithmetic over binary naturals (N)

Quantitative

Bench.	Number		Time		
	Theorems	Lemmas	Rec.	Exp.	Comp.
Stdlib	1,726	195,317	2 min 30	6 min 30	10h
Model	2,121	322,428	6 min 30	29 min	44h
Vectors	2,606	338,087	6 min 30	21 min	39h

Bench.	Memory		
	H.D.D.	Virt. OCaml	Virt. Coq
Stdlib	218 Mb	1.8 Gb	4.5 Gb
Model	372 Mb	5.0 Gb	7.6 Gb
Vectors	329 Mb	3.0 Gb	7.5 Gb

Outline

Conclusion

HOL Light:

- record and export proof terms
- reusable

Coq:

- a model of HOL Light
- sound

$\hookrightarrow$ good understanding of the two formalisms required

Interface:

- standard library
- intelligibility of theorem's statements
- users facilities

Perspectives

Enhancements:

- **smaller certificates**
- certificates not kept in memory
- better analysis of dependences

Scaling:

- big developments (Flyspeck?)

Thanks

http://www.lix.polytechnique.fr/~keller/Recherche/hollightcoq.html

http://hol-light.googlecode.com/svn/trunk

Thank you for your attention!

Any questions?

Types

```
Inductive type :  Type :=
    | B : type
    | A : type -> type -> type
```

Types

```
Inductive type :  Type :=
    | B : type
    | A : type -> type -> type
    | TVar :  string -> type
```

- Type schemes (polymorphism)

Types

```
Inductive type :  Type :=
    | B : type
    | A : type -> type -> type
    | TVar :  string -> type
    | TDef :  string -> list type -> type.
```

- Type definitions: schema of specification
  - `T` existing type
  - `P: A T B`
  - `T' = {x:  T | P x}`
- Easy to translate to concrete Coq types

Terms

```
Inductive term :  Type :=
    | Var:  string -> term
    | Lam:  string -> type -> term -> term
    | App:  term -> term -> term
```

Terms

```
Inductive term :  Type :=
    | Var:  string -> term
    | Lam:  string -> type -> term -> term
    | App:  term -> term -> term
    | Eq:  type -> term
    | Eps:  type -> term
```

- Equality and Hilbert's epsilon

Terms

```
Inductive term :  Type :=
    | Var:  string -> term
    | Lam:  string -> type -> term -> term
    | App:  term -> term -> term
    | Eq:  type -> term
    | Eps:  type -> term
    | Def:  string -> type -> term.
```

- Term definitions: constants
- Easy to translate to concrete Coq terms