PML : A new proof assistant and deduction system

Christophe Raffalli

LAMA

PLMMS 2007
Motivation

Drawback of current proof assistants

- Limited module system
- Equationnaal reasoning difficult
- Limited expressive power (compared to ZF)
- In general, no good integration between the proof assistant and the programming language
- Different languages (often 3-4)
- Much harder to learn the programming languages
Idea: start from a programming language

Programming when doing proof:
- To write tactics
- To prove programs
- In math: a lot of algorithm

Design choices:
- Start from a programming language (ML like)
- Turn it into a deduction system.
- HOL is build this way from simply typed λ-calculus
- Do this as best as we can!
Idea: start from a programming language

Programming when doing proof:
- To write tactics
- To prove programs
- In math: a lot of algorithm

Design choices:
- Start from a programming language (ML like)
- Turn it into a deduction system.
- HOL is build this way from simply typed \( \lambda \)-calculus
- Do this as best as we can!
Idea: start from a programming language

Programming when doing proof:
- To write tactics
- To prove programs
- In math: a lot of algorithm

Design choices:
- Start from a programming language (ML like)
- Turn it into a deduction system.
- HOL is build this way from simply typed $\lambda$-calculus
- Do this as best as we can!
Outline

1 Brief Description of PML’s Language

2 Other ideas and consequences
   - bool = prop (propositions as programs)
   - No Abstract Type
   - Restricted Inductive Type
   - Termination Check

3 Judgment and proofs
   - Three sorts of judgment?
   - Proofs as Programs

4 Conclusion
Types as programs

PML’s type system
Based on a new dedicated constraint consistency check algorithm (polynomial)

Problem:
Types are too complex for users
Types as programs

PML’s type system
Based on a new dedicated constraint consistency check algorithm (polynomial)

Solution:
- See the type system as a black box
- Recover types from programs:
  Types are partial identity maps.
Types as programs

PML’s type system

Based on a new dedicated constraint consistency check algorithm (polynomial)

Example

type nat = [ Z[] | S[nat] ]
val x : nat = ...

means

let rec nat = fun Z[] -> Z[]
  | S[n] -> S[nat n]
val x = nat (...
How does it looks like

Better than ML?

- Polymorphic variant (with subtyping and inheritance)
- Records (with subtyping and inheritance)
- Tuples, modules and object encoded using records
- Functors encoded as functions
- Open on records
- Exceptions and errors
- No type annotation needed (excepted for open and multiple inheritance)
- ML like polymorphism
Few examples

(* two classes encoded using records *)
val point pos = {
    val p = pos
    val move self = { self with
        val p = match self.p with
            P[x] -> x
            | x -> S[x]}
    |
    x -> S[x]}
}

val bpoint pos = {
    include point pos
    val back self = { self with
        val p = match self.p with
            S[x] -> x
            | x -> P[x]}
    |
    x -> P[x]}
}
Brief Description of PML's Language

Other ideas and consequences

Judgment and proofs

Conclusion

Few examples

(* nat subtype of int, with unique representation *

type rec nat = [ End[] | Zero[nat’] | One[nat] ]

and nat’ = [ One[nat] | Zero[nat’] ]

val rec succ : (nat => nat’) = fun
  Zero[x] -> One[x]
  | One[x] -> Zero[ (succ x) ]
  | End[] -> One[End[]]

val rec pred : (nat => nat’) = ...

type int = [ nat | Minus[nat’] ]

val succ:(int=>int) = fun
  Minus[n] -> opp(pred n)
  | n -> succ n
Few examples

(* red black trees as a subtype of trees *)

type rec tree (A) = [ Nil[] | Node[tree A * A * tree A] ]

type rec red_black_tree (A) = [ Nil[] | Node[red_black_tree A * A * red_black_tree A with val color : [Red[] | Black[]]] ]
The logic is part of the language

Avoid duplication
- Booleans can be defined in PML
- Propositions are needed in a deduction system

Identify them? Why not?
A lot of consequences...
Consistency

**Problem**

Consistency is easier to lose when $\text{bool} = \text{prop}$

**First step toward a solution:**

Interpretation using sets:

- Types as sets
- Function types as the set of all functions
- Record types as products
- Variant types as sums

... Why not?

More problems ...
Abstract type are inconsistent

Abstract type implies existential (this is not enough)

\{ \text{type } t; \text{ val } x : t; \ldots \} \simeq \exists t.(t \times \ldots)

A fact:

\exists \text{ and } \forall \text{ over types in types are inconsistent in HOL}

Existential type unnecessary!

- We have specification to replace them
- With existential type one can not extend library
Abstract type are inconsistent

Abstract type implies existential (this is not enough)

\{ \text{type } t; \ \text{val } x : \ t; \ \ldots \ \} \simeq \exists t.(t \times \ldots )

A fact:

∃ and ∀ over types in types are inconsistent in HOL

Existential type unnecessary!

- We have specification to replace them
- With existential type one can not extend library
Abstract type are inconsistent

Abstract type implies existential (this is not enough)
\[
\{ \text{type } t; \ \text{val} \ x : \ t; \ \ldots \} \simeq \exists t. (t \times \ldots)
\]

A fact:
\exists \text{ and } \forall \text{ over types in types are inconsistent in HOL}

Existential type unnecessary!

- We have specification to replace them
- With existential type one can not extend library
Inductive type are inconsistent

Same fact:
\( \mu \) and \( \nu \) (fix-points) over types are inconsistent in HOL

Solution:
No solution using sets for:

\[ \alpha = \alpha \rightarrow \beta \]
Inductive type are inconsistent

**Same fact:**

\( \mu \) and \( \nu \) (fix-points) over types are inconsistent in HOL

**Solution:**

No solution using sets for:

\[
\alpha \supset \alpha \rightarrow \beta
\]

But solutions for

\[
\alpha \subset \alpha \rightarrow \beta
\]
Inductive type are inconsistent

Same fact:
\( \mu \) and \( \nu \) (fix-points) over types are inconsistent in HOL

Solution:
From typing constraints, construct *interpreted after* order \((\succ, \succeq)\)

\[
\begin{align*}
\alpha \supset \beta & \implies \alpha \succeq \beta \\
\alpha \supset \beta \to \gamma & \implies \alpha \succ \beta, \gamma \\
\alpha \supset \beta \times \gamma & \implies \alpha \succeq \beta, \gamma \\
\alpha \subset \beta \to \gamma & \implies \top \\
\alpha \subset \beta \times \gamma & \implies \top
\end{align*}
\]

Reject program if \( \succ \) is cyclic
Object encoding is preserved!
Fixpoint of negation:

An inconsistency:

```ml
val not A = match A with
    True[] -> False[]
  | False[] -> True[]
val rec A = not A
```

Solution:

In proofs, propositions must be terminating:

- Implement a termination check
- When this test fails infer Loop as a possible error
- Enforce that proof terms do not trigger Loop
What to prove?

We need three sorts of judgment:

- Truth of proposition
- Type reinforcement
- Termination
What to prove?

We need three sorts of judgment:
- Truth of proposition
- Type reinforcement
- Termination

One sort is enough:
In record we may have:

\[
\text{prop ident : expression} \gg \text{value} \\
\text{proof ...}
\]

- \textit{expression}: any PML's expression
- \textit{value}: a pattern
An example

val rec add_zero_right x = {
  prop eq : eq_nat (add x Z[]) x >> True[]
  proof match x with
    Z[] -> True[]
    | S[x'] -> use (add_zero_right x').eq in True[]

val rec add_succ_right x y = ...

val rec add_commutative x y = {
  prop eq : eq_nat (add x y) (add y x) >> True[]
  proof match x y with
    Z[] -> use (add_zero_right x).eq in True[]
    | S[x'] ->
      open add_succ_right x' y in
      use (add_commutative x' y).eq in True[]


An example illustrated

```ML
val rec add_zero_right x = {
  prop eq : eq_nat (add x Z[]) x >> True[]
proof
  [* |- eq_nat (add x Z[]) x >> True[] *]
}
```
An example illustrated

val rec add_zero_right x = {
  prop eq : eq_nat (add x Z[]) x >> True[]
proof match x with
  Z[] ->
  [* x >> Z[]
    |- eq_nat (add x Z[]) x >> True[] *]
  S[x'] ->
  [* x >> S[x']
    |- eq_nat (add x Z[]) x >> True[] *]
}
val rec add_zero_right x = {
prop eq : eq_nat (add x Z[]) x >> True[]
proof match x with
  Z[] ->
  [* x >> Z[]
    |- True[] >> True[] *]
| S[x'] ->
  [* x >> S[x']
    |- eq_nat (add x Z[]) x >> True[] *]
}
val rec add_zero_right x = {
  prop eq : eq_nat (add x Z[]) x >> True[
  proof match x with
    Z[] -> True[
    | S[x'] ->
      [* x >> S[x']
      | - eq_nat (add x Z[]) x >> True[] *]
  }
}
An example illustrated

```ml
val rec add_zero_right x = {
  prop eq : eq_nat (add x Z[]) x >> True[]
  proof match x with
    Z[] -> True[]
  | S[x'] ->
    [* x >> S[x']
      |- eq_nat (add x' Z[]) x' >> True[] *]
}
```
An example illustrated

```ocaml
val rec add_zero_right x = {
  prop eq : eq_nat (add x Z[]) x >> True[]
  proof match x with
    Z[] -> True[]
    | S[x'] ->
      use (add_zero_right x').eq in True[]
  }
```
Proving termination

val f x =
  (* termination check fails for f *)
let rec f y = ... in
  (* proof that f terminates *)
let rec ft y = {
    prop fr : f y >> z
    proof ... (* proof that f terminates *)
    val result = z
  }
  in (ft x).result
Exceptions in proof

(* example to illustrate "let try" rather than "try" to do case analysis between exceptional and normal values *)

val lemma1 = {
  prop th : try u
    with e -> v >> True[

proof
  [* try u with e -> v >> True[ ] *]
}


Exceptions in proof

(* example to illustrate "let try" rather than "try" to do case analysis between exceptional and normal values *)

val lemma1 = {
    prop th : let try x = u in x
               with e -> v >> True[]
       proof
           [* let try x = u in x
                with e -> v >> True[] *
       }
}
Exceptions in proof

(* example to illustrate "let try" rather than "try" to do case analysis between exceptional and normal values *)

val lemma1 = {
  prop th : let try x = u in x
  with e -> v >> True[]

proof
  let try x = u in
  [* u >> x
    |- u >> True[] *]
  with e ->
  [* u >> raise e
    |- v >> True[] *]
}


The dependent type problem

val F (M : \x:nat -> \{ prop th : P x >> True[\] \}) =
{ prop th : and (P 2) (P 3) >> True[]
proof
  open M 2 in open M 3 in True[]
}
val S = F S'

Who is in charge of checking that S' is OK for F

PML’s typing can not.
The dependent type problem

val F (M : \x:nat -> { prop th : P x >> True[] }) = 
{ 
    prop th : and (P 2) (P 3) >> True[]
    proof 
        open M 2 in open M 3 in True[]
    }
val S = F S'

Who is in charge of checking that $S'$ is OK for $F$
Check it at runtime: Works but bad!
The dependent type problem

val F (M : $\forall x : \text{nat} \rightarrow \{ \text{prop th} : P x \implies \text{True} \}) = \{
    \text{prop th} : \text{and} (P 2) (P 3) \implies \text{True}[]
    \text{proof}
    \quad \text{open M 2 in open M 3 in True[]} \}
val S = F S'

Who is in charge of checking that $S'$ is OK for $F$

Check the all program as if it where a proof
Proving without a proof

Example

val rec even = fun
  Z[] -> True[]
| S[Z[]] -> False[]
| S[S[x]] -> even x

type even_nat =
  { nat as x with prop is_even |- even x }

val rec half x : even_nat = match x with
  Z[] -> True[]
| S[S[y]] -> open x in half y
| S[Z[y]] -> [* *]
What’s next?

- Finish to implement proof checking
- Quantification via choice operators and exceptions
- Infer computability
- Leibniz and computable equality
- Termination check (In progress by Marc Lasson)
- Macros and tactics
- Interface
- Extensible grammar (Using dypgen by Emmanuel Onzon)
- Theoretical strength and relation with Quine’s NF and Jensen’s NFU
- Compilation of PML
- ... Bootstrap! (actual 11208 loc, estimated total ¡30000 loc)
Help or ideas are welcome

Inspiration from
- Alfa/Agda
- Phd of S. Baro (directed by P. Manoury)
- Boyer-Moore (nqthm/acl2)

Other contributors
- Emmanuel Onzon (dypgen)
- Pierre Hyvernat (doc, grammar, ...)
- Marc Lasson (termination check)

URL
www.lama.univ-savoie.fr/~raffalli/pml