A Call-by-Value Realizability Model with Equivalence (and Subtyping) for PML

Rencontre CHoCoLa du 03/12/2015 - ENS Lyon

Rodolphe Lepigre - Université Savoie Mont Blanc
What does the acronym PML stands for?

Obviously, ML stands for ML.

We are not so sure about the P yet...

Some ideas:
- pedestrian,
- perverted,
- phantasmagoric,
- pleasurable,
- presumptuous,
- ...

PML is a programming language

PML is similar to OCaml or SML:
- call-by-value evaluation,
- ML-like polymorphism,
- Curry-style syntax (no types in terms),
- effects.

Example of program:

```ml
type rec nat = Zero | Succ of nat

val rec add n m =
  match n with
  | Zero    -> n
  | Succ[n'] -> Succ[add n' m]
```
PML is a proof system

The mechanism for program proving relies on:
- dependent product type ($\Pi$-type),
- equational reasoning (equivalence of programs).

The system follows the “program as proof” principle.
(As opposed to the “proof as program” principle.)

Ultimate goal: formalization of mathematics (untyped terms as objects).
Why another proof system?

We want a programing language centered system:
- an efficient, convenient programming language (ML),
- in which properties of programs can be proved (occasionally),
- in the same (programming) language.

Proofs can be composed with programs (i.e. tactics).

Other systems:
- in Coq the proof-terms are hidden behind tactics,
- in Agda the syntax of proof-terms is limited,
- in HOL light, HOL, Isabelle/HOL there are no proof-terms,
- in Why3 proofs are not programs.
PART 1

The type system of PML
Starting point: ML

Three base types:
- function type \( A \Rightarrow B \),
- product (record) type \( \{ l_1 : A_1, \ldots, l_n : A_n \} \),
- sum (variant) type \( [C_1 \text{ of } A_1 | \ldots | C_n \text{ of } A_n ] \),
- \( \{ \} \) and \( [] \) are “unit” and the empty type.

Effects:
- syntax of the \( \lambda \mu \)-calculus (\( \mu \alpha \ t, [\alpha]t \)),
- access to the evaluation context,
- future work: references.

Polymorphism (universal quantifier).
\[
\lambda x \lambda y \{ \text{fst} = x; \text{snd} = y \} : \forall X \forall Y (X \rightarrow Y \rightarrow \{ \text{fst} : X; \text{snd} : Y \})
\]
Terms as individuals

Equality types $t \equiv u$ and $t \not\equiv u$:
- interpreted with observational equivalence,
- $t$ and $u$ are (possibly untyped) terms,
- these types are equivalent to $\{}$ when the equivalence is true
- and to $[]$ when it is false.

First-order quantification:

$\Gamma \vdash v : A \quad a \not\in FV(\Gamma) \quad \Gamma \vdash t : \forall a \ A$

$\Gamma \vdash v : \forall a \ A$

$\Gamma \vdash t : A[a := u]$

Example:

$\vdash : \forall n \ (\text{Succ } n \not\equiv \text{Zero})$. 
Working with equality

Automatic decision procedure for $t \equiv u$:

- not decidable since $(\equiv)$ contains function extensionality,
- the term $\triangleleft$ can be introduced when an equivalence can be derived.

$$
\frac{}{\Gamma \vdash t \equiv u} \quad \frac{}{\Gamma \vdash \triangleleft : t \equiv u} \quad \frac{}{\Gamma \vdash t \not\equiv u} \quad \frac{}{\Gamma \vdash \triangleleft : t \not\equiv u}
$$

Example:

$$
\frac{}{\vdash \text{add Zero } x \equiv x} \quad \frac{}{\vdash \triangleleft : \text{add Zero } x \equiv x} \quad x \notin \text{FV}(\emptyset)
$$

$$
\frac{}{\vdash \triangleleft : \forall x \ (\text{add Zero } x \equiv x)}
$$
Dependent product type

We want to be able to prove properties of typed terms.

The system includes a $\Pi$-type.

$$
\begin{align*}
\Gamma, x : A & \vdash t : B[a := x] & \Gamma \vdash \lambda x\ t : \Pi_{a:A}B \\
\Gamma & \vdash t : \Pi_{a:A}B & \Gamma \vdash \nu : A & \Gamma \vdash t\ \nu : B[a := \nu]
\end{align*}
$$

Example:

$$
\begin{align*}
x : \mathbb{N} & \vdash \text{add Zero } x \equiv x \\
\Gamma & \vdash \forall : \Pi_{n: \mathbb{N}} \text{ add Zero } n \equiv n
\end{align*}
$$

PML proof of $\Pi_{n: \mathbb{N}} \text{ add } n \text{ Zero } \equiv n$:

$$
Y\ \lambda r\ \lambda x\ \text{case } x\ \text{of Zero } \rightarrow \forall \mid \text{Succ } y \rightarrow r\ y
$$
Soundness issue

Care should be taken when combining:
- call-by-value evaluation,
- side-effects (references, control operators...),
- polymorphism.

The problem extends to the \( \Pi \)-type.

Some typing rules cannot be proved safe:

\[
\begin{align*}
\Gamma \vdash t : A & \quad X \notin \text{FV}(\Gamma) \\
\text{then} \quad \Gamma \vdash t : \forall X \ A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : \Pi_{a:A} B & \quad \Gamma \vdash u : A \\
\text{then} \quad \Gamma \vdash t \ u : B[a := u]
\end{align*}
\]
Counter-example

If we extend a pure ML language with references:

```ml
val ref : 'a -> 'a ref
val (!) : 'a ref -> 'a
val (:=) : 'a ref -> 'a -> unit
```

The following program is accepted:

```ml
let r = ref [] in
r := [true];
42 + (List.hd !r)
```

A more complex counter-example is required with control operators.
Value restriction

The problem can be solved by restricting some rules to values:

\[
\frac{\Gamma \vdash v : A \quad X \notin \text{FV}(\Gamma)}{\Gamma \vdash v : \forall X \ A}_{v\text{value}} \quad \frac{\Gamma \vdash t : \Pi_\alpha : A \ B \quad \Gamma \vdash v : A}{\Gamma \vdash t \ v : B[\alpha := v]}_{v\text{value}}
\]

Equivalently we may consider having two forms of judgements:
- \( \Gamma \vdash t : A \) where \( t \) is any term (maybe a value),
- \( \Gamma \vdash^\text{val} v : A \) where \( v \) can only be a value.

The rules become the following.

\[
\frac{\Gamma \vdash^\text{val} v : A \quad X \notin \text{FV}(\Gamma)}{\Gamma \vdash^\text{val} v : \forall X \ A}_{v\text{value}} \quad \frac{\Gamma \vdash t : \Pi_\alpha : A \ B \quad \Gamma \vdash^\text{val} v : A}{\Gamma \vdash t \ v : B[\alpha := v]}_{v\text{value}}
\]

Remark: we need an extra rule:

\[
\frac{\Gamma \vdash^\text{val} v : A}{\Gamma \vdash v : A}.
\]
Is value restriction satisfactory?

We can cope with value restriction for polymorphism.

Value restriction is too restrictive on the \( \Pi \)-type.

\[
\begin{align*}
\Gamma & \vdash t : \Pi_{a:A} B \\
\Gamma & \mid_{\text{val}} v : A \\
\Gamma & \vdash t\,v : B[a := v]
\end{align*}
\]

We cannot apply \( \lambda x \, \infty : \Pi_{n: \mathbb{N}} \text{add Zero } n \equiv n \) to \( 2 \times 21 \) (which is not a value).

We need to relax value restriction:

\[
\begin{align*}
\Gamma, u \equiv v & \vdash t : \Pi_{a:A} B \\
\Gamma, u \equiv v & \vdash u : A \\
\Gamma, u \equiv v & \vdash t\,u : B[a := u]
\end{align*}
\]

Remark: we do not encode \( A \Rightarrow B \) using the \( \Pi \)-type.
Part 2

A realizability model for PML
Syntax and Krivine machine

Values, terms and stacks:

\[ v, w ::= x \mid \lambda x \ t \mid C[v] \mid \{l_i = v_i\}_{i \in I} \mid \pi \]
\[ t, u ::= \alpha \mid v \mid tu \mid \mu \alpha t \mid [\pi] t \mid v. l \mid \text{case } v \text{ of } [C_i[x] \rightarrow t_i]_{i \in I} \]
\[ \pi ::= \alpha \mid v \cdot \pi \mid [t] \pi \]

The state of the machine is a process \( t \ast \pi \).
Operational semantics

\[(t \ u) \ast \pi > u \ast [t] \pi\]
\[\nu \ast [t] \pi > t \ast \nu \cdot \pi\]
\[(\lambda x \ t) \ast \nu \cdot \pi > t[x \leftarrow \nu] \ast \pi\]
\[(\mu \alpha \ t) \ast \pi > t[\alpha \leftarrow \pi] \ast \pi\]
\[[\pi]t \ast \rho > t \ast \pi\]

\[\text{case } C_k[\nu] \text{ of } [C_i[x] \rightarrow t_i]_{i \in I} \ast \pi > t_k[x \leftarrow \nu] \ast \pi\]
\[\{ l_i = \nu_i \}_{i \in I} \cdot l_k \ast \pi > \nu_k \ast \pi\]
Interpretation of types

Three levels of interpretation:
- raw semantics $\llbracket A \rrbracket$,
- falsity value $\llbracket A \rrbracket = \{ \pi \mid \forall \nu \in \llbracket A \rrbracket , \nu * \pi \in \bot \}$,
- truth value $\| A \| = \{ t \mid \forall \pi \in \llbracket A \rrbracket , t * \pi \in \bot \}$.

Here, $\bot$ is a set of well-behaved processes.

$$\bot = \{ t * \pi \mid \exists \nu \in \mathcal{V}, t * \pi \succ^* \nu * \epsilon \}$$
Raw semantics

\[
\begin{align*}
\llbracket A \Rightarrow B \rrbracket & := \{ \lambda x \; t \mid \forall v \in \llbracket A \rrbracket \; t[x := v] \in |B|\} \\
\llbracket \{l_i : A_i\}_{i \in I} \rrbracket & := \{ \{l_i = v_i\}_{i \in I} \mid \forall i \in I, v_i \in \llbracket A_i \rrbracket\} \\
\llbracket [C_i \text{ of } A_i]_{i \in I} \rrbracket & := \bigcup_{i \in I} \{C_i[v] \mid v \in \llbracket A_i \rrbracket\} \\
\llbracket \forall a \; A \rrbracket & := \bigcap_{t \in \Lambda_c} [A[a := t]] \\
\llbracket \exists a \; A \rrbracket & := \bigcup_{t \in \Lambda_c} [A[a := t]] \\
\llbracket t \equiv u \rrbracket & := \{\{\}\} \text{ when } t \equiv u \text{ and } \llbracket \{\}\rrbracket = \emptyset \text{ otherwise} \\
\llbracket t \in A \rrbracket & := \{v \in \llbracket A \rrbracket \mid v \equiv t\}
\end{align*}
\]

Remark: the type \(\Pi_{a: A} B\) is encoded as \(\forall a \; (a \in A \Rightarrow B)\).
Soundness

**Theorem (Adequacy Lemma):**
- if $t$ is a term such that $\vdash t : A$ then $t \in |A|$, 
- if $v$ is a value such that $\vdash_{\text{val}} v : A$ then $v \in \llbracket A \rrbracket$.

Remark: $\llbracket A \rrbracket \subseteq |A|$ by definition.

Intuition: a typed program behaves well (in any well-typed evaluation context).
Observational equivalence

Two programs are equivalent if they behave the same on every input.

We define the equivalence of $t$ and $u$ as:
\[
\forall \pi \ t * \pi \text{ behaves well } \iff u * \pi \text{ behaves well}.
\]

Required properties for the equivalence:
- extensionality (if $v \equiv w$ then $t[x := v] \equiv t[x := w]$),
- if $v \in \llbracket A \rrbracket$ and $v \equiv w$ then $w \in \llbracket A \rrbracket$.

\[
\frac{\Gamma, v \equiv w \vdash t[x := v] : A}{\Gamma, v \equiv w \vdash t[x := w] : A}
\]

\[
\frac{\Gamma, v \equiv w \vdash t : A[x := v]}{\Gamma, v \equiv w \vdash t : A[x := w]}
\]
Implementation of the decision procedure

We derive rules from the definition of ($\equiv$):
- $(\lambda x \, t) \, v \equiv t[x := v]$,
- {...l = v...}.l $\equiv$ v,
- C[v] $\not\equiv$ D[w] if C $\not\equiv$ D,
- ...

Pseudo-decision algorithm for equivalence:
- efficiency is critical (bottleneck in first implementation),
- data structure: graph with maximal sharing (union find),
- proof by contradiction,
- we can only approximate equivalence,
- the user can help by giving hints.
Relaxing value restriction

With value restriction, some rules are restricted to values.

Idea: a term that is equivalent to a value may be considered a value.

Informal proof:

\[
\begin{align*}
\Gamma, t \equiv v & \vdash t : A \\
\Gamma, t \equiv v & \vdash v : A \\
\Gamma, t \equiv v & \vdash v : \forall a \ A \\
\Gamma, t \equiv v & \vdash t : \forall a \ A
\end{align*}
\]
Semantical value restriction

In every realizability model \([A] \subseteq |A|\).

This provides a semantical justification to the rule

\[
\frac{\Gamma \vdash v : A}{\Gamma \vdash v : A^{\uparrow}}.
\]

We need to have \(|A| \cap \mathcal{V} \subseteq [A]|\) to obtain the rule

\[
\frac{\Gamma \vdash v : A}{\Gamma \vdash v : A^{\downarrow}}.
\]

With this rule we can lift the value restriction to the semantics.

\[
\frac{\Gamma, t \equiv v \vdash t : A}{\Gamma, t \equiv v \vdash v : A^{\downarrow}}\quad \frac{\Gamma, t \equiv v \vdash v : A^{\uparrow}}{\Gamma, t \equiv v \vdash v : \forall a \ A}
\]

\[
\frac{\Gamma, t \equiv v \vdash v : \forall a \ A}{\Gamma, t \equiv v \vdash t : \forall a \ A}
\]
The new instruction trick

The property $|A| \cap \mathcal{V} \subseteq \llbracket A \rrbracket$ is not true in every realizability model.

To obtain it we extend the system with a new term constructor $\delta_{v,w}$.

We will have $\delta_{v,w} \cdot \pi \succ v \cdot \pi$ if and only if $v \not\equiv w$.

Idea of the proof:
- suppose $v \not\in \llbracket A \rrbracket$ and show $v \not\in |A|$,  
- we need to find $\pi$ such that $v \cdot \pi \not\in \bot$ and $\forall w \in \llbracket A \rrbracket$, $w \cdot \pi \in \bot$,  
- we can take $\pi = [\lambda x \delta_{x,v}]\varepsilon$,  
- $v \cdot [\lambda x \delta_{x,v}]\varepsilon \succ \lambda x \delta_{x,v} \cdot v.\varepsilon \succ \delta_{v,v} \cdot \varepsilon$,  
- $w \cdot [\lambda x \delta_{x,v}]\varepsilon \succ \lambda x \delta_{x,v} \cdot w.\varepsilon \succ \delta_{w,v} \cdot \varepsilon \succ w \cdot \varepsilon$. 
Stratified reduction and equivalence

**Problem:** the definitions of $(>)$ and $(\equiv)$ are circular.

We need to rely on a stratified construction of the two relations

$$(\Rightarrow_i) = (>) \cup \{(\delta_{v,w} \ast \pi, v \ast \pi) \mid \exists j < i, v \neq_j w\}$$

$$(\equiv_i) = \{(t, u) \mid \forall j \leq i, \forall \pi \in \Pi, \forall \sigma, t\sigma \ast \pi \downarrow_j \iff u\sigma \ast \pi \downarrow_j\}$$

We then take

$$(\Rightarrow) = \bigcup_{i \in \mathbb{N}} (\Rightarrow_i) \quad (\equiv) = \bigcap_{i \in \mathbb{N}} (\equiv_i)$$

With these definitions, $(\equiv)$ is indeed extensional...
Current and future work

Subtyping without coercions (almost finished):
- useful for programming (modules, classes...),
- provide injections between types for free,
- judgement $\vdash A \subseteq B$ interpreted as $\llbracket A \rrbracket \subseteq \llbracket B \rrbracket$ in the semantics.

Recursion and (co-)inductive types (in progress):
- the types $\mu X A$ and $\forall X A$ will be handled by subtyping,
- we need to extend the language with a fixpoint,
- termination needs to be ensured to preserve soundness.

Theoretical investigation (for later):
- can we use $\delta_{v,w}$ to realize new formulas,
- how do we encode real maths in the system?
THANK YOU!

HTTP://PATOLINE.ORG