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

Logic and Semantics Seminar - Cambridge 26/02/2016

Rodolphe Lepigre - Université Savoie Mont Blanc

What does PML stands for?

Obviously, ML stands for ML.

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

Some ideas:

- pedestrian,
- perverted,
- phantasmagoric,
- pleasurable,
- presumptuous,
- ...

Full list at *http://adjectivesstarting.com/with-p/*.

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:

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

val rec add n m =
  match n with
  | Zero    -> m
  | Succ nn -> Succ (add nn m)
```

PML is a proof system

The mechanism for program proving relies on:

- equational reasoning (equivalence of programs),
- dependent product type (Π-type).

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 $[C_1 \text{ of } A_1 \mid \ldots \mid C_n \text{ of } A_n]$,
- $\{\}$ 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).

$$\lambda x\, \lambda y\, \{fst = x\,;\, snd = y\} : \forall X\, \forall Y\, (X \to Y \to \{fst : X;\, snd : Y\})$$

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:

$$\frac{\Gamma \vdash v : A \qquad a \notin FV(\Gamma)}{\Gamma \vdash v : \forall a\ A} \qquad\qquad \frac{\Gamma \vdash t : \forall a\ A}{\Gamma \vdash t : A[a := u]}$$

Example:

$$- : \forall n\ (\mathrm{Succ}\, n \not\equiv \mathrm{Zero}).$$

Working with equality

Automatic decision procedure for $t \equiv u$:

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

$$\frac{\Gamma \vdash t \equiv u}{\Gamma \vdash ✄ : t \equiv u} \qquad \frac{\Gamma \vdash t \not\equiv u}{\Gamma \vdash ✄ : t \not\equiv u}$$

Example:

$$\frac{\dfrac{\vdash \text{add Zero}\, x \equiv x}{\vdash ✄ : \text{add Zero}\, x \equiv x} \qquad x \notin FV(\emptyset)}{\vdash ✄ : \forall x\ (\text{add Zero}\, x \equiv x)}$$

Dependent product type

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

The system includes a Π-type.

$$\frac{\Gamma, x : A \vdash t : B[a := x]}{\Gamma \vdash \lambda x\, t : \Pi_{a:A} B} \qquad \frac{\Gamma \vdash t : \Pi_{a:A} B \quad \Gamma \vdash v : A}{\Gamma \vdash t\, v : B[a := v]}$$

Example:

$$\frac{\dfrac{x : \mathbb{N} \vdash \text{add Zero}\, x \equiv x}{x : \mathbb{N} \vdash ✄ : \text{add Zero}\, x \equiv x}}{\vdash \lambda x\, ✄ : \Pi_{n:\mathbb{N}}\ \text{add Zero}\, n \equiv n}$$

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

$$Y\, \lambda r\, \lambda x\, \text{case}\, x\, \text{of Zero} \to ✄ \mid \text{Succ}\, y \to r\, y$$

Soundness issue

Care should be taken when combining:

- call-by-value evaluation,
- side-effects (references, control operators...),
- polymorphism.

The problem extends to the Π-type.

Some typing rules cannot be proved safe:

$$\frac{\Gamma \vdash t : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X\ A} \qquad \frac{\Gamma \vdash t : \Pi_{a:A} B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[a := u]}$$

Counter-example

If we extend a pure ML language with references:

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

The following program is accepted:

```
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:

$$\frac{\Gamma \vdash v : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash v : \forall X\ A}\, v \text{ value} \qquad \frac{\Gamma \vdash t : \Pi_{a:A} B \qquad \Gamma \vdash v : A}{\Gamma \vdash t\, v : B[a := v]}\, v \text{ value}$$

Equivalently we may consider having two forms of judgements:

- $\Gamma \vdash t : A$ where t is an arbitrary term (maybe a value),
- $\Gamma \vdash_{\text{val}} v : A$ where v can only be a value.

The rules become the following.

$$\frac{\Gamma \vdash_{\text{val}} v : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash_{\text{val}} v : \forall X\ A} \qquad \frac{\Gamma \vdash t : \Pi_{a:A} B \qquad \Gamma \vdash_{\text{val}} v : A}{\Gamma \vdash t\, v : B[a := v]}$$

Remark: we need an extra rule: $\dfrac{\Gamma \vdash_{\text{val}} v : A}{\Gamma \vdash v : A}$.

Is value restriction satisfactory?

We can cope with value restriction for polymorphism.

Value restriction is too restrictive on the Π-type.

$$\frac{\Gamma \vdash t : \Pi_{a:A} B \qquad \Gamma \vdash_{\text{val}} v : A}{\Gamma \vdash t\, v : B[a := v]}$$

We cannot apply λx ✄ $: \Pi_{n:\mathbb{N}}\, \text{add Zero}\, n \equiv n$ to 2×21 (which is not a value).

We need to relax value restriction:

$$\frac{\Gamma, u \equiv v \vdash t : \Pi_{a:A} B \qquad \Gamma, u \equiv v \vdash u : A}{\Gamma, u \equiv v \vdash t\, u : B[a := u]}$$

Remark: we do not encode $A \Rightarrow B$ using the Π-type.

Part 2

A realizability model for PML

Syntax and Krivine machine

Values, terms and stacks:

$$
\begin{aligned}
v, w &::= x \mid \lambda x\, t \mid C[v] \mid \{l_i = v_i\}_{i \in I} \mid ✂ \\
t, u &::= a \mid v \mid t\, u \mid \mu\alpha\, t \mid [\pi]t \mid v\,.\,l \mid \text{case}\, v \,\text{of}\, [C_i[x] \to t_i]_{i \in I} \\
\pi &::= \alpha \mid v \cdot \pi \mid [t]\,\pi
\end{aligned}
$$

The state of the machine is a process $t * \pi$.

Operational semantics

$$
\begin{aligned}
t\,u * \pi &\succ u * [t]\,\pi \\
v * [t]\,\pi &\succ t * v \cdot \pi \\
(\lambda x\, t) * v \cdot \pi &\succ t[x \leftarrow v] * \pi \\
\mu\alpha\, t * \pi &\succ t[\alpha \leftarrow \pi] * \pi \\
[\pi]t * \rho &\succ t * \pi \\
\text{case}\, C_k[v]\, \text{of}\, [C_i[x] \rightarrow t_i]_{i \in I} * \pi &\succ t_k[x \leftarrow v] * \pi \\
\{l_i = v_i\}_{i \in I}.\, l_k * \pi &\succ v_k * \pi
\end{aligned}
$$

Interpretation of types

Three levels of interpretation:

- raw semantics $[\![A]\!]$,
- falsity value $\|A\| = \{\pi \mid \forall v \in [\![A]\!], v * \pi \in \perp\!\!\!\perp\}$,
- truth value $|A| = \{t \mid \forall \pi \in \|A\|, t * \pi \in \perp\!\!\!\perp\}$.

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

$$\perp\!\!\!\perp = \{t * \pi \mid \exists v \in \mathcal{V}, t * \pi \succ^* v * \varepsilon\}$$

Raw semantics

$$
\begin{aligned}
[\![A \Rightarrow B]\!] &:= \{\lambda x\, t \mid \forall v \in [\![A]\!]\ t[x := v] \in |B|\}\\
[\![\{l_i : A_i\}_{i \in I}]\!] &:= \left\{\{l_i = v_i\}_{i \in I} \mid \forall i \in I,\, v_i \in [\![A_i]\!]\right\}\\
[\![[C_i \text{ of } A_i]_{i \in I}]\!] &:= \cup_{i \in I}\{C_i[v] \mid v \in [\![A_i]\!]\}\\
[\![\forall a\ A]\!] &:= \cap_{t \in \Lambda_c}[\![A[a := t]]\!]\\
[\![\exists a\ A]\!] &:= \cup_{t \in \Lambda_c}[\![A[a := t]]\!]\\
[\![t \equiv u]\!] &:= [\![\{\}]\!] \text{ when } t \equiv u \text{ and } [\![[]]\!] = \emptyset \text{ otherwise}\\
[\![t \in A]\!] &:= \{v \in [\![A]\!] \mid v \equiv t\}
\end{aligned}
$$

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 $\vdash_{val} v : A$ then $v \in [\![A]\!]$.

Remark: $[\![A]\!] \subseteq |A|$ 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:

$$\forall \pi \;\; t * \pi \text{ behaves well } \Leftrightarrow u * \pi \text{ behaves well}.$$

Required properties for the equivalence:

- extensionality (if $v \equiv w$ then $t[x := v] \equiv t[x := w]$),
- if $v \in [\![A]\!]$ and $v \equiv w$ then $w \in [\![A]\!]$.

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

Implementation of the decision procedure

We derive rules from the definition of $(\equiv)$:

- $(\lambda x\, t)\, v \equiv t[x := v]$,
- $\{\dots l = v \dots\}.l \equiv v$,
- $C[v] \not\equiv D[w]$ if $C \neq D$,
- ...

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:

$$\dfrac{\dfrac{\dfrac{\Gamma, t \equiv v \vdash t : A}{\Gamma, t \equiv v \vdash v : A} \quad a \notin FV(\Gamma)}{\Gamma, t \equiv v \vdash v : \forall a\ A}}{\Gamma, t \equiv v \vdash t : \forall a\ A}$$

Semantical value restriction

In every realizability model $[\![A]\!] \subseteq |A|$.

This provides a semantical justification to the rule $\dfrac{\Gamma \vdash_{\text{val}} v : A}{\Gamma \vdash v : A}\uparrow$.

We need to have $|A| \cap \mathscr{V} \subseteq [\![A]\!]$ to obtain the rule $\dfrac{\Gamma \vdash v : A}{\Gamma \vdash_{\text{val}} v : A}\downarrow$.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, t \equiv v \vdash t : A}{\Gamma, t \equiv v \vdash v : A}\equiv}{\Gamma, t \equiv v \vdash_{\text{val}} v : A}\downarrow \quad a \notin FV(\Gamma)}{\Gamma, t \equiv v \vdash_{\text{val}} v : \forall a\ A}\forall_e}{\Gamma, t \equiv v \vdash v : \forall a\ A}\uparrow}{\Gamma, t \equiv v \vdash t : \forall a\ A}\equiv$$

The new instruction trick

The property $|A| \cap \mathscr{V} \subseteq [\![A]\!]$ 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 $\delta_{v,w} * \pi \succ v * \pi$ if and only if $v \not\equiv w$.

Idea of the proof:

- suppose $v \notin [\![A]\!]$ and show $v \notin |A|$,
- we need to find π such that $v * \pi \notin \perp\!\!\!\perp$ and $\forall w \in [\![A]\!], w * \pi \in \perp\!\!\!\perp$,
- we can take $\pi = [\lambda x\, \delta_{x,v}]\varepsilon$,
- $v * [\lambda x\, \delta_{x,v}]\varepsilon \succ \lambda x\, \delta_{x,v} * v.\varepsilon \succ \delta_{v,v} * \varepsilon$,
- $w * [\lambda x\, \delta_{x,v}]\varepsilon \succ \lambda x\, \delta_{x,v} * w.\varepsilon \succ \delta_{w,v} * \varepsilon \succ w * \varepsilon$.

Stratified reduction and equivalence

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

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

$$(\twoheadrightarrow_i) = (\succ) \cup \left\{(\delta_{v,w} * \pi, v * \pi) \mid \exists j < i, v \not\equiv_j w\right\}$$

$$(\equiv_i) = \left\{(t, u) \mid \forall j \leqslant i, \forall \pi \in \Pi, \forall \sigma, t\sigma * \pi \Downarrow_j \Leftrightarrow u\sigma * \pi \Downarrow_j\right\}$$

We then take

$$(\twoheadrightarrow) = \bigcup_{i \in \mathbb{N}} (\twoheadrightarrow_i) \qquad (\equiv) = \bigcap_{i \in \mathbb{N}} (\equiv_i)$$

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 $[\![A]\!] \subseteq [\![B]\!]$ in the semantics.

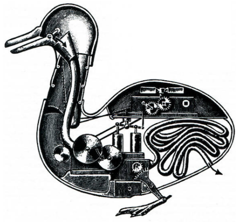
Recursion and (co-)inductive types (in progress):

- the types $\mu X\, A$ and $\nu X\, A$ 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