Theory and Demo of PML$_2$

Proving Programs in ML

TYPES2017 (Budapest 01/06/2017)

Rodolphe Lepigre
rodolphe.lepigre@univ-smb.fr
LAboratoire de MAthématiques, UMR 5127 CNRS

Programming Language, with Proving Features

An ML-like programming language:

- General recursion, records and variants
- Call-by-value evaluation
- Effects (control operators)
- Curry-style language
- Subtyping

Programming Language, with Proving Features

An ML-like programming language:

- General recursion, records and variants
- Call-by-value evaluation
- Effects (control operators)
- Curry-style language
- Subtyping

An enriched type system for program proving:

- Higher-order layer with programs as individuals
- Equality types $t \equiv u$ (observational equivalence)
- Dependent function type (typed quantification)
- Termination checking (only required for proofs)

Example of Program and Proof

```
type rec nat = [Z ; S of nat]
val rec add : nat ⇒ nat ⇒ nat = fun n m →
  case n { Z[_] → m | S[k] → S[add k m] }
```

Example of Program and Proof

```
type rec nat = [Z ; S of nat]
val rec add : nat ⇒ nat ⇒ nat = fun n m →
  case n { Z[_] → m | S[k] → S[add k m] }

val add_Z_n : ∀n:ι , add Z n ≡ n = {}
```

Example of Program and Proof

```
type rec nat = [Z ; S of nat]
val rec add : nat ⇒ nat ⇒ nat = fun n m →
  case n { Z[_] → m | S[k] → S[add k m] }

val add_Z_n : ∀n:ι , add Z n ≡ n = {}

val rec add_n_Z : ∀n∈nat, add n Z ≡ n = fun n →
  case n {
    Z[_] → {}
    S[p] → add_n_Z p
  }
```

Detailed Proof Using (Higher-Order) Macros

```
def tac_deduce<f:ο> : τ = ({} : f)
def tac_show<f:ο, p:τ> : τ = (p : f)
def tac_qed : τ = {}
```

Detailed Proof Using (Higher-Order) Macros

```
def tac_deduce<f:ο> : τ = ({} : f)
def tac_show<f:ο, p:τ> : τ = (p : f)
def tac_qed : τ = {}

val rec add_n_Z : ∀n∈nat, add n Z ≡ n = fun n →
  case n {
    Z[_] → deduce add Z Z ≡ Z; qed
    S[k] → show add k Z ≡ k using add_n_Z k;
           deduce S[add k Z] ≡ S[k];
           deduce add S[k] Z ≡ S[k]; qed
  }
```

Fine-grained Specification Using Equivalence

```
val rec is_even : nat ⇒ bool = fun n →
  case n {
    Z[_] → true
    S[p] → case p { Z[_] → false | S[p] → is_even p }
  }
```

```
val rec is_even : nat ⇒ bool = fun n →
  case n {
    Z[_] → true
    S[p] → case p { Z[_] → false | S[p] → is_even p }
  }

type even_nat = ∃v:ι, (v∈nat | is_even v ≡ true)
```

```
val rec is_even : nat ⇒ bool = fun n →
  case n {
    Z[_] → true
    S[p] → case p { Z[_] → false | S[p] → is_even p }
  }

type even_nat = ∃v:ι, (v∈nat | is_even v ≡ true)

val rec double : nat ⇒ even_nat = fun n →
  case n {
    Z[_] → Z
    S[p] → let r : even_nat = double p in S[S[r]]
  }
```

More Examples of Specifications

```
type rec list<a> = [Nil ; Cons of {hd : a ; tl : list}]

// Vectors (as a subtype of lists)
val length : ∀a:ο, list<a> ⇒ nat = {- ... -}
type vec<a:ο, s:τ> = ∃l:ι, l∈list<a> | length l ≡ s

// Sorted lists (as a subtype of lists)
val increasing : list<nat> ⇒ bool = {- ... -}
type sorted_list = ∃l:ι, l∈list<nat> | increasing l ≡ true
```

Realisability is about computation:

- Call-by-value Krivine Machine (for classical logic)
- States of the form $t * \pi$ with a reduction relation $(\succ)$

Realisability is about computation:

- Call-by-value Krivine Machine (for classical logic)
- States of the form $t * \pi$ with a reduction relation $(\succ)$

We also require a notion of observational equivalence:

- We write $t * \pi \Downarrow$ for $\exists\, v, t * \pi \succ^* v * \varepsilon$ (successful computation)
- $(\equiv)$ is defined as $\{(t, u) \mid \forall\, \pi, \forall\, \rho, t\rho * \pi \Downarrow \Leftrightarrow u\rho * \pi \Downarrow\}$

Realisability is about computation:

- Call-by-value Krivine Machine (for classical logic)
- States of the form $t * \pi$ with a reduction relation $(\succ)$

We also require a notion of observational equivalence:

- We write $t * \pi \Downarrow$ for $\exists\, v, t * \pi \succ^* v * \varepsilon$ (successful computation)
- $(\equiv)$ is defined as $\{(t, u) \mid \forall \pi, \forall \rho, t\rho * \pi \Downarrow \Leftrightarrow u\rho * \pi \Downarrow\}$

A type A is interpreted using two sets (in fact three):

- The set of its "canonical" values $[\![A]\!]$
- A set of terms $[\![A]\!]^{\perp\!\!\!\perp}$ defined as a form of completion of $[\![A]\!]$
- Closure under $(\equiv)$ is required on those sets

Interpretation of the (Usual) Types

$$\left[\!\left[\{(l_i : A_i)_{i \in I}\}\right]\!\right] = \left\{\{(l_i = v_i)_{i \in I}\} \mid \forall\, i \in I,\, v_i \in [\![A_i]\!]\right\}$$

$$\left[\!\left[[(C_i : A_i)_{i \in I}]\right]\!\right] = \cup_{i \in I} \{C_i[v] \mid v \in [\![A_i]\!]\}$$

$$[\![A \Rightarrow B]\!] = \{\lambda x.t \mid \forall\, v \in [\![A]\!],\, t[x := v] \in [\![B]\!]^{\perp\!\!\perp}\}$$

$$[\![\forall \chi^s.A]\!] = \cap_{\Phi \in [\![s]\!]} [\![A[\chi := \Phi]]\!]$$

$$[\![\exists \chi^s.A]\!] = \cup_{\Phi \in [\![s]\!]} [\![A[\chi := \Phi]]\!]$$

$$[\![\mu_\tau X.A]\!] = \cup_{\kappa < \tau} (X \mapsto [\![A]\!])^\kappa (\emptyset)$$

$$[\![\nu_\tau X.A]\!] = \cap_{\kappa < \tau} (X \mapsto [\![A]\!])^\kappa (\Lambda_\iota)$$

A new membership type $\mathtt{t}{\in}A$:

- Built using a term $\mathtt{t}$ and a type A
- Denotes the equivalence class of $\mathtt{t}$ in A
- Interpreted as $[\![\mathtt{t}{\in}A]\!] = \{v \in [\![A]\!] \mid \mathtt{t} \equiv v\}$

Membership Type and Dependent Functions

A new membership type $t \in A$:

- Built using a term t and a type A
- Denotes the equivalence class of t in A
- Interpreted as $[\![t \in A]\!] = \{ v \in [\![A]\!] \mid t \equiv v \}$

Only way to link the "word of terms" and the "world of types"

A new membership type $t \in A$:

- Built using a term t and a type A
- Denotes the equivalence class of t in A
- Interpreted as $[\![t \in A]\!] = \{v \in [\![A]\!] \mid t \equiv v\}$

Only way to link the "word of terms" and the "world of types"

The dependent function type is encoded using membership:

- $\forall a \in A, B$ is defined as $\forall a.(a \in A \Rightarrow B)$
- Related to the relativised quantification scheme

A new restriction type $A \restriction P$:

- Built using a type A and a "semantic predicate" P
- $[\![A \restriction P]\!]$ is equal to $[\![A]\!]$ if P is satisfied and to $[\![\forall X.X]\!]$ otherwise
- Examples of predicates: $t \equiv u$, $\kappa \neq 0$, $A \subseteq B$, $\neg P$, $P \wedge Q$

A new restriction type $A \upharpoonright P$:

- Built using a type A and a "semantic predicate" P
- $[\![A \upharpoonright P]\!]$ is equal to $[\![A]\!]$ if P is satisfied and to $[\![\forall X.X]\!]$ otherwise
- Examples of predicates: $t \equiv u$, $\kappa \neq 0$, $A \subseteq B$, $\neg P$, $P \wedge Q$

The equality type $t \equiv u$ is encoded as $\{\} \upharpoonright t \equiv u$

A new restriction type $A \upharpoonright P$:

- Built using a type A and a "semantic predicate" P
- $[\![A \upharpoonright P]\!]$ is equal to $[\![A]\!]$ if P is satisfied and to $[\![\forall X.X]\!]$ otherwise
- Examples of predicates: $t \equiv u$, $\kappa \neq 0$, $A \subseteq B$, $\neg P$, $P \wedge Q$

The equality type $t \equiv u$ is encoded as $\{\} \upharpoonright t \equiv u$

Restriction and membership can be combined into a subset type:

- It is possible to define $\{x \in A \mid P\}$ as $\exists x^{\iota}.x \in A \upharpoonright P$
- Note that $\{x \in A \mid P\}$ is always a subtype of A
- A similar constructor can be used in nuPRL

Internal Totality Proofs

```
val rec add_total : ∀n m∈nat, ∃v:ι , add n m ≡ v = fun n m →
  case n {
    Z[_] → qed
    S[k] → use add_total k m; qed
  }
```

```
val rec add_total : ∀n m∈nat, ∃v:ι , add n m ≡ v = fun n m →
  case n {
    Z[_] → qed
    S[k] → use add_total k m; qed
  }

val rec add_asso : ∀n m p∈nat, add n (add m p) ≡ add (add n m) p =
  fun n m p →
    use add_total m p;
    case n {
      Z[_] → qed
      S[k] → use add_total k m; use add_asso k m p; qed
    }
```

Subtyping and termination checking are handled using circular proofs:

- Types (and judgments) are parametrised by ordinals sizes
- A proof forms a directed acyclic graph of atomic proof blocks
- The edges carry size relations between matching ordinals

Subtyping and termination checking are handled using circular proofs:

- Types (and judgments) are parametrised by ordinals sizes
- A proof forms a directed acyclic graph of atomic proof blocks
- The edges carry size relations between matching ordinals

We use an external check to show that typing derivations are well-founded

Subtyping and termination checking are handled using circular proofs:

- Types (and judgments) are parametrised by ordinals sizes
- A proof forms a directed acyclic graph of atomic proof blocks
- The edges carry size relations between matching ordinals

We use an external check to show that typing derivations are well-founded

A semantic proof by induction on the typing derivation gives normalisation

References for Technical Details

A Classical Realizability Model for a Semantical Value Restriction
R. Lepigre (ESOP 2016)

Practical Subtyping for System F with Sized (Co-)Induction
R. Lepigre and C. Raffalli (2016 - 2017)
https://lama.univ-smb.fr/subml/

Semantics and Implementation of an Extension of ML for Proving Programs
R. Lepigre, PhD manuscript
https://lama.univ-smb.fr/~lepigre/these/

Thanks!