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
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

\begin{verbatim}
  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] }
\end{verbatim}
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 : , 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:i , 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

```python
def tac_deduce<f:o> : τ = ({} : f)
def tac_show<f:o, p:τ> : τ = (p : f)
def tac qed : τ = {}
```
Detailed Proof Using (Higher-Order) Macros

\[
def \text{tac\_deduce}<f:o> : \tau = (\{\} : f)\\
def \text{tac\_show}<f:o, p:\tau> : \tau = (p : f)\\
def \text{tac\_qed} : \tau = {}\\
\]

\[
val \text{rec add\_n\_Z} : \forall n \in \text{nat}, \text{add } n \ Z \equiv \text{n } = \text{fun } n \rightarrow\]

\[
\text{case } n \{\\
\text{Z[_] } \rightarrow \text{tac\_deduce add Z Z } \equiv \text{ Z; qed}\\
\text{S[k] } \rightarrow \text{tac\_show add k Z } \equiv \text{ k using add\_n\_Z k;\\
\text{tac\_deduce S[add k Z] } \equiv \text{ S[k];\\
\text{tac\_deduce add S[k] Z } \equiv \text{ S[k]; qed}}\\
\}
\]
Fine-grained Specification Using Equivalence

val rec is_even : nat \Rightarrow bool = fun n \Rightarrow

  \begin{cases}
    \text{case } n \{ \\
    Z[_] \Rightarrow \text{true} \\
    S[p] \Rightarrow \text{case } p \{ Z[_] \Rightarrow \text{false} | S[p] \Rightarrow \text{is_even } p \} \\
  \end{cases}
**Fine-grained Specification Using Equivalence**

```ocaml
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_n = ∃v:ι, (v∈nat | is_even v ≡ true)
```
**Fine-grained Specification Using Equivalence**

val rec is_even : nat \(\Rightarrow\) bool = fun n \(\Rightarrow\)

  case n {
    Z[_] \(\Rightarrow\) true
    S[p] \(\Rightarrow\) case p { Z[_] \(\Rightarrow\) false | S[p] \(\Rightarrow\) is_even p }
  }

type even_nat = \(\exists\)v:\nat, (v\\in\nat \mid is_even v \equiv\) true

val rec double : nat \(\Rightarrow\) even_nat = fun n \(\Rightarrow\)

  case n {
    Z[_] \(\Rightarrow\) Z
    S[p] \(\Rightarrow\) 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:₀, list<a> ⇒ nat = {- ... -}
type vec<a:₀, 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
Classical Realisability Semantics

Realisability is about computation:
- Call-by-value Krivine Machine (for classical logic)
- States of the form $t \ast \pi$ with a reduction relation ($\rightarrow$)
Classical Realisability Semantics

Realisability is about computation:
- Call-by-value Krivine Machine (for classical logic)
- States of the form $t * \pi$ with a reduction relation ($\Rightarrow$)

We also require a notion of observational equivalence:
- We write $t * \pi \Downarrow$ for $\exists \nu$, $t * \pi \Rightarrow \nu * \varepsilon$ (successful computation)
- $(\equiv)$ is defined as $\{(t, u) \mid \forall \pi, \forall \rho, t\rho * \pi \Downarrow \iff u\rho * \pi \Downarrow\}$
**Classical Realisability Semantics**

Realisability is about computation:
- Call-by-value Krivine Machine (for classical logic)
- States of the form \( t \ast \pi \) with a reduction relation \( (\Rightarrow) \)

We also require a notion of observational equivalence:
- We write \( t \ast \pi \Downarrow \) for \( \exists \nu, t \ast \pi \Rightarrow* \nu \ast e \) (successful computation)
- \( (\equiv) \) is defined as \( \{(t, u) \mid \forall \pi, \forall \rho, t\rho \ast \pi \Downarrow \iff u\rho \ast \pi \Downarrow\} \)

A type \( A \) is interpreted using two sets (in fact three):
- The set of its “canonical” values \( \llbracket A \rrbracket \)
- A set of terms \( \llbracket A \rrbracket^{\Pi,\Lambda} \) defined as a form of completion of \( \llbracket A \rrbracket \)
- Closure under \( (\equiv) \) is required on those sets
INTERPRETATION OF THE (USUAL) TYPES

\[\llbracket \{ (l_i : A_i)_{i \in I} \} \rrbracket = \{ (l_i = n_i)_{i \in I} \mid \forall i \in I, n_i \in \llbracket A_i \rrbracket \}\]

\[\llbracket \{(c_i : A_i)_{i \in I}\} \rrbracket = \bigcup_{i \in I} \{c_i[n] \mid n \in \llbracket A_i \rrbracket \}\]

\[\llbracket A \Rightarrow B \rrbracket = \{ \lambda x . t \mid \forall n \in \llbracket A \rrbracket, t[x := n] \in \llbracket B \rrbracket \}\]

\[\llbracket \forall \chi^s . A \rrbracket = \bigcap_{\Phi \in [s]} \llbracket A[\chi := \Phi] \rrbracket\]

\[\llbracket \exists \chi^s . A \rrbracket = \bigcup_{\Phi \in [s]} \llbracket A[\chi := \Phi] \rrbracket\]

\[\llbracket \mu \tau X . A \rrbracket = \bigcup_{\kappa < \tau} (X \mapsto \llbracket A \rrbracket)^\kappa (\emptyset)\]

\[\llbracket \nu \tau X . A \rrbracket = \bigcap_{\kappa < \tau} (X \mapsto \llbracket A \rrbracket)^\kappa (\Lambda_i)\]
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 $\llbracket t \in A \rrbracket = \{ v \in \llbracket A \rrbracket \mid t \equiv v \}$
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 $\llbracket t \in A \rrbracket = \{ v \in \llbracket A \rrbracket \mid t \equiv v \}$

Only way to link the “word of terms” and the “world of types”
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 $\llbracket t \in A \rrbracket = \{v \in \llbracket A \rrbracket \mid t \equiv v\}$

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
SEMANTIC RESTRICTION AND SUBSETS

A new restriction type $A \upharpoonright P$:
- Built using a type $A$ and a “semantic predicate” $P$
- $\llbracket A \upharpoonright P \rrbracket$ is equal to $\llbracket A \rrbracket$ if $P$ is satisfied and to $\llbracket \forall X.X \rrbracket$ otherwise
- Examples of predicates: $t \equiv u$, $\kappa \neq 0$, $A \subseteq B$, $\neg P$, $P \land Q
A new restriction type $A \downarrow P$:
- Built using a type $A$ and a “semantic predicate” $P$
- $\llbracket A \downarrow P \rrbracket$ is equal to $\llbracket A \rrbracket$ if $P$ is satisfied and to $\llbracket \forall X.X \rrbracket$ otherwise
- Examples of predicates: $t \equiv u$, $\kappa \neq 0$, $A \subseteq B$, $\neg P$, $P \land Q$

The equality type $t \equiv u$ is encoded as $\{\} \downarrow t \equiv u$
**Semantic Restriction and Subsets**

A new restriction type $A \vdash P$:
- Built using a type $A$ and a “semantic predicate” $P$
- $\llbracket A \vdash P \rrbracket$ is equal to $\llbracket A \rrbracket$ if $P$ is satisfied and to $\llbracket \forall X.X \rrbracket$ otherwise
- Examples of predicates: $t \equiv u$, $\kappa \neq 0$, $A \subseteq B$, $\neg P$, $P \land Q$

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

Restriction and membership can be combined into a subset type:
- It is possible to define $\{x \in A \mid P\}$ as $\exists x^t . x \in A \vdash P$
- Note that $\{x \in A \mid P\}$ is always a subtype of $A$
- A similar constructor can be used in nuPRL
**Internal Totality Proofs**

```ocaml
val rec add_total : \forall n m \in \text{nat}, \exists v : i , \text{add} n m \equiv v = \text{fun} n m \rightarrow
case n {
    Z[_] \rightarrow \text{qed}
    S[k] \rightarrow \text{use} add_total k m; \text{qed}
}
```
**Internal Totality Proofs**

```
val rec add_total : \forall n m \in \text{nat}, \exists v : i, \text{add n m} \equiv v = \text{fun n m -> }
    \text{case n { }
    Z[_] \rightarrow \text{qed }
    S[k] \rightarrow \text{use add_total k m; qed }
    }

val rec add_asso : \forall n m p \in \text{nat}, \text{add n (add m p)} \equiv \text{add (add n m) p} =
    \text{fun n m p -> }
    \text{use add_total m p; }
    \text{case n { }
    Z[_] \rightarrow \text{qed }
    S[k] \rightarrow \text{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
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!