The PML Language:
Realizability at the Service of Program Proofs

Rodolphe Lepigre

Realizability workshop – 08/03/2018 – Marseille

Using OCaml as a Proof System: A Silly Idea?

```
(* Empty type (logical absurdity). *)
type empty = { empty : 'a. 'a }

(* Logical negation. *)
type 'a neg = 'a → empty

(* The law of the excluded middle. *)
type 'a excluded_middle =
  | True  of 'a
  | False of 'a neg
```

Using OCaml as a Proof System: A Silly Idea?

```
(* Empty type (logical absurdity). *)
type empty = { empty : 'a. 'a }

(* Logical negation. *)
type 'a neg = 'a → empty

(* The law of the excluded middle. *)
type 'a excluded_middle =
  | True  of 'a
  | False of 'a neg

(* The law of the excluded middle implies double negation elimination. *)
let proof : 'a excluded_middle → 'a neg neg → 'a = fun em h →
  match em with
  | True  a     → a
  | False not_a → (h not_a).empty
```

Toward a Programming Language, with Program Proving Features

An ML-like programming language with:

- records, variants (constructors), inductive types,
- polymorphism, general recursion,
- a call-by-value evaluation strategy,
- effects (control operators),
- a Curry-style syntax (light) and subtyping.

Toward a Programming Language, with Program Proving Features

An ML-like programming language with:

- records, variants (constructors), inductive types,
- polymorphism, general recursion,
- a call-by-value evaluation strategy,
- effects (control operators),
- a Curry-style syntax (light) and subtyping.

For proving program, the type system is enriched with:

- programs as individuals (higher-order layer),
- an equality type $t \equiv u$ (observational equivalence),
- a dependent function type (typed quantification).
- Termination checking is required for proofs.

Toward a Programming Language, with Program Proving Features

An ML-like programming language with:

- records, variants (constructors), inductive types,
- polymorphism, general recursion,
- a call-by-value evaluation strategy,
- effects (control operators),
- a Curry-style syntax (light) and subtyping.

For proving program, the type system is enriched with:

- programs as individuals (higher-order layer),
- an equality type $t \equiv u$ (observational equivalence),
- a dependent function type (typed quantification).
- Termination checking is required for proofs.

Example of Program and Proof

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

Example of Program and Proof

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

val add_Zero_m : ∀m∈nat, add Zero m ≡ m =
  fun m { {} }
```

Example of Program and Proof

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

val add_Zero_m : ∀m∈nat, add Zero m ≡ m =
  fun m { {} }

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

Part I

Specificities of the Type System

Properties as Program Equivalences

Examples of (equational) program properties:

- `add (add m n) k` $\equiv$ `add m (add n k)` (associativity of `add`)
- `rev (rev l)` $\equiv$ `l` (`rev` is an involution)
- `map g (map f l)` $\equiv$ `map (fun x {g (f x)}) l` (`map` and composition)
- `sort (sort l)` $\equiv$ `sort l` (`sort` is idempotent)

Properties as Program Equivalences

Examples of (equational) program properties:

- `add (add m n) k` $\equiv$ `add m (add n k)` (associativity of add)
- `rev (rev l)` $\equiv$ `l` (rev is an involution)
- `map g (map f l)` $\equiv$ `map (fun x {g (f x)}) l` (map and composition)
- `sort (sort l)` $\equiv$ `sort l` (sort is idempotent)

Specification of a sorting function using predicates:

- `is_increasing (sort l)` $\equiv$ `true` (sort produces a sorted list)
- `is_perm (sort l) l` $\equiv$ `true` (sort yields a permutation)

Equality Types and Equivalence

We consider a new type former $t \equiv u$ (where t and u are untyped terms).

Equality Types and Equivalence

We consider a new type former $t \equiv u$ (where t and u are untyped terms).

It is interpreted as:

- $\top$ (the *unit type*) if t and u are “equivalent”,
- $\bot$ (the *empty type*) otherwise.

Equality Types and Equivalence

We consider a new type former $t \equiv u$ (where t and u are untyped terms).

It is interpreted as:

- $\top$ (the *unit type*) if t and u are "equivalent",
- $\bot$ (the *empty type*) otherwise.

$$\frac{\Gamma ; \Xi \vdash t : \top \qquad \dfrac{\text{dec. proc. says "yes"}}{\Xi \vdash u_1 \equiv u_2}}{\Gamma ; \Xi \vdash t : u_1 \equiv u_2}$$

Equality Types and Equivalence

We consider a new type former $t \equiv u$ (where t and u are untyped terms).

It is interpreted as:

- $\top$ (the *unit type*) if t and u are “equivalent”,
- $\bot$ (the *empty type*) otherwise.

$$\frac{\Gamma ; \Xi \vdash t : \top \qquad \dfrac{\text{dec. proc. says “yes”}}{\Xi \vdash u_1 \equiv u_2}}{\Gamma ; \Xi \vdash t : u_1 \equiv u_2} \qquad \frac{\Gamma, x : \top ; \Xi , u_1 \equiv u_2 \vdash t : C}{\Gamma, x : u_1 \equiv u_2 ; \Xi \vdash t : C}$$

Equality Types and Equivalence

We consider a new type former $t \equiv u$ (where t and u are untyped terms).

It is interpreted as:

- $\top$ (the *unit type*) if t and u are "equivalent",
- $\bot$ (the *empty type*) otherwise.

$$\frac{\Gamma ; \Xi \vdash t : \top \qquad \dfrac{\text{dec. proc. says "yes"}}{\Xi \vdash u_1 \equiv u_2}}{\Gamma ; \Xi \vdash t : u_1 \equiv u_2} \qquad\qquad \frac{\Gamma, x : \top ; \Xi , u_1 \equiv u_2 \vdash t : C}{\Gamma, x : u_1 \equiv u_2 ; \Xi \vdash t : C}$$

Remark: equivalence is undecidable.

Equality Types and Equivalence

We consider a new type former $t \equiv u$ (where t and u are untyped terms).

It is interpreted as:

- $\top$ (the *unit type*) if t and u are “equivalent”,
- $\bot$ (the *empty type*) otherwise.

$$\frac{\Gamma; \Xi \vdash t : \top \qquad \dfrac{\text{dec. proc. says “yes”}}{\Xi \vdash u_1 \equiv u_2}}{\Gamma; \Xi \vdash t : u_1 \equiv u_2} \qquad\qquad \frac{\Gamma, x : \top; \Xi, u_1 \equiv u_2 \vdash t : C}{\Gamma, x : u_1 \equiv u_2; \Xi \vdash t : C}$$

Remark: equivalence is undecidable.

Remark: decision of equivalence only needs to be correct.

First-Order Quantification is not Enough

```
val rec add : nat ⇒ nat ⇒ nat =
  fun n m { case n { Zero → m | S[k] → S[add k m] } }
```

First-Order Quantification is not Enough

```
val rec add : nat ⇒ nat ⇒ nat =
  fun n m { case n { Zero → m | S[k] → S[add k m] } }

val add_Zero_m : ∀m, add Zero m ≡ m = {- ??? -}
```

First-Order Quantification is not Enough

```
val rec add : nat ⇒ nat ⇒ nat =
  fun n m { case n { Zero → m | S[k] → S[add k m] } }

val add_Zero_m : ∀m, add Zero m ≡ m = {}
// Immediate by definition
```

First-Order Quantification is not Enough

```
val rec add : nat ⇒ nat ⇒ nat =
  fun n m { case n { Zero → m | S[k] → S[add k m] } }

val add_Zero_m : ∀m, add Zero m ≡ m = {}
// Immediate by definition

val add_n_Zero : ∀n, add n Zero ≡ n = {- ??? -}
```

First-Order Quantification is not Enough

```
val rec add : nat ⇒ nat ⇒ nat =
  fun n m { case n { Zero → m | S[k] → S[add k m] } }

val add_Zero_m : ∀m, add Zero m ≡ m = {}
// Immediate by definition

val add_n_Zero : ∀n, add n Zero ≡ n = {- ??? -}
// Nothing we can do
```

First-Order Quantification is not Enough

```
val rec add : nat ⇒ nat ⇒ nat =
  fun n m { case n { Zero → m | S[k] → S[add k m] } }

val add_Zero_m : ∀m, add Zero m ≡ m = {}
// Immediate by definition

val add_n_Zero : ∀n, add n Zero ≡ n = {- ??? -}
// Nothing we can do
```

We need a form of typed quantification!

Dependent Functions for Typed Quantification

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

Dependent Functions for Typed Quantification

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

Remark: we may inspect the elements of the domain.

Dependent Functions for Typed Quantification

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

Remark: we may inspect the elements of the domain.

$$\frac{\Gamma, x : A ; \Xi \vdash t : B}{\Gamma ; \Xi \vdash \lambda x.t : \forall x {\in} A.B}$$

Dependent Functions for Typed Quantification

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

Remark: we may inspect the elements of the domain.

$$\frac{\Gamma, x : A; \Xi \vdash t : B}{\Gamma; \Xi \vdash \lambda x.t : \forall x{\in}A.B} \qquad \frac{\Gamma; \Xi \vdash t : \forall x{\in}A.B \qquad \Gamma; \Xi \vdash v : A}{\Gamma; \Xi \vdash t\ v : B[x := v]}$$

Structuring Proofs with Dummy Programs

```
val rec add_n_Sm : ∀n m∈nat, add n S[m] ≡ S[add n m] =
  fun n m {
    case n {
      Zero → {}
      S[k] → add_n_Sm k m
    }
  }

val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero → add_n_Zero m
      S[k] → add_n_Sm m k; add_comm k m
    }
  }
```

Part II

Formalisation of the System and Semantics

Realizability Model

We build a model to prove that the language has the expected properties.

Realizability Model

We build a model to prove that the language has the expected properties.

To construct the model, we need to:

1) give the syntax of programs and types,
2) define the interpretation of types as sets of terms (uses reduction),
3) define adequate typing rules,
4) deduce *termination*, *type safety* and *consistency*.

Realizability Model

We build a model to prove that the language has the expected properties.

To construct the model, we need to:

1) give the syntax of programs and types,
2) define the interpretation of types as sets of terms (uses reduction),
3) define adequate typing rules,
4) deduce *termination*, *type safety* and *consistency*.

Advantage: it is modular (contrary to *type preservation*).

Call-by-Value Abstract Machine

Values	(Λ_ι)	$v, w ::= x \mid \lambda x.t \mid \{(l_i = v_i)_{i \in I}\} \mid C_k[v]$	
Terms	(Λ)	$t, u ::= v \mid t\ u \mid v.l_k \mid [v \mid (C_i[x_i] \rightarrow t_i)_{i \in I}] \mid \mu\alpha.t \mid [\pi]t$	
Stacks	(Π)	$\pi, \xi ::= \alpha \mid \varepsilon \mid v \, . \, \pi \mid [t]\pi$	(evaluation context)
Processes		$p, q ::= t * \pi$	

Call-by-value Reduction Relation

$$
\begin{array}{rcll}
t\;u * \pi & \succ & u * [t]\pi & \\
v * [t]\pi & \succ & t * v . \pi & \\
\lambda x.t * v . \pi & \succ & t[x := v] * \pi & \\
\{(l_i = v_i)_{i \in I}\}.l_k * \pi & \succ & v_k * \pi & (k \in I) \\
[C_k[v] \mid (C_i[x_i] \rightarrow t_i)_{i \in I}] * \pi & \succ & t_k[x_k := v] * \pi & (k \in I) \\
\mu\alpha.t * \pi & \succ & t[\alpha := \pi] * \pi & \\
[\pi]t * \xi & \succ & t * \pi &
\end{array}
$$

Successful Computation and Observational Equivalence

The abstract machine may either:

- successfully compute a result (it converges),
- fail with a *runtime error* or never terminate (it diverges).

Successful Computation and Observational Equivalence

The abstract machine may either:

- successfully compute a result (it converges),
- fail with a *runtime error* or never terminate (it diverges).

Definition: we write $t * \pi \Downarrow$ iff $t * \pi \succ^* v * \varepsilon$ for some value v ($t * \pi \Uparrow$ otherwise).

Successful Computation and Observational Equivalence

The abstract machine may either:

- successfully compute a result (it converges),
- fail with a *runtime error* or never terminate (it diverges).

Definition: we write $t \ast \pi \Downarrow$ iff $t \ast \pi \succ^{*} v \ast \varepsilon$ for some value v ($t \ast \pi \Uparrow$ otherwise).

$$(\lambda x.x)\ \{\} \ast \varepsilon \Downarrow \qquad (\lambda x.x\ x)\ (\lambda x.x\ x) \ast \varepsilon \Uparrow \qquad (\lambda x.t).l_1 \ast \varepsilon \Uparrow$$

Successful Computation and Observational Equivalence

The abstract machine may either:

- successfully compute a result (it converges),
- fail with a *runtime error* or never terminate (it diverges).

Definition: we write $t \ast \pi \Downarrow$ iff $t \ast \pi \succ^* v \ast \varepsilon$ for some value v ($t \ast \pi \Uparrow$ otherwise).

$$(\lambda x.x)\ \{\} \ast \varepsilon \Downarrow \qquad (\lambda x.x\ x)\ (\lambda x.x\ x) \ast \varepsilon \Uparrow \qquad (\lambda x.t).l_1 \ast \varepsilon \Uparrow$$

Definition: two terms are equivalent if they converge in the same contexts.

Successful Computation and Observational Equivalence

The abstract machine may either:

- successfully compute a result (it converges),
- fail with a *runtime error* or never terminate (it diverges).

Definition: we write $t \ast \pi \Downarrow$ iff $t \ast \pi \succ^* v \ast \varepsilon$ for some value v ($t \ast \pi \Uparrow$ otherwise).

$$(\lambda x.x)\ \{\} \ast \varepsilon \Downarrow \qquad (\lambda x.x\ x)\ (\lambda x.x\ x) \ast \varepsilon \Uparrow \qquad (\lambda x.t).l_1 \ast \varepsilon \Uparrow$$

Definition: two terms are equivalent if they converge in the same contexts.

$$(\equiv) \quad = \quad \big\{(t, u) \mid \forall \pi, t \ast \pi \Downarrow \Leftrightarrow u \ast \pi \Downarrow\big\}$$

Successful Computation and Observational Equivalence

The abstract machine may either:

- successfully compute a result (it converges),
- fail with a *runtime error* or never terminate (it diverges).

Definition: we write $t \ast \pi \Downarrow$ iff $t \ast \pi \succ^{*} v \ast \varepsilon$ for some value v ($t \ast \pi \Uparrow$ otherwise).

$$(\lambda x.x)\ \{\} \ast \varepsilon \Downarrow \qquad (\lambda x.x\ x)\ (\lambda x.x\ x) \ast \varepsilon \Uparrow \qquad (\lambda x.t).l_1 \ast \varepsilon \Uparrow$$

Definition: two terms are equivalent if they converge in the same contexts.

$$(\equiv) \quad = \quad \left\{(t\,,\,u) \mid \forall \pi, \forall \rho\,,\, t\rho \ast \pi \Downarrow \Leftrightarrow u\rho \ast \pi \Downarrow\right\}$$

Types as Sets of Canonical Values

Definition: a type A is interpreted as a set of values $[\![A]\!]$ closed under $(\equiv)$.

Types as Sets of Canonical Values

Definition: a type A is interpreted as a set of values $[\![A]\!]$ closed under $(\equiv)$.

$$
\begin{aligned}
[\![\{l_1 : A_1 \,;\, l_2 : A_2\}]\!] &= \left\{\{l_1 = v_1 \,;\, l_2 = v_2\} \mid v_1 \in [\![A_1]\!] \;\wedge\; v_2 \in [\![A_2]\!]\right\} \\
[\![[C_1 : A_1 \mid C_2 : A_2]]\!] &= \left\{C_i[v] \mid i \in \{1, 2\} \;\wedge\; v \in [\![A_i]\!]\right\} \\
[\![\forall X.A]\!] &= \bigcap_{\Phi \text{ type}} [\![A[X := \Phi]]\!] \\
[\![\exists X.A]\!] &= \bigcup_{\Phi \text{ type}} [\![A[X := \Phi]]\!] \\
[\![\forall x.A]\!] &= \bigcap_{v \text{ value}} [\![A[a := t]]\!] \\
[\![\exists x.A]\!] &= \bigcup_{v \text{ value}} [\![A[a := t]]\!]
\end{aligned}
$$

Membership Types and Dependency

We consider a new *membership type* $t \in A$ (with t a term, A a type).

- It is interpreted as $[\![t \in A]\!] = \{v \in [\![A]\!] \mid t \equiv v\}$,
- and allows the introduction of dependency.

Membership Types and Dependency

We consider a new *membership type* $t \in A$ (with t a term, A a type).

- It is interpreted as $[\![t \in A]\!] = \{v \in [\![A]\!] \mid t \equiv v\}$,
- and allows the introduction of dependency.

The dependent function type $\forall x \in A.B$

- is defined as $\forall x.(x \in A \Rightarrow B)$,
- this is a form of *relativised quantification* scheme.

Semantic Restriction Type and Equalities

We also consider a new *restriction type* $A \upharpoonright P$:

- it is build using a type A and a “semantic predicate” P,
- $[\![A \upharpoonright P]\!]$ is equal to $[\![A]\!]$ if P is satisfied and to $[\![\bot]\!]$ otherwise.
- We can use predicates like $t \equiv u$, $\neg P$ or $P \wedge Q$.

Semantic Restriction Type and Equalities

We also consider a new *restriction type* $A \restriction P$:

- it is build using a type A and a "semantic predicate" P,
- $[\![A \restriction P]\!]$ is equal to $[\![A]\!]$ if P is satisfied and to $[\![\bot]\!]$ otherwise.
- We can use predicates like $t \equiv u$, $\neg P$ or $P \land Q$.

The equality type $t \equiv u$ is encoded as $\top \restriction t \equiv u$.

Interpretation of the Function Type

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

Interpretation of the Function Type

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

What about λ-abstractions which bodies are terms?

Interpretation of the Function Type

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

What about λ-abstractions which bodies are terms?

We define a completion operation $[\![A]\!] \mapsto [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.

Interpretation of the Function Type

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

What about λ-abstractions which bodies are terms?

We define a completion operation $[\![A]\!] \mapsto [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.

The set $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ contains terms “behaving” as values of $[\![A]\!]$.

Interpretation of the Function Type

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

What about λ-abstractions which bodies are terms?

We define a completion operation $[\![A]\!] \mapsto [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.

The set $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ contains terms “behaving” as values of $[\![A]\!]$.

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

Pole and Orthogonality

The definition of $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ is parametrised by a set of processes $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$.

Pole and Orthogonality

The definition of $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ is parametrised by a set of processes $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$.

We require that $p \in \perp\!\!\!\perp$ and $q \succ p$ implies $q \in \perp\!\!\!\perp$.

Pole and Orthogonality

The definition of $[\![A]\!]^{⫫⫫}$ is parametrised by a set of processes $⫫ \subseteq \Lambda \times \Pi$.

We require that $p \in ⫫$ and $q \succ p$ implies $q \in ⫫$.

Intuitively, $⫫$ is a set of processes that “behave well”.

Pole and Orthogonality

The definition of $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ is parametrised by a set of processes $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$.

We require that $p \in \perp\!\!\!\perp$ and $q \succ p$ implies $q \in \perp\!\!\!\perp$.

Intuitively, $\perp\!\!\!\perp$ is a set of processes that “behave well”.

The set $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$ is a good choice.

Pole and Orthogonality

The definition of $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ is parametrised by a set of processes $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$.

We require that $p \in \perp\!\!\!\perp$ and $q \succ p$ implies $q \in \perp\!\!\!\perp$.

Intuitively, $\perp\!\!\!\perp$ is a set of processes that "behave well".

The set $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$ is a good choice.

$$\begin{aligned}
[\![A]\!] &\in \{\Phi \subseteq \Lambda_\iota \mid v \in \Phi \wedge v \equiv w \Rightarrow w \in \Phi\} \\
[\![A]\!]^{\perp\!\!\!\perp} &= \{\pi \in \Pi \mid \forall v \in [\![A]\!], v * \pi \in \perp\!\!\!\perp\} \\
[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp} &= \{t \in \Lambda \mid \forall \pi \in [\![A]\!]^{\perp\!\!\!\perp}, t * \pi \in \perp\!\!\!\perp\}
\end{aligned}$$

Value Restriction and Typing Judgments

Combining call-by-value and effects leads to soundness issues (well-known).

Value Restriction and Typing Judgments

Combining call-by-value and effects leads to soundness issues (well-known).

Usual solution: “value restriction” on some typing rules.

Value Restriction and Typing Judgments

Combining call-by-value and effects leads to soundness issues (well-known).

Usual solution: "value restriction" on some typing rules.

This is encoded with two forms judgments:

- $\Gamma; \Xi \vdash_{\mathrm{val}} v : A$ for values only,
- $\Gamma; \Xi \vdash \mathrm{t} : A$ for terms (including values).

Value Restriction and Typing Judgments

Combining call-by-value and effects leads to soundness issues (well-known).

Usual solution: “value restriction” on some typing rules.

This is encoded with two forms judgments:

- $\Gamma; \Xi \vdash_{\text{val}} v : A$ for values only,
- $\Gamma; \Xi \vdash t : A$ for terms (including values).

$$\frac{\Gamma; \Xi \vdash_{\text{val}} v : A}{\Gamma; \Xi \vdash v : A}$$

Value Restriction and Typing Judgments

Combining call-by-value and effects leads to soundness issues (well-known).

Usual solution: “value restriction” on some typing rules.

This is encoded with two forms judgments:

- $\Gamma; \Xi \vdash_{\text{val}} v : A$ for values only,
- $\Gamma; \Xi \vdash t : A$ for terms (including values).

$$\frac{\Gamma; \Xi \vdash_{\text{val}} v : A}{\Gamma; \Xi \vdash v : A} \qquad \frac{\Gamma; \Xi \vdash t : A \Rightarrow B \qquad \Gamma; \Xi \vdash u : A}{\Gamma; \Xi \vdash t\,u : B}$$

$$\frac{}{\Gamma, x : A; \Xi \vdash_{\text{val}} x : A} \qquad \frac{\Gamma, x : A; \Xi \vdash t : B}{\Gamma; \Xi \vdash_{\text{val}} \lambda x.t : A \Rightarrow B}$$

Adequate Typing Rule

Theorem (adequacy lemma):

- if $\vdash t : A$ is derivable then $t \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$,
- if $\vdash_{\text{val}} v : A$ is derivable then $v \in [\![A]\!]$.

Adequate Typing Rule

Theorem (adequacy lemma):

- if $\vdash t : A$ is derivable then $t \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$,
- if $\vdash_{\text{val}} v : A$ is derivable then $v \in [\![A]\!]$.

Proof by induction on the typing derivation.

Theorem (adequacy lemma):

- if $\vdash t : A$ is derivable then $t \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$,
- if $\vdash_{\mathrm{val}} v : A$ is derivable then $v \in [\![A]\!]$.

Proof by induction on the typing derivation.

We only need to check that our typing rules are "correct".

Adequate Typing Rule

Theorem (adequacy lemma):

- if $\vdash t : A$ is derivable then $t \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$,
- if $\vdash_{\mathrm{val}} v : A$ is derivable then $v \in [\![A]\!]$.

Proof by induction on the typing derivation.

We only need to check that our typing rules are “correct”.

For example $\dfrac{\vdash_{\mathrm{val}} v : A}{\vdash v : A}$ is correct since $[\![A]\!] \subseteq [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.

Adequacy of For All Introduction

$$\frac{\Gamma;\ \Xi \vdash_{\text{val}} v : A}{\Gamma;\ \Xi \vdash_{\text{val}} v : \forall X.A}\, X \notin \Gamma$$

Adequacy of For All Introduction

$$\frac{X \vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : \forall X.A}$$

Adequacy of For All Introduction

$$\frac{X \vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : \forall X.A}$$

We suppose $v \in [\![A[X := \Phi]]\!]$ for all Φ, and show $v \in [\![\forall X.A]\!]$.

Adequacy of For All Introduction

$$\frac{X \vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : \forall X.A}$$

We suppose $v \in [\![A[X := \Phi]]\!]$ for all Φ, and show $v \in [\![\forall X.A]\!]$.

This is immediate since $[\![\forall X.A]\!] = \cap_{\Phi} [\![A[X := \Phi]]\!]$.

Adequacy of For All Introduction

$$\frac{\mathrm{X} \vdash_{\mathrm{val}} v : A}{\vdash_{\mathrm{val}} v : \forall \mathrm{X}.A}$$

We suppose $v \in [\![A[\mathrm{X} := \Phi]]\!]$ for all Φ, and show $v \in [\![\forall \mathrm{X}.A]\!]$.

This is immediate since $[\![\forall \mathrm{X}.A]\!] = \cap_{\Phi} [\![A[\mathrm{X} := \Phi]]\!]$.

$$\frac{\mathrm{X} \vdash t : A}{\vdash t : \forall \mathrm{X}.A}\,\mathrm{bad}$$

Adequacy of For All Introduction

$$\frac{X \vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : \forall X.A}$$

We suppose $v \in [\![A[X := \Phi]]\!]$ for all Φ, and show $v \in [\![\forall X.A]\!]$.

This is immediate since $[\![\forall X.A]\!] = \cap_\Phi [\![A[X := \Phi]]\!]$.

$$\frac{X \vdash t : A}{\vdash t : \forall X.A}\,\text{bad}$$

We suppose $t \in [\![A[X := \Phi]]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ for all Φ, and show $t \in [\![\forall X.A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.

Adequacy of For All Introduction

$$\frac{X \vdash_{\mathrm{val}} v : A}{\vdash_{\mathrm{val}} v : \forall X.A}$$

We suppose $v \in [\![A[X := \Phi]]\!]$ for all Φ, and show $v \in [\![\forall X.A]\!]$.

This is immediate since $[\![\forall X.A]\!] = \cap_\Phi [\![A[X := \Phi]]\!]$.

$$\frac{X \vdash t : A}{\vdash t : \forall X.A}\,\mathrm{bad}$$

We suppose $t \in [\![A[X := \Phi]]\!]^{\perp\!\!\perp}$ for all Φ, and show $t \in [\![\forall X.A]\!]^{\perp\!\!\perp}$.

However we have $\cap_\Phi [\![A[X := \Phi]]\!]^{\perp\!\!\perp} \not\subseteq [\![\forall X.A]\!]^{\perp\!\!\perp} = \left(\cap_\Phi [\![A[X := \Phi]]\!]\right)^{\perp\!\!\perp}$.

Properties of the System

Theorem (normalisation):

$t : A$ implies $t \ast \varepsilon \succ^{*} v \ast \varepsilon$ for some value v.

Properties of the System

Theorem (normalisation):

$t : A$ implies $t * \varepsilon \succ^* v * \varepsilon$ for some value v.

Theorem (safety for simple datatypes):

$t : A$ implies $t * \varepsilon \succ^* v * \varepsilon$ for some value $v : A$.

Properties of the System

Theorem (normalisation):

$t : A$ implies $t * \varepsilon \succ^* v * \varepsilon$ for some value v.

Theorem (safety for simple datatypes):

$t : A$ implies $t * \varepsilon \succ^* v * \varepsilon$ for some value $v : A$.

Theorem (consistency):

there is no closed term $t : \bot$.

Part III

Semantical Value Restriction

Derived Rules for Dependent Functions

$$\frac{x : A \vdash t : B[a := x]}{\vdash_{\text{val}} \lambda x.t : \forall a \in A.B}$$

$$\frac{\vdash t : \forall a \in A.B \qquad \vdash_{\text{val}} v : A}{\vdash t\ v : B[a := v]}$$

Derived Rules for Dependent Functions

$$\frac{x : A \vdash t : B[a := x]}{\vdash_{\text{val}} \lambda x.t : \forall a \in A.B} \qquad \frac{\vdash t : \forall a \in A.B \qquad \vdash_{\text{val}} v : A}{\vdash t\, v : B[a := v]}$$

$$\dfrac{\dfrac{\dfrac{\vdash t : \forall a \in A.B}{\vdash t : \forall a.(a \in A \Rightarrow B)}\,\text{Def}}{\vdash t : v \in A \Rightarrow B[a := v]}\,\forall_e \qquad \dfrac{\dfrac{\vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : v \in A}\,\in_i}{\vdash v : v \in A}\,\uparrow}{\vdash t\, v : B[a := v]}\,\Rightarrow_e$$

Derived Rules for Dependent Functions

$$\frac{x : A \vdash t : B[a := x]}{\vdash_{\mathrm{val}} \lambda x.t : \forall a \in A.B} \qquad \frac{\vdash t : \forall a \in A.B \quad \vdash_{\mathrm{val}} v : A}{\vdash t\ v : B[a := v]}$$

$$\dfrac{\dfrac{\dfrac{\vdash t : \forall a \in A.B}{\vdash t : \forall a.(a \in A \Rightarrow B)}\,\mathrm{Def}}{\vdash t : v \in A \Rightarrow B[a := v]}\,\forall_e \qquad \dfrac{\dfrac{\vdash_{\mathrm{val}} v : A}{\vdash_{\mathrm{val}} v : v \in A}\,\in_i}{\vdash v : v \in A}\,\uparrow}{\vdash t\ v : B[a := v]}\,\Rightarrow_e$$

Value restriction breaks the compositionality of dependent functions.

```
// add_n_Zero : ∀n∈nat, add n Zero ≡ n
add_n_Zero (add Zero S[Zero]) : add (add Zero S[Zero]) Zero ≡ add Zero S[Zero]
```

Semantical Value Restriction

We replace $\dfrac{\vdash t : \forall a \in A.B \qquad \vdash_{\mathrm{val}} v : A}{\vdash t\ v : B[a := v]}$ by $\dfrac{\vdash t : \forall a \in A.B \qquad \vdash u : A \qquad \vdash u \equiv v}{\vdash t\ u : B[a := u]}$.

Semantical Value Restriction

We replace $\dfrac{\vdash t : \forall a{\in}A.B \qquad \vdash_{\text{val}} v : A}{\vdash t\ v : B[a := v]}$ by $\dfrac{\vdash t : \forall a{\in}A.B \qquad \vdash u : A \qquad \vdash u \equiv v}{\vdash t\ u : B[a := u]}$.

This requires changing $\dfrac{\vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : v{\in}A}$ into $\dfrac{\vdash t : A \qquad \vdash t \equiv v}{\vdash t : t{\in}A}$.

Semantical Value Restriction

We replace $\dfrac{\vdash t : \forall a{\in}A.B \qquad \vdash_{\text{val}} v : A}{\vdash t\ v : B[a := v]}$ by $\dfrac{\vdash t : \forall a{\in}A.B \qquad \vdash u : A \qquad \vdash u \equiv v}{\vdash t\ u : B[a := u]}$.

This requires changing $\dfrac{\vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : v{\in}A}$ into $\dfrac{\vdash t : A \qquad \vdash t \equiv v}{\vdash t : t{\in}A}$.

Can this rule be derived in the system?

We replace $\dfrac{\vdash t : \forall a{\in}A.B \quad \vdash_{\text{val}} v : A}{\vdash t\ v : B[a := v]}$ by $\dfrac{\vdash t : \forall a{\in}A.B \quad \vdash u : A \quad \vdash u \equiv v}{\vdash t\ u : B[a := u]}$.

This requires changing $\dfrac{\vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : v{\in}A}$ into $\dfrac{\vdash t : A \quad \vdash t \equiv v}{\vdash t : t{\in}A}$.

Can this rule be derived in the system?

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash t : A \quad \vdash t \equiv v}{\vdash v : A}\equiv}{\vdash_{\text{val}} v : A}}{\vdash_{\text{val}} v : v{\in}A}\in_i}{\vdash v : v{\in}A}\uparrow \quad \vdash t \equiv v}{\vdash t : t{\in}A}\equiv$$

Biorthogonal Completion Closed for Values

Everything goes down to having a rule $\dfrac{\vdash v : A}{\vdash_{\text{val}} v : A}$.

Biorthogonal Completion Closed for Values

Everything goes down to having a rule $\dfrac{\vdash v : A}{\vdash_{\mathrm{val}} v : A}$.

It should not be confused with $\dfrac{\vdash_{\mathrm{val}} v : A}{\vdash v : A}$.

Biorthogonal Completion Closed for Values

Everything goes down to having a rule $\dfrac{\vdash v : A}{\vdash_{\mathrm{val}} v : A}$.

It should not be confused with $\dfrac{\vdash_{\mathrm{val}} v : A}{\vdash v : A}$.

Semantically, this requires that $v \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ implies $v \in [\![A]\!]$.

Biorthogonal Completion Closed for Values

Everything goes down to having a rule $\dfrac{\vdash v : A}{\vdash_{\mathrm{val}} v : A}$.

It should not be confused with $\dfrac{\vdash_{\mathrm{val}} v : A}{\vdash v : A}$.

Semantically, this requires that $v \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ implies $v \in [\![A]\!]$.

The biorthogonal completion should not introduce new values.

Biorthogonal Completion Closed for Values

Everything goes down to having a rule $\dfrac{\vdash v : A}{\vdash_{\mathrm{val}} v : A}$.

It should not be confused with $\dfrac{\vdash_{\mathrm{val}} v : A}{\vdash v : A}$.

Semantically, this requires that $v \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ implies $v \in [\![A]\!]$.

The biorthogonal completion should not introduce new values.

The rule seems reasonable, but it is hard to justify semantically.

The New Instruction Trick

We do not have $v \in [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ implies $v \in [\![A]\!]$ in every realizability model.

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p\Downarrow\}$:

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:

- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:

- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp\!\!\!\perp}$ such that $v * \pi \Uparrow$.

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:

- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp\!\!\!\perp}$ such that $v * \pi \Uparrow$.
- We need to find π such that $v * \pi \Uparrow$ and $\forall w \in [\![A]\!], w * \pi \Downarrow$.

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:

- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp\!\!\!\perp}$ such that $v * \pi \Uparrow$.
- We need to find π such that $v * \pi \Uparrow$ and $\forall w \in [\![A]\!], w * \pi \Downarrow$.
- We can take $\pi = [\lambda x.\delta_{x,v}]\varepsilon$.

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:

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

The New Instruction Trick

We do not have $v \notin [\![A]\!]$ implies $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ in every realizability model.

We extend the system with a new term constructor $\delta_{v,w}$ such that

$$\delta_{v,w} * \pi \succ v * \pi \qquad \text{iff} \qquad v \not\equiv w.$$

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:

- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp\!\!\!\perp}$ such that $v * \pi \Uparrow$.
- We need to find π such that $v * \pi \Uparrow$ and $\forall\, w \in [\![A]\!]$, $w * \pi \Downarrow$.
- We can take $\pi = [\lambda x.\delta_{x,v}]\varepsilon$.
- $v * [\lambda x.\delta_{x,v}]\varepsilon \succ \lambda x.\delta_{x,v} * v \,.\, \varepsilon \succ \delta_{v,v} * \varepsilon \Uparrow$
- $w * [\lambda x.\delta_{x,v}]\varepsilon \succ \lambda x.\delta_{x,v} * w \,.\, \varepsilon \succ \delta_{w,v} * \varepsilon \succ w * \varepsilon \Downarrow$ if $w \in [\![A]\!]$

Well-defined construction of equivalence and reduction

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

Well-defined construction of equivalence and reduction

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

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

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

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

We then take

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

“Demo” (?) and Conclusion

Things That I did not Show

1) Syntax directed typing and subtyping rules using:
   - local subtyping judgments of the form $t \in A \subset B$,
   - choice operators like $\varepsilon_{x \in A}(t \notin B)$ or $\varepsilon_X(t \notin A)$,
   - an encoding of “neutral terms” into reduction.

2) Inductive types, coinductive types and recursion (more recent) using:
   - circular typing and subtyping proofs,
   - well-foundedness established using the *size change principle*.

3) Unreachable code and refutation of patterns.

Future Work

Practical issues (work in progress):

- Composing programs that are proved terminating.
- Extensible records and variant types (inference).

Toward a practical language:

- Compiler using typing informations for optimisations.
- Built-in types (int64, float) with their specification.

Theoretical questions:

- Can we handle more side-effects? (mutable cells, arrays)
- What can we realise with (variations of) $\delta_{v,w}$?
- Can we extend the system with quotient types?
- Can we formalise mathematics in the system?

References for Technical Details

A Classical Realizability Model for a Semantical Value Restriction
R. Lepigre (ESOP 2016)
https://lepigre.fr/files/docs/lepigre2016_svr.pdf

Practical Subtyping for Curry-Style Languages
R. Lepigre and C. Raffalli (submitted to TOPLAS)
https://lepigre.fr/files/docs/lepigre2017_subml.pdf

Semantics and Implementation of an Extension of ML for Proving Programs
R. Lepigre, PhD manuscript
http://lepigre.fr/files/docs/phd.pdf

Thanks!