An Overview of the PML₂ Language

Realisability, subtyping and cyclic proofs

Rodolphe Lepigre

Journées inaugurales du GT Scalp – 26/11/2018

SEMANTICS AND IMPLEMENTATION OF AN EXTENSION OF ML FOR PROVING PROGRAMS

UNIVERSITÉ
SAVOIE
MONT BLANC

RODOLPHE LEPIGRE – 18/07/2017

SUPERVISED BY CHRISTOPHE RAFFALLI, PIERRE HYVERNAT (AND KARIM NOUR)

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 light, Curry-style syntax and subtyping.

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 light, Curry-style syntax 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.

```
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

Specific Type Constructors

Properties as Program Equivalences

Examples of (equational) program properties:

- add (add m n) k ≡ add m (add n k)   (associativity of add)
- rev (rev l) ≡ l   (rev is an involution)
- map g (map f l) ≡ map (fun x {g (f x)}) l   (map and composition)
- sort (sort l) ≡ 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:

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

Equality Types and Equivalence

We consider the type former $t \equiv u$ (where t and u are arbitrary terms).

Equality Types and Equivalence

We consider the type former t ≡ u (where t and u are arbitrary terms).

It is interpreted as:
- the *unit type* ⊤ if t and u are "equivalent",
- the *empty type* ⊥ otherwise.

EQUALITY TYPES AND EQUIVALENCE

We consider the type former $t \equiv u$ (where t and u are arbitrary terms).

It is interpreted as:
- the *unit type* $\top$ if t and u are "equivalent",
- the *empty type* $\bot$ 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 the type former $t \equiv u$ (where t and u are arbitrary terms).

It is interpreted as:
- the *unit type* $\top$ if t and u are "equivalent",
- the *empty type* $\bot$ otherwise.

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

$$\frac{\Gamma, x : \top; \Xi, u_1 \equiv u_2 \vdash t : C}{\Gamma, x : u_1 \equiv u_2; \Xi \vdash t : C}$$

Equality Types and Equivalence

We consider the type former $t \equiv u$ (where t and u are arbitrary terms).

It is interpreted as:
- the *unit type* $\top$ if t and u are "equivalent",
- the *empty type* $\bot$ otherwise.

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

Remark: cannot be complete since equivalence is undecidable.

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 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 = {- ??? -}
```

```
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!

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

```
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.

```
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.

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

```
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.

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

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 a very flexible approach.

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) $t, u ::= v \mid t\, u \mid v.l_k \mid [v \mid (C_i[x_i] \to t_i)_{i \in I}] \mid \mu\alpha.t \mid [\pi]t$

Stacks (Π) $\pi, \xi ::= \alpha \mid \varepsilon \mid v.\pi \mid [t]\pi$ (evaluation context)

Processes $p, q ::= t * \pi$

$$t\,u * \pi \;>\; u * [t]\pi$$

$$v * [t]\pi \;>\; t * v\,.\,\pi$$

$$\lambda x.t * v\,.\,\pi \;>\; t[x \coloneqq v] * \pi$$

$$\{(l_i = v_i)_{i \in I}\}.l_k * \pi \;>\; v_k * \pi \qquad (k \in I)$$

$$[C_k[v] \,|\, (C_i[x_i] \to t_i)_{i \in I}] * \pi \;>\; t_k[x_k \coloneqq v] * \pi \qquad (k \in I)$$

$$\mu\alpha.t * \pi \;>\; t[\alpha \coloneqq \pi] * \pi$$

$$[\pi]t * \xi \;>\; t * \pi$$

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).

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 $t * \pi >^* v * \varepsilon$ for some value v ($t * \pi \Uparrow$ otherwise).

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 $t * \pi >^* v * \varepsilon$ for some value v ($t * \pi \Uparrow$ otherwise).

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

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 $t * \pi \Downarrow$ iff $t * \pi >^* v * \varepsilon$ for some value v ($t * \pi \Uparrow$ otherwise).

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

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

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

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

$$(\equiv)\ =\ \big\{(t,\, u)\ |\ \forall\, \pi,\, t * \pi \Downarrow \Leftrightarrow u * \pi \Downarrow \big\}$$

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 $t * \pi \Downarrow$ iff $t * \pi >^* v * \varepsilon$ for some value v ($t * \pi \Uparrow$ otherwise).

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

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

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

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 $(\equiv)$.

$$[\![\{l_1 : A_1 ; l_2 : A_2\}]\!] = \Big\{\{l_1 = v_1 ; l_2 = v_2\} \mid v_1 \in [\![A_1]\!] \ \wedge \ v_2 \in [\![A_2]\!]\Big\}$$

$$[\![[C_1 : A_1 \mid C_2 : A_2]]\!] = \Big\{C_i[v] \mid i \in \{1, 2\} \ \wedge \ v \in [\![A_i]\!]\Big\}$$

$$[\![\forall X.A]\!] = \bigcap_{\Phi \text{ type}} [\![A[X := \Phi]]\!]$$

$$[\![\exists X.A]\!] = \bigcup_{\Phi \text{ type}} [\![A[X := \Phi]]\!]$$

$$[\![\forall x.A]\!] = \bigcap_{v \text{ value}} [\![A[a := t]]\!]$$

$$[\![\exists x.A]\!] = \bigcup_{v \text{ value}} [\![A[a := t]]\!]$$

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.

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.

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 $P \wedge Q$.

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 $P \wedge Q$.

Remark: equality types $t \equiv u$ are encoded as $\top \upharpoonright t \equiv u$.

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 $P \wedge Q$.

Remark: equality types $t \equiv u$ are encoded as $\top \upharpoonright t \equiv u$.

Remark: refinement types $\{x \in A \mid P\}$ are encoded as $\exists x.(x \in A \upharpoonright P)$.

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

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

What about λ-abstractions which bodies are terms?

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

What about λ-abstractions which bodies are terms?

We define a completion operation $\llbracket A \rrbracket \mapsto \llbracket A \rrbracket^{\perp\!\perp}$.

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

What about λ-abstractions which bodies are terms?

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

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

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

What about λ-abstractions which bodies are terms?

We define a completion operation $\llbracket A \rrbracket \mapsto \llbracket A \rrbracket^{\perp\!\perp}$.

The set $\llbracket A \rrbracket^{\perp\!\perp}$ contains terms "behaving" as values of $\llbracket A \rrbracket$.

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

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 $[\![A]\!]^{\perp\!\!\!\perp}$ 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 $[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp}$ 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".

Pole and Orthogonality

The definition of $[\![A]\!]^{\perp\!\!\!\perp}$ 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.

$$[\![A]\!] \quad \in \quad \{\Phi \subseteq \Lambda_\iota \mid v \in \Phi \wedge v \equiv w \Rightarrow w \in \Phi\}$$

$$[\![A]\!]^{\perp\!\!\!\perp} \quad = \quad \{\pi \in \Pi \mid \forall v \in [\![A]\!], v * \pi \in \perp\!\!\!\perp\}$$

$$[\![A]\!]^{\perp\!\!\!\perp\perp\!\!\!\perp} \quad = \quad \{t \in \Lambda \mid \forall \pi \in [\![A]\!]^{\perp\!\!\!\perp}, t * \pi \in \perp\!\!\!\perp\}$$

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,
- $\Gamma; \Xi \vdash t : A$ 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).

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

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

Theorem (adequacy lemma):

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

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

Theorem (adequacy lemma):

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

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

Proof by induction on the typing derivation.

Theorem (adequacy lemma):
- if $\vdash t : A$ is derivable then $t \in [\![A]\!]^{\perp\perp}$,
- 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".

Theorem (adequacy lemma):
- if $\vdash t : A$ is derivable then $t \in [\![A]\!]^{\perp\!\perp}$,
- if $\vdash_{\text{val}} v : A$ 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 $\quad \dfrac{\vdash_{\text{val}} v : A}{\vdash v : A} \quad$ is correct since $[\![A]\!] \subseteq [\![A]\!]^{\perp\!\perp}$.

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

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

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 $v \in [\![\forall X.A]\!]$.

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 $[\![\forall X.A]\!] = \cap_{\Phi} [\![A[X := \Phi]]\!]$.

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 $[\![\forall X.A]\!] = \cap_{\Phi} [\![A[X := \Phi]]\!]$.

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

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

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

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

Theorem (normalisation):

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

Theorem (normalisation):

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

Theorem (safety for simple datatypes):

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

Theorem (normalisation):

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

Theorem (safety for simple datatypes):

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

Theorem (consistency):

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

Part III

Semantical Value Restriction

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

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

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

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

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

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

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

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]
```

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

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

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

Can this rule be derived in the system?

Semantical Value Restriction

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

Can this rule be derived in the system?

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

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

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

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

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

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

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

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 $v \in [\![A]\!]^{\perp\!\!\!\perp}$ implies $v \in [\![A]\!]$.

The biorthogonal completion should not introduce new values.

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

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

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

The biorthogonal completion should not introduce new values.

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

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

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

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

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

Idea of the proof with $\perp\!\perp = \{p \mid p \Downarrow\}$:
- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\perp}$.

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

Idea of the proof with $\perp\!\perp = \{p \mid p \Downarrow\}$:
- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp}$ such that $v * \pi \Uparrow$.

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

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:
- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp}$ such that $v * \pi \Uparrow$.
- We need to find π such that $v * \pi \Uparrow$ and $\forall w \in [\![A]\!], w * \pi \Downarrow$.

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

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:
- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp}$ 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 $\pi = [\lambda x.\delta_{x,v}]\varepsilon$.

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

Idea of the proof with $\perp\!\!\!\perp = \{p \mid p \Downarrow\}$:
- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp}$ 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 $\pi = [\lambda x.\delta_{x,v}]\varepsilon$.
- $v * [\lambda x.\delta_{x,v}]\varepsilon \succ \lambda x.\delta_{x,v} * v \cdot \varepsilon \succ \delta_{v,v} * \varepsilon \Uparrow$

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

Idea of the proof with $\perp\!\perp = \{p \mid p \Downarrow\}$:
- We assume $v \notin [\![A]\!]$ and show $v \notin [\![A]\!]^{\perp\!\perp}$.
- We need to find $\pi \in [\![A]\!]^{\perp}$ 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 $w \in [\![A]\!]$

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

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

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

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

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

We then take

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

PART IV

LOCAL SUBTYPING AND CHOICE OPERATORS

A Syntax-directed Presentation

PML_2 is hard to implement for several reasons:
- it is a Curry-style language (quantifiers are not reflected in terms),
- many of its type constructors don't have "algorithmic contents".

A Syntax-directed Presentation

PML_2 is hard to implement for several reasons:

- it is a Curry-style language (quantifiers are not reflected in terms),
- many of its type constructors don't have "algorithmic contents".

$$\frac{\Gamma\,;\, \Xi \vdash t : A \quad a \notin FV(\Gamma\,;\, \Xi) \quad \Xi \vdash t \equiv \nu}{\Gamma\,;\, \Xi \vdash t : \forall a.A} \qquad \frac{\Gamma\,;\, \Xi \vdash t : \forall a.A}{\Gamma\,;\, \Xi \vdash t : A[a := u]}$$

A Syntax-directed Presentation

PML_2 is hard to implement for several reasons:

- it is a Curry-style language (quantifiers are not reflected in terms),
- many of its type constructors don't have "algorithmic contents".

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

Solution: handle these connectives using *local subtyping*.

A Syntax-directed Presentation

PML_2 is hard to implement for several reasons:
- it is a Curry-style language (quantifiers are not reflected in terms),
- many of its type constructors don't have "algorithmic contents".

$$\frac{\Gamma; \Xi \vdash t : A \quad a \notin \mathrm{FV}(\Gamma; \Xi) \quad \Xi \vdash t \equiv v}{\Gamma; \Xi \vdash t : \forall a.A} \qquad \frac{\Gamma; \Xi \vdash t : \forall a.A}{\Gamma; \Xi \vdash t : A[a := u]}$$

Solution: handle these connectives using *local subtyping*.

We then obtain a type system with:
- one typing for each term (or value) constructor,
- one typing rule for each pair of type constructors (up to commutation).

We replace free variables with "choice operators":

- $\varepsilon_{x \in A}(t \notin B)$ denotes some $v \in [\![A]\!]$ such that $[\![t[x := a]]\!] \notin [\![B]\!]^{\perp\!\perp}$ (if possible),
- and similar things are defined for types and other syntactic elements.
- Choice operators are interpreted using elements of the semantic domain.

Choice Operators and Local Subtyping

We replace free variables with "choice operators":

- $\varepsilon_{x \in A}(t \notin B)$ denotes some $v \in [\![A]\!]$ such that $[\![t[x := a]]\!] \notin [\![B]\!]^{\perp\!\perp}$ (if possible),
- and similar things are defined for types and other syntactic elements.
- Choice operators are interpreted using elements of the semantic domain.

We modify the system by:

- eliminating typing contexts (in favor of choice operators),
- introducing local subtyping judgments of the form $\Xi \vdash t : A \subseteq B$.
- They are interpreted as: "if $\Xi \vdash t : A$ holds, then $\Xi \vdash t : B$ also holds."

CHOICE OPERATORS AND LOCAL SUBTYPING

We replace free variables with "choice operators":

- $\varepsilon_{x \in A}(t \notin B)$ denotes some $v \in [\![A]\!]$ such that $[\![t[x := a]]\!] \notin [\![B]\!]^{\perp\!\!\!\perp}$ (if possible),
- and similar things are defined for types and other syntactic elements.
- Choice operators are interpreted using elements of the semantic domain.

We modify the system by:

- eliminating typing contexts (in favor of choice operators),
- introducing local subtyping judgments of the form $\Xi \vdash t : A \subseteq B$.
- They are interpreted as: "if $\Xi \vdash t : A$ holds, then $\Xi \vdash t : B$ also holds."

Remark: choice operators may not be necessary, but they makes the semantics simpler.

Choice Operators and Local Subtyping

We replace free variables with "choice operators":

- $\varepsilon_{x \in A}(t \notin B)$ denotes some $v \in \llbracket A \rrbracket$ such that $\llbracket t[x := a] \rrbracket \notin \llbracket B \rrbracket^{\perp\!\perp}$ (if possible),
- and similar things are defined for types and other syntactic elements.
- Choice operators are interpreted using elements of the semantic domain.

We modify the system by:

- eliminating typing contexts (in favor of choice operators),
- introducing local subtyping judgments of the form $\Xi \vdash t : A \subseteq B$.
- They are interpreted as: "if $\Xi \vdash t : A$ holds, then $\Xi \vdash t : B$ also holds."

Remark: choice operators may not be necessary, but they makes the semantics simpler.

Remark: $\Xi \vdash A \subseteq B$ can be encoded as $\Xi \vdash \varepsilon_{x \in A}(x \notin B) : A \subseteq B$.

$$\frac{\Xi \vdash \lambda x.t : A \Rightarrow B \subseteq C \qquad \Xi, \varepsilon_{x \in A}(t \notin B) \neq \square \vdash t[x := \varepsilon_{x \in A}(t \notin B)] : B}{\Xi \vdash \lambda x.t : C} \Rightarrow_i$$

$$\dfrac{\Xi \vdash \lambda x.t : A \Rightarrow B \subseteq C \quad \Xi, \varepsilon_{x \in A}(t \notin B) \neq \square \vdash t[x := \varepsilon_{x \in A}(t \notin B)] : B}{\Xi \vdash \lambda x.t : C} \Rightarrow_i$$

$$\dfrac{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : A \subseteq C \quad \Xi \vdash \varepsilon_{x \in A}(t \notin B) \neq \square}{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : C} \text{Ax}$$

Examples of Syntax-directed Typing Rules

$$\frac{\Xi \vdash \lambda x.t : A \Rightarrow B \subseteq C \quad \Xi, \varepsilon_{x \in A}(t \notin B) \neq \square \vdash t[x := \varepsilon_{x \in A}(t \notin B)] : B}{\Xi \vdash \lambda x.t : C} \Rightarrow_i$$

$$\frac{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : A \subseteq C \quad \Xi \vdash \varepsilon_{x \in A}(t \notin B) \neq \square}{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : C} {}_{Ax}$$

$$\frac{\Xi \vdash t : A \Rightarrow B \quad \Xi \vdash u : A}{\Xi \vdash t\, u : B} \Rightarrow_e$$

$$\frac{\Xi \vdash \lambda x.t : A \Rightarrow B \subseteq C \quad \Xi, \varepsilon_{x \in A}(t \notin B) \neq \square \vdash t[x := \varepsilon_{x \in A}(t \notin B)] : B}{\Xi \vdash \lambda x.t : C} \Rightarrow_i$$

$$\frac{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : A \subseteq C \quad \Xi \vdash \varepsilon_{x \in A}(t \notin B) \neq \square}{\Xi \vdash \varepsilon_{x \in A}(t \notin B) : C} Ax$$

$$\frac{\Xi \vdash t : A \Rightarrow B \quad \Xi \vdash u : A}{\Xi \vdash t\, u : B} \Rightarrow_e$$

$$\frac{\Xi \vdash v : A \quad \Xi \vdash C_k[v] : [C_k : A] \subseteq B}{\Xi \vdash C_k[v] : B} +_i \qquad \frac{\Xi \vdash v : \{l_k : A \, ; \cdots\}}{\Xi \vdash v.l_k : A} \times_e$$

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X.A \subseteq B} \forall_l \qquad \frac{\Xi \vdash t : A \subseteq B[X := \varepsilon_X(t \notin B)] \quad \Xi \vdash v \equiv t}{\Xi \vdash t : A \subseteq \forall X.B} \forall_r$$

Examples of Syntax-directed (Local) Subtyping Rules

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X.A \subseteq B} \, \forall_l \qquad \frac{\Xi \vdash t : A \subseteq B[X := \varepsilon_X(t \notin B)] \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t : A \subseteq \forall X.B} \, \forall_r$$

$$\frac{\Xi, u_1 \equiv u_2 \vdash t : A \subseteq B \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t : A \upharpoonright u_1 \equiv u_2 \subseteq B} \, \upharpoonright_l \qquad \frac{\Xi \vdash t : A \subseteq B \quad \Xi \vdash u_1 \equiv u_2}{\Xi \vdash t : A \subseteq B \upharpoonright u_1 \equiv u_2} \, \upharpoonright_r$$

Examples of Syntax-directed (Local) Subtyping Rules

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X.A \subseteq B} \forall_l \qquad \frac{\Xi \vdash t : A \subseteq B[X := \varepsilon_X(t \notin B)] \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t : A \subseteq \forall X.B} \forall_r$$

$$\frac{\Xi, u_1 \equiv u_2 \vdash t : A \subseteq B \quad \Xi \vdash \nu \equiv t}{\Xi \vdash t : A \restriction u_1 \equiv u_2 \subseteq B} \restriction_l \qquad \frac{\Xi \vdash t : A \subseteq B \quad \Xi \vdash u_1 \equiv u_2}{\Xi \vdash t : A \subseteq B \restriction u_1 \equiv u_2} \restriction_r$$

$$\frac{\Xi, t \equiv u \vdash t : A \subseteq B \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t : u \in A \subseteq B} \in_l \qquad \frac{\Xi \vdash t : A \subseteq B \quad \Xi \vdash t \equiv u \quad \Xi \vdash t \equiv \nu}{\Xi \vdash t : A \subseteq u \in B} \in_r$$

EXAMPLES OF SYNTAX-DIRECTED (LOCAL) SUBTYPING RULES

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X.A \subseteq B} \forall_l \qquad \frac{\Xi \vdash t : A \subseteq B[X := \varepsilon_X(t \notin B)] \quad \Xi \vdash v \equiv t}{\Xi \vdash t : A \subseteq \forall X.B} \forall_r$$

$$\frac{\Xi, u_1 \equiv u_2 \vdash t : A \subseteq B \quad \Xi \vdash v \equiv t}{\Xi \vdash t : A \upharpoonright u_1 \equiv u_2 \subseteq B} \upharpoonright_l \qquad \frac{\Xi \vdash t : A \subseteq B \quad \Xi \vdash u_1 \equiv u_2}{\Xi \vdash t : A \subseteq B \upharpoonright u_1 \equiv u_2} \upharpoonright_r$$

$$\frac{\Xi, t \equiv u \vdash t : A \subseteq B \quad \Xi \vdash t \equiv v}{\Xi \vdash t : u \in A \subseteq B} \in_l \qquad \frac{\Xi \vdash t : A \subseteq B \quad \Xi \vdash t \equiv u \quad \Xi \vdash t \equiv v}{\Xi \vdash t : A \subseteq u \in B} \in_r$$

$$\frac{\Xi, w \neq \square \vdash w : A_2 \subseteq A_1 \quad \Xi, w \neq \square \vdash t \, w : B_1 \subseteq B_2 \quad \Xi \vdash t \equiv v}{\Xi \vdash t : A_1 \Rightarrow B_1 \subseteq A_2 \Rightarrow B_2} \Rightarrow$$

(where $w = \varepsilon_{x \in A_2}(t \, x \notin B_2)$)

Part V

Cyclic Proofs and Termination Checking

Recursive programs rely on a term $\varphi a.v$ (binding a term in a value).

$$\varphi a.v * \pi \quad \twoheadrightarrow \quad v[a := \varphi a\, v] * \pi$$

$$\frac{\Xi \vdash v[a := \varphi a.v] : A}{\Xi \vdash \varphi a.v : A}\varphi$$

Recursive programs rely on a term $\varphi a.v$ (binding a term in a value).

$$\varphi a.v * \pi \quad \twoheadrightarrow \quad v[a := \varphi a\, v] * \pi \qquad\qquad \dfrac{\Xi \vdash v[a := \varphi a.v] : A}{\Xi \vdash \varphi a.v : A}\,\varphi$$

Problem: we need to work with infinite proofs.

Recursive programs rely on a term $\varphi a.v$ (binding a term in a value).

$$\varphi a.v * \pi \quad \twoheadrightarrow \quad v[a := \varphi a\, v] * \pi \qquad\qquad \frac{\Xi \vdash v[a := \varphi a.v] : A}{\Xi \vdash \varphi a.v : A}\,\varphi$$

Problem: we need to work with infinite proofs.

We introduce a cyclic structure in our proofs.

$$\frac{\forall \alpha\ (\Xi \vdash t : A)}{(\Xi \vdash t : A)[\alpha := \kappa]}\,\text{Gen} \qquad\qquad \frac{\begin{array}{c}[\forall \alpha\ (\Xi \vdash t : A)]^i \\ \vdots \\ \hline (\Xi \vdash t : A)[\alpha := \varepsilon_\alpha(t \notin A)]\end{array}}{\forall \alpha\ (\Xi \vdash t : A)}\,\text{Ind[i]}$$

$$\frac{\Xi \vdash t : A \subseteq B[X := \mu_\infty X.B]}{\Xi \vdash t : A \subseteq \mu_\infty X.B} \, \mu_{\tau,\infty}$$

$$\frac{\Xi \vdash t : A \subseteq B[X := \mu_\infty X.B]}{\Xi \vdash t : A \subseteq \mu_\infty X.B} \mu_{\tau,\infty}$$

$$\frac{\Xi \vdash t : A \subseteq B[X := \mu_\upsilon X.B] \quad \Xi \vdash \upsilon < \tau}{\Xi \vdash t : A \subseteq \mu_\tau X.B} \mu_\tau$$

$$\frac{\Xi \vdash t : A \subseteq B[X := \mu_\infty X.B]}{\Xi \vdash t : A \subseteq \mu_\infty X.B} \mu_{\tau,\infty}$$

$$\frac{\Xi \vdash t : A \subseteq B[X := \mu_\upsilon X.B] \quad \Xi \vdash \upsilon < \tau}{\Xi \vdash t : A \subseteq \mu_\tau X.B} \mu_r$$

$$\frac{\Xi ; \tau > 0 \vdash t : A[X := \mu_{\varepsilon_{\theta<\tau}(t \in A[X := \mu_\theta X.A])} X.A] \subseteq B \quad \Xi \vdash \nu \equiv t}{\gamma ; \Xi \vdash t : \mu_\tau X.A \subseteq B} \mu_l$$

Example of Cyclic Proof

Let us consider the "map" function: $\varphi m.\lambda f.\lambda l.[l \mid [\,] \rightarrow [\,] \mid x :: l \rightarrow f\, x :: m\, f\, l]$.

It can be given either of the types:

- $\forall X.Y(X \Rightarrow Y) \Rightarrow \text{List}(X) \Rightarrow \text{List}(X)$,
- $\forall \alpha.\forall X.Y(X \Rightarrow Y) \Rightarrow \text{List}(\alpha, X) \Rightarrow \text{List}(X)$,
- $\forall \alpha.\forall X.Y(X \Rightarrow Y) \Rightarrow \text{List}(\alpha, X) \Rightarrow \text{List}(\alpha, X)$.

$\text{List}(\alpha, X)$ is defined as $\mu_\alpha L.[([\,]) : \{\} \mid (::) : X \times L]$.

Conclusion

Future Work

1) Practical issues (work in progress):
- Composing programs that are proved terminating.
- Extensible records and variant types (inference).

2) Toward a practical language:
- Compiler using type information for optimisations.
- Built-in types (`int64`, `float`) with their formal specification.

3) 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?

Practical Subtyping for Curry-Style Languages
https://lepigre.fr/files/publications/LepRaf2018a.pdf

PML₂: Integrated Program Verification in ML
https://lepigre.fr/files/publications/Lepigre2018.pdf

Semantics and Implementation of an Extension of ML for Proving Programs
https://lepigre.fr/files/publications/Lepigre2017PhD.pdf

A Classical Realizability Model for a Semantical Value Restriction
https://lepigre.fr/files/publications/Lepigre2016.pdf

Rodolphe Lepigre

Thanks!