

THE  $PML_2$  LANGUAGE  
INTEGRATED PROGRAM VERIFICATION IN ML

*Inria*

RODOLPHE LEPIGRE  
PROSECCO SEMINAR – 03/12/2018

# SEMANTICS AND IMPLEMENTATION OF AN EXTENSION OF ML FOR PROVING PROGRAMS



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.

PART I    SPECIFIC TYPE CONSTRUCTORS

PART II    FORMALISATION OF THE SYSTEM AND SEMANTICS

PART III   SEMANTICAL VALUE RESTRICTION

PART IV    LOCAL SUBTYPING AND CHOICE OPERATORS

PART V    CYCLIC PROOFS AND TERMINATION CHECKING

# PART I

## SPECIFIC TYPE CONSTRUCTORS

## PROPERTIES AS PROGRAM EQUIVALENCES

Examples of (equational) program properties:

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

## PROPERTIES AS PROGRAM EQUIVALENCES

Examples of (equational) program properties:

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

Specification of a sorting function using predicates:

- $\text{sorted} (\text{sort } l) \equiv \text{true}$  (sort produces a sorted list)
- $\text{permutation} (\text{sort } l) \ l \equiv \text{true}$  (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  $t \equiv u$  (where  $t$  and  $u$  are arbitrary terms).

It is interpreted as:

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

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

## 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*  $\perp$  otherwise.

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

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

## 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*  $\perp$  otherwise.

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

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

**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] } }
```

## 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  $\Rightarrow$  nat  $\Rightarrow$  nat =  
  fun n m { case n { Zero  $\rightarrow$  m | S[k]  $\rightarrow$  S[add k m] } }
```

```
val add_Zero_m :  $\forall$ m, add Zero m  $\equiv$  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 :  $\forall n \in \text{nat}, \text{add } n \text{ Zero} \equiv n =$   
  fun n {  
    case n {  
      Zero  $\rightarrow \{\}$   
      S[p]  $\rightarrow \text{add\_n\_Zero } p$   
    }  
  }
```

## DEPENDENT FUNCTIONS FOR TYPED QUANTIFICATION

```
val rec add_n_Zero :  $\forall n \in \text{nat}, \text{add } n \text{ Zero} \equiv n =$   
  fun n {  
    case n {  
      Zero  $\rightarrow \{\}$   
      S[p]  $\rightarrow \text{add\_n\_Zero } p$   
    }  
  }
```

**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 :  $\forall n \in \text{nat}, \text{add } n \text{ Zero} \equiv n =$   
  fun n {  
    case n {  
      Zero  $\rightarrow \{\}$   
      S[p]  $\rightarrow \text{add\_n\_Zero } p$   
    }  
  }
```

**Remark:** we may inspect the elements of the domain.

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

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

## STRUCTURING PROOFS WITH DUMMY PROGRAMS

```
val rec add_n_Sm :  $\forall n m \in \text{nat}, \text{add } n \ S[m] \equiv S[\text{add } n \ m] =$   
  fun n m {  
    case n { Zero  $\rightarrow$  {} | S[k]  $\rightarrow$  add_n_Sm k m }  
  }
```

```
val rec add_comm :  $\forall n m \in \text{nat}, \text{add } n \ m \equiv \text{add } m \ n =$   
  fun n m {  
    case n {  
      Zero  $\rightarrow$  add_n_Zero m  
      S[k]  $\rightarrow$  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	$(\Lambda_v)$	$v, w ::= x \mid \lambda x.t \mid \{(l_i = v_i)_{i \in I}\} \mid C_k[v]$
Terms	$(\Lambda)$	$t, u ::= v \mid t u \mid v.l_k \mid [v \mid (C_i[x_i] \rightarrow t_i)_{i \in I}] \mid \mu \alpha.t \mid [\pi]t$
Stacks	$(\Pi)$	$\pi, \xi ::= \alpha \mid \varepsilon \mid v.\pi \mid [t]\pi$ (evaluation context)
Processes		$p, q ::= t * \pi$

## CALL-BY-VALUE REDUCTION RELATION

$$t \ u * \pi \succ u * [t] \pi$$

$$v * [t] \pi \succ t * v . \pi$$

$$\lambda x . t * v . \pi \succ t[x := v] * \pi$$

$$\{(l_i = v_i)_{i \in I}\} . l_k * \pi \succ v_k * \pi \quad (k \in I)$$

$$[C_k[v] \mid (C_i[x_i] \rightarrow t_i)_{i \in I}] * \pi \succ t_k[x_k := v] * \pi \quad (k \in I)$$

$$\mu \alpha . t * \pi \succ t[\alpha := \pi] * \pi$$

$$[\pi] t * \xi \succ t * \pi$$

## 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).

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

$$(\lambda x.x) \{\} * \varepsilon \Downarrow$$

$$(\lambda x.x x) (\lambda x.x x) * \varepsilon \Uparrow$$

$$(\lambda x.t).l_1 * \varepsilon \Uparrow$$

## 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  $t * \pi \succ^* v * \varepsilon$  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).

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

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

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

## 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).

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

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

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

## TYPES AS SETS OF CANONICAL VALUES

**Definition:** a type  $A$  is interpreted as a set of values  $\llbracket A \rrbracket$  closed under  $(\equiv)$ .

## TYPES AS SETS OF CANONICAL VALUES

**Definition:** a type  $A$  is interpreted as a set of values  $\llbracket A \rrbracket$  closed under ( $\equiv$ ).

$$\llbracket \{l_1 : A_1 ; l_2 : A_2\} \rrbracket = \{ \{l_1 = v_1 ; l_2 = v_2\} \mid v_1 \in \llbracket A_1 \rrbracket \wedge v_2 \in \llbracket A_2 \rrbracket \}$$

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

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

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

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

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

## MEMBERSHIP TYPES AND DEPENDENCY

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

- It is interpreted as  $\llbracket t \in A \rrbracket = \{v \in \llbracket A \rrbracket \mid t \equiv v\}$ ,
- 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  $\llbracket t \in A \rrbracket = \{v \in \llbracket A \rrbracket \mid t \equiv v\}$ ,
- 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”  $P$ ,
- $\llbracket A \upharpoonright P \rrbracket$  is equal to  $\llbracket A \rrbracket$  if  $P$  is satisfied and to  $\llbracket \perp \rrbracket$  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*  $A \upharpoonright P$ :

- it is build 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  $\llbracket \perp \rrbracket$  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”  $P$ ,
- $\llbracket A \upharpoonright P \rrbracket$  is equal to  $\llbracket A \rrbracket$  if  $P$  is satisfied and to  $\llbracket \perp \rrbracket$  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  $\exists x.(x \in A \upharpoonright P)$ .

## INTERPRETATION OF THE FUNCTION TYPE

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

## INTERPRETATION OF THE FUNCTION TYPE

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

What about  $\lambda$ -abstractions which bodies are terms?

## INTERPRETATION OF THE FUNCTION TYPE

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

What about  $\lambda$ -abstractions which bodies are terms?

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

## INTERPRETATION OF THE FUNCTION TYPE

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

What about  $\lambda$ -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  $\llbracket A \rrbracket$ .

## INTERPRETATION OF THE FUNCTION TYPE

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

What about  $\lambda$ -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  $\llbracket A \Rightarrow B \rrbracket = \{\lambda x.t \mid \forall v \in \llbracket A \rrbracket, t[x := v] \in \llbracket B \rrbracket^{\perp\perp}\}$ .



## POLE AND ORTHOGONALITY

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

## POLE AND ORTHOGONALITY

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

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

## POLE AND ORTHOGONALITY

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

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

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

## POLE AND ORTHOGONALITY

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

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

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

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

## POLE AND ORTHOGONALITY

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

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

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

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

$$\llbracket A \rrbracket \in \{\Phi \subseteq \Lambda_l \mid v \in \Phi \wedge v \equiv w \Rightarrow w \in \Phi\}$$

$$\llbracket A \rrbracket^{\perp} = \{\pi \in \Pi \mid \forall v \in \llbracket A \rrbracket, v * \pi \in \perp\}$$

$$\llbracket A \rrbracket^{\perp\perp} = \{t \in \Lambda \mid \forall \pi \in \llbracket A \rrbracket^{\perp}, t * \pi \in \perp\}$$

## 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:

- $\Gamma; \Xi \vdash_{\text{val}} v : A$  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).

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

$$\frac{\Gamma; \Xi \vdash t : A \Rightarrow B \quad \Gamma; \Xi \vdash u : A}{\Gamma; \Xi \vdash t u : B}$$

$$\frac{}{\Gamma, x : A; \Xi \vdash_{\text{val}} x : A}$$

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

## ADEQUATE TYPING RULE

### **Theorem (adequacy lemma):**

- if  $\vdash t : A$  is derivable then  $t \in \llbracket A \rrbracket^{\text{tt}}$ ,
- if  $\vdash_{\text{val}} v : A$  is derivable then  $v \in \llbracket A \rrbracket$ .

## ADEQUATE TYPING RULE

### **Theorem (adequacy lemma):**

- if  $\vdash t : A$  is derivable then  $t \in \llbracket A \rrbracket^{\text{tt}}$ ,
- if  $\vdash_{\text{val}} v : A$  is derivable then  $v \in \llbracket A \rrbracket$ .

*Proof* by induction on the typing derivation.

## ADEQUATE TYPING RULE

### **Theorem (adequacy lemma):**

- if  $\vdash t : A$  is derivable then  $t \in \llbracket A \rrbracket^{\text{ll}}$ ,
- if  $\vdash_{\text{val}} v : A$  is derivable then  $v \in \llbracket A \rrbracket$ .

*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  $t \in \llbracket A \rrbracket^{\perp\perp}$ ,
- if  $\vdash_{\text{val}} v : A$  is derivable then  $v \in \llbracket A \rrbracket$ .

*Proof* by induction on the typing derivation.

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

For example  $\frac{\vdash_{\text{val}} v : A}{\vdash v : A}$  is correct since  $\llbracket A \rrbracket \subseteq \llbracket A \rrbracket^{\perp\perp}$ .

## ADEQUACY OF FOR ALL INTRODUCTION

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

## 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  $v \in \llbracket A[X := \Phi] \rrbracket$  for all  $\Phi$ , and show  $v \in \llbracket \forall X.A \rrbracket$ .

## ADEQUACY OF FOR ALL INTRODUCTION

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

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

This is immediate since  $\llbracket \forall X.A \rrbracket = \bigcap_{\Phi} \llbracket A[X := \Phi] \rrbracket$ .

## ADEQUACY OF FOR ALL INTRODUCTION

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

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

This is immediate since  $\llbracket \forall X.A \rrbracket = \bigcap_{\Phi} \llbracket A[X := \Phi] \rrbracket$ .

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

## ADEQUACY OF FOR ALL INTRODUCTION

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

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

This is immediate since  $\llbracket \forall X.A \rrbracket = \bigcap_{\Phi} \llbracket A[X := \Phi] \rrbracket$ .

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

We suppose  $t \in \llbracket A[X := \Phi] \rrbracket^{\text{uu}}$  for all  $\Phi$ , and show  $t \in \llbracket \forall X.A \rrbracket^{\text{uu}}$ .

## ADEQUACY OF FOR ALL INTRODUCTION

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

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

This is immediate since  $\llbracket \forall X.A \rrbracket = \bigcap_{\Phi} \llbracket A[X := \Phi] \rrbracket$ .

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

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

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

## PROPERTIES OF THE SYSTEM

### **Theorem (normalisation):**

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

## PROPERTIES OF THE SYSTEM

### **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  $v : A$ .

## PROPERTIES OF THE SYSTEM

### **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 (consistency):**

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



# PART III

## SEMANTICAL VALUE RESTRICTION

## DERIVED RULES FOR DEPENDENT FUNCTIONS

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

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

## DERIVED RULES FOR DEPENDENT FUNCTIONS

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

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

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

## DERIVED RULES FOR DEPENDENT FUNCTIONS

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

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

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

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

## SEMANTICAL VALUE RESTRICTION

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

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

## SEMANTICAL VALUE RESTRICTION

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

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

Can this rule be derived in the system?

## SEMANTICAL VALUE RESTRICTION

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

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

Can this rule be derived in the system?

$$\frac{\vdash t : A \quad \vdash t \equiv v}{\vdash v : A} \equiv$$

$$\frac{\vdash_{\text{val}} v : A}{\vdash_{\text{val}} v : v \in A} \stackrel{\epsilon_i}{\vdash v : v \in A} \uparrow \vdash t \equiv v \equiv$$

$$\frac{\vdash v : v \in A \quad \vdash t \equiv v}{\vdash t : t \in A} \equiv$$



## BIORTHOGONAL COMPLETION CLOSED FOR VALUES

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

## BIORTHOGONAL COMPLETION CLOSED FOR VALUES

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

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

## BIORTHOGONAL COMPLETION CLOSED FOR VALUES

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

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

Semantically, this requires that  $v \in \llbracket A \rrbracket^{\text{val}}$  implies  $v \in \llbracket A \rrbracket$ .

## BIORTHOGONAL COMPLETION CLOSED FOR VALUES

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

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

Semantically, this requires that  $v \in \llbracket A \rrbracket^{\text{ll}}$  implies  $v \in \llbracket A \rrbracket$ .

The biorthogonal completion should not introduce new values.

## BIORTHOGONAL COMPLETION CLOSED FOR VALUES

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

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

Semantically, this requires that  $v \in \llbracket A \rrbracket^{\text{ll}}$  implies  $v \in \llbracket A \rrbracket$ .

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  $v \in \llbracket A \rrbracket^{\perp\perp}$  implies  $v \in \llbracket A \rrbracket$  in every realizability model.

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$



## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

- We assume  $v \notin \llbracket A \rrbracket$  and show  $v \notin \llbracket A \rrbracket^{\perp\perp}$ .

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

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

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

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

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

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

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

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

## THE NEW INSTRUCTION TRICK

We do not have  $v \notin \llbracket A \rrbracket$  implies  $v \notin \llbracket A \rrbracket^{\perp\perp}$  in every realizability model.

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

$$\delta_{v,w} * \pi > v * \pi \quad \text{iff} \quad v \neq w.$$

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

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

## WELL-DEFINED CONSTRUCTION OF EQUIVALENCE AND REDUCTION

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



## WELL-DEFINED CONSTRUCTION OF EQUIVALENCE AND REDUCTION

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

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

$$(\rightarrow_i) = (>) \cup \{(\delta_{v,w} * \pi, v * \pi) \mid \exists j < i, v \not\equiv_j w\}$$

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

We then take

$$(\rightarrow) = \bigcup_{i \in \mathbb{N}} (\rightarrow_i) \quad \text{and} \quad (\equiv) = \bigcap_{i \in \mathbb{N}} (\equiv_i).$$

# PART IV

## LOCAL SUBTYPING AND CHOICE OPERATORS

## A SYNTAX-DIRECTED PRESENTATION

PML<sub>2</sub> 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<sub>2</sub> 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”.

$$\frac{\Gamma; \Xi \vdash t : A \quad \alpha \notin \text{FV}(\Gamma; \Xi) \quad \Xi \vdash t \equiv v}{\Gamma; \Xi \vdash t : \forall \alpha. A}$$

$$\frac{\Gamma; \Xi \vdash t : \forall \alpha. A}{\Gamma; \Xi \vdash t : A[\alpha := u]}$$

## A SYNTAX-DIRECTED PRESENTATION

PML<sub>2</sub> 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”.

$$\frac{\Gamma; \Xi \vdash t : A \quad \alpha \notin \text{FV}(\Gamma; \Xi) \quad \Xi \vdash t \equiv v}{\Gamma; \Xi \vdash t : \forall \alpha. A}$$

$$\frac{\Gamma; \Xi \vdash t : \forall \alpha. A}{\Gamma; \Xi \vdash t : A[\alpha := u]}$$

**Solution:** handle these connectives using *local subtyping*.

## A SYNTAX-DIRECTED PRESENTATION

PML<sub>2</sub> 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”.

$$\frac{\Gamma; \Xi \vdash t : A \quad a \notin \text{FV}(\Gamma; \Xi) \quad \Xi \vdash t \equiv v}{\Gamma; \Xi \vdash t : \forall a. A}$$

$$\frac{\Gamma; \Xi \vdash t : \forall a. A}{\Gamma; \Xi \vdash t : A[a := u]}$$

**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).

## 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.

## 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  $\Xi \vdash t : A \sqsubseteq B$ .
- 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  $v \in \llbracket A \rrbracket$  such that  $\llbracket t[x := a] \rrbracket \notin \llbracket B \rrbracket^{\sharp\sharp}$  (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  $\Xi \vdash \varepsilon_{x \in A}(x \notin B) : A \subseteq B$ .

## EXAMPLES OF SYNTAX-DIRECTED TYPING RULES

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

## EXAMPLES OF SYNTAX-DIRECTED TYPING RULES

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

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

## EXAMPLES OF SYNTAX-DIRECTED TYPING RULES

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

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

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

## EXAMPLES OF SYNTAX-DIRECTED TYPING RULES

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

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

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

$$\frac{\Xi \vdash v : A \quad \Xi \vdash C_k[v] : [C_k : A] \subseteq B}{\Xi \vdash C_k[v] : B} +_i$$

$$\frac{\Xi \vdash v : \{l_k : A; \dots\}}{\Xi \vdash v.l_k : A} \times_e$$

## EXAMPLES OF SYNTAX-DIRECTED (LOCAL) SUBTYPING RULES

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X. A \subseteq B} \nu_l$$

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

## EXAMPLES OF SYNTAX-DIRECTED (LOCAL) SUBTYPING RULES

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X. A \subseteq B} \forall_l$$

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

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

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



## EXAMPLES OF SYNTAX-DIRECTED (LOCAL) SUBTYPING RULES

$$\frac{\Xi \vdash t : A[X := C] \subseteq B}{\Xi \vdash t : \forall X. A \subseteq B} \nu_l$$

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

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

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

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

$$\frac{\Xi \vdash t : A \subseteq B \quad \Xi \vdash t \equiv u \quad \Xi \vdash t \equiv v}{\Xi \vdash t : A \subseteq u \in B} \epsilon_r$$

## 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

## GENERAL RECURSION AND FIXPOINT UNFOLDING

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

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

## GENERAL RECURSION AND FIXPOINT UNFOLDING

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

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

**Problem:** we need to work with infinite proofs.

## GENERAL RECURSION AND FIXPOINT UNFOLDING

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

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

**Problem:** we need to work with infinite proofs.

We introduce a cyclic structure in our proofs.

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

## ORDINALS AND INDUCTIVE TYPES

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

## ORDINALS AND INDUCTIVE TYPES

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

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



## ORDINALS AND INDUCTIVE TYPES

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

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

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

## EXAMPLE OF CYCLIC PROOF

Let us consider the “map” function:  $\varphi m. \lambda f. \lambda l. [l | [] \rightarrow [] | x :: l \rightarrow f x :: m f l]$ .

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  $\mu_{\alpha} L. [([], \{ \}) | (::) : X \times L]$ .

# 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>

*PML<sub>2</sub>: 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>

Thanks!