

# Functional programming and type systems

Master Parisien de Recherche en Informatique  
(MPRI)

Giuseppe Castagna

CNRS  
Laboratoire PPS  
Université Paris Diderot

Janvier 2012

# Subtyping

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects
- 4 Parametric Types
- 5 Generics
- 6 Recursive Types
- 7 Bibliography

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects
- 4 Parametric Types
- 5 Generics
- 6 Recursive Types
- 7 Bibliography

# Simply Typed $\lambda$ -calculus

## Syntax

<i>Types</i>	$T ::= T \rightarrow T$	function types
	$\text{Bool} \mid \text{Int} \mid \text{Real} \mid \dots$	basic types
<i>Terms</i>	$a, b ::= \text{true} \mid \text{false} \mid 1 \mid 2 \mid \dots$	constants
	$  x$	variable
	$  ab$	application
	$  \lambda x:T.a$	abstraction

## Reduction

*Contexts*  $C[] ::= [] \mid a[] \mid []a \mid \lambda x:t.[]$

BETA  
 $(\lambda x:T.a)b \longrightarrow a[b/x]$

CONTEXT  
 $\frac{a \longrightarrow b}{C[a] \longrightarrow C[b]}$

## Typing

$$\begin{array}{c} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array} \qquad \begin{array}{c} \rightarrow\text{INTRO} \\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x : S. a : S \rightarrow T} \end{array} \qquad \begin{array}{c} \rightarrow\text{ELIM} \\ \frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T} \end{array}$$

(plus the typing rules for constants).

### Theorem (Subject Reduction)

If  $\Gamma \vdash a : T$  and  $a \longrightarrow^* b$ , then  $\Gamma \vdash b : T$ .

We will essentially focus on the subject reduction property (a.k.a. *type preservation*), though well-typed programs also satisfy *progress*:

### Theorem (Progress)

If  $\emptyset \vdash a : T$  and  $a \not\rightarrow$ , then  $a$  is a value

where a value is either a constant or a lambda abstraction

$$v ::= \lambda x : T. a \mid \text{true} \mid \text{false} \mid 1 \mid 2 \mid \dots$$

# Type checking algorithm

The deduction system is *syntax directed* and satisfies the *subformula property*.  
As such it describes a deterministic algorithm.

```
let rec typecheck gamma = function
  | x -> gamma(x) (* Var rule *)
  |  $\lambda x:T.a$  -> typecheck (gamma, x:T) a (* Intro rule *)
  |  $ab$  -> let  $T_1 \rightarrow T_2 =$  typecheck gamma a in (* Elim rule *)
            let  $T_3 =$  typecheck gamma b in
            if  $T_1 == T_3$  then  $T_2$  else fail
```

**Exercise.** Write the *typecheck* function for the following definitions:

```
type stype = Int | Bool | Arrow of stype * stype
```

```
type term =
  Num of int | BVal of bool | Var of string
  | Lam of string * stype * term | App of term * term
```

```
exception Error
```

Use `List.assoc` for environments.

# Subtyping

The rule for application requires the argument of the function to be *exactly of the same type* as the domain of the function:

$$\frac{\rightarrow\text{ELIM} \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

So, for instance, we cannot

- Apply a function of type `Int`  $\rightarrow$  `Int` to an argument of type `Odd` even though every odd number is an integer number, too.
- If we have records, apply the function  $\lambda x : \{\ell : \text{Int}\}. (3 + x.\ell)$  to a record of type  $\{\ell : \text{Int}, \ell' : \text{Bool}\}$
- If we are in OOP, send a message defined for objects `Persons` to an instance of the subclass `Student`.

## Subtyping polymorphism

We need a kind of polymorphism different from the ML one (parametric polymorphism).

- Define a pre-order (ie, a reflexive and transitive binary relation)  $\leq$  on types:  $\leq \subset \text{Types} \times \text{Types}$  (some literature uses the notation  $<:$ )

- This *subtyping relation* has two possible interpretations:

**Containment:** If  $S \leq T$ , then every value of type  $S$  *is also* of type  $T$ .

For instance an odd number *is also* an integer, a student *is also* a person.

Sometimes called a “**is\_a**” relation.

**Substitutability:** If  $S \leq T$ , then every value of type  $S$  can be *safely* used where a value of type  $T$  is expected.

Where “safely” means, without disrupting type preservation and progress.

- We'll see how each interpretation has a formal counterpart.

- We suppose to have a predefined preorder  $\mathcal{B} \subset \text{Basic} \times \text{Basic}$  for basic types (given by the language designer).

For instance take the reflexive and transitive closure of  $\{(\text{Odd}, \text{Int}), (\text{Even}, \text{Int}), (\text{Int}, \text{Real})\}$

- To extend it to function types, we resort to the substitutability interpretation. We will try to deduce when we can safely replace a function of some type by a term of a different type

# Subtyping of arrows: intuition

## Problem

Determine for which type  $S$  we have  $S \leq T_1 \rightarrow T_2$

Let  $g : S$  and  $f : T_1 \rightarrow T_2$ . Let us follow the **substitutability interpretation**:

- 1 If  $a : T_1$ , then we can apply  $f$  to  $a$ . If  $S \leq T_1 \rightarrow T_2$ , then we can apply  $g$  to  $a$ , as well.  
 $\Rightarrow g$  is a function, therefore  $S = S_1 \rightarrow S_2$
- 2 If  $a : T_1$ , then  $f(a)$  is well typed. If  $S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$ , then also  $g(a)$  is well-typed.  $g$  expects arguments of type  $S_1$  but  $a$  is of type  $T_1$   
 $\Rightarrow$  we can safely use  $T_1$  where  $S_1$  is expected, ie  $T_1 \leq S_1$
- 3  $f(a) : T_2$ , but since  $g$  returns results in  $S_2$ , then  $g(a) : S_2$ . If I use  $g$  where  $f$  is expected, then it must be safe to use  $S_2$  results where  $T_2$  results are expected  
 $\Rightarrow S_2 \leq T_2$  must hold.

## Solution

$$S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \quad \Leftrightarrow \quad T_1 \leq S_1 \wedge S_2 \leq T_2$$

$$S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \Leftrightarrow T_1 \leq S_1 \wedge S_2 \leq T_2$$

Notice the different orientation of containment on domains and co-domains.

We say that the type constructor  $\rightarrow$  is

- *covariant* on codomains, since it preserves the direction of the relation;
- *contravariant* on domains, since it reverses the direction of the relation.

## Containment interpretation:

The *containment interpretation* yields exactly the same relation as obtained by the *substitutability interpretation*. For instance a function that maps integers to integers ...

- *is also* a function that maps integers to reals: it returns results in `Int` so they will be also in `Real`.

$\text{Int} \rightarrow \text{Int} \leq \text{Int} \rightarrow \text{Real}$  (covariance of the codomains)

- *is also* a function that maps odds to integers: when fed with integers it returns integers, so will do the same when fed with odd numbers.

$\text{Int} \rightarrow \text{Int} \leq \text{Odd} \rightarrow \text{Int}$  (contravariance of the codomains)

# Subtyping deduction system

$$\text{BASIC} \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2}$$

$$\text{ARROW} \frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

$$\text{REFL} \frac{}{T \leq T}$$

$$\text{TRANS} \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}$$

This system is neither *syntax directed* nor satisfies the *subformula* property

How do we define an algorithm to check the subtyping relation?

## Theorem (Admissibility of Refl and Trans)

*In the system composed just by the rules Arrow and Basic:*

- 1)  $T \leq T$  is provable for all types  $T$
- 2) If  $T_1 \leq T_2$  and  $T_2 \leq T_3$  are provable, so is  $T_1 \leq T_3$ .

The rules Refl and Trans are *admissible*

# Proof of admissibility of transitivity

Suppose we already proved that REFL is admissible (and thus useless). We prove the theorem by induction on depth of the proof of  $T_1 \leq T_3$  that there exists a proof of  $T_1 \leq T_3$  of smaller or equal depth and that does not use TRANS. We proceed by a case analysis on the last applied rule

**Case BASIC** The proof already does not use transitivity.

**Case ARROW** Straightforward application of induction hypothesis.

**Case TRANS** Two subcases:

- 1  $T_1 \leq T_3$  is of the form  $B_1 \leq B_2$ . Since  $\mathcal{B}$  is transitive then we can directly prove  $T_1 \leq T_3$  by a single application of BASIC.
- 2  $T_1 \leq T_3$  is of the form  $S_1 \rightarrow S'_1 \leq S_3 \rightarrow S'_3$ . Since the last applied rule is transitivity, then it must have the form

$$\text{TRANS} \frac{S_1 \rightarrow S'_1 \leq S_2 \rightarrow S'_2 \quad S_2 \rightarrow S'_2 \leq S_3 \rightarrow S'_3}{S_1 \rightarrow S'_1 \leq S_3 \rightarrow S'_3}$$

Apply the induction hypothesis and deduce that you can prove  $S_3 \leq S_2$ ,  $S_2 \leq S_1$ , and  $S'_1 \leq S'_2$ ,  $S'_2 \leq S'_3$ . By induction we can prove without transitivity  $S_3 \leq S_1$  and  $S'_1 \leq S'_3$ . The result follows by an application of the ARROW rule.

# Type system

We defined the subtyping relation and we know how to decide it. How do we use it for typing our programs?

$$\begin{array}{c} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array} \qquad \frac{\text{→INTRO} \quad \Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x : S. a : S \rightarrow T} \qquad \frac{\text{→ELIM} \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$
$$\frac{\text{SUBSUMPTION} \quad \Gamma \vdash a : S \quad S \leq T}{\Gamma \vdash a : T}$$

This corresponds to the *containment relation*:

if  $S \leq T$  and  $a$  is of type  $S$  then  $a$  *is also* of type  $T$

**Subject reduction:** If  $\Gamma \vdash a : T$  and  $a \longrightarrow^* b$ , then  $\Gamma \vdash b : T$ .

**Progress property:** If  $\emptyset \vdash a : T$  and  $a \not\rightarrow$ , then  $a$  is a value

# Typing algorithm

$$\begin{array}{c} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array} \quad \begin{array}{c} \rightarrow\text{INTRO} \\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x : S. a : S \rightarrow T} \end{array} \quad \begin{array}{c} \rightarrow\text{ELIM}_{\leq} \\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \rightarrow T \quad \Gamma \vdash_{\mathcal{A}} b : U \quad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T} \end{array}$$
  
$$\begin{array}{c} \rightarrow\text{ELIM} \\ \frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T} \end{array} \quad \begin{array}{c} \text{SUBSUMPTION} \\ \frac{\Gamma \vdash a : S \quad S \leq T}{\Gamma \vdash a : T} \end{array}$$

Subsumption makes the type system non-algorithmic:

- it is not *syntax directed*: subsumption can be applied whatever the term.
- it does not satisfy the *subformula property*: even if we know that we have to apply subsumption which  $T$  shall we choose?

How do we define the typechecking algorithm?

# Typing algorithm

$$\begin{array}{c} \text{VAR} \\ \Gamma \vdash_{\mathcal{A}} x : \Gamma(x) \end{array} \quad \begin{array}{c} \rightarrow\text{INTRO} \\ \frac{\Gamma, x : S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x : S. a : S \rightarrow T} \end{array} \quad \begin{array}{c} \rightarrow\text{ELIM}_{\leq} \\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \rightarrow T \quad \Gamma \vdash_{\mathcal{A}} b : U \quad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T} \end{array}$$

- 1 The system is algorithmic: it describes a typing algorithm (exercise: program `typecheck` and `subtype` by using the previous structures)
- 2 The system conforms the substitutability interpretation: we *use* an expression of a subtype  $U$  where a supertype  $S$  is expected (note “use” = elimination rule).

How do we relate the two systems?

For subtyping, admissibility ensured that the system and the algorithm prove the same judgements. Here it is no longer true. For instance:

$\emptyset \vdash \lambda x : \text{Int}. x : \text{Odd} \rightarrow \text{Real}$     but     $\emptyset \not\vdash_{\mathcal{A}} \lambda x : \text{Int}. x : \text{Odd} \rightarrow \text{Real}$ .

**This is expected:** Algorithm = one type returned for each typable term.

# Soundness and completeness of the typing algorithm

$a$  is typable by  $\vdash \iff a$  is typable by  $\vdash_{\mathcal{A}}$

$\Leftarrow$  = soundness

$\Rightarrow$  = completeness

## Theorem (Soundness)

If  $\Gamma \vdash_{\mathcal{A}} a : T$ , then  $\Gamma \vdash a : T$

**Proof.** Simply observe that  $\rightarrow\text{ELIM}_{\leq}$  is *derivable* in  $\vdash$ :

$$\rightarrow\text{ELIM} \frac{\Gamma \vdash a : S \rightarrow T \quad \frac{\Gamma \vdash b : U \quad U \leq S}{\Gamma \vdash b : S} \text{SUBSUMPTION}}{\Gamma \vdash ab : T}$$

(this is just one possible way to derive  $\rightarrow\text{ELIM}_{\leq}$ : can you find a second one?)

Notice the difference between *admissible* and *derivable* rules:

**Derivable:** A rule is *derivable* (or *derived*) when it can be obtained by a given *composition* of other rules.

**Admissible:** A rule is *admissible* when it is possible to derive its conclusion from its premisses by using other rules.

A derivable rule is also admissible but not viceversa

## Theorem (Completeness)

*If  $\Gamma \vdash a : T$ , then  $\Gamma \vdash_{\mathcal{A}} a : S$  with  $S \leq T$*

To prove the theorem we need the following lemma

**Lemma.** *If  $\Gamma \vdash a : T$  is provable, then there exists a proof of it that never uses two consecutive applications of the SUBSUMPTION rule.*

**Proof of the theorem.** We prove by induction on the depth of the proof of  $\Gamma \vdash a : T$  that there exists  $S \leq T$  and a proof of  $\Gamma \vdash_{\mathcal{A}} a : S$  of smaller or equal depth. If the depth of the proof of  $\Gamma \vdash_{\mathcal{A}} a : S$  is 1, then it consists of a single application of VAR, so it is also a proof in  $\vdash_{\mathcal{A}}$ .

If the depth is strictly larger than 1, then we proceed by a case analysis on the form of  $a$  and on the last applied rule. W.l.o.g we can consider that the hypotheses of the previous lemma hold.

# Completeness of the type checking algorithm II

- If  $a = x$ , then the only possibility (no consecutive application of subsumption) is that the proof has the form

$$\text{SUBSUMPTION} \frac{\text{VAR} \frac{}{\Gamma \vdash x : \Gamma(x)} \quad \Gamma(x) \leq T}{\Gamma \vdash x : T}$$

Then take  $\Gamma(x)$  for  $S$ , we have that  $\Gamma \vdash_{\mathcal{A}} a : \Gamma(x)$  is provable.

- If  $a = a_1 a_2$  then we have two subcases:
  - (1.) the proof ends by an application of  $\rightarrow\text{ELIM}$

$$\rightarrow\text{ELIM} \frac{\Gamma \vdash a_1 : S \rightarrow T \quad \Gamma \vdash a_2 : S}{\Gamma \vdash a_1 a_2 : T}$$

By induction  $\Gamma \vdash_{\mathcal{A}} a_1 : S' \rightarrow T' \leq S \rightarrow T$  and  $\Gamma \vdash_{\mathcal{A}} a_2 : S'' \leq S$ . By the definition of subtyping  $S'' \leq S \leq S'$  and  $T' \leq T$ . We can then use  $\rightarrow\text{ELIM}_{\leq}$ , to deduce  $\Gamma \vdash_{\mathcal{A}} a_1 a_2 : T' \leq T$ .

- (2.) the proof ends by an application of subsumption. By the previous lemma the only possible case is the following one:

$$\text{SUBSUMPTION} \frac{\text{→ELIM} \frac{\Gamma \vdash a_1 : S \rightarrow U \quad \Gamma \vdash a_2 : S}{\Gamma \vdash a_1 a_2 : U} \quad U \leq T}{\Gamma \vdash a_1 a_2 : T}$$

Then proceed as in the previous case.

- If  $a = \lambda x:S.b$ , then proceed as in the previous case (left as an exercise).

# Minimum type and soundness

## Corollary (Minimum type)

*If  $\Gamma \vdash_{\mathcal{A}} a : T$  then  $T = \min\{S \mid \Gamma \vdash a : S\}$*

**Proof.** Let  $\mathcal{S} = \{S \mid \Gamma \vdash a : S\}$ . Soundness ensures that  $\mathcal{S}$  is not empty. Completeness states that  $T$  is a lower bound of  $\mathcal{S}$ . Minimality follows by using soundness once more.

The corollary above explains that the typing algorithm works with the minimum types of the terms. It keeps track of the best type information available

## Theorem (Algorithmic subject reduction)

*If  $\Gamma \vdash_{\mathcal{A}} a : T$  and  $a \longrightarrow^* b$ , then  $\Gamma \vdash_{\mathcal{A}} b : S$  with  $S \leq T$ .*

The theorem above explains that the computation reduces the minimum type of a program. As such it increases the type information about it.

The proof of (algorithmic) subject reduction cannot be done directly by induction since substitutions may increase the size of the terms. We need a substitution lemma.

## Lemma (Substitution)

If  $\Gamma, x : S \vdash_{\mathcal{A}} a : T$  and  $\Gamma \vdash b : S' \leq S$ , then  $\Gamma \vdash_{\mathcal{A}} a[x/b] : T'$  with  $T' \leq T$ .

*Proof. Exercise.*

## Proof of algorithmic subject reduction.

It suffices to prove the theorem for  $\longrightarrow$ . The result for  $\longrightarrow^*$  follows by a straightforward induction on the number of steps of reduction. We proceed by induction on the structure of  $a$ .

- $a = x$ : straightforward since  $a$  is in normal form (that is, it cannot be reduced).
- $a = \lambda x:S.b$ : *Exercise*.
- $a = a_1 a_2$  where  $\Gamma \vdash_{\mathcal{A}} a_1 : U \rightarrow T$ ,  $\Gamma \vdash_{\mathcal{A}} a_2 : V \leq U$ . Three subcases:
  - $a_1 \longrightarrow a'_1$ . By induction hypothesis  $\Gamma \vdash_{\mathcal{A}} a_1 : U' \rightarrow T'$  with  $U \leq U'$  and  $T' \leq T$ . Since  $V \leq U \leq U'$  then by  $\rightarrow\text{ELIM}_{\leq}$  we obtain  $\Gamma \vdash_{\mathcal{A}} a'_1 a_2 : T' \leq T$ .
  - $a_2 \longrightarrow a'_2$ . By induction hypothesis  $\Gamma \vdash_{\mathcal{A}} a'_2 : V' \leq V$ . Since  $V' \leq V \leq U$  then by  $\rightarrow\text{ELIM}_{\leq}$  we obtain  $\Gamma \vdash_{\mathcal{A}} a_1 a'_2 : T$ .
  - $a_1 = \lambda x:U.a_3$  and  $b = a_3[x/a_1]$  with  $\Gamma, x : U \vdash_{\mathcal{A}} a_3 : T$ . The result follows from the application of the substitution lemma.

Subject reduction of the “logical” system is a corollary of the algorithmic one

## Corollary (Subject reduction)

*If  $\Gamma \vdash a : T$  and  $a \longrightarrow^* b$ , then  $\Gamma \vdash b : T$ .*

Proof.

$\Gamma \vdash a : T$	
$\Rightarrow \Gamma \vdash_{\mathcal{A}} a : S \leq T$	completeness
$\Rightarrow \Gamma \vdash_{\mathcal{A}} b : U \leq S$	algorithmic subject reduction
$\Rightarrow \Gamma \vdash b : U$	soundness
$\Rightarrow \Gamma \vdash b : T$	subsumption (since $U \leq T$ )

## Summary for simply-typed $\lambda$ -calculs + $\leq$

- The *containment* interpretation of the subtyping relation corresponds to the “logical” view of the type system embodied by subsumption.
- The *substitutability* interpretation of the subtyping relation corresponds to the “algorithmic” view of the type system.
- To *define* the type system one usually starts from the “logical” system, which is simpler since subtyping is concentrated in the subsumption rule
- To *implement* the type system one passes to the substitutability view. Subsumption is eliminated and the check of the subtyping relation is distributed in the places where values are used/consumed. This in general correspond to embed subtype checking into elimination rules.
- The obtained algorithm works on the *minimum types* of the logical system
- Computation reduces the (algorithmic) type thus increasing type information (the result of a computation represents the best possible type information: it is the *singleton type* containing the result).
- The last point makes *dynamic dispatch* (aka, dynamic binding) meaningful.

- 1 Simple Types
- 2 Extensions: products, records, references**
- 3 A simple model of objects
- 4 Parametric Types
- 5 Generics
- 6 Recursive Types
- 7 Bibliography

## Syntax

*Types*  $T ::= \dots \mid T \times T$  product types

*Terms*  $a, b ::= \dots$   
|  $(a, a)$  pair  
|  $\pi_i(a)$  ( $i=1,2$ ) projection

## Reduction

$$\pi_i((a_1, a_2)) \longrightarrow a_i \quad (i=1,2)$$

## Typing

$$\frac{\times \text{INTRO} \quad \Gamma \vdash a_1 : T_1 \quad \Gamma \vdash a_2 : T_2}{\Gamma \vdash (a_1, a_2) : T_1 \times T_2}$$
$$\frac{\times \text{ELIM}_i \quad \Gamma \vdash a : T_1 \times T_2}{\Gamma \vdash \pi_i(a) : T_i} \quad (i=1,2)$$

## Subtyping

$$\frac{\text{PROD} \quad S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2}$$

**Exercise:** Check whether the above rule is compatible with the containment and/or the substitutability interpretation of the subtyping relation.

The subtyping rule above is also algorithmic. Similarly, for the typing rules there is no need to embed subtyping in the elimination rules since  $\pi_i$  is an operator that works on all products, not a particular one (cf. with the application of a function, which requires a particular domain).

Of course subject reduction and progress still hold.

**Exercise:** Define values and reduction contexts for this extension.

# Records

Up to now subtyping rules « lift » the subtyping relation  $\mathcal{B}$  on basic types to constructed types. But if  $\mathcal{B}$  is the identity relation, so is the whole subtyping relation. Record subtyping is non-trivial even when  $\mathcal{B}$  is the identity relation.

## Syntax

<i>Types</i>	$T ::= \dots \mid \{l : T, \dots, l : T\}$	record types
<i>Terms</i>	$a, b ::= \dots$	
	$\{l = a, \dots, l = a\}$	record
	$a.l$	field selection

## Reduction

$$\{\dots, l = a, \dots\}.l \longrightarrow a$$

## Typing

{ }INTRO

$$\Gamma \vdash a_1 : T_1 \dots \Gamma \vdash a_n : T_n$$

$$\frac{}{\Gamma \vdash \{l_1 = a_1, \dots, l_n = a_n\} : \{l_1 : T_1, \dots, l_n : T_n\}}$$

{ }ELIM

$$\Gamma \vdash a : \{\dots, l : T, \dots\}$$

$$\frac{}{\Gamma \vdash a.l : T}$$

# Record Subtyping

To define subtyping we resort once more on the substitutability relation. A record is “used” by selecting one of its labels.

We can replace some record by a record of different type if in the latter we can select the same fields as in the former and their contents can substitute the respective contents in the former.

## Subtyping

RECORD

$$\frac{S_1 \leq T_1 \dots S_n \leq T_n}{\{\ell_1:S_1, \dots, \ell_n:S_n, \dots, \ell_{n+k}:S_{n+k}\} \leq \{\ell_1:T_1, \dots, \ell_n:T_n\}}$$

**Exercise.** Which are the algorithmic typing rules?

References or *locations* are ranged over by  $\ell$  and can be: allocated and initialized (`ref`), written (`:=`), read (`!`)

<i>Terms</i>	$a, b$	$::=$	...	
			$\ell^T$	Location
			$\text{ref}^T a$	Allocate and initialize
			$a := b$	Write
			$!a$	Read
			$()$	Unit value
<i>Types</i>	$T$	$::=$	...	
			$\text{ref} T$	Location type
			$\text{unit}$	Side effect

Locations are not available to the programmer (they can only be manipulated by the other constructions).

## References II

We need *strict evaluation* (function arguments must be values) since otherwise  $(\lambda x:T.(x:=1;!x))(\text{ref } 3)$  might reduce to  $(\text{ref } 3):=1;!(\text{ref } 3)$ .

### Syntactic sugar

$$\begin{aligned} \text{let } x:T = a \text{ in } b &\stackrel{\text{def}}{=} (\lambda x:T.b)a \\ a;b &\stackrel{\text{def}}{=} \text{let } x:T = a \text{ in } b \quad (x \# b, T \text{ any type}). \end{aligned}$$

### Reduction

$$\text{Values} \quad v ::= x \mid \lambda x:T.a \mid \ell^T$$

$$\text{Eval Contexts} \quad E[] ::= [] \mid vE[] \mid E[]a \mid \text{ref}^T E[] \mid E[]:=a \mid v:=E[] \mid !E[]$$

Let  $\sigma$  denote a *store*, that is a mapping from locations to values.

$$\begin{array}{ll} \text{(CBV)} & (\lambda x:T.a)v \mid \sigma \longrightarrow a[x/v] \mid \sigma \\ \text{(ALLOC)} & \text{ref}^T v \mid \sigma \longrightarrow \ell \mid \sigma[\ell^T \mapsto v] \quad \text{if } \ell^T \notin \text{dom}(\sigma) \\ \text{(WRITE)} & \ell := v \mid \sigma \longrightarrow () \mid \sigma[\ell \mapsto v] \\ \text{(READ)} & !\ell \mid \sigma \longrightarrow \sigma(\ell) \mid \sigma \end{array} \quad \begin{array}{l} \text{(CONTEXT)} \\ a \longrightarrow b \\ \hline E[a] \longrightarrow E[b] \end{array}$$

## Typing

$$\text{LOC} \quad \frac{}{\Gamma \vdash \ell^T : T}$$

$$\text{REF} \quad \frac{\Gamma \vdash a : T}{\Gamma \vdash \text{ref}^T a : \text{ref } T}$$

$$\text{GET} \quad \frac{\Gamma \vdash a : \text{ref } T}{\Gamma \vdash !a : T}$$

$$\text{SET} \quad \frac{\Gamma \vdash a : \text{ref } T \quad \Gamma \vdash b : T}{\Gamma \vdash a := b : \text{unit}}$$

## Reference subtyping

How to define subtyping for references? A wrong answer is “covariantly.”

$$\frac{\text{WRONG} \quad S \leq T}{\text{ref } S \leq \text{ref } T}$$

This rule is unsound. For example:

```
let x: ref{a:Int,b:Bool} = ref{a=1,b=true} in
let y: ref{a:Int} = x in
  y:={a=0};
  (!x).b
```

Or, without the `let` syntactic sugar.

```
(λx:ref{a:Int,b:Bool}.
  (λy:ref{a:Int}.y:={a=0})x ; (!x).b
)(ref{a:Int,b:Bool}{a=1,b=true})
```

With the rule [WRONG] the term above is typable but reduces to `{a=0}.b`

## Soundness requires invariance of reference types

- Type `ref T` is akin to the record type  $\{\text{set}: T \rightarrow () , \text{get}: () \rightarrow T\}$ 
  - the `set` method imposes contravariance
  - the `get` method imposes covariance

It is possible to use one of the two disciplines by suppressing the corresponding method.

- read-only locations can be covariantly specialized (no `set` method);
  - write-only locations can be contravariantly specialized (no `get` method)
- Reference invariance can be met in OOLs that allow the addition of new instance variables/fields in subclasses, but forbid the specialization of the types of the existing ones (exercise: use the example in the previous slide to define two classes with an instance variable and an inherited method to show unsoundness of field type specialization).
- The problem is similar to that of references in a polymorphic calculus, which must have a fixed type: see the so-called *value restriction* by Andrew K. Wright [1995].

# Typing references with subtyping

As usual, to enrich the typing relation with subtyping it suffices to add the subsumption rule.

## Algorithmic typing rules

$$\frac{\text{REF}_{\leq} \quad \Gamma \vdash a : S \quad S \leq T}{\Gamma \vdash \text{ref}^T a : \text{ref } T} \quad \frac{\text{SET}_{\leq} \quad \Gamma \vdash a : \text{ref } T \quad \Gamma \vdash b : S \quad S \leq T}{\Gamma \vdash a := b : \text{unit}}$$

Notice that the following rule is unsound in presence of subsumption

$$\frac{\text{REF-INTRO} \quad \Gamma \vdash a : T}{\Gamma \vdash \text{ref } a : \text{ref } T}$$

similarly it makes subject reduction fail with algorithmic system since reduction is allowed under `ref` terms.

## Whence the need to fix the type of allocations by indexing ref-terms

(the types indexing locations,  $\ell^T$ , instead are not necessary but just convenient to avoid the addition of a new typing environment for locations).

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects**
- 4 Parametric Types
- 5 Generics
- 6 Recursive Types
- 7 Bibliography

## An example in Scala

```
class Point(a: Int, b: Int) {
  var x: Int = a           // mutable instance variable
  var y: Int = b           // mutable instance variable
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))           // method
  def erase(): Point = { x = 0; y = 0; return this }       // method
  def move(dx: Int): Point = new Point(x+dx,y)           // method
  def isOrigin(): Boolean = (this.norm == 0)             // method
}

class ColPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c           // non-mutable instance variable
  def isWhite(): Boolean = c == "white"
  override def norm(): Double = {
    if (this.isWhite) return 0 else return sqrt(pow(x,2)+pow(y,2))
  }
  override def move(dx: Int): ColPoint=new ColPoint(x+dx,y,"red")
}
```

Notice the late binding of `norm` in `isOrigin`.

# Encoding of Point objects

## Recursive type for Point:

```
type Point = { norm: Double ,           // actually unit -> Double
               erase: Point ,          // actually unit -> Point
               move: Int -> Point ,
               isOrigin: Boolean       // actually unit -> Boolean
            }
```

## Generator:

$$\mathcal{G}_{\text{Point}} = \lambda \text{mkobj} : \{x : \text{ref Int}, y : \text{ref Int}\} \rightarrow \text{Point}.$$
$$\lambda \text{state} : \{x : \text{ref Int}, y : \text{ref Int}\}.$$
$$\{ \text{norm} = \sqrt{! \text{state}.x^2 + ! \text{state}.y^2},$$
$$\text{erase} = \text{state}.x := 0; \text{state}.y := 0; \text{mkobj}(\text{state}),$$
$$\text{move} = \lambda dx : \text{Int}. \text{mkobj}(\{x = ! \text{state}.x + dx, y = ! \text{state}.y\})$$
$$\text{isOrigin} = (\text{mkobj}(\text{state}).\text{norm} == 0) \}$$

## Object creation = generator wrapping:

$$\text{new Point} = \lambda(a, b). \mathbf{Y}(\mathcal{G}_{\text{Point}}) \{x = \text{ref}^{\text{Int}} a, y = \text{ref}^{\text{Int}} b\}$$

# Excursus: on the fix-point combinator

## Fixpoints:

$x$  is a *fixpoint* of  $f$  iff  $x = f(x)$

## Static and dynamic semantics:

$$\begin{array}{ll} \text{FIX} & [\text{FIX}] \\ \mathbf{Y}(a) \longrightarrow a(\mathbf{Y}(a)) & \mathbf{Y} : (T \rightarrow T) \rightarrow T \end{array}$$

## Encoding of rec:

$$\begin{aligned} \text{rec } x : T = a & \stackrel{\text{def}}{=} \mathbf{Y}(\lambda x : T. a) \\ & \longrightarrow (\lambda x : T. a)(\mathbf{Y}(\lambda x : T. a)) \\ & \longrightarrow a[(\mathbf{Y}(\lambda x : T. a))/x] \\ & \equiv a[(\text{rec } x : T = a)/x] \end{aligned}$$

## Compare the definitions

```
class Point(a: Int, b: Int) {  
  var x: Int = a  
  var y: Int = b  
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))  
  def erase(): Point = { x = 0; y = 0; return this}  
  def move(dx: Int): Point = new Point(x+dx,y)  
  def isOrigin(): Boolean = (this.norm == 0)  
}
```

### Generator:

$$\mathcal{G}_{\text{Point}} = \lambda \text{mkobj}. \{x : \text{ref Int}, y : \text{ref Int}\} \rightarrow \text{Point}.$$
$$\lambda \text{state}. \{x : \text{ref Int}, y : \text{ref Int}\}.$$
$$\{ \text{norm} = \sqrt{!state.x^2 + !state.y^2},$$
$$\text{erase} = \text{state.x}:=0; \text{state.y}:=0; \text{mkobj}(\text{state}),$$
$$\text{move} = \lambda dx : \text{Int}. \text{mkobj}(\{x=!state.x+dx, y=!state.y\})$$
$$\text{isOrigin} = (\text{mkobj}(\text{state}).\text{norm}==0) \}$$

## Generator:

$\mathcal{G}_{\text{ColPoint}} =$

$\lambda \text{mkobj} : \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\} \rightarrow \text{ColPoint}.$

$\lambda \text{state} : \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\}.$

let super:Point =  $\mathcal{G}_{\text{Point}}(\lambda s. \text{mkobj}(\text{state} \oplus s)) \text{state}$

in super  $\oplus$

{norm = if  $\text{mkobj}(\text{state}).\text{isWhite}$  then 0 else ... ,

move =  $\lambda dx : \text{Int}.\text{mkobj}(\{x = !\text{state}.x + dx, y = !\text{state}.y, c = \text{"red"}\})$

isWhite =  $\text{state}.c == \text{"white"}$

}

## Notice the type:

```
type ColPoint = { norm: Double ,
                  erase: Point ,
                  isWhite: Boolean ,
                  move: Int -> ColPoint ,
                  isOrigin : Boolean
                }
```

# The essence of inheritance

<b>A</b>	
m1	... <b>this.m2()</b> ...
m2	

wrapping

<b>A</b>	
m1	... <b>this.m2()</b> ...
m2	

<b>B</b>	
<b>A</b>	
m1	... <b>this.m2()</b> ...
m2	
m2	
m3	... <b>this.m2()</b> ...

wrapping

<b>B</b>	
<b>A</b>	
m1	... <b>this.m2()</b> ...
m2	
m2	
m3	... <b>this.m2()</b> ...

Generator wrapping

# Dynamic binding and loss of information

- The dynamic binding of `self/this` is performed by generator rewrapping
- While the semantics of `self` is updated, this is not true for its type:

```
type ColPoint = { norm: Double ,  
                  erase: Point ,  
                  isWhite: Boolean ,  
                  move: Int -> ColPoint ,  
                  isOrigin : Boolean  
                }
```

In spite that `erase` returns `this` and therefore a `ColPoint`

- The following is ill-typed

```
(newColPoint(0,0,"white")).erase.isWhite
```

## Rationale

The method `erase` returns a result *of the same type as the type of its receiver*

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects
- 4 Parametric Types**
- 5 Generics
- 6 Recursive Types
- 7 Bibliography

# Loss of information

Consider the function

$$\text{id} = \lambda x:T.x : T \rightarrow T$$

By the typing rule of application

$$\frac{\vdash \text{id} : T \rightarrow T \quad \vdash a : S < T}{\vdash \text{id}(a) : T}$$

Therefore

$$(\lambda x:\{a : \text{Int}\}.x)\{a = 1, b = 2\} : \{a : \text{Int}\}$$

## Rationale

The function `id` returns a result *of the same type as the type of its argument*

## Second order types

$$\text{id} : \forall X \leq T. X \rightarrow X$$

Two ways:

- 1 Implicit polymorphism
- 2 Explicit polymorphism

# Implicit polymorphism

## Type schema

No types in terms

$$\lambda x.x : \forall \alpha. \alpha \rightarrow \alpha$$

Therefore

$$(\lambda x.x)3 : \text{Int}$$

## Subtyping

$$\lambda x.((\lambda y.x)(x.l + 3)) : \forall \alpha \leq \{\ell : \text{Int}\}. \alpha \rightarrow \alpha$$

Therefore

$$\lambda x.((\lambda y.x)(x.l + 3))(\{\ell = 1, m = \text{true}\}) : \{\ell : \text{Int}, m : \text{Bool}\}$$

To deduce these types see constraint programming

## Example of deduction of constraints

$$\alpha = \beta \frac{\frac{x : \alpha, y : \gamma \vdash x : \alpha}{x : \alpha \vdash \lambda y. x : \gamma \rightarrow \beta} \quad \frac{x : \alpha \vdash 3 : \text{Int} \quad \frac{x : \alpha \vdash x : \alpha}{x : \alpha \vdash x.l : \varepsilon}}{x : \alpha \vdash x.l + 3 : \delta}}{x : \alpha \vdash ((\lambda y. x)(x.l + 3)) : \beta}}{\emptyset \vdash \lambda x. ((\lambda y. x)(x.l + 3)) : \alpha \rightarrow \beta}$$

$\alpha \leq \{\ell : \varepsilon\}$   
 $\delta = \text{Int}, \varepsilon \leq \text{Int}$   
 $\delta \leq \gamma$

### Resulting type:

$$\forall \varepsilon \leq \text{Int}. \forall \alpha \leq \{\ell : \varepsilon\}. \alpha \rightarrow \alpha$$

By the subtyping on record types, this can be simplified into:

$$\forall \alpha \leq \{\ell : \text{Int}\}. \alpha \rightarrow \alpha$$

# Explicit polymorphism

## Type abstractions.

$$\Lambda X. \lambda x: X. x : \forall X. X \rightarrow X$$

The programmer specifies the type to be substituted for  $X$

$$(\Lambda X. \lambda x: X. x)[\text{Int}](3) \longrightarrow (\lambda x: \text{Int}. x)(3)$$

And

$$(\Lambda X. \lambda x: X. x)[\text{Int}] : \text{Int} \rightarrow \text{Int}$$

## Subtyping

$$\lambda x. ((\lambda y. x)(x.l + 3))$$

$$\Lambda X \leq \{\ell : \text{Int}\}. \lambda x: X. ((\lambda y: \text{Int}. x)(x.l + 3)) : \forall X \leq \{\ell : \text{Int}\}. X \rightarrow X$$

Therefore

$$\Lambda X \leq \{\ell : \text{Int}\}. \lambda x: X. ((\lambda y: \text{Int}. x)(x.l + 3)) [\{\ell : \text{Int}, m : \text{Bool}\}](\{\ell = 1, m = \text{true}\})$$

has type  $\{\ell : \text{Int}, m : \text{Bool}\}$

## Syntax

<i>Types</i>	$T ::= T \rightarrow T$	function types
	$\quad   \forall X \leq T. T$	bounded quantification
	$\quad   X$	type variables
	$\quad   \text{Any}$	top type
<i>Terms</i>	$a, b ::= x \mid ab \mid \lambda x:T.a$	
	$\quad   \text{top}$	top value
	$\quad   a[T]$	type application
	$\quad   \Lambda X \leq T.a$	type abstraction

## Reduction

BETA

$$(\lambda x:T.a)b \longrightarrow a[b/x]$$

BETA $_{\forall}$

$$(\Lambda X \leq T.a)[S] \longrightarrow a[S/X]$$

Notice that in  $\beta_{\forall}$  no subtype checking is performed (similarly to  $\beta$  where no type checking is performed). Other notations for **Any** in the litterature:  $\top$ , **Top**

$$\begin{array}{c} \text{TOP} \\ C \vdash T \leq \text{Any} \end{array}$$

$$\begin{array}{c} \text{VAR} \\ C \vdash X \leq C(X) \end{array}$$

$$\begin{array}{c} \text{ARROW} \\ \frac{C \vdash T_1 \leq S_1 \quad C \vdash S_2 \leq T_2}{C \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \end{array}$$

$$\begin{array}{c} \text{FORALL} \\ \frac{C \vdash T_1 \leq S_1 \quad C, X \leq T_1 \vdash S_2 \leq T_2}{C \vdash \forall X \leq S_1. S_2 \leq \forall X \leq T_1. T_2} \end{array}$$

$$\begin{array}{c} \text{REFL} \\ C \vdash T \leq T \end{array}$$

$$\begin{array}{c} \text{TRANS} \\ \frac{C \vdash T_1 \leq T_2 \quad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3} \end{array}$$

$$\begin{array}{c} \text{TVAR} \\ C; \Gamma \vdash x : \Gamma(x) \end{array}$$

$$\begin{array}{c} \text{TOP} \\ C; \Gamma \vdash \text{top} : \text{Any} \end{array}$$

$$\begin{array}{c} \rightarrow\text{INTRO} \\ C; \Gamma, (x : S) \vdash a : T \\ \hline C; \Gamma \vdash \lambda x : S. a : S \rightarrow T \end{array}$$

$$\begin{array}{c} \forall\text{INTRO} \\ C, X \leq S; \Gamma \vdash a : T \\ \hline C; \Gamma \vdash \Lambda X \leq S. a : \forall X \leq S. T \end{array}$$

$$\begin{array}{c} \rightarrow\text{ELIM} \\ C; \Gamma \vdash a : S \rightarrow T \quad C; \Gamma \vdash b : S \\ \hline C; \Gamma \vdash ab : T \end{array}$$

$$\begin{array}{c} \forall\text{ELIM} \\ C; \Gamma \vdash a : \forall X \leq S. T \\ \hline C; \Gamma \vdash a[S] : T[S/X] \end{array}$$

$$\begin{array}{c} \text{SUBSUMPTION} \\ C; \Gamma \vdash a : S \quad C \vdash S \leq T \\ \hline C; \Gamma \vdash a : T \end{array}$$

- **Subtype checking is centralized in subsumption.**
- **Subsumption is used to make application types match.**
- **It uses the relation built on type variables by  $\Lambda$ -abstractions.**

# How to define the subtyping algorithm?

$$\begin{array}{c} \text{TOP} \\ C \vdash T \leq \text{Any} \end{array}$$

$$\begin{array}{c} \text{VAR} \\ C \vdash X \leq C(X) \end{array}$$

$$\begin{array}{c} \text{ARROW} \\ \frac{C \vdash T_1 \leq S_1 \quad C \vdash S_2 \leq T_2}{C \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \end{array}$$

$$\begin{array}{c} \text{FORALL} \\ \frac{C \vdash T_1 \leq S_1 \quad C, X \leq T_1 \vdash S_2 \leq T_2}{C \vdash \forall X \leq S_1. S_2 \leq \forall X \leq T_1. T_2} \end{array}$$

$$\begin{array}{c} \text{REFL} \\ C \vdash T \leq T \end{array}$$

$$\begin{array}{c} \text{TRANS} \\ \frac{C \vdash T_1 \leq T_2 \quad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3} \end{array}$$

Eliminating REFL and TRANS is not a solution. While REFL can be replaced by reflexivity just for variables  $C \vdash X \leq X$ , without TRANS there is no way to prove, say:

$$X \leq Y, Y \leq Z \vdash X \leq Z$$

and therefore to type

$$\forall Y \leq Z. \forall X \leq Y. \lambda x:Z \rightarrow T. \lambda y:X. xy$$

Notice that the transitivity rule is a *logical cut*: we must *eliminate* it.

# Transitivity elimination

Coerce expressions  $c ::= \text{Id}_X \mid X_T \mid \text{Any}_T \mid c \rightarrow c \mid \forall(X \leq c)c \mid cc$

$$\begin{array}{c} \text{TOP} \\ C \vdash \text{Any}_T : T \leq \text{Any} \end{array}$$

$$\begin{array}{c} \text{VAR} \\ C \vdash X_{C(X)} : X \leq C(X) \end{array}$$

$$\begin{array}{c} \text{ARROW} \\ \frac{C \vdash c_1 : T_1 \leq S_1 \quad C \vdash c_2 : S_2 \leq T_2}{C \vdash c_1 \rightarrow c_2 : S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \end{array}$$

$$\begin{array}{c} \text{FORALL} \\ \frac{C \vdash c_1 : T_1 \leq S_1 \quad C, X \leq T_1 \vdash c_c : S_2 \leq T_2}{C \vdash \forall(X \leq c_1)c_2 : \forall X \leq S_1. S_2 \leq \forall X \leq T_1. T_2} \end{array}$$

$$\begin{array}{c} \text{REFL} \\ C \vdash \text{Id}_X : X \leq X \end{array}$$

$$\begin{array}{c} \text{TRANS} \\ \frac{C \vdash c_2 : T_1 \leq T_2 \quad C \vdash c_1 : T_2 \leq T_3}{C \vdash c_1 c_2 : T_1 \leq T_3} \end{array}$$

## Theorem

*There is a 1-1 correspondence between well-typed coerce expressions and subtyping derivations.*

$$\begin{array}{lll} \text{(Asc)} & (cd)e & \rightsquigarrow c(de) \\ (\rightarrow') & (c \rightarrow d)(c' \rightarrow d') & \rightsquigarrow (c'c) \rightarrow (dd') \\ (\rightarrow'') & (c \rightarrow d)((c' \rightarrow d')e) & \rightsquigarrow ((c'c) \rightarrow (dd'))e \\ (\forall') & (\forall(X \leq c)d)((\forall X \leq c')d') & \rightsquigarrow \forall(X \leq c'c)(dd'[cX_S/X_T]) \\ (\forall'') & ((\forall(X \leq c)d)((\forall X \leq c')d')e) & \rightsquigarrow (\forall(X \leq c'c)(dd'[cX_S/X_T]))e \end{array}$$

- Normal forms are subterms of  $(c \rightarrow d)e_1 \dots e_n$  or of  $(\forall(X \leq c)d)e_1 \dots e_n$  where  $c, d$  are in normal form and  $e_i$ 's are either  $X_T$  or  $\text{Any}_T$ .
- These normal forms corresponds to derivation in which every left premise of a [TRANS] rule is a leaf.
- Thus the rewriting system pushes transitivity towards left and up to the leaves.

## Example

Let us develop the following example at the blackboard:

$$(c \rightarrow d)((c' \rightarrow d')e) \rightsquigarrow ((c'c) \rightarrow (dd'))e$$

### Theorem (Soundness)

*If  $c \rightsquigarrow^* d$  and  $C \vdash c : \Delta$ , then  $C \vdash d : \Delta$*

### Theorem (Weak normalization)

*Every innermost strategy for  $\rightsquigarrow$  terminates.*

# Cleaning the proofs up

## Housekeeping reduction:

Let  $c : S \leq T$

$$\begin{array}{llll} (\text{ID}_l) & \text{Id}_X c & \rightsquigarrow & c \\ (\text{ID}_r) & c \text{Id}_X & \rightsquigarrow & c \\ (\text{TOP}) & \text{Any}_T c & \rightsquigarrow & \text{Any}_S \\ (\text{VARTOP}) & X_{\text{Any}} & \rightsquigarrow & \text{Any}_X \end{array}$$

Consider the composition of the two rewriting systems, then:

## Theorem (Normal forms)

*Every well-typed coerce expression in normal form has the form  $c_0 c_1 \dots c_n$  (with  $n \geq 0$ ), where  $c_0$  can be any coerce expression different from a composition (of coerce expressions) whose subformulae are in normal form, and  $c_1 \dots c_n$  are variables.*

## Lemma

*For every provable subtyping judgment, there exists only one coerce expression in normal form proving it.*

## Theorem (Coherence)

*Let  $\Pi_1$  and  $\Pi_2$  be two proofs of the same subtyping judgement  $C \vdash \Delta$ . If  $c_1$  and  $c_2$  are the corresponding coerce expressions, then they are equal modulo the rewriting system*

## Shape of NFs and subtyping algorithm

The normal forms of the “Normal forms” theorem correspond to derivations in which every application of a [TRANS] rule has as left premise an application of the rule [TAUT].

**This can be used to define the following subtyping algorithm.**

# Subtyping algorithm

$$\text{TOP} \\ C \vdash T \leq \text{Any}$$

$$\text{ARROW} \\ \frac{C \vdash T_1 \leq S_1 \quad C \vdash S_2 \leq T_2}{C \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

$$\text{FORALL} \\ \frac{C \vdash T_1 \leq S_1 \quad C, X \leq T_1 \vdash S_2 \leq T_2}{C \vdash \forall X \leq S_1. S_2 \leq \forall X \leq T_1. T_2}$$

$$\text{ALGREFL} \\ C \vdash X \leq X$$

$$\text{ALGTRANS} \\ \frac{C \vdash C(X) \leq T}{C \vdash X \leq T}$$

**The rules above describe a sound and complete algorithm for the subtyping relation**  
**(Just choose a priority order for TOP, ALGREFL, and ALGTRANS.)**

# Typing algorithm

$$\begin{array}{c} \text{TVAR} \\ C; \Gamma \vdash x : \Gamma(x) \end{array}$$

$$\begin{array}{c} \text{TOP} \\ C; \Gamma \vdash \text{top} : \text{Any} \end{array}$$

$$\begin{array}{c} \rightarrow\text{INTRO} \\ C; \Gamma, (x : S) \vdash a : T \\ \hline C; \Gamma \vdash \lambda x : S. a : S \rightarrow T \end{array}$$

$$\begin{array}{c} \forall\text{INTRO} \\ C, X \leq S; \Gamma \vdash a : T \\ \hline C; \Gamma \vdash \Lambda X \leq S. a : \forall X \leq S. T \end{array}$$

$$\begin{array}{c} \rightarrow\text{ELIM}_{\leq} \\ C; \Gamma \vdash a : V \quad C; \Gamma \vdash b : U \quad C \vdash U \leq S \quad \mathcal{B}_C(V) = S \rightarrow T \\ \hline C; \Gamma \vdash ab : T \end{array}$$

$$\begin{array}{c} \forall\text{ELIM} \\ C; \Gamma \vdash a : U \quad C \vdash S' \leq S \quad \mathcal{B}_C(U) = \forall X \leq S. T \\ \hline C; \Gamma \vdash a[S'] : T[S'/X] \end{array}$$

where

$$\mathcal{B}_C(T) = \begin{cases} \mathcal{B}_C(C(X)) & \text{if } T = X \\ T & \text{otherwise} \end{cases}$$

# Undecidability

Typing and subtyping algorithms are sound and complete

**Sound and complete does not mean decidable!!**

Let  $\neg T$  and  $\forall X. T$  denote  $T \rightarrow \text{Any}$  and  $\forall X \leq \text{Any}. T$ , respectively.

$$X_0 \leq \forall Y. \neg(\forall Z \leq Y. \neg Y) \quad \vdash \quad X_0 \leq \forall X_1 \leq X_0. \neg X_0$$

By applying ALGTRANS

$$X_0 \leq \forall Y. \neg(\forall Z \leq Y. \neg Y) \quad \vdash \quad \forall X_1. \neg(\forall X_2 \leq X_1. \neg X_1) \leq \forall X_1 \leq X_0. \neg X_0$$

By applying FORALL

$$X_0 \leq \forall Y. \neg(\forall Z \leq Y. \neg Y), X_1 \leq X_0 \quad \vdash \quad \neg(\forall X_2 \leq X_1. \neg X_1) \leq \neg X_0$$

By the contravariance of  $\rightarrow$

$$X_0 \leq \forall Y. \neg(\forall Z \leq Y. \neg Y), X_1 \leq X_0 \quad \vdash \quad X_0 \leq \forall X_2 \leq X_1. \neg X_1$$

This judgment is the same as the one we started from, with a larger environment.

**Just semi-decidability holds**

## Back to the loss of information

The problem of loss of information is that wrappers are parametric in the *value* of self, but not on the *type* of self:

```
 $\mathcal{G}_{\text{Point}} = \lambda \text{mkobj} : \{x : \text{ref Int}, y : \text{ref Int}\} \rightarrow \text{Point}.$   
   $\lambda \text{state} : \{x : \text{ref Int}, y : \text{ref Int}\}.$   
    {norm =  $\sqrt{! \text{state}.x^2 + ! \text{state}.y^2}$ ,  
     erase = state.x:=0; state.y:=0; mkobj(state),  
     move =  $\lambda dx : \text{Int} . \text{mkobj}(\{x=! \text{state}.x+dx, y=! \text{state}.y\})$   
     isOrigin = (mkobj(state).norm==0)}
```

```
 $\mathcal{G}_{\text{ColPoint}} =$   
   $\lambda \text{mkobj} : \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\} \rightarrow \text{ColPoint}.$   
   $\lambda \text{state} : \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\}.$   
  let super:Point =  $\mathcal{G}_{\text{Point}}(\lambda s . \text{mkobj}(\text{state} \oplus s)) \text{state}$   
  in super  $\oplus$   
    {norm = if mkobj(state).isWhite then 0 else ... ,  
     move =  $\lambda dx : \text{Int} . \text{mkobj}(\{x=! \text{state}.x+dx, y=! \text{state}.y, c="red"\})$   
     isWhite = state.c == "white" }
```

# Make generators parametric also in the type of self

$\mathcal{G}_{\text{Point}} = \Lambda \text{Mytype} \leq \text{Point}.$

$\lambda \text{mkobj}: \{x : \text{ref Int}, y : \text{ref Int}\} \rightarrow \text{Mytype}.$

$\lambda \text{state}: \{x : \text{ref Int}, y : \text{ref Int}\}.$

$\{\text{norm} = \sqrt{! \text{state}.x^2 + ! \text{state}.y^2},$

$\text{erase} = \text{state}.x := 0; \text{state}.y := 0; \text{mkobj}(\text{state}),$

$\text{move} = \lambda dx : \text{Int}. \text{mkobj}(\{x = ! \text{state}.x + dx, y = ! \text{state}.y\})$

$\text{isOrigin} = (\text{mkobj}(\text{state}).\text{norm} == 0)\}$

## Object creation = generator wrapping:

$\text{new Point} = \lambda(a, b). \mathbf{Y}(\mathcal{G}_{\text{Point}}[\text{Point}])\{x = \text{ref}^{\text{Int}} a, y = \text{ref}^{\text{Int}} b\}$

When inheriting a method, update the type of self:

$\mathcal{G}_{\text{ColPoint}} =$

$\wedge \text{Mytype} \leq \text{ColPoint}.$

$\lambda \text{mkobj}: \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\} \rightarrow \text{Mytype}.$

$\lambda \text{state}: \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\}.$

let  $\text{super}:\{\text{norm}: \text{Double}, \text{erase}: \text{Mytype}, \text{move}: \text{Int} \rightarrow \text{Mytype}, \dots\}$

$= \mathcal{G}_{\text{Point}}[\text{Mytype}] (\lambda s. \text{mkobj}(\text{state} \oplus s)) \text{state}$

in  $\text{super} \oplus$

$\{\text{norm} = \text{if } \text{mkobj}(\text{state}).\text{isWhite} \text{ then } 0 \text{ else } \dots,$

$\text{move} = \lambda dx: \text{Int}. \text{mkobj}(\{x = !\text{state}.x + dx, y = !\text{state}.y, c = \text{"red"}\})$

$\text{isWhite} = \text{state}.c == \text{"white"} \}$

**As usual, object creation = generator wrapping:**

$\text{new ColPoint} =$

$\lambda(a, b, c). \mathbf{Y}(\mathcal{G}_{\text{ColPoint}}[\text{ColPoint}])\{x = \text{ref}^{\text{Int}} a, y = \text{ref}^{\text{Int}} b, \text{color} = c\}$

## Application to Scala

No explicit parameter, but a keyword `this.type` to denote the type of self/this

```
class Point(a: Int, b: Int) {
  var x: Int = a
  var y: Int = b
  def norm(): Double = sqrt(pow(x,2) + pow(y,2))
  def erase(): this.type = { x = 0; y = 0; return this}
  def move(dx: Int): Point = new Point(x+dx,y)
  def isOrigin(): Boolean = (this.norm == 0)
}

class ColPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c
  def isWhite(): Boolean = c == "white"
  override def norm(): Double = {
    if(this.isWhite) return 0 else return sqrt(pow(x,2)+pow(y,2))}
  override def move(dx: Int): ColPoint=new ColPoint(x+dx,y,"red")
}
```

In particular if we load the two previous definitions in Scala we have:

```
scala> (new ColorPoint(0, 0, "white")).erase  
res0: ColorPoint = ColorPoint@1b3e02ed
```

```
scala> (new ColorPoint(0, 0, "white")).erase.isWhite  
res1: Boolean = true
```

## $F_{\leq}$ in practice

$F_{\leq}$  fostered the introduction of *generics* in OOL (first in Java-like languages), in particular in its *F-bounded* variant. It allows us to understand what is “under the hood” of recent OOLs.

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects
- 4 Parametric Types
- 5 Generics**
- 6 Recursive Types
- 7 Bibliography

## Why in C# and not in Java?

### Direct support in the CLR and IL (intermediate language)

The CLR implementation pushes support for generics into almost all feature areas, including serialization, remoting, reflection, reflection emit, profiling, debugging, and pre-compilation.

### Java Generics based on GJ

Rather than extend the JVM with support for generics, the feature is "compiled away" by the Java compiler

### Consequences:

- generic types can be instantiated only with reference types (e.g. string or object) and not with primitive types
- type information is not preserved at runtime, so objects with distinct source types such as `List<string>` and `List<object>` cannot be distinguished by run-time
- Clearer syntax

# Generics Problem Statement

```
public class Stack
{
    object [] m_Items;
    public void Push(object item)
    {...}
    public object Pop()
    {...}
}
```

- runtime cost (boxing/unboxing, garbage collection)
- type safety

```
Stack stack = new Stack();
stack.Push(1);
stack.Push(2);
int number = (int)stack.Pop();

Stack stack = new Stack();
stack.Push(1);
string number = (string)stack.Pop();           // exception thrown
```

# Heterogenous translation

You can overcome these two problems by writing type-specific stacks. For integers:

```
public class IntStack
{
    int[] m_Items;
    public void Push(int item){...}
    public int Pop(){...}
}
IntStack stack = new IntStack();
stack.Push(1);
int number = stack.Pop();
```

For strings:

```
public class StringStack
{
    string[] m_Items;
    public void Push(string item){...}
    public string Pop(){...}
}
StringStack stack = new StringStack();
stack.Push("1");
string number = stack.Pop();
```

## Problem

Writing type-specific data structures is a tedious, repetitive, and error-prone task.

## Solution

### Generics

```
public class Stack<T>
{
    T[] m_Items;
    public void Push(T item)
    {...}
    public T Pop()
    {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

You have to instruct the compiler which type to use instead of the generic type parameter T, both when declaring the variable and when instantiating it:

```
Stack<int> stack = new Stack<int>();
```

```
public class Stack<T>{
    readonly int m_Size;
    int m_StackPointer = 0;
    T[] m_Items;
    public Stack():this(100){
    }
    public Stack(int size){
        m_Size = size;
        m_Items = new T[m_Size];
    }
    public void Push(T item){
        if(m_StackPointer >= m_Size)
            throw new StackOverflowException();
        m_Items[m_StackPointer] = item;
        m_StackPointer++;
    }
    public T Pop(){
        m_StackPointer--;
        if(m_StackPointer >= 0) {
            return m_Items[m_StackPointer]; }
        else {
            m_StackPointer = 0;
            throw new InvalidOperationException("Cannot pop an empty stack");
        }
    }
}
```

# Generator

This can be easily obtained by adding a type parameter for stack elements

$G_{Stack} =$

AS.

$\Lambda Mytype \leq Stack[S/X].$

$\lambda mkobj: \dots$

$\vdots$

where  $Stack = \forall X. \{ Push: X \rightarrow (), Pop: () \rightarrow X \}$

The creation of a new object requires a type parameter to be passed to the wrapper

## Note

The type parameter is uncostrained (just System F). But bounded quantification is sometime necessary.

## Exercise

Fill the ellipsis in the definition and write the “new” function

# Multiple Generic Type Parameters

```
class Node<K,T> {
    public K Key;
    public T Item;
    public Node<K,T> NextNode;
    public Node() {
        Key      = default(K);           // the "default" value of type K
        Item     = default(T);           // the "default" value of type K
        NextNode = null;
    }
    public Node(K key,T item,Node<K,T> nextNode) {
        Key      = key;
        Item     = item;
        NextNode = nextNode;
    }
}

public class LinkedList<K,T> {
    Node<K,T> m_Head;
    public LinkedList() {
        m_Head = new Node<K,T>();
    }
    public void AddHead(K key,T item){
        Node<K,T> newNode = new Node<K,T>(key,item,m_Head.NextNode);
        m_Head.NextNode = newNode;
    }
}
```

# Generic Type Constraints

Suppose you would like to add searching by key to the linked list class

```
public class LinkedList<K,T>
public T Find(K key) {
    Node<K,T> current = m_Head;
    while(current.NextNode != null) {
        if(current.Key == key)           //Will not compile
            break;
        else
            current = current.NextNode;
    }
    return current.Item;
}
```

The compiler will refuse to compile this line

```
if(current.Key == key)
```

because the compiler does not know whether K (or the actual type supplied by the client) supports the == operator.

We must ensure that K implements the following interface

```
public interface IComparable {
    int CompareTo(Object other);
    bool Equals(Object other);
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : IComparable {
    public T Find(K key) {
        Node<K,T> current = m_Head;
        while(current.NextNode != null) {
            if(current.Key.CompareTo(key) == 0)
                break;
            else
                current = current.NextNode;
        }
        return current.Item;
    }
    //Rest of the implementation
}
```

This corresponds to add constraints in the generator:

$$\begin{aligned} G_{\text{LinkedList}} &= \Lambda T. \Lambda K \leq \text{IComparable}. \\ &\quad \Lambda \text{Mytype} \leq \text{LinkedList} [K/X, T/Y]. \\ &\quad \lambda \text{mkobj}: \dots \end{aligned}$$

## The previous solution has two problems

- 1 key is boxed/unboxed when it is a value (i.e. not an object)
- 2 The static information that key is of type K is not used (CompareTo requires a parameter just of type Object).

```
public interface IComparable {
    int CompareTo(Object other);
    bool Equals(Object other);
}

public class LinkedList<K,T> where K : IComparable {
    public T Find(K key) {
        Node<K,T> current = m_Head;
        while(current.NextNode != null) {
            if(current.Key.CompareTo(key) == 0)
                break;
            else
                current = current.NextNode;
        }
        return current.Item;
    }
    //Rest of the implementation
}
```

# F-bounded polymorphism

In order to enhance type-safety (in particular, enforce the argument of `K.CompareTo` to have type `K` rather than `Object`) and avoid boxing/unboxing when the key is a value we can use a generic version of `Comparable`.

```
public interface Comparable<T> {  
    int CompareTo(T other);  
    bool Equals(T other);  
}
```

This can be done by specifying a constraint:

```
public class LinkedList<K,T> where K : Comparable<K> {  
    public T Find(K key) {  
        Node<K,T> current = m_Head;  
        while(current.NextNode != null) {  
            if(current.Key.CompareTo(key) == 0)  
                break;  
            else  
                current = current.NextNode;  
        }  
        return current.Item;  
    }  
    //Rest of the implementation  
}
```

# F-bounded polymorphism

**Formally, it requires a new form of quantification:**

Add to terms  $\lambda t \leq F(t).a$

Add to type  $\forall t \leq F(t).T$

## Intuition

The term/type above accepts as argument any type  $S$  such that  $S \leq F(S)$

It permits *inheritance* (ie code reuse in the definition of the generators) also without subtyping.

**Example:** add an equal method to our (Col)Points (simplified)

$$\begin{aligned} P[t] &= \{\text{equal}:t \rightarrow \text{Bool}\} \\ CP[t] &= \{\text{equal}:t \rightarrow \text{Bool}; \text{isWhite}:\text{Bool}\} \end{aligned}$$

Notice that  $\text{ColPoint} = \mu t. CP[t] \not\leq \mu t. P[t] = \text{Point}$  however it is possible to define ColPoint so that it “inherits” equal from Point.

$$\begin{aligned}
\mathcal{G}_{\text{Point}} = & \\
& \wedge \text{Mytype} \leq P[\text{Mytype}]. \\
& \lambda \text{mkobj}: \{x : \text{ref Int}, y : \text{ref Int}\} \rightarrow \text{Mytype}. \\
& \lambda \text{state}: \{x : \text{ref Int}, y : \text{ref Int}\}. \\
& \{ \\
& \quad \vdots \\
& \quad \text{equal} = \dots \\
& \quad \vdots \}
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}_{\text{ColPoint}} = & \\
& \wedge \text{Mytype} \leq CP[\text{Mytype}]. \\
& \lambda \text{mkobj}: \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\} \rightarrow \text{Mytype}. \\
& \lambda \text{state}: \{x : \text{ref Int}, y : \text{ref Int}, c : \text{String}\}. \\
& \text{let } \text{super}: P[\text{Mytype}] = \mathcal{G}_{\text{Point}}[\text{Mytype}] (\lambda s. \text{mkobj}(\text{state} \oplus s)) \text{state} \\
& \text{in } \text{super} \oplus \{ \text{isWhite} = \text{state.c} == \text{"white"} \}
\end{aligned}$$

# Generic methods

You can define method-specific (possibly constrained) generic type parameters even if the containing class does not use generics at all:

```
public class MyClass
{
    public void MyMethod<T>(T t) where T : IComparable<T>
    {
        ...
    }
}
```

When calling a method that defines generic type parameters, you can provide the type to use at the call site:

```
MyClass obj = new MyClass();
obj.MyMethod<int>(3)
```

## Generator

Formally this corresponds to use bounded quantification inside the method definitions of the generator.

Generics are *invariant*:

```
List<string> ls = new List<string>();  
ls.Add("test");  
List<object> lo = ls;    // Can't do this in C#  
object o1 = lo[0];      // ok - converting string to object  
lo[0] = new object();   // ERROR - can't convert object to string
```

This is the right decision as the example above shows.

Thus

$S$  is a subtype of  $T$  *does not imply*  $\text{Class}\langle S \rangle$  is a subtype of  $\text{Class}\langle T \rangle$ .

If this (covariance) were allowed, the last line would have to result in an exception (eg. `InvalidCastException`).

## Excursus: Beware of self-proclaimed type-safety

Since  $S$  is a subtype of  $T$  *implies*  $S[]$  is subtype of  $T[]$ . (*covariance*)

Do not we have the same problem with arrays? **Yes**

From Jim Miller CLI book

*The decision to support covariant arrays was primarily to allow Java to run on the VES (Virtual Execution System). The covariant design is not thought to be the best design in general, but it was chosen in the interest of broad reach*

Regretful (and regretted) decision:

```
class Test {
    static void Fill(object[] array, int index, int count, object val) {
        for (int i = index; i < index + count; i++) array[i] = val;
    }
    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0); //→System.ArrayTypeMismatchException
    }
}
```

# Variant annotations

## Add variants (C# 4.0)

```
// Covariant parameters can be used as result types
interface IEnumerator<out T> {
    T Current { get; }
    bool MoveNext();
}

// Covariant parameters can be used in covariant result types
interface IEnumerable<out T> {
    IEnumerator<T> GetEnumerator();
}

// Contravariant parameters can be used as argument types
interface IComparer<in T> {
    bool Compare(T x, T y);
}
```

This means we can write code like the following:

```
IEnumerable<string> stringCollection = ...; //smaller type
IEnumerable<object> objectCollection = stringCollection; // larger type
foreach( object o in objectCollection ) { ... }

IComparer<object> objectComparer = ...; //smaller type
IComparer<string> stringComparer = objectComparer; // larger type
bool b = stringComparer.Compare( "x", "y" );
```

## Exercise

Think about how to modify the definition of  $F_{\leq}$  to account for variant annotations. Add also reference types.

## Features becoming standard in modern OOLs ...

In Scala we have generics classes and methods with annotations and bounds

```
case class ListNode[+T](h: T, t: ListNode[T]) {
  def head: T = h
  def tail: ListNode[T] = t
  def prepend[U >: T](elem: U): ListNode[U] =
    ListNode(elem, this)
}
```

and F-bounded polymorphism as well:

```
class GenCell[T](init: T) {
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}

trait Ordered[T] {
  def < (x: T): boolean
}

def updateMax[T <: Ordered[T]](c: GenCell[T], x: T) =
  if (c.get < x) c.set(x)
```

... but also in FP.

All these characteristics are present in different flavours in OCaml

Generics are close to parametrized classes:

```
# exception Empty;;
```

```
class ['a] stack =
```

```
  object
```

```
    val mutable p : 'a list = []
```

```
    method push x = p <- x :: p
```

```
    method pop =
```

```
      match p with
```

```
      | [] -> raise Empty
```

```
      | x::t -> p <- t; x
```

```
  end;;
```

```
class ['a] stack :
```

```
object val mutable p : 'a list method pop : 'a method push : 'a -> unit end
```

```
# new stack # push 3;;
```

```
- : unit = ()
```

```
# let x = new stack;;
```

```
val x : '_a stack = <obj>
```

```
# x # push 3;;
```

```
- : unit = ()
```

```
# x;;
```

```
- : int stack = <obj>
```

## Constraints can be deduced by the type-checker

```
#class ['a] circle (c : 'a) =  
  object  
    val mutable center = c  
    method center = center  
    method set_center c = center <- c  
    method move = (center#move : int -> unit)  
  end;;  
class ['a] circle :  
  'a ->  
  object  
    constraint 'a = < move : int -> unit; .. >  
    val mutable center : 'a  
    method center : 'a  
    method move : int -> unit  
    method set_center : 'a -> unit  
  end
```

## Constraints can be imposed by the programmer

```
#class point x_init =
  object
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end

#class ['a] circle (c : 'a) =
  object
    constraint 'a = #point    (* = < get_x : int; move : int->unit; .. > *)
    val mutable center = c
    method center = center
    method set_center c = center <- c
    method move = center#move
  end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = #point
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

## Explicit instantiation is done just for inheritance

```
#class colored_point x (c : string) =
  object
    inherit point x
    val c = c
    method color = c
  end;;
class colored_point :
  int ->
  string ->
  object
  :
end

#class colored_circle c =
  object
    inherit [colored_point] circle c
    method color = center#color
  end;;
class colored_circle :
  colored_point ->
  object
    val mutable center : colored_point
    method center : colored_point
    method color : string
    method move : int -> unit
    method set_center : colored_point -> unit
  end
```

## Variance constraints

- Variance constraint are meaningful only with subtyping (i.e. objects, polymorphic variants, ...).
- They can be used in OCaml (not well documented): useful on abstract types to describe the expected behaviour of the type with respect to subtyping.
- For instance, an immutable container type (like lists) will have a covariant type:

```
type (+'a) container
```

meaning that if  $s$  is a subtype of  $t$  then  $s$  container is a subtype of  $t$  container. On the other hand an acceptor will have a contravariant type:

```
type (-'a) acceptor
```

meaning that if  $s$  is a subtype of  $t$  then  $t$  acceptor is a subtype  $s$  acceptor.

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects
- 4 Parametric Types
- 5 Generics
- 6 Recursive Types**
- 7 Bibliography

# Iso-recursive and Equi-recursive types

Lists are a classic example of recursive types:

$$X \approx (\text{Int} \times X) \vee \text{Nil}$$

also written as  $\mu X.((\text{Int} \times X) \vee \text{Nil})$

Two different approaches according to whether  $\approx$  is interpreted as an isomorphism or an equality:

**Iso-recursive types:**  $\mu X.((\text{Int} \times X) \vee \text{Nil})$  is considered *isomorphic* to its one-step unfolding  $(\text{Int} \times \mu X.((\text{Int} \times X) \vee \text{Nil})) \vee \text{Nil}$ . Terms include a pair of built-in coercion functions for each recursive type  $\mu X.T$ :

$$\text{unfold} : \mu X.T \rightarrow T[\mu X.T/X] \quad \text{fold} : T[\mu X.T/X] \rightarrow \mu X.T$$

**Equi-recursive types:**  $\mu X.((\text{Int} \times X) \vee \text{Nil})$  is considered *equal* to its one-step unfolding  $(\text{Int} \times \mu X.((\text{Int} \times X) \vee \text{Nil})) \vee \text{Nil}$ . The two types are completely interchangeable. No support needed from terms.

Subtyping for recursive types generalizes the equi-recursive approach.

The  $\approx$  relation corresponds to subtyping in both directions:

$$\mu X.T \leq T[\mu X.T/X] \quad T[\mu X.T/X] \leq \mu X.T$$

# Recursive types are weird

- To add recursive types you do not need to add any new term
- You don't even need to have recursion on terms:

$$\mu X.((\text{Int} \times X) \vee \text{Nil})$$

interpret the type above as the *finite* lists of integers.

Then  $\mu X.(\text{Int} \times X)$  is the empty type.

- Actually if you have recursive terms and allow infinite values you can easily jeopardize decidability of the subtyping relation (which resorts to checking type emptiness)
- This contrasts with their intuition which looks simple: we always uniformly applied a rule such as:

$$\frac{A, X \leq Y \vdash S \leq T}{A \vdash \mu X.S \leq \mu Y.T}$$

# Subtyping recursive types

## Syntax

<i>Types</i>	$T$	$::=$	Any	top type
			$T \rightarrow T$	function types
			$T \times T$	product types
			$X$	type variables
			$\mu X. T$	recursive types

where  $T$  is *contractive*, that is (two equivalent definitions):

- 1  $T$  is contractive iff for every subexpression  $\mu X. \mu X_1 \dots \mu X_n. S$  it holds  $S \neq X$ .
- 2  $T$  is contractive iff every type variable  $X$  occurring in it is separated from its binder by a  $\rightarrow$  or a  $\times$ .

# Subtyping recursive types

The subtyping relation is defined *COINDUCTIVELY* by the rules

$$\begin{array}{c} \text{TOP} \\ \frac{}{T \leq \text{Any}} \end{array} \quad \begin{array}{c} \text{PROD} \\ \frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \end{array} \quad \begin{array}{c} \text{ARROW} \\ \frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \end{array}$$
$$\begin{array}{c} \text{UNFOLD LEFT} \\ \frac{S[\mu X.S/X] \leq T}{\mu X.S \leq T} \end{array} \quad \begin{array}{c} \text{UNFOLD RIGHT} \\ \frac{S \leq T[\mu X.T/X]}{S \leq \mu X.T} \end{array}$$

## Coinductive definition

- 1 Why coinduction?
- 2 Why no reflexivity/transitivity rules?
- 3 Why no rule to compare two  $\mu$ -types?

## Short answers (more detailed answers to come):

- 1 Because we compare infinite expansions
- 2 Because it would be unsound
- 3 Useless since obtained by coinduction and unfold

## Example of coinductive derivation

$$\begin{array}{l} \text{ARROW} \frac{\text{Even} \leq \text{Int} \quad \mu X.\text{Int} \rightarrow X \leq \mu Y.\text{Even} \rightarrow Y}{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y.\text{Even} \rightarrow Y)} \\ \text{UNFOLD RIGHT} \frac{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y.\text{Even} \rightarrow Y)}{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \mu Y.\text{Even} \rightarrow Y} \\ \text{UNFOLD LEFT} \frac{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \mu Y.\text{Even} \rightarrow Y}{\mu X.\text{Int} \rightarrow X \leq \mu Y.\text{Even} \rightarrow Y} \end{array}$$

**Notice the use of coinduction**

# Amadio and Cardelli's subtyping algorithm

Let  $A \subset \text{Types} \times \text{Types}$

$$\frac{}{A \vdash S \leq T} (S, T) \in A$$

$$\frac{}{A \vdash S \leq \text{Any}} (S, \text{Any}) \notin A$$

$$\frac{A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \times S_2 \leq T_1 \times T_2} A' = AU(S_1 \times S_2, T_1 \times T_2); A \neq A'$$

$$\frac{A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} A' = AU(S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A'$$

$$\frac{A' \vdash S[\mu X.S/X] \leq T}{A \vdash \mu X.S \leq T} A' = AU(\mu X.S, T); A \neq A'; T \neq \text{Any}$$

$$\frac{A' \vdash S \leq T[\mu X.T/X]}{A \vdash S \leq \mu X.T} A' = AU(S, \mu X.T); A \neq A'; S \neq \mu Y.U$$

## Theorem (Soundness and Completeness)

*Let  $S$  and  $T$  be closed types.  $S \leq T$  belongs the relation coinductively defined by the rules in slide 98 if and only if  $\emptyset \vdash S \leq T$  is provable*

The rest of the lecture aims to prove the above theorem.

Notice that the algorithm above is exponential. We will show how to define an  $O(n^2)$  algorithm to decide  $S \leq T$ , where  $n$  is the total number of different subexpressions of  $S \leq T$ .

# Induction and coinduction

## Intuition

Given a deduction system, it characterizes two possible distinct sets (of provable judgements) according to whether an inductive or a coinductive approach is used.

Given a deduction system  $\mathcal{F}$  and a universe,  $\mathcal{U}$  a set  $X \in \mathcal{P}(\mathcal{U})$  is:

**$\mathcal{F}$ -closed** if it contains all the elements that can be deduced by  $\mathcal{F}$  with hypothesis in  $X$ .

**$\mathcal{F}$ -consistent** if every element of  $X$  can be deduced by  $\mathcal{F}$  from other elements in  $X$ .

## Induction and coinduction

A deduction system

- *inductively* defines the least  $\mathcal{F}$ -closed set
- *coinductively* defines the greatest  $\mathcal{F}$ -consistent set

# Induction and coinduction

**induction:** start from  $\emptyset$ , add all the consequences of the deduction system, and iterate.

**coinduction:** start from  $\mathcal{U}$ , remove all elements that are not consequence of other elements, and iterate.

## Observation

In all the (algorithmic, ie without refl and trans) subtyping system met so far, the two coincide. This is not true in general, due to the presence of *self-justifying sets*, that is sets in which the deductions do not start just by axioms.

## Example:

$$\mathcal{U} = \{a, b, c, d, e, f, g\}$$

$$\begin{array}{cccccc} a & b & c & & d & f \\ \frac{}{b} & \frac{}{c} & \frac{}{a} & \frac{}{d} & \frac{}{e} & \frac{}{g} \end{array}$$

Inductively:

$$\{d, e\}$$

Coinductively:

$$\{a, b, c, d, e\}$$

Self-justifying set:

$$\{a, b, c\}$$

- 1 Let  $\mathcal{U} = \mathbb{Z}$  and take as deduction system all the instances of the rule

$$\frac{n}{n+1}$$

for  $n \in \mathbb{Z}$ . Which are the sets inductively and coinductively defined by it?

- 2 Same question but with  $\mathcal{U} = \mathbb{N}$ .
- 3 Same question but with  $\mathcal{U} = \mathbb{N}^2$  and as deduction system all the rules instance of

$$\frac{(m, n) \quad (n, o)}{(m, o)}$$

for  $m, n, o \in \mathbb{N}$

# Why Coinduction for Recursive types?

We want to use  $S = \mu X. \text{Int} \rightarrow X$  where  $T = \mu Y. \text{Even} \rightarrow Y$  is expected.

Use the substitutability interpretation.

Let  $e : T$  then  $e$ :

- 1 waits for an **Even** number,
- 2 fed by an **Even** number returns a function that behaves similarly: (1) wait for an **Even** ...

Now consider  $f : S$ , then  $f$ :

- 1 waits for an **Int** number,
- 2 fed by an **Int** (or a **Even**) number returns a function that behaves similarly: (1) wait for ...

$S$  and  $T$  are in subtyping relation because their infinite expansions are in subtyping relation.

$$S \leq T \implies \text{Int} \rightarrow S \leq \text{Even} \rightarrow T \implies S \leq T \wedge \text{Even} \leq \text{Int}$$

This is exactly the proof we saw at the beginning:

$$\begin{array}{l}
 \text{ARROW} \frac{\text{Even} \leq \text{Int} \quad \overbrace{\mu X.\text{Int} \rightarrow X}^S \leq \overbrace{\mu Y.\text{Even} \rightarrow Y}^T}{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y.\text{Even} \rightarrow Y)} \\
 \text{UNFOLD RIGHT} \frac{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y.\text{Even} \rightarrow Y)}{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \mu Y.\text{Even} \rightarrow Y} \\
 \text{UNFOLD LEFT} \frac{\text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \mu Y.\text{Even} \rightarrow Y}{\underbrace{\mu X.\text{Int} \rightarrow X}_S \leq \underbrace{\mu Y.\text{Even} \rightarrow Y}_T}
 \end{array}$$

## Coinduction

$S \leq T$  is not an axiom but  $\{S \leq T, \text{Even} \leq \text{Int}\}$  is a *self-justifying set*.

## Observation:

- 1 The deduction above shows why a specific rule for  $\mu$  is useless (apply consecutively the two unfold rules).
- 2 If we added reflexivity and/or transitivity rules, then  $\mathcal{U}$  would be  $\mathcal{F}$ -consistent (cf. the third exercise few slides before).

## Definition (Deduction system)

A deduction system on a universe  $\mathcal{U}$  is univocally characterized by a (total) **monotone generating function**  $\mathcal{F} : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ .

Intuitively,  $\mathcal{F}$  describes one step of deduction. Namely, all the elements that can be deduced in one step starting from a given subset of the universe.

### Example:

$$\begin{array}{ccc} \overline{\quad} & \frac{c}{b} & \frac{b \quad c}{a} \\ c & & \end{array}$$

is characterized by

$$\begin{array}{ll} \mathcal{F}(\emptyset) = \{c\} & \mathcal{F}(\{a, b\}) = \{c\} \\ \mathcal{F}(\{a\}) = \{c\} & \mathcal{F}(\{a, c\}) = \{b, c\} \\ \mathcal{F}(\{b\}) = \{c\} & \mathcal{F}(\{b, c\}) = \{a, b, c\} \\ \mathcal{F}(\{c\}) = \{b, c\} & \mathcal{F}(\{a, b, c\}) = \{a, b, c\} \end{array}$$

Monotonicity represents the fact that the larger the set of the premises one starts from, the larger the set of the consequences that can be deduced.

## Definition (Fixed point)

Let  $\mathcal{F}$  be a generating function on  $\mathcal{U}$ . If  $X \subseteq \mathcal{U}$ , then

- $X$  is  *$\mathcal{F}$ -closed* iff  $\mathcal{F}(X) \subseteq X$ .
- $X$  is  *$\mathcal{F}$ -consistent* iff  $X \subseteq \mathcal{F}(X)$ .
- $X$  is a *fixed point of  $\mathcal{F}$*  iff  $X = \mathcal{F}(X)$ .

A fixed point of  $\mathcal{F}$  is thus a set that is both  $\mathcal{F}$ -closed and  $\mathcal{F}$ -consistent: it contains all the consequences of its elements ( $\mathcal{F}$ -closed), it contains justifications for all its elements ( $\mathcal{F}$ -consistent), and nothing else.

A deduction system (actually, a *generating function*) characterizes a set of fixed points among which two are of particular interest.

Let  $\mathcal{F}$  be a generating function on  $\mathcal{U}$ . We use:

$\mu\mathcal{F}$  to denote the *least fixed point* of  $\mathcal{F}$ .

$\nu\mathcal{F}$  to denote the *greatest fixed point* of  $\mathcal{F}$ .

Notice that the greatest and least fixed point always exist since  $(\subseteq, \mathcal{U})$  is a complete lattice.

### Theorem (Knaster-Tarski [Tarski 1955])

Let  $\mathcal{U}$  be a complete lattice and  $\mathcal{F} : \mathcal{U} \rightarrow \mathcal{U}$  a monotone function. Then the set of fixed points of  $\mathcal{F}$  in  $\mathcal{U}$  is also a complete lattice.

In particular we have

$$\mu\mathcal{F} = \bigcap \{X \mid X \text{ is } \mathcal{F}\text{-closed}\} \quad (1)$$

$$\nu\mathcal{F} = \bigcup \{X \mid X \text{ is } \mathcal{F}\text{-consistent}\} \quad (2)$$

**Proof.** We consider only part (2); part (1) is symmetric. Let  $C = \{X \mid X \subseteq \mathcal{F}(X)\}$  be the collection of all  $\mathcal{F}$ -consistent sets, and let  $P$  be the union of all these sets. For any  $X \in C$ , we have  $X \subseteq \mathcal{F}(X) \subseteq \mathcal{F}(P)$ , since  $X$  is  $\mathcal{F}$ -consistent,  $X \subseteq P$ , and  $\mathcal{F}$  is monotone. Consequently,  $P = \bigcup_{X \in C} X \subseteq \mathcal{F}(P)$ , i.e.  $P$  is  $\mathcal{F}$ -consistent. By construction  $P$  is the largest  $\mathcal{F}$ -consistent set and the monotonicity of  $\mathcal{F}$  yields  $\mathcal{F}(P) \subseteq \mathcal{F}(\mathcal{F}(P))$ . Therefore  $\mathcal{F}(P) \in C$ . Hence, as for any member of  $C$ , we have  $\mathcal{F}(P) \subseteq P$ , i.e.  $P$  is  $\mathcal{F}$ -closed. Now we have established both that  $P$  is the largest  $\mathcal{F}$ -consistent set and that  $P$  is a fixed point of  $\mathcal{F}$ , so  $P$  is the largest fixed point.

# Principles of Induction and Coinduction

The interest of these two sets is given by the following Corollary:

## Corollary

**Principle of induction:** *if  $X$  is  $\mathcal{F}$ -closed, then  $\mu\mathcal{F} \subseteq X$ .* (1)

**Principle of coinduction:** *if  $X$  is  $\mathcal{F}$ -consistent, then  $X \subseteq \nu\mathcal{F}$ .* (2)

**The Principle of Induction** states that  $\mu\mathcal{F}$  can be constructed inductively and shows that every property that is preserved by  $\mathcal{F}$  holds for all elements in  $\mu\mathcal{F}$ : if  $X_\varphi$  is the set of all elements that satisfy a property  $\varphi$ , and the property is preserved by  $\mathcal{F}$  (ie.  $\mathcal{F}(X_\varphi) \subseteq X_\varphi$ ), then by (1),  $\mu\mathcal{F} \subseteq X_\varphi$ .

**The Principle of Coinduction** states that in order to prove that  $x \in \nu\mathcal{F}$  it suffices to build an  $\mathcal{F}$ -consistent set  $X$  such that  $x \in X$ .

## Remark:

The principle of coinduction is used for instance to prove equivalence of infinite reductions (eg. prove bisimilarity by providing a bisimulation). Here we use it to prove a relation on infinite expansions by providing a self-justifying (ie, an  $\mathcal{F}$ -consistent) subset of the relation.

We start our formal treatment without using the  $\mu$ -notation: by  $\mu$ -notation we can represent just regular trees, while our results are stated and holds for more general infinite trees.

### Definition (Tree type)

A tree  $T \in \mathcal{T}$  is a partial mapping  $T : \{1, 2\}^* \rightarrow \{\times, \rightarrow, \text{Any}\}$  such that

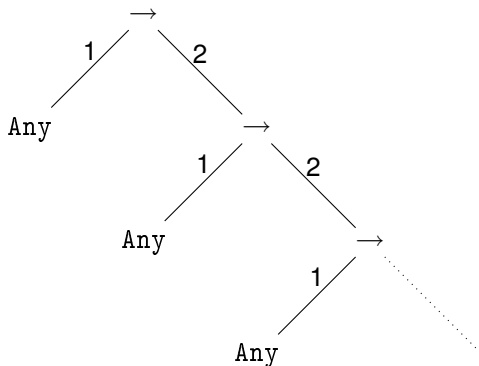
- 1  $T(\varepsilon)$  is defined ( $\varepsilon$  denotes the empty sequence)
- 2 if  $T(\pi \cdot \pi')$  is defined, then  $T(\pi)$  is defined
- 3 if  $T(\pi) \neq \text{Any}$ , then  $T(\pi \cdot i)$  is defined (for  $i = 1, 2$ )
- 4 if  $T(\pi) = \text{Any}$ , then  $T(\pi \cdot i)$  is **undefined** (for  $i = 1, 2$ )

### Definition (Finite and regular trees)

A tree  $T$  is *finite* ( $T \in \mathcal{T}_f$ ) if  $\text{dom}(T)$  is finite. It is *regular* if it has finitely many distinct subtrees.

# Example

$\text{Any} \rightarrow (\text{Any} \rightarrow (\text{Any} \rightarrow \dots))$



It is not finite but it is regular (just two distinct subtrees: `Any` and itself).

## Compactly:

$$T ::= \text{Any} \mid T \rightarrow T \mid T \times T$$

⇓

Deduction system

$$\frac{}{\text{Any}} \qquad \frac{T_1 \quad T_2}{T_1 \times T_2} \qquad \frac{T_1 \quad T_2}{T_1 \rightarrow T_2}$$

⇓

Let  $F$  be the corresponding generating function  
from labeled trees to labeled trees

⇓

$$\mathcal{T} = \nu F \qquad \mathcal{T}_f = \mu F$$

# Subtyping

- $\mathcal{S}_f$  generating function for subtyping finite trees ( $\mathcal{U} = \mathcal{T}_f \times \mathcal{T}_f$ )
- $\mathcal{S}$  generating function for subtyping infinite trees ( $\mathcal{U} = \mathcal{T} \times \mathcal{T}$ )

They are defined as follows:

$$\begin{aligned}\mathcal{S}_f(R) &= \{(T, \text{Any}) \mid T \in \mathcal{T}_f\} \cup \\ &\quad \{(\mathcal{S}_1 \times \mathcal{S}_2, T_1 \times T_2) \mid (\mathcal{S}_1, T_1), (\mathcal{S}_2, T_2) \in R\} \cup \\ &\quad \{(\mathcal{S}_1 \rightarrow \mathcal{S}_2, T_1 \rightarrow T_2) \mid (T_1, \mathcal{S}_1), (\mathcal{S}_2, T_2) \in R\}\end{aligned}$$

$$\begin{aligned}\mathcal{S}(R) &= \{(T, \text{Any}) \mid T \in \mathcal{T}\} \cup \\ &\quad \{(\mathcal{S}_1 \times \mathcal{S}_2, T_1 \times T_2) \mid (\mathcal{S}_1, T_1), (\mathcal{S}_2, T_2) \in R\} \cup \\ &\quad \{(\mathcal{S}_1 \rightarrow \mathcal{S}_2, T_1 \rightarrow T_2) \mid (T_1, \mathcal{S}_1), (\mathcal{S}_2, T_2) \in R\}\end{aligned}$$

In both cases they characterize:

$$\begin{array}{c} \text{TOP} \\ \hline T \leq \text{Any} \end{array} \quad \begin{array}{c} \text{PROD} \\ \hline \frac{\mathcal{S}_1 \leq T_1 \quad \mathcal{S}_2 \leq T_2}{\mathcal{S}_1 \times \mathcal{S}_2 \leq T_1 \times T_2} \end{array} \quad \begin{array}{c} \text{ARROW} \\ \hline \frac{T_1 \leq \mathcal{S}_1 \quad \mathcal{S}_2 \leq T_2}{\mathcal{S}_1 \rightarrow \mathcal{S}_2 \leq T_1 \rightarrow T_2} \end{array}$$

**But on different universes (in particular on infinite trees for  $\mathcal{S}$ ).**

## Subtyping relations

The subtyping relation on  $\mathcal{T}_f$  is  $\mu\mathcal{S}_f$

The subtyping relation on  $\mathcal{T}$  is  $\nu\mathcal{S}$

## Note

- $\mu\mathcal{S}_f = \nu\mathcal{S}_f$  (Exercise)
- $\mu\mathcal{S} \subsetneq \nu\mathcal{S}$ : for instance  $\nu\mathcal{S}_f$  is not reflexive (on infinite types) since it relates only finite types (use the principle of induction).

Recall that transitivity of the subtyping relation is a cornerstone of the theory of subtyping. Without it one could not prove, say, the subject reduction property. Imagine  $S \leq T$ ,  $T \leq U$ , but  $S \not\leq U$ . Let  $a : U \rightarrow \text{Any}$  and  $b : S$ ; consider

$$(\lambda x : T. ax)b$$

The application above is well-typed, but its reductum, that is  $ab$ , it is not

## Definition (transitivity)

A relation  $R \subseteq \mathcal{U} \times \mathcal{U}$  is *transitive* if it is  $TR$ -closed, where

$$TR(X) = \{(x, y) \mid \exists z \in \mathcal{U}. (x, z), (z, y) \in X\}$$

Intuitively,  $TR$  performs a step of transitivity closure.

## Lemma

Let  $\mathcal{F} : \mathcal{P}(\mathcal{U} \times \mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U} \times \mathcal{U})$ . If  $\forall X \text{TR}(\mathcal{F}(X)) \subseteq \mathcal{F}(\text{TR}(X))$ , then  $\forall \mathcal{F}$  is transitive.

Notice the connection between this lemma and the cut-elimination technique:

- $y \in \text{TR}(\mathcal{F}(X))$  means that  $y$  is obtained by transitivity from something that was deduced (in 1 step) from  $X$ .
- $y \in \mathcal{F}(\text{TR}(X))$  means that  $y$  is deduced (in 1 step) from something that was obtained by transitivity from  $X$ .

So the lemma states that if we can “push transitivity upward” in  $\mathcal{F}$ -deductions, then  $\mathcal{F}$  is transitive.

We need the lemma above to prove that  $\forall S_{(f)}$  is transitive.

**Exercise:** Prove that  $\forall S$  is reflexive.

## Transitivity: digression

Subtyping relation that are inductively defined admit to distinct presentations:

- declarative (with transitive rule);
- algorithmic (without the transitive rule)

This works with inductive definitions since they preserve *closure* properties:

### Theorem

*Let  $\mathcal{F}$  and  $\mathcal{G}$  be two generating functions. Define  $\mathcal{H}(X) = \mathcal{F}(x) \cup \mathcal{G}(X)$ . Then  $\mu\mathcal{H}$  is the smallest set that is both  $\mathcal{F}$ -closed and  $\mathcal{G}$ -closed.*

So adding the transitivity rule (the one corresponding a *TR*) to an inductive systems extends transitivity to the whole resulting system.

### Caveat

**This does not work with coinductive systems**  
we will just consider algorithmic systems

The next step is to develop *algorithms* to check membership of lfp and gfp of a generating function  $\mathcal{F}$ .

## Intuition

The algorithms “run  $\mathcal{F}$  backward”, so as to check how  $\mathcal{F}$  generated a given element  $x$ .

In order to avoid *backtracking* (ie, non deterministic algorithms) we focus on **invertible functions**, that is, generating functions that characterize *syntax-directed* deduction systems.

## Definition

Let  $\mathcal{F}$  be a generating function  $\mathcal{F} : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$  and  $x \in \mathcal{U}$ .

- 1  $X \subseteq \mathcal{U}$  is a *generating set* for  $x$  if  $x \in \mathcal{F}(X)$ ;
- 2 We use  $\mathbb{G}_x$  to denote the collection of generating sets for  $x$ , that is

$$\mathbb{G}_x = \{X \subseteq \mathcal{U} \mid x \in \mathcal{F}(X)\}$$

- 3  $\mathcal{F}$  is *invertible* if  $\forall x \in \mathcal{U}$ ,  $\mathbb{G}_x$  is either empty or has a minimum
- 4 Let  $\mathcal{F}$  be an invertible generating function, define:

$$\text{support}_{\mathcal{F}}(x) = \begin{cases} \uparrow & \text{if } \mathbb{G}_x = \emptyset \\ \min \mathbb{G}_x & \text{otherwise} \end{cases}$$

$$\text{support}_{\mathcal{F}}(X) = \begin{cases} \bigcup_{x \in X} \text{support}_{\mathcal{F}}(x) & \text{if } \forall x, \text{support}_{\mathcal{F}}(x) \neq \uparrow \\ \uparrow & \text{otherwise} \end{cases}$$

## Algorithms for $gfp_{\mathcal{F}}$ and $lfp_{\mathcal{F}}$

Let  $\mathcal{F} : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$  be an invertible generating function and  $X \subseteq \mathcal{U}$ :

$gfp_{\mathcal{F}}(X) =$  **if**  $\text{support}_{\mathcal{F}}(X) = \uparrow$  **then** false  
**else if**  $\text{support}_{\mathcal{F}}(X) \subseteq X$  **then** true  
**else**  $gfp_{\mathcal{F}}(\text{support}_{\mathcal{F}}(X) \cup X)$

$lfp_{\mathcal{F}}(X) =$  **if**  $\text{support}_{\mathcal{F}}(X) = \uparrow$  **then** false  
**else if**  $X = \emptyset$  **then** true  
**else**  $lfp_{\mathcal{F}}(\text{support}_{\mathcal{F}}(X))$

## Theorem

- If  $gfp_{\mathcal{F}}(X)$ , then  $X \subseteq \nu\mathcal{F}$ ;
- If  $\neg gfp_{\mathcal{F}}(X)$ , then  $X \not\subseteq \nu\mathcal{F}$ ;
- If  $lfp_{\mathcal{F}}(X)$ , then  $X \subseteq \mu\mathcal{F}$ ;
- If  $\neg lfp_{\mathcal{F}}(X)$ , then  $X \not\subseteq \mu\mathcal{F}$ .

# Termination for $v\mathcal{F}$

We want to find properties to ensure that the algorithms terminate (try  $lfp_{\mathcal{F}}$ , where  $\mathcal{F}$  is the function defined in the first exercise of slide 104)

## Definition

Let  $\mathcal{F}$  be an invertible generating function, define:

$$\textcircled{1} \quad \text{pred}_{\mathcal{F}}(x) = \begin{cases} \emptyset & \text{if } \text{support}_{\mathcal{F}}(x) = \uparrow \\ \text{support}_{\mathcal{F}}(x) & \text{otherwise} \end{cases}$$

$$\text{pred}_{\mathcal{F}}(X) = \bigcup_{x \in X} \text{pred}_{\mathcal{F}}(x)$$

$$\text{reachable}_{\mathcal{F}}(X) = \bigcup_{n \geq 0} \text{pred}_{\mathcal{F}}^n(x)$$

$\textcircled{2}$   $\mathcal{F}$  is *finite state* if  $\forall X \subseteq \mathcal{U}. \text{reachable}_{\mathcal{F}}(X)$  is finite.

So  $\text{reachable}_{\mathcal{F}}(X)$  is the set of all elements that are reachable from  $X$  via  $\text{support}_{\mathcal{F}}(\cdot)$ . Clearly, the algorithm terminates if this set is always finite.

## Theorem

*If  $\mathcal{F}$  is finite state, then  $gfp_{\mathcal{F}}(X)$  terminates for every  $X$  finite.*

# Efficiency for $\forall\mathcal{F}$

Consider:

$$\frac{b \quad c}{a} \quad \frac{d}{b} \quad \frac{e}{d} \quad \frac{b}{e}$$

and follow the computation of

$$gfp_{\mathcal{F}}(\{a\}) = GFP_{\mathcal{F}}(\{a, b, c\}) = GFP_{\mathcal{F}}(\{a, b, c, d\}) = GFP_{\mathcal{F}}(\{a, b, c, d, e\}) = \text{true}$$

$gfp_{\mathcal{F}}(X) =$  **if**  $\text{support}_{\mathcal{F}}(X) = \uparrow$  **then** false  
**else if**  $\text{support}_{\mathcal{F}}(X) \subseteq X$  **then** true  
**else**  $gfp_{\mathcal{F}}(\text{support}_{\mathcal{F}}(X) \cup X)$

Nota bene

$\text{support}_{\mathcal{F}}(a)$  is computed 4 times!

## Use memoization:

### Without memoization:

$$\begin{aligned} \text{gfp}_{\mathcal{F}}(X) = & \text{ if } \text{support}_{\mathcal{F}}(X) = \uparrow \text{ then false} \\ & \text{ else if } \text{support}_{\mathcal{F}}(X) \subseteq X \text{ then true} \\ & \text{ else } \text{gfp}_{\mathcal{F}}(\text{support}_{\mathcal{F}}(X) \cup X) \end{aligned}$$

### With memoization:

$$\begin{aligned} \text{gfp}_{\mathcal{F}}^M(A, X) = & \text{ if } \text{support}_{\mathcal{F}}(X) = \uparrow \text{ then false} \\ & \text{ else if } X = \emptyset \text{ then true} \\ & \text{ else } \text{gfp}_{\mathcal{F}}^M(A \cup X, \text{support}_{\mathcal{F}}(X) \setminus (A \cup X)) \end{aligned}$$

The problem with the definition above is that the actual computation is performed on single elements rather than sets of elements. The formulation above hides possible repetition on single elements. Let us expose this computation.

$$\begin{aligned} \text{gfp}_{\mathcal{F}}^{\text{elm}}(A, x) &= \text{if } \text{support}_{\mathcal{F}}(x) = \uparrow \text{ then false} \\ &\text{else if } \text{support}_{\mathcal{F}}(x) \subseteq A \text{ then true} \\ &\text{else let } \{x_1, \dots, x_n\} = \text{support}_{\mathcal{F}}(x) \text{ in} \\ &\quad \text{gfp}_{\mathcal{F}}^{\text{elm}}(A \cup \{x\}, x_1) \wedge \dots \wedge \text{gfp}_{\mathcal{F}}^{\text{elm}}(A \cup \{x\}, x_n) \end{aligned}$$

The formulation above still presents two problems:

- 1 The definition is not tail-recursive
- 2 Much worse: the calculations for each  $x_1, \dots, x_n$  are independent, which yields exponential complexity (*cf.* Amadio-Cardelli algorithm).

Idea to solve both problems:

We *thread* the different recursive calls as follows:

$$\text{gfp}_{\mathcal{F}}^{\text{thrd}}(A \cup \{x\}, x_1) \wedge \text{gfp}_{\mathcal{F}}^{\text{thrd}}(\quad, x_2) \wedge \text{gfp}_{\mathcal{F}}^{\text{thrd}}(\quad, x_3) \wedge \dots$$

so as to reuse the assumption sets calculated in the previous calls.

$gfp_{\mathcal{F}}^{\text{thrd}}(A, x) =$  **if**  $x \in A$  **then**  $A$  **else**  
**if**  $\text{support}_{\mathcal{F}}(x) = \uparrow$  **then fail else**  
**let**  $\{x_1, \dots, x_n\} = \text{support}_{\mathcal{F}}(x)$  **in**  
 $gfp_{\mathcal{F}}^{\text{thrd}}(gfp_{\mathcal{F}}^{\text{thrd}}(\dots(gfp_{\mathcal{F}}^{\text{thrd}}(gfp_{\mathcal{F}}^{\text{thrd}}(A \cup \{x\}, x_1), x_2)\dots), x_{n-1}), x_n)$

Equivalently:

$$\begin{aligned} \text{gfp}_{\mathcal{F}}^{\text{thrd}}(A, x) = & \text{ if } x \in A \text{ then } A \text{ else} \\ & \text{ if } \text{support}_{\mathcal{F}}(x) = \uparrow \text{ then fail else} \\ & \text{ let } \{x_1, \dots, x_n\} = \text{support}_{\mathcal{F}}(x) \text{ in} \\ & \text{ let } A_1 = A \cup \{x\} \text{ in} \\ & \text{ let } A_2 = \text{gfp}_{\mathcal{F}}^{\text{thrd}}(A_1, x_1) \text{ in} \\ & \quad \vdots \\ & \text{ let } A_{n-1} = \text{gfp}_{\mathcal{F}}^{\text{thrd}}(A_{n-2}, x_{n-2}) \text{ in} \\ & \text{ let } A_n = \text{gfp}_{\mathcal{F}}^{\text{thrd}}(A_{n-1}, x_{n-1}) \text{ in } \text{gfp}_{\mathcal{F}}^{\text{thrd}}(A_n, x_n) \end{aligned}$$

## Theorem

- 1 If  $\text{gfp}_{\mathcal{F}}^{\text{thrd}}(\emptyset, x) = A'$  then  $x \in \text{v}\mathcal{F}$ .
- 2 If  $\text{gfp}_{\mathcal{F}}^{\text{thrd}}(\emptyset, x) = \text{fail}$  then  $x \notin \text{v}\mathcal{F}$ .

## Goal

Instantiate the generic algorithm for  $vF$  to trees

We do not want to work on generic infinite tree types, but on trees that are representable in a machine, that is, trees *with a finite representation*.

## Definition (Regular tree)

A tree  $T \in \mathcal{T}$  is *regular* if it has finitely many distinct subtrees. We use  $\mathcal{T}_r$  to denote the set of regular trees.

## Examples:

$T = \text{Any} \rightarrow (\text{Any} \times \text{Any})$       $T = \text{Any} \times (\text{Any} \times (\text{Any} \times \dots))$

subtrees:

$T$

Any

Any  $\times$  Any

$T = (A \times (B \times (A \times (B \times (B \times (A \times (B \times (B \times B(\dots))))))))))$  is not regular

subtrees:

$T$

Any

# Regular Trees

Let  $\mathcal{S}$  be the generating function on infinite trees ( $\mathcal{U} = \mathcal{T} \times \mathcal{T}$ ) characterizing the deduction system (see slide 114):

$$\text{TOP} \frac{}{T \leq \text{Any}} \quad \text{PROD} \frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \quad \text{ARROW} \frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

and define  $\mathcal{S}_r = \mathcal{S}|_{\mathcal{T}_r}$

## Theorem

$\mathcal{S}_r$  is finite state.

*Proof:* exercise.

## Corollary

We have an algorithm to decide for every  $T \in \mathcal{T}_r$  whether  $T \in \nu \mathcal{S}_r$

**It remains to find a way to represent regular tree types.**

$\mu$ Types are a representation of regular tree types.

$$T ::= \text{Any} \mid X \mid T \rightarrow T \mid T \times T \mid \mu X. T$$

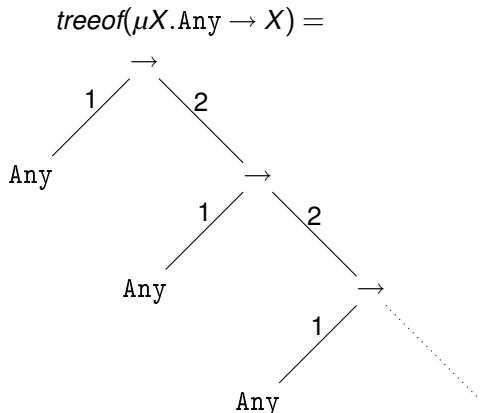
Let  $\mathcal{T}_\mu$  denote the set of all *closed* and *contractive* terms inductively generated by the productions above. It is possible to define a mapping from  $\mu$ Types to regular trees, which maps  $\mu$ Types into their denotations.

## Definition (treeof)

$\text{treeof}(\text{Any})(\varepsilon)$	$=$	$\text{Any}$	
$\text{treeof}(T_1 \rightarrow T_2)(\varepsilon)$	$=$	$\rightarrow$	
$\text{treeof}(T_1 \times T_2)(\varepsilon)$	$=$	$\times$	
$\text{treeof}(T_1 \rightarrow T_2)(i \cdot \pi)$	$=$	$\text{treeof}(T_i)(\pi)$	for $i = 1, 2$
$\text{treeof}(T_1 \times T_2)(i \cdot \pi)$	$=$	$\text{treeof}(T_i)(\pi)$	for $i = 1, 2$
$\text{treeof}(\mu X. T)(\pi)$	$=$	$\text{treeof}(T[\mu X. T/X])(\pi)$	
$\text{treeof}(T)(\pi)$	$=$	$\perp$	otherwise

**Exercise:** explain the need for contractivity.

# Example



## Exercise

Show that *treeof* is not injective and how to associate to each regular tree a “canonical”  $\mu$ Type.

# $\mu$ Types subtyping algorithm

Intuitively, the subtyping relation on  $\mathcal{T}_\mu$  is obtained by adding to the rules of the deduction system in slide 114 (used coinductively) the rules

$$\frac{S[\mu X.S/X] \leq T}{\mu X.S \leq T} \qquad \frac{S \leq T[\mu X.T/X]}{S \leq \mu X.T}$$

Formally speaking the subtyping relation on  $\mathcal{T}_\mu$  is the greatest fixpoint of  $S_\mu : \mathcal{P}(\mathcal{T}_\mu \times \mathcal{T}_\mu) \rightarrow \mathcal{P}(\mathcal{T}_\mu \times \mathcal{T}_\mu)$  defined as follows:

$$\begin{aligned} S_\mu(R) = & \{(T, \text{Any}) \mid T \in \mathcal{T}_\mu\} \cup \\ & \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\} \cup \\ & \{(S_1 \rightarrow S_2, T_1 \rightarrow T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \cup \\ & \{(S, \mu X.T) \mid (S, T[\mu X.T/X]) \in R\} \cup \\ & \{(\mu X.S, T) \mid (S[\mu X.S/X] \leq T) \in R, T \neq \text{Any}, T \neq \mu Y.U\} \end{aligned}$$

## Theorem

For all  $S, T \in \mathcal{T}_\mu$ ,  $(S, T) \in \nu S_\mu \iff (\text{treeof}(S), \text{treeof}(T)) \in \mu S$

To obtain a subtyping algorithm for the  $\mu$ Types, it suffices to instantiate  $gfp_{\mathcal{F}}^{\text{thrd}}$  by taking  $S_{\mu}$  for  $\mathcal{F}$ . This yields the following algorithm:

```
subtype(A, S, T) = if (S, T)  $\in$  A then A else  
  let  $A_0 = A \cup \{(S, T)\}$  in  
  if  $T = \text{Any}$  then  $A_0$   
    else if  $S = S_1 \times S_2$  and  $T = T_1 \times T_2$  then  
      subtype(subtype( $A_0$ ,  $S_1$ ,  $T_1$ ),  $S_2$ ,  $T_2$ )  
    else if  $S = S_1 \rightarrow S_2$  and  $T = T_1 \rightarrow T_2$  then  
      subtype(subtype( $A_0$ ,  $T_1$ ,  $S_1$ ),  $S_2$ ,  $T_2$ )  
    else if  $T = \mu X. T_1$  then  
      subtype( $A_0$ , S,  $T_1[\mu X. T_1 / X]$ )  
    else if  $S = \mu X. S_1$  then  
      subtype( $A_0$ ,  $S_1[\mu X. S_1 / X]$ , T)  
  else fail
```

## Compare the previous algorithm with the Amadio-Cardelli algorithm:

$$\frac{}{A \vdash S \leq T} (S, T) \in A$$

$$\frac{}{A \vdash S \leq \text{Any}} (S, \text{Any}) \notin A$$

$$\frac{A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \times S_2 \leq T_1 \times T_2} A' = AU(S_1 \times S_2, T_1 \times T_2); A \neq A'$$

$$\frac{A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} A' = AU(S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A'$$

$$\frac{A' \vdash S[\mu X.S/X] \leq T}{A \vdash \mu X.S \leq T} A' = AU(\mu X.S, T); A \neq A'; T \neq \text{Any}$$

$$\frac{A' \vdash S \leq T[\mu X.T/X]}{A \vdash S \leq \mu X.T} A' = AU(S, \mu X.T); A \neq A'; S \neq \mu Y.U$$



They both check containment in the relation coinductively defined by:

$$\text{TOP} \frac{}{T \leq \text{Any}} \quad \text{PROD} \frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \quad \text{ARROW} \frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

$$\text{UNFOLD LEFT} \frac{S[\mu X.S/X] \leq T}{\mu X.S \leq T} \quad \text{UNFOLD RIGHT} \frac{S \leq T[\mu X.T/X]}{S \leq \mu X.T}$$

But the former is far more efficient.

- 1 Simple Types
- 2 Extensions: products, records, references
- 3 A simple model of objects
- 4 Parametric Types
- 5 Generics
- 6 Recursive Types
- 7 Bibliography**

-  R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 14(4):575-631, 1993.
-  Pierce et al. Recursive types revealed, *Journal of Functional Programming*, 12(6):511-548, 2002.