Subtyping

Outline

Outline

Simply Typed λ-calculus

Syntax

Types	T	$::=$	$T \to T$	function types
			$\texttt{Bool} \mid \texttt{Int} \mid \texttt{Real} \mid \ldots$	basic types
Terms	a, b	$::=$	$\texttt{true} \mid \texttt{false} \mid 1 \mid 2 \mid \ldots$	constants
		$\mid$	x	variable
		$\mid$	$a\,b$	application
		$\mid$	$\lambda x{:}T.a$	abstraction

Reduction

$$\textit{Contexts} \quad C[\,] \quad ::= \quad [\,] \quad \mid \quad a[\,] \quad \mid \quad [\,]a \quad \mid \quad \lambda x{:}T.[\,]$$

$$\textsc{Beta} \qquad (\lambda x{:}T.a)b \longrightarrow a[b/x]$$

$$\textsc{Context} \qquad \frac{a \longrightarrow b}{C[a] \longrightarrow C[b]}$$

Type system

Typing

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x)$$

$$\to\textsc{Intro}\quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T}$$

$$\to\textsc{Elim}\quad \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

(plus the typing rules for constants).

Typing

VAR
$$\Gamma \vdash x : \Gamma(x)$$

$\rightarrow$INTRO
$$\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \rightarrow T}$$

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

(plus the typing rules for constants).

Theorem (Subject Reduction)

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

Type system

Typing

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \to\textsc{Intro}\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad \to\textsc{Elim}\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

(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 $\varnothing \vdash a : T$ and $a \not\longrightarrow$, then a is a value

where a value is either a constant or a lambda abstraction

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

Type checking algorithm

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

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   *)
  | λx:T.a -> typecheck (gamma, x:T) a             (* Intro rule *)
  | ab -> let T1→T2 = typecheck gamma a in         (* Elim rule  *)
          let T3 = typecheck gamma b in
            if T1==T3 then T2 else fail
```

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   *)
  | λx:T.a -> typecheck (gamma, x:T) a             (* Intro rule *)
  | ab -> let T1→T2 = typecheck gamma a in         (* Elim rule  *)
          let T3 = typecheck gamma b in
            if T1==T3 then T2 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:

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}\ (\to\text{Elim})$$

So, for instance, we **cannot:**

Subtyping

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

$$\to\textsc{Elim}\quad \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

So, for instance, we **cannot:**

- Apply a function of type `Int` $\to$ `Int` to an argument of type `Odd` even though every odd number is an integer number, too.

Subtyping

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

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}\ (\to\textsc{Elim})$$

So, for instance, we **cannot:**

- Apply a function of type $\mathtt{Int} \to \mathtt{Int}$ to an argument of type $\mathtt{Odd}$ even though every odd number is an integer number, too.
- If we have records, apply the function $\lambda x{:}\{\ell : \mathtt{Int}\}.(3 + x.\ell)$ to a record of type $\{\ell : \mathtt{Int}, \ell' : \mathtt{Bool}\}$

Subtyping

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

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}\ (\to\text{ELIM})$$

So, for instance, we **cannot:**

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

Subtyping

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

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

So, for instance, we **cannot:**

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

Subtyping polymorphism

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

Subtyping relation

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

Subtyping relation

- Define a pre-order (*ie*, a reflexive and transitive binary relation) $\leq$ on types: $\leq \subset$ *Types* $\times$ *Types* (some literature uses the notation `<:`)
- This *subtyping relation* has two possible interpretations:

Subtyping relation

- Define a pre-order (*ie*, a reflexive and transitive binary relation) $\leq$ on types: $\leq \subset$ *Types* $\times$ *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.

Subtyping relation

- Define a pre-order (*ie*, a reflexive and transitive binary relation) $\leq$ on types: $\leq \subset$ *Types* $\times$ *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.

Subtyping relation

- Define a pre-order (*ie*, a reflexive and transitive binary relation) $\leq$ on types: $\leq \subset$ *Types* $\times$ *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.

Subtyping for simply typed λ-calculus

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

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

Subtyping for simply typed λ-calculus

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

  For instance take the reflexive and transitive closure of $\{(\mathtt{Odd},\mathtt{Int}),(\mathtt{Even},\mathtt{Int}),(\mathtt{Int},\mathtt{Real})\}$
- To extend it to function types, we resort to the sustitutability 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 $S \leq T_1 \to T_2$

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

Subtyping of arrows: intuition

Problem

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

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

1. If $a : T_1$, then we can apply f to a. If $S \leq T_1 \to T_2$, then we can apply g to a, as well.
   $\Rightarrow$ g is a function, therefore $S = S_1 \to S_2$

Subtyping of arrows: intuition

Problem

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

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

1. If $a : T_1$, then we can apply f to a. If $S \leq T_1 \to T_2$, then we can apply g to a, as well.
   $\Rightarrow$ g is a function, therefore $S = S_1 \to S_2$
2. If $a : T_1$, then $f(a)$ is well typed. If $S_1 \to S_2 \leq T_1 \to T_2$, 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 $T_1 \leq S_1$

Subtyping of arrows: intuition

Problem

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

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

1. If $a : T_1$, then we can apply f to a. If $S \leq T_1 \to T_2$, then we can apply g to a, as well.
   $\Rightarrow$ g is a function, therefore $S = S_1 \to S_2$
2. If $a : T_1$, then $f(a)$ is well typed. If $S_1 \to S_2 \leq T_1 \to T_2$, 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.

Subtyping of arrows: intuition

Problem

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

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

1. If $a : T_1$, then we can apply f to a. If $S \leq T_1 \to T_2$, then we can apply g to a, as well.
   $\Rightarrow$ g is a function, therefore $S = S_1 \to S_2$
2. If $a : T_1$, then $f(a)$ is well typed. If $S_1 \to S_2 \leq T_1 \to T_2$, 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

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

Covariance and contravariance

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

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

We say that the type constructor $\to$ is

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

Covariance and contravariance

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

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

We say that the type constructor $\to$ 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 ...

Covariance and contravariance

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

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

We say that the type constructor $\to$ 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`.

  $\texttt{Int}\to\texttt{Int} \leq \texttt{Int}\to\texttt{Real}$ (covariance of the codomains)

Covariance and contravariance

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

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

We say that the type constructor $\to$ 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`.
  `Int`$\to$`Int`$\leq$ `Int`$\to$`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.
  `Int`$\to$`Int`$\leq$ `Odd`$\to$`Int` (contravariance of the codomains)

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

$$\textsc{Refl}\ \frac{}{T \leq T} \qquad \textsc{Trans}\ \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

$$\textsc{Refl}\ \frac{}{T \leq T} \qquad \textsc{Trans}\ \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

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

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

$$\textsc{Refl}\ \frac{}{T \leq T} \qquad \textsc{Trans}\ \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

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

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

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

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

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

These rules describe a deterministic and terminating algorithm (we say that the system is algorithmic).

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

Subtyping deduction system

$$\textsc{Basic}\ \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2} \qquad\qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

These rules describe a deterministic and terminating algorithm (we say that the system is algorithmic).

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*

Type system

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

Type system

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

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \textsc{$\rightarrow$Intro}\quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \rightarrow T} \qquad \textsc{$\rightarrow$Elim}\quad \frac{\Gamma \vdash a : S \rightarrow T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

Type system

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

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \textsc{$\to$Intro}\quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad \textsc{$\to$Elim}\quad \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

Type system

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

VAR
$$\Gamma \vdash x : \Gamma(x)$$

$\rightarrow$INTRO
$$\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \rightarrow T}$$

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

SUBSUMPTION
$$\frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

This corresponds to the *containment relation*:

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

Type system

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

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \rightarrow\textsc{Intro}\quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \rightarrow T} \qquad \rightarrow\textsc{Elim}\quad \frac{\Gamma \vdash a : S \rightarrow T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

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 $\varnothing \vdash a : T$ and $a \not\longrightarrow$, then a is a value

Typing algorithm

VAR

$$\Gamma \vdash x : \Gamma(x)$$

→INTRO

$$\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T}$$

→ELIM

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

SUBSUMPTION

$$\frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

Typing algorithm

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x) \qquad \rightarrow\textsc{Intro}\quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \rightarrow T}$$

$$\rightarrow\textsc{Elim}\quad \frac{\Gamma \vdash a : S \rightarrow T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T} \qquad \textsc{Subsumption}\quad \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

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?

Typing algorithm

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x)$$

$$\textsc{$\to$Intro}\quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T}$$

$$\textsc{$\to$Elim}\quad \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$\textsc{Subsumption}\quad \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

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

VAR
$$\Gamma \vdash_{\mathcal{A}} x : \Gamma(x)$$

$\to$INTRO
$$\frac{\Gamma, x : S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}S.a : S \to T}$$

$\to$ELIM$_{\leq}$
$$\frac{\Gamma \vdash_{\mathcal{A}} a : S{\to}T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U{\leq}S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

$\to$ELIM
$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

SUBSUMPTION
$$\frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

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

$$\textsc{Var}\quad \Gamma \vdash_{\mathcal{A}} x : \Gamma(x) \qquad \to\textsc{Intro}\quad \frac{\Gamma, x : S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}S.a : S\to T} \qquad \to\textsc{Elim}_{\leq}\quad \frac{\Gamma \vdash_{\mathcal{A}} a : S\to T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

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).

Typing algorithm

VAR
$$\Gamma \vdash_{\mathcal{A}} x : \Gamma(x)$$

→INTRO
$$\frac{\Gamma, x : S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}S.a : S \to T}$$

→ELIM$_{\leq}$
$$\frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

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?

Typing algorithm

$$\textsc{Var}\quad \Gamma \vdash_{\mathcal{A}} x : \Gamma(x) \qquad \to\textsc{Intro}\quad \frac{\Gamma, x:S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}S.a : S \to T} \qquad \to\textsc{Elim}_{\leq}\quad \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

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:

$$\varnothing \vdash \lambda x{:}\mathtt{Int}.x : \mathtt{Odd} \to \mathtt{Real} \qquad \text{but} \qquad \varnothing \nvdash_{\mathcal{A}} \lambda x{:}\mathtt{Int}.x : \mathtt{Odd} \to \mathtt{Real}.$$

Typing algorithm

$$\textsc{Var}\quad \Gamma \vdash_{\mathcal{A}} x : \Gamma(x) \qquad \rightarrow\textsc{Intro}\quad \frac{\Gamma, x : S \vdash_{\mathcal{A}} a : T}{\Gamma \vdash_{\mathcal{A}} \lambda x{:}S.a : S \rightarrow T} \qquad \rightarrow\textsc{Elim}_{\leq}\quad \frac{\Gamma \vdash_{\mathcal{A}} a : S \rightarrow T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

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:

$$\varnothing \vdash \lambda x{:}\texttt{Int}.x : \texttt{Odd} \rightarrow \texttt{Real} \qquad \text{but} \qquad \varnothing \nvdash_{\mathcal{A}} \lambda x{:}\texttt{Int}.x : \texttt{Odd} \rightarrow \texttt{Real}.$$

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

Soundness and completeness of the typing algorithm

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

$\Leftarrow$ = soundness

$\Rightarrow$ = completeness

Soundness and completeness of the typing algorithm

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

$\Leftarrow$ = soundness

$\Rightarrow$ = completeness

Theorem (Soundness)

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

Theorem (Completeness)

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

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.

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

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.

Summary for simply-typed λ-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.

Summary for simply-typed λ-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 corresponds to embed subtype checking into elimination rules.

Summary for simply-typed λ-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 corresponds 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.

Products I

Syntax

Types	T	$::=$	$\ldots \mid T \times T$	product types
Terms	a, b	$::=$	$\ldots$	
		$\mid$	(a, a)	pair
		$\mid$	$\pi_i(a) \quad (i{=}1{,}2)$	projection

Reduction

$$\pi_i((a_1, a_2)) \longrightarrow a_i \qquad (i{=}1{,}2)$$

Typing

$$\times\textsc{Intro}\ \frac{\Gamma \vdash a_1 : T_1 \qquad \Gamma \vdash a_2 : T_2}{\Gamma \vdash (a_1, a_2) : T_1 \times T_2} \qquad\qquad \times\textsc{Elim}_i\ \frac{\Gamma \vdash a : T_1 \times T_2}{\Gamma \vdash \pi_i(a) : T_i}\ (i{=}1{,}2)$$

Products II

Subtyping

$$\textsc{Prod}\quad \frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2}$$

Exercise: *Check whether the above rule is compatible with the containement 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 π_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

Types	T	$::=$	$... \mid \{\ell : T, ..., \ell : T\}$	record types
Terms	a, b	$::=$	$...$	
		$\mid$	$\{\ell = a, ..., \ell = a\}$	record
		$\mid$	$a.\ell$	field selection

Reduction

$$\{..., \ell = a, ...\}.\ell \longrightarrow a$$

Typing

{}Intro

$$\frac{\Gamma \vdash a_1 : T_1 \ \ldots\ \Gamma \vdash a_n : T_n}{\Gamma \vdash \{\ell_1 = a_1, ..., \ell_n = a_n\} : \{\ell_1 : T_1, ..., \ell_n : T_n\}}$$

{}Elim

$$\frac{\Gamma \vdash a : \{..., \ell : T, ...\}}{\Gamma \vdash a.\ell : T}$$

Record Subtyping

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

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

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

Exercise. *Which are the algorithmic typing rules?*

Outline

Iso-recursive and Equi-recursive types

Lists are a classic example of recursive types:

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

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

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

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

$$\texttt{unfold} : \mu X.T \to T[\mu X.T/X] \qquad \texttt{fold} : T[\mu X.T/X] \to \mu X.T$$

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

Iso-recursive and Equi-recursive types

Lists are a classic example of recursive types:

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

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

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

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

$$\texttt{unfold} : \mu X.T \to T[\mu X.T/X] \qquad \texttt{fold} : T[\mu X.T/X] \to \mu X.T$$

Equi-recursive types: $\mu X.((\texttt{Int} \times X) \vee \texttt{Nil})$ is considered *equal* to its one-step unfolding $(\texttt{Int} \times \mu X.((\texttt{Int} \times X) \vee \texttt{Nil})) \vee \texttt{Nil})$. 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:

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

Recursive types are weird

- To add (equi-)recursive types you do not need to add any new term

Recursive types are weird

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

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

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

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

Recursive types are weird

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

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

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

  Then $\mu X.(\mathtt{Int} \times X)$ 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 informally 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

Types	T	$::=$	$\texttt{Any}$	top type
		$\mid$	$T \to T$	function types
		$\mid$	$T \times T$	product types
		$\mid$	X	type variables
		$\mid$	$\mu X.T$	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 $\to$ or a $\times$.

Subtyping recursive types

The subtyping relation is defined *COINDUCTIVELY* by the rules

$$\textsc{Top}\ \frac{}{T \leq \texttt{Any}} \qquad \textsc{Prod}\ \frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

$$\textsc{Unfold Left}\ \frac{S[\mu X.S/X] \leq T}{\mu X.S \leq T} \qquad \textsc{Unfold Right}\ \frac{S \leq T[\mu X.T/X]}{S \leq \mu X.T}$$

Subtyping recursive types

The subtyping relation is defined *COINDUCTIVELY* by the rules

$$\textsc{Top}\ \frac{}{T \leq \texttt{Any}} \qquad \textsc{Prod}\ \frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

$$\textsc{Unfold Left}\ \frac{S[\mu X.S/X] \leq T}{\mu X.S \leq T} \qquad \textsc{Unfold Right}\ \frac{S \leq T[\mu X.T/X]}{S \leq \mu X.T}$$

Coinductive definition

1. Why coinduction?
2. Why no reflexivity/transitivity rules?
3. Why no rule to compare two μ-types?

Subtyping recursive types

The subtyping relation is defined *COINDUCTIVELY* by the rules

$$\text{TOP}\ \frac{}{T \leq \texttt{Any}} \qquad \text{PROD}\ \frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \qquad \text{ARROW}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

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

Coinductive definition

1. Why coinduction?
2. Why no reflexivity/transitivity rules?
3. Why no rule to compare two μ-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

$$
\textsc{Unfold Left}\ \dfrac{\textsc{Unfold Right}\ \dfrac{\textsc{Arrow}\ \dfrac{\mathtt{Even} \leq \mathtt{Int} \qquad \mu X.\mathtt{Int} \to X \leq \mu Y.\mathtt{Even} \to Y}{\mathtt{Int} \to (\mu X.\mathtt{Int} \to X) \leq \mathtt{Even} \to (\mu Y.\mathtt{Even} \to Y)}}{\mathtt{Int} \to (\mu X.\mathtt{Int} \to X) \leq \mu Y.\mathtt{Even} \to Y}}{\mu X.\mathtt{Int} \to X \leq \mu Y.\mathtt{Even} \to Y}
$$

Example of coinductive derivation

$$
\textsc{Unfold Left}\;\dfrac{\textsc{Unfold Right}\;\dfrac{\textsc{Arrow}\;\dfrac{\texttt{Even} \leq \texttt{Int} \qquad \color{red}{\mu X.\texttt{Int} \to X \leq \mu Y.\texttt{Even} \to Y}}{\texttt{Int} \to (\mu X.\texttt{Int} \to X) \leq \texttt{Even} \to (\mu Y.\texttt{Even} \to Y)}}{\texttt{Int} \to (\mu X.\texttt{Int} \to X) \leq \mu Y.\texttt{Even} \to Y}}{\color{red}{\mu X.\texttt{Int} \to X \leq \mu Y.\texttt{Even} \to Y}}
$$

Notice the use of coinduction

Amadio and Cardelli's subtyping algorithm

Let $A \subset Types \times Types$

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Amadio and Cardelli's subtyping algorithm

Determinization of the rules

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Amadio and Cardelli's subtyping algorithm

Memoization

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2} \; A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Amadio and Cardelli's subtyping algorithm

Determinization of the rules

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Memoization

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2} \; A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Amadio and Cardelli's subtyping algorithm

The rest is similar

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Amadio and Cardelli's subtyping algorithm

Let $A \subset Types \times Types$

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

Properties

Theorem (Soundness and Completeness)

Let S and T be closed types. $S \leq T$ belongs the relation coinductively defined by the rules in slide 374 if and only if $\varnothing \vdash S \leq T$ is provable

Properties

Theorem (Soundness and Completeness)

Let S and T be closed types. $S \leq T$ belongs the relation coinductively defined by the rules in slide 374 if and only if $\varnothing \vdash S \leq T$ is provable

To see the proof of the above theorem you can refer to the following reference Pierce et al. Recursive types revealed, Journal of Functional Programming, 12(6):511-548, 2002.

Properties

Theorem (Soundness and Completeness)

Let S and T be closed types. $S \leq T$ belongs the relation coinductively defined by the rules in slide 374 if and only if $\varnothing \vdash S \leq T$ is provable

To see the proof of the above theorem you can refer to the following reference Pierce et al. Recursive types revealed, Journal of Functional Programming, 12(6):511-548, 2002.

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.

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 decution 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

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 decution 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 $\varnothing$, 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.

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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.

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:

$\{\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:

$\{d\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:

$\{d, e\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:

$\{d, e\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:
$\{d, e\}$

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

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:
$\{d, e\}$

Coinductively:
$\{a, b, c, d, e, f, g\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:
$\{d, e\}$

Coinductively:
$\{a, b, c, d, e, g\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:
$\{d, e\}$

Coinductively:
$\{a, b, c, d, e, g\}$

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively:
$\{d, e\}$

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

Induction and coinduction

induction: start from $\varnothing$, 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 (algorithimic, 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:

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

Inductively: $\{d, e\}$

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

Self-justifying set: $\{a, b, c\}$

Exercises

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

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

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

Why Coinduction for Recursive types?

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

Why Coinduction for Recursive types?

We want to use $S = \mu X.\mathtt{Int} \to X$ where $T = \mu Y.\mathtt{Even} \to Y$ 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` ...

Why Coinduction for Recursive types?

We want to use $S = \mu X.\texttt{Int} \to X$ where $T = \mu Y.\texttt{Even} \to Y$ 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 ...

Why Coinduction for Recursive types?

We want to use $S = \mu X.\mathtt{Int} \to X$ where $T = \mu Y.\mathtt{Even} \to Y$ 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.

$$S \leq T \quad \Longrightarrow \quad \mathtt{Int} \to S \leq \mathtt{Even} \to T \quad \Longrightarrow \quad S \leq T \wedge \mathtt{Even} \leq \mathtt{Int}$$

This is exactly the proof we saw at the beginning:

$$\textsc{Unfold Left}\ \dfrac{\textsc{Unfold Right}\ \dfrac{\textsc{Arrow}\ \dfrac{\texttt{Even} \leq \texttt{Int} \qquad \overbrace{\mu X.\texttt{Int} \to X}^{S} \leq \overbrace{\mu Y.\texttt{Even} \to Y}^{T}}{\texttt{Int} \to (\mu X.\texttt{Int} \to X) \leq \texttt{Even} \to (\mu Y.\texttt{Even} \to Y)}}{\texttt{Int} \to (\mu X.\texttt{Int} \to X) \leq \mu Y.\texttt{Even} \to Y}}{\underbrace{\mu X.\texttt{Int} \to X}_{S} \leq \underbrace{\mu Y.\texttt{Even} \to Y}_{T}}$$

This is exactly the proof we saw at the beginning:

$$\textsc{Unfold Left}\ \dfrac{\textsc{Unfold Right}\ \dfrac{\textsc{Arrow}\ \dfrac{\mathtt{Even} \leq \mathtt{Int} \qquad \overbrace{\mu X.\mathtt{Int} \to X}^{S} \leq \overbrace{\mu Y.\mathtt{Even} \to Y}^{T}}{\mathtt{Int} \to (\mu X.\mathtt{Int} \to X) \leq \mathtt{Even} \to (\mu Y.\mathtt{Even} \to Y)}}{\mathtt{Int} \to (\mu X.\mathtt{Int} \to X) \leq \mu Y.\mathtt{Even} \to Y}}{\underbrace{\mu X.\mathtt{Int} \to X}_{S} \leq \underbrace{\mu Y.\mathtt{Even} \to Y}_{T}}$$

Coinduction

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

This is exactly the proof we saw at the beginning:

$$\textsc{Unfold Left}\ \dfrac{\textsc{Unfold Right}\ \dfrac{\textsc{Arrow}\ \dfrac{\mathtt{Even} \leq \mathtt{Int} \qquad \overbrace{\mu X.\mathtt{Int} \to X}^{S} \leq \overbrace{\mu Y.\mathtt{Even} \to Y}^{T}}{\mathtt{Int} \to (\mu X.\mathtt{Int} \to X) \leq \mathtt{Even} \to (\mu Y.\mathtt{Even} \to Y)}}{\mathtt{Int} \to (\mu X.\mathtt{Int} \to X) \leq \mu Y.\mathtt{Even} \to Y}}{\underbrace{\mu X.\mathtt{Int} \to X}_{S} \leq \underbrace{\mu Y.\mathtt{Even} \to Y}_{T}}$$

Coinduction

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

Observation:

1. The deduction above shows why a specific rule for μ 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).

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we “thread” the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$\mathit{subtype}(A, S, T) \quad = \quad \textbf{if } (S, T) \in A \textbf{ then } A \textbf{ else}$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we “thread” the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$\begin{aligned} subtype(A, S, T) \quad = \quad & \textbf{if } (S, T) \in A \textbf{ then } A \textbf{ else} \\ & \textbf{let } A_0 = A \cup \{(S, T)\} \textbf{ in} \end{aligned}$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we “thread” the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$
\begin{aligned}
\textit{subtype}(A, S, T) \quad = \quad & \textbf{if } (S, T) \in A \textbf{ then } A \textbf{ else} \\
& \textbf{let } A_0 = A \cup \{(S, T)\} \textbf{ in} \\
& \textbf{if } T = \texttt{Any} \textbf{ then } A_0
\end{aligned}
$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we "thread" the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$
\begin{array}{rcl}
\mathit{subtype}(A,S,T) & = & \textbf{if } (S,T)\in A \textbf{ then } A \textbf{ else} \\
 & & \textbf{let } A_0 = A\cup\{(S,T)\} \textbf{ in} \\
 & & \textbf{if } T = \mathtt{Any} \textbf{ then } A_0 \\
 & & \quad \textbf{else if } S = S_1\times S_2 \textbf{ and } T = T_1\times T_2 \textbf{ then} \\
 & & \qquad \mathit{subtype}(\mathit{subtype}(A_0,S_1,T_1),S_2,T_2)
\end{array}
$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we “thread” the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$
\begin{array}{rcl}
\mathit{subtype}(A,S,T) & = & \textbf{if } (S,T)\in A \textbf{ then } A \textbf{ else} \\
& & \textbf{let } A_0 = A\cup\{(S,T)\} \textbf{ in} \\
& & \textbf{if } T = \texttt{Any} \textbf{ then } A_0 \\
& & \quad \textbf{else if } S = S_1\times S_2 \textbf{ and } T = T_1\times T_2 \textbf{ then} \\
& & \qquad \mathit{subtype}(\mathit{subtype}(A_0,S_1,T_1),S_2,T_2) \\
& & \quad \textbf{else if } S = S_1\to S_2 \textbf{ and } T = T_1\to T_2 \textbf{ then} \\
& & \qquad \mathit{subtype}(\mathit{subtype}(A_0,T_1,S_1),S_2,T_2)
\end{array}
$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we “thread” the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$
\begin{array}{rcl}
\mathit{subtype}(A, S, T) & = & \textbf{if } (S, T) \in A \textbf{ then } A \textbf{ else} \\
 & & \textbf{let } A_0 = A \cup \{(S, T)\} \textbf{ in} \\
 & & \textbf{if } T = \texttt{Any} \textbf{ then } A_0 \\
 & & \quad \textbf{else if } S = S_1 \times S_2 \textbf{ and } T = T_1 \times T_2 \textbf{ then} \\
 & & \qquad \mathit{subtype}(\mathit{subtype}(A_0, S_1, T_1), S_2, T_2) \\
 & & \quad \textbf{else if } S = S_1 \to S_2 \textbf{ and } T = T_1 \to T_2 \textbf{ then} \\
 & & \qquad \mathit{subtype}(\mathit{subtype}(A_0, T_1, S_1), S_2, T_2) \\
 & & \quad \textbf{else if } T = \mu X.T_1 \textbf{ then} \\
 & & \qquad \mathit{subtype}(A_0, S, T_1[\mu X.T_1/X])
\end{array}
$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we "thread" the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$
\begin{array}{rcl}
\mathit{subtype}(A,S,T) & = & \textbf{if } (S,T)\in A \textbf{ then } A \textbf{ else} \\
 & & \textbf{let } A_0 = A\cup\{(S,T)\} \textbf{ in} \\
 & & \textbf{if } T = \mathtt{Any} \textbf{ then } A_0 \\
 & & \quad \textbf{else if } S = S_1\times S_2 \textbf{ and } T = T_1\times T_2 \textbf{ then} \\
 & & \qquad \mathit{subtype}(\mathit{subtype}(A_0,S_1,T_1),S_2,T_2) \\
 & & \quad \textbf{else if } S = S_1\to S_2 \textbf{ and } T = T_1\to T_2 \textbf{ then} \\
 & & \qquad \mathit{subtype}(\mathit{subtype}(A_0,T_1,S_1),S_2,T_2) \\
 & & \quad \textbf{else if } T = \mu X.T_1 \textbf{ then} \\
 & & \qquad \mathit{subtype}(A_0,S,T_1[\mu X.T_1/X]) \\
 & & \quad \textbf{else if } S = \mu X.S_1 \textbf{ then} \\
 & & \qquad \mathit{subtype}(A_0,S_1[\mu X.S_1/X],T)
\end{array}
$$

A naive implementation of the Amadio-Cardelli algorithm is exponential (why?). If we "thread" the computation of the memoization environments we obtain a quadratic complexity. This is done as follows:

$$
\begin{array}{rcl}
\mathit{subtype}(A,S,T) & = & \textbf{if } (S,T)\in A \textbf{ then } A \textbf{ else}\\
& & \textbf{let } A_0 = A\cup\{(S,T)\} \textbf{ in}\\
& & \textbf{if } T = \texttt{Any} \textbf{ then } A_0\\
& & \quad \textbf{else if } S = S_1\times S_2 \textbf{ and } T = T_1\times T_2 \textbf{ then}\\
& & \qquad \mathit{subtype}(\mathit{subtype}(A_0,S_1,T_1),S_2,T_2)\\
& & \quad \textbf{else if } S = S_1\to S_2 \textbf{ and } T = T_1\to T_2 \textbf{ then}\\
& & \qquad \mathit{subtype}(\mathit{subtype}(A_0,T_1,S_1),S_2,T_2)\\
& & \quad \textbf{else if } T = \mu X.T_1 \textbf{ then}\\
& & \qquad \mathit{subtype}(A_0,S,T_1[\mu X.T_1/X])\\
& & \quad \textbf{else if } S = \mu X.S_1 \textbf{ then}\\
& & \qquad \mathit{subtype}(A_0,S_1[\mu X.S_1/X],T)\\
& & \textbf{else } \texttt{fail}
\end{array}
$$

Compare the previous algorithm with the Amadio-Cardelli algorithm:

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

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

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

$$\frac{A' \vdash T_1 \leq S_1 \qquad A' \vdash S_2 \leq T_2}{A \vdash S_1 \to S_2 \leq T_1 \to T_2}\ A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A'$$

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

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

They both check containment in the relation coinductively defined by:

$$\textsc{Top}\ \frac{}{T \leq \texttt{Any}} \qquad \textsc{Prod}\ \frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \qquad \textsc{Arrow}\ \frac{T_1 \leq S_1 \qquad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$$

$$\textsc{Unfold Left}\ \frac{S[\mu X.S/X] \leq T}{\mu X.S \leq T} \qquad \textsc{Unfold Right}\ \frac{S \leq T[\mu X.T/X]}{S \leq \mu X.T}$$

But the former is far more efficient.

Outline

References

- 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.