Four Forms of Polymorphism

SIGPL Summer School 2019

Giuseppe Castagna

CNRS

Outline of the course

- Background and Motivations

  Polymorphism - Motivating Examples - A Refresher Course on Operational Semantics

- Subtyping polymorphism

  Simple Types - Recursive Types - Bibliography

- Parametric polymorphism

  Introduction - Hindley-Milner System - Inference algorithm

- Ad-Hoc polymorphism

  Set-theoretic types - Semantic Subtyping - Application to a language. - Adding Parametric Polymorphism: the Types - Adding Parametric Polymorphism: the Language

- Gradual Typing (dynamic type polymorphism)

  Main ideas - Formal system - Algorithmic Aspects - Criteria for Gradual Typing - Implementation issues - References

Background and Motivations

Outline

Outline

What is polymorphism?

Merriam-Webster Dictionary

The quality or state of existing in or assuming different forms

What is polymorphism?

Merriam-Webster Dictionary

The quality or state of existing in or assuming different forms

In computing: the capability of a programming entity to act as of being of different types.

What is polymorphism?

Merriam-Webster Dictionary

The quality or state of existing in or assuming different forms

In computing: the capability of a programming entity to act as of being of different types.

There exists several polymorphic programming entities:

- polymorphic functions (e.g., a function of type `int`$\to$`int` and of type `bool`$\to$`bool`)
- polymorphic data structures (e.g., a list whose elements are of any possible type)
- polymorphic classes (e.g. a class whose instances are stack of `int` and stacks of `bool`
- polymorphic operators (e.g., the symbol + to denote arithmetic sum and string concatenation
- ...

What is polymorphism?

Merriam-Webster Dictionary

The quality or state of existing in or assuming different forms

In computing: the capability of a programming entity to act as of being of different types.

There exists several polymorphic programming entities:

- **polymorphic functions** (e.g., a function of type `int`→`int` and of type `bool`→`bool`)
- polymorphic data structures (e.g., a list whose elements are of any possible type)
- polymorphic classes (e.g. a class whose instances are stack of `int` and stacks of `bool`
- polymorphic operators (e.g., the symbol + to denote arithmetic sum and string concatenation
- ...

In this course I focus on functions.

Polymorphic functions

Polymorphic functions

Functions that can be applied to arguments of different types

Polymorphic functions

Polymorphic functions

Functions that can be applied to arguments of different types

GOAL

How to define sound type system for polymorphic functions

Sound = all expressions that pass type-checking will never reduce to *stuck* terms such as `3(true)`

Polymorphic functions

Polymorphic functions

Functions that can be applied to arguments of different types

GOAL

How to define sound type system for polymorphic functions

Sound = all expressions that pass type-checking will never reduce to *stuck* terms such as `3(true)`

Four forms of polymorphism:

1. parametric,
2. subtyping,
3. ad-hoc,
4. dynamic

Four kinds of polymorphism

1. **Parametric polymorphism:**
   Functions that work with arguments of any type.

Four kinds of polymorphism

1. **Parametric polymorphism:**

   Functions that work with arguments of any type.

   They do not inspect “parametric” arguments, they just:
   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result

Four kinds of polymorphism

1. **Parametric polymorphism:**

   Functions that work with arguments of any type.

   They do not inspect “parametric” arguments, they just:

   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result

2. **Subtyping polymorphism:**

   Functions that work with arguments having certain properties:

Four kinds of polymorphism

1. **Parametric polymorphism:**

   Functions that work with arguments of any type.

   They do not inspect "parametric" arguments, they just:

   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result

2. **Subtyping polymorphism:**

   Functions that work with arguments having certain properties:

   They use the known properties of the arguments

Four kinds of polymorphism

1. **Parametric polymorphism:**
   Functions that work with arguments of any type.
   They do not inspect "parametric" arguments, they just:
   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result
2. **Subtyping polymorphism:**
   Functions that work with arguments having certain properties:
   They use the known properties of the arguments
3. **Ad-hoc polymorphism (a.k.a. overloading):**
   Functions that work with arguments belonging to a specific (finite) set of different types

Four kinds of polymorphism

1. **Parametric polymorphism:**
   Functions that work with arguments of any type.
   They do not inspect "parametric" arguments, they just:
   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result
2. **Subtyping polymorphism:**
   Functions that work with arguments having certain properties:
   They use the known properties of the arguments
3. **Ad-hoc polymorphism (a.k.a. overloading):**
   Functions that work with arguments belonging to a specific (finite) set of different types
   They execute different code for each type of the argument

Four kinds of polymorphism

1. **Parametric polymorphism:**
   Functions that work with arguments of any type.
   They do not inspect "parametric" arguments, they just:
   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result
2. **Subtyping polymorphism:**
   Functions that work with arguments having certain properties:
   They use the known properties of the arguments
3. **Ad-hoc polymorphism (a.k.a. overloading):**
   Functions that work with arguments belonging to a specific (finite) set of different types
   They execute different code for each type of the argument
4. **Dynamic/Unknow type:**
   Functions that make no assumption about the type *of some specific arguments*

Four kinds of polymorphism

1. **Parametric polymorphism:**
   Functions that work with arguments of any type.
   They do not inspect "parametric" arguments, they just:
   - either ignore them
   - or pass them to other polymorphic functions
   - or return them in the result
2. **Subtyping polymorphism:**
   Functions that work with arguments having certain properties:
   They use the known properties of the arguments
3. **Ad-hoc polymorphism (a.k.a. overloading):**
   Functions that work with arguments belonging to a specific (finite) set of different types
   They execute different code for each type of the argument
4. **Dynamic/Unknow type:**
   Functions that make no assumption about the type *of some specific arguments*
   They delay the check to the type of these arguments at run-time

Outline

1. Parametric polymorphism

Functions that work with arguments of any type.

They do not inspect "parametric" arguments, they just:

- either ignore them
- or pass them to other polymorphic functions
- or return them in the result

```
function first (x , y) {
  return x;
}
```

It can be applied to pairs of type $\mathtt{S} \times \mathtt{T} \to \mathtt{S}$ and returns a result of type $\mathtt{S}$, whatever types $\mathtt{S}$ and $\mathtt{T}$ are.

1. Parametric polymorphism

Functions that work with arguments of any type.

They do not inspect "parametric" arguments, they just:

- either ignore them
- or pass them to other polymorphic functions
- or return them in the result

```
function first (x , y) {
  return x;
}
```

It can be applied to pairs of type $\mathtt{S} \times \mathtt{T} \to \mathtt{S}$ and returns a result of type $\mathtt{S}$, whatever types $\mathtt{S}$ and $\mathtt{T}$ are.

Intuition

Add type variables and quantify them universally:

$$\forall\alpha,\beta\,.\,\alpha\times\beta\to\alpha$$

2. Subtyping polymorphism

Functions that work with arguments of with certain properties: They use the known properties of the arguments

```
function size (x) {
  return x.length;
}
```

It can be applied to objects with the property `lenght` and return (in general) an integer.

2. Subtyping polymorphism

Functions that work with arguments of with certain properties: They use the known properties of the arguments

```
function size (x) {
  return x.length;
}
```

It can be applied to objects with the property `lenght` and return (in general) an integer.

Intuition

Define an order relation on types and accept arguments of any subtype

$$\texttt{\{ length: number \}} \to \texttt{number}$$

Accepts arguments of any type $\texttt{T} \leq \texttt{\{ length: number \}}$
(e.g. `{ length: number, concat: string→string}`)

Combined usage

```
function size (x) {
  return x.length;
}
```

Subtyping + Parametric

Possibility two combine the two form of polymorphism

$$\forall\alpha.\{\ \texttt{length}\ :\ \ \alpha\ \} \to \alpha$$

Combined usage

```
function size (x) {
  return x.length;
}
```

Subtyping + Parametric

Possibility two combine the two form of polymorphism

$$\forall\alpha.\{\ \texttt{length}\ :\ \alpha\ \} \to \alpha$$

```
function doOnLength (x) {
  if (x.length > 4) { <do something> }
  return x
}
```

Bounded parametric

$$\forall\alpha \leq \{\ \texttt{length}\ :\ \texttt{number}\ \}.\ \ \alpha \to \alpha$$

3. *Ad hoc* polymorphism

Functions for arguments in a specific (finite) set of different types

They execute different code for each type of the argument

```
function double (x) {
    (typeof(x) === "number") ?  2*x  :  x.concat(x)
}
```

If applied to an integer returns an integer, if applied to a string returns a string

3. *Ad hoc* polymorphism

Functions for arguments in a specific (finite) set of different types

They execute different code for each type of the argument

```
function double (x) {
    (typeof(x) === "number") ?  2*x  :  x.concat(x)
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

3. *Ad hoc* polymorphism

Functions for arguments in a specific (finite) set of different types

They execute different code for each type of the argument

```
function double (x) {
    (typeof(x) === "number") ?  2*x  :  x.concat(x)
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

$$\texttt{(number|string)}\rightarrow\texttt{(number|string)}$$

3. *Ad hoc* polymorphism

Functions for arguments in a specific (finite) set of different types

They execute different code for each type of the argument

```
function double (x) {
    (typeof(x) === "number") ?  2*x   :   x.concat(x)
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

$$\texttt{(number|string)} \rightarrow \texttt{(number|string)}$$

3. *Ad hoc* polymorphism

Functions for arguments in a specific (finite) set of different types

They execute different code for each type of the argument

```
function double (x) {
    (typeof(x) === "number") ?  2*x  :  x.concat(x)
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

  $$(\texttt{number}|\texttt{string}) \to (\texttt{number}|\texttt{string})$$

- Better solution: intersection types

  $$(\texttt{number} \to \texttt{number})\ \&\ (\texttt{string} \to \texttt{string})$$

3. *Ad hoc* polymorphism

Functions for arguments in a specific (finite) set of different types

They execute different code for each type of the argument

```
function double (x) {
    (typeof(x) === "number") ?  2*x   :   x.concat(x)
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

  `(number|string)→(number|string)`

- Better solution: intersection types

  `(number→number) & (string→string)`

needs some form of occurrence typing

Combined usage

```
function double (x) {
  (typeof(x) === "number") ? 2*x : x.concat(x)
}
```

Set-theoretic + Subtyping

```
( number→number ) &
( (not(number) & {concat: string→string}) → string )
```

Actually, set-theoretic types are defined by subtyping

Combined usage

```
function double (x) {
  (typeof(x) === "number") ? 2*x : x.concat(x)
}
```

Set-theoretic + Subtyping

```
( number→number ) &
( (not(number) & {concat: string→string}) → string )
```

Actually, set-theoretic types are defined by subtyping

Set-theoretic + Parametric

```
∀α,β.  ( number→number ) &
       ( (α & not(number) & {concat: α → β}) → β)
```

Combined usage

```
function double (x) {
  (typeof(x) === "number") ? 2*x : x.concat(x)
}
```

Set-theoretic + Subtyping

```
( number→number ) &
( (not(number) & {concat: string→string}) → string )
```

Actually, set-theoretic types are defined by subtyping

Set-theoretic + Parametric

$\forall\alpha,\beta.$ `( number→number ) &`
`( (`α` & not(number) & {concat: `$\alpha \to \beta$`}) → `β`)`

a sophisticated way to write bounded polymorphism and recursive types:

$\forall\beta,\forall(\gamma \leq$ **not(number) &** $\mu X.$**{concat:** $X \to \beta$**}).**
(number→number) & ($\gamma \to \beta$**)**

4. Dynamic types

Functions that *for some specific arguments* delay the check of types at run-time

```
function double (x) {
    ( typeof(x) === "number" ) ?  2*x :  x.concat(x)
}
```

4. Dynamic types

Functions that *for some specific arguments* delay the check of types at run-time

```
function double (x) {
    (<some twisted condition>) ?  2*x :  x.concat(x)
}
```

4. Dynamic types

Functions that *for some specific arguments* delay the check of types at run-time

```
function double (x) {
    (<some twisted condition>) ?  2*x :  x.concat(x)
}
```

Cannot give a type to `x` that works with both `2*x` and `x.concat(x)`

4. Dynamic types

Functions that *for some specific arguments* delay the check of types at run-time

```
function double (x : ?) {
    (<some twisted condition>) ?  2*x :  x.concat(x)
}
```

Cannot give a type to `x` that works with both `2*x` and `x.concat(x)`

Solution

Add an unknown/type "?"

4. Dynamic types

Functions that *for some specific arguments* delay the check of types at run-time

```
function double (x : ?) {
    (<some twisted condition>) ?  2*x  :  x.concat(x)
}
```

Cannot give a type to `x` that works with both `2*x` and `x.concat(x)`

Solution

Add an unknown/type "?"

Develop a type theory for "?" such that:

- No solution for ? for some execution ⇒ statically reject
- No problem for any solution for ? ⇒ statically accept, do nothing
- For each possible execution there exists some solution for ? ⇒ statically accept and add run-time checks

Reject at compile time:

```
function wrong (x : ?) {
  return (2*x + x(2));  //cannot be a number and a function
}
```

Reject at compile time:

```
function wrong (x : ?) {
  return (2*x + x(2));  //cannot be a number and a function
}
```

Accept as is:

```
function ok (x : ?) {
  if (typeof(x) === "number"){ return 42 } else { return x }
}
```

Intuitively the function has type: **?** $\to$ (number | **?**)

Reject at compile time:

```
function wrong (x : ?) {
  return (2*x + x(2));  //cannot be a number and a function
}
```

Accept as is:

```
function ok (x : ?) {
  if (typeof(x) === "number"){ return 42 } else { return x }
}
```

Intuitively the function has type: $? \to (\texttt{number} \mid ?)$

Accept and insert checks:

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)
}
```

Compile as

```
function double (x : ?) {
  (<condition>) ? 2*(x⟨number⟩) : (x⟨string⟩).concat(x⟨string⟩)
}
```

Combined usage: all 4 together! (OCaml style)

```
let mymap (condition) (f) (x : ?) =
   if condition then Array.map f x else List.map f x
```

Combined usage: all 4 together! (OCaml style)

```
let mymap (condition) (f) (x : ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to \textbf{?} \to \textbf{?}$

Combined usage: all 4 together! (OCaml style)

```
let mymap (condition) (f) (x : ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to ? \to ?$

- x can be bound to anything (though only α list or α array work)
- no information on the type of the result (though only β list or β array are possible)

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to ((\alpha\ \texttt{array} | \alpha\ \texttt{list})\ \&\ ?) \to (\beta\ \texttt{array} | \beta\ \texttt{list})$

Combined usage: all 4 together! (OCaml style)

```
let mymap (condition) (f) (x : ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to ? \to ?$

- x can be bound to anything (though only α list or α array work)
- no information on the type of the result (though only β list or β array are possible)

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to (\,(\alpha\,\texttt{array} | \alpha\,\texttt{list})\,\&\,?\,) \to (\beta\,\texttt{array} | \beta\,\texttt{list})$

Compiled as:

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
   if condition then Array.map f (x⟨α array⟩)
   else List.map f (x⟨α list⟩)
```

Combined usage: all 4 together! (OCaml style)

```
let mymap (condition) (f) (x : ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to ? \to ?$

- $\texttt{x}$ can be bound to anything (though only $\alpha\,\texttt{list}$ or $\alpha\,\texttt{array}$ work)
- no information on the type of the result (though only $\beta\,\texttt{list}$ or $\beta\,\texttt{array}$ are possible)

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
   if condition then Array.map f x else List.map f x
```

Type: $\texttt{bool} \to (\alpha \to \beta) \to (\,(\alpha\,\texttt{array} | \alpha\,\texttt{list})\ \&\ ?\,) \to (\beta\,\texttt{array} | \beta\,\texttt{list})$

Compiled as:

```
let mymap (condition) (f) (x : (α array | α list) & ?) =
   if condition then Array.map f (x⟨α array⟩)
   else List.map f (x⟨α list⟩)
```

Cutting edge research: *Gradual typing, a new perspective*, POPL 19

Outline

Syntax and small-step semantics

Syntax

Terms	a, b	$::=$	N	Numeric constant
		$\mid$	x	Variable
		$\mid$	$a\,b$	Application
		$\mid$	$\lambda x.a$	Abstraction
Values	v	$::=$	$\lambda x.a \mid N$	

Syntax and small-step semantics

Syntax

$$
\begin{array}{llcll}
\textit{Terms} & a,b & ::= & N & \text{Numeric constant}\\
 & & \mid & x & \text{Variable}\\
 & & \mid & a\,b & \text{Application}\\
 & & \mid & \lambda x.a & \text{Abstraction}\\
\textit{Values} & v & ::= & \lambda x.a \mid N &
\end{array}
$$

Small step semantics for strict functional languages

$$\textit{Evaluation Contexts} \quad E \quad ::= \quad [\,] \mid E\,a \mid v\,E$$

$$\textsc{Beta}_v \qquad (\lambda x.a)\,v \rightarrow a[v/x]$$

$$\textsc{Context} \qquad \frac{a \rightarrow b}{E[a] \rightarrow E[b]}$$

Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under λ-abstractions;

Call-by-value: In an application $(\lambda x.a)\,b$, the argument b must be fully reduced to a value before β-reduction can take place.

Left-most reduction: In an application $a\,b$, we must reduce a to a value first before we can start reducing b.

Deterministic: For every term a, there is at most one b such that $a \to b$.

Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under λ-abstractions;

Call-by-value: In an application $(\lambda x.a)\,b$, the argument b must be fully reduced to a value before β-reduction can take place.

Left-most reduction: In an application $a\,b$, we must reduce a to a value first before we can start reducing b.

Deterministic: For every term a, there is at most one b such that $a \to b$.

Big step semantics for strict functional languages

$$N \Rightarrow N \qquad\qquad \lambda x.a \Rightarrow \lambda x.a \qquad\qquad \frac{a \Rightarrow \lambda x.c \quad b \Rightarrow v_\circ \quad c[v_\circ/x] \Rightarrow v}{a\,b \Rightarrow v}$$

Interpreter

The big step semantics induces an efficient implementation

```
type term =
  Const of int | Var of string | Lam of string * term | App of term * term

exception Error

let rec subst x v = function        (* assumes v is closed *)
  | Const n -> Const n
  | Var y -> if x = y then v else Var y
  | Lam(y, b) -> if x = y then Lam(y, b) else Lam(y, subst x v b)
  | App(b, c) -> App(subst x v b, subst x v c)

let rec eval = function
  | Const n -> Const n
  | Var x -> raise Error
  | Lam(x, a) -> Lam(x, a)
  | App(a, b) ->
      match eval a with
      | Lam(x, c) -> let v = eval b in eval (subst x v c)
      | _ -> raise Error
```

Exercises

1. Define the small-step and big-step semantics for the call-by-name
2. Deduce from the latter the interpreter
3. Use the technique introduced for the type `'a delayed` earlier in the course to implement an interpreter with lazy evaluation.

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash ab \Rightarrow v}$$

Giving up substitutions in favor of environments does not come for free

Improving implementation

Environments

- Implementing textual substitution $a[x/v]$ is *inefficient*. This is why compilers and interpreters *do not* implement it.
- Alternative: record the binding $x \mapsto v$ in an *environment* e

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_\circ \quad e; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash ab \Rightarrow v}$$

Giving up substitutions in favor of environments does not come for free

- **Lexical scoping** requires careful handling of environments

```
let x = 1 in
let f = λy.(x+1) in
let x = "foo" in
f 2
```

In the environment used to evaluate `f 2` the variable `x` is bound to 1.

Exercise

Try to evaluate

```
let x = 1 in
let f = λy.(x+1) in
let x = "foo" in
f 2
```

by the big-step semantics in the previous slide,
where `let` x `=` a `in` b is syntactic sugar for $(\lambda x.b)a$

let us outline it together

Function closures

To implement *lexical scoping in the presence of environments*, function abstractions $\lambda x.a$ must not evaluate to themselves, but to a function *closure*: a pair $(\lambda x.a)[e]$ (ie, the function and the *environment of its definition*)

Big step semantics with environments and closures

$$\begin{array}{lccl} \textit{Values} & v & ::= & N \mid (\lambda x.a)[e] \\ \textit{Environments} & e & ::= & x_1 \mapsto v_1 ; \ldots ; x_n \mapsto v_n \end{array}$$

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e_\circ] \quad e \vdash b \Rightarrow v_\circ \quad e_\circ ; x \mapsto v_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

De Bruijn indexes

Identify variable not by names but by the number $\underline{n}$ of λ's that separate the variable from its binder in the syntax tree.

$$\lambda x.(\lambda y.y\,x)x \quad \text{is} \quad \lambda.(\lambda.\underline{0}\,\underline{1})\underline{0}$$

$\underline{n}$ is the variable bound by the n-th enclosing λ. Environments become sequences of values, the n-th value of the sequence being the value of variable $\underline{n-1}$.

$$\begin{array}{lrcl} \textit{Terms} & a,b & ::= & N \mid \underline{n} \mid \lambda.a \mid a\,b \\ \textit{Values} & v & ::= & N \mid (\lambda.a)[e] \\ \textit{Environments} & e & ::= & v_0;v_1;\ldots;v_n \end{array}$$

$$\frac{e = v_0;\ldots;v_n;\ldots;v_m}{e \vdash \underline{n} \Rightarrow v_n} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda.a \Rightarrow (\lambda.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda.c)[e_\circ] \quad e \vdash b \Rightarrow v_\circ \quad v_\circ;e_\circ \vdash c \Rightarrow v}{e \vdash a\,b \Rightarrow v}$$

The canonical, efficient interpreter

```
# type term = Const of int | Var of int | Lam of term | App of term * term
   and value = Vint of int | Vclos of term * environment
   and environment = value list                      (* use Vec instead *)

# exception Error

# let rec eval e a =
    match a with
    | Const n -> Vint n
    | Var n -> List.nth e n                       (* will fail for open terms *)
    | Lam a -> Vclos(Lam a, e)
    | App(a, b) ->
        match eval e a with
        | Vclos(Lam c, e') ->
            let v = eval e b in
            eval (v :: e') c
        | _ -> raise Error



# eval [] (App ( Lam (Var 0), Const (2)));;          (*  (λx.x)2 → 2  *)
- : value = Vint 2
```

Note: To obtain improved performance one should implement environments by persistent extensible arrays: for instance by the `Vec` library by Luca de Alfaro.

Subtyping

Outline

Outline

Simply Typed λ-calculus

Syntax

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

Reduction

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

Beta

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

Context

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

Type system

Typing

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

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

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

(plus the typing rules for constants).

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 $\Gamma \vdash b : T$.

Type system

Typing

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

(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 must 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

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

Subject Reduction + Progress = Soundness

Soundness [Wright & Felleisen 1994]

A type system is *sound* if every well-typed expression either diverges or reduces to a value of type

Soundness is a corollary of subject reduction and progress

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 -> T → (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 -> T → (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:

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

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:

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

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

- Apply a function of type `Int` → `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:

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

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

- Apply a function of type $\texttt{Int} \to \texttt{Int}$ to an argument of type $\texttt{Odd}$ 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 $\{\ell : \texttt{Int}, \ell' : \texttt{Bool}\}$

Subtyping

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

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

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

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

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

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

- Apply a function of type `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: $\leq \subset$ *Types* $\times$ *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 $\mathcal{B} \subset \textit{Basic} \times \textit{Basic}$ for basic types (given by the language designer).

  For instance take the reflexive and transitive closure of $\{(\texttt{Odd}, \texttt{Int}), (\texttt{Even}, \texttt{Int}), (\texttt{Int}, \texttt{Real})\}$
- 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 $S \leq T_1 \rightarrow T_2$

Let $g : S$ and $f : T_1 \rightarrow T_2$. 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

$$S_1 \to S_2 \leq T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leq S_1 \text{ and } S_2 \leq T_2$$

Covariance and contravariance

$$S_1 \to S_2 \leq T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leq S_1 \text{ and } S_2 \leq T_2$$

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

$$S_1 \to S_2 \leq T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leq S_1 \text{ and } S_2 \leq T_2$$

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

$$S_1 \to S_2 \leq T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leq S_1 \text{ and } S_2 \leq T_2$$

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)

Covariance and contravariance

$$S_1 \to S_2 \leq T_1 \to T_2 \quad \Leftrightarrow \quad T_1 \leq S_1 \text{ and } S_2 \leq T_2$$

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

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

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

Subtyping deduction system

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

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?

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

Type system

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

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

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

Type system

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

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

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

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?

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

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

This corresponds to the *containment relation*:

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

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

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

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

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

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

Typing algorithm

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

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

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

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

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

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

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

$\to$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

VAR

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

$\rightarrow$INTRO

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

$\rightarrow$ELIM$_{\leq}$

$$\frac{\Gamma \vdash_{\mathcal{A}} a : S \rightarrow T \quad \Gamma \vdash_{\mathcal{A}} b : U \quad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

$\rightarrow$ELIM

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

SUBSUMPTION

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

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

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

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

$$\textsc{Var}\quad \Gamma \vdash_{\mathcal{A}} x : \Gamma(x)$$

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

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

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?

Typing algorithm

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

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:

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

Typing algorithm

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

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:

$$\varnothing \vdash \lambda x{:}\mathtt{Int}.x : \mathtt{Odd} \to \mathtt{Real} \qquad \text{but} \qquad \varnothing \not\vdash_{\mathcal{A}} \lambda x{:}\mathtt{Int}.x : \mathtt{Odd} \to \mathtt{Real}.$$

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

$$
\begin{array}{llcll}
\textit{Types} & T & ::= & \ldots \mid T \times T & \text{product types} \\
\textit{Terms} & a,b & ::= & \ldots & \\
 & & \mid & (a,a) & \text{pair} \\
 & & \mid & \pi_i(a) \quad {\scriptstyle (i=1,2)} & \text{projection}
\end{array}
$$

Reduction

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

Typing

$$\times\text{INTRO}\quad \frac{\Gamma \vdash a_1 : T_1 \qquad \Gamma \vdash a_2 : T_2}{\Gamma \vdash (a_1, a_2) : T_1 \times T_2}$$

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

Products II

Subtyping

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

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

Reduction

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

Typing

{}INTRO

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

{}ELIM

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

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

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

Exercise. *Which are the algorithmic typing rules?*

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:

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

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

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

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

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

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

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

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

Then $\mu X.(\texttt{Int} \times X)$ 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 $\mu X.\mu X_1.\ldots.\mu X_n.S$ 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

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

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

Coinductive definition

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

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

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

Example of coinductive derivation

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

Notice the use of coinduction

Amadio and Cardelli's subtyping algorithm

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

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

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

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

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

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

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

Amadio and Cardelli's subtyping algorithm

Determinization of the rules

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

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

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

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

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

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

Amadio and Cardelli's subtyping algorithm

Store the type to implement coinduction

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

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

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

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

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

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

Determinization of the rules

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

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

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

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

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

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

Amadio and Cardelli's subtyping algorithm

Store the type to implement coinduction

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

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

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

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

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

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

Amadio and Cardelli's subtyping algorithm

The rest is similar

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

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

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

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

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

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

Amadio and Cardelli's subtyping algorithm

Let $A \subset Types \times Types$

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

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

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

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

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

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

Properties

Theorem (Soundness and Completeness)

Let S and T be closed types. $S \leq T$ belongs the relation coinductively defined by the rules on slide 55 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 on slide 55 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 on slide 55 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.

Let $\mathcal{F}$ be a deduction system on a universe $\mathcal{U}$ (i.e. a monotone function from $\mathcal{P}(\mathcal{U})$ to $\mathcal{P}(\mathcal{U})$). 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.

Let $\mathcal{F}$ be a deduction system on a universe $\mathcal{U}$ (i.e. a monotone function from $\mathcal{P}(\mathcal{U})$ to $\mathcal{P}(\mathcal{U})$). 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:

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

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:

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

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:

$\{d, e\}$

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:

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

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:

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

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:

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

Inductively:
$\{d, e\}$

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

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

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 $S = \mu X.\mathtt{Int} \to X$ 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 ...

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 $\mathtt{Even}$ number,
2. fed by an $\mathtt{Even}$ number returns a function that behaves similarly: (1) wait for an $\mathtt{Even}$ ...

Now consider $f : S$, then f:

1. waits for an $\mathtt{Int}$ number,
2. fed by an $\mathtt{Int}$ (or a $\mathtt{Even}$) 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:

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

Coinduction

$S \leq T$ is not an axiom but $\{S \leq T\ ,\ \texttt{Even} \leq \texttt{Int}\}$ 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 on slide 61).

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

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

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

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

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

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.

Parametric polymorphism

Outline

Outline

Monomorphic calculus

Types	T	$::=$	$\texttt{Bool} \mid \texttt{Int} \mid \texttt{Real} \mid \ldots$	basic types
		$\mid$	$T \to T$	function 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
		$\mid$	$\texttt{let}\ x : T\ \texttt{=}\ a\ \texttt{in}\ b$	let

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

$$\frac{\Gamma \vdash a : S \quad \Gamma, x : S \vdash b : T}{\Gamma \vdash \texttt{let}\ x : S\ \texttt{=}\ a\ \texttt{in}\ b : T}$$

Parametric polymorphism

It is a pity to use the identity function just with a single type.

$$\texttt{let}\ f : \texttt{Int} \to \texttt{Int}\ \texttt{=}\ \lambda x{:}\texttt{Int}.x\ \texttt{in}\ b$$

In particular, if we get rid of type annotations we see that the identity function can be given several different types.

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

$$\frac{\Gamma \vdash a : S \quad \Gamma, x : S \vdash b : T}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b : T}$$

In particular, $\lambda x.x$ can be given all the types of the form $T \to T$ for every T.

Parametric polymorphism

We extend the syntax of types

Types	T	$::=$	$\texttt{Bool} \mid \texttt{Int} \mid \texttt{Real} \mid \ldots$	basic types
		$\mid$	$T \to T$	function types
		$\mid$	α	type variables
		$\mid$	$\forall\alpha.T$	polymorphic types

We add to the previous rules these two rules

$$\frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

The resulting system is called System F (Girard/Reynolds)

We can for instance derive

$$\lambda x.xx : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

and supposing we have pairs:

$$\texttt{let } f \texttt{ = } \lambda x.x \texttt{ in } (f\,3, f\,\mathsf{true}) : \texttt{Int} \times \texttt{Bool}$$

Remark

The condition $\alpha \notin \mathrm{fv}(\Gamma)$ in the rule

$$\frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T}$$

is crucial ... without it we can derive

$$\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{x : \alpha \vdash \forall\alpha.\alpha}}{\vdash \lambda x.x : \alpha \to (\forall\alpha.\alpha)}$$

and therefore type, for instance, $(\lambda x.x)1\,2$ with any type we wish

Bad news

For terms without type anotations the problems:

- type inference: given an expression a find if there exists a type T such that $a : T$
- type checking: given and expression a and a type T check whether $a : T$ holds

are both undecidable

(J. B. Wells. *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, 1994.)

Bad news

For terms without type anotations the problems:

- type inference: given an expression a find if there exists a type T such that $a : T$
- type checking: given and expression a and a type T check whether $a : T$ holds

are both undecidable

(J. B. Wells. *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, 1994.)

Solution 1: use explicit type abstractions and instantiations (e.g., generics)
Solution 2: restrict the power of the system (e.g., Hindley-Milner)

Bad news

For terms without type anotations the problems:

- type inference: given an expression a find if there exists a type T such that $a : T$
- type checking: given and expression a and a type T check whether $a : T$ holds

are both undecidable

(J. B. Wells. *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, 1994.)

Solution 1: use explicit type abstractions and instantiations (e.g., generics)
Solution 2: restrict the power of the system (e.g., Hindley-Milner)

Hindley-Milner

We restrict the power of System F to have decidable type inference and type checking

(used in OCaml, SML, Haskell, etc ...)

Outline

The quantification can only be prenex:

Types	T	$::=$	$\texttt{Bool} \mid \texttt{Int} \mid \texttt{Real} \mid \ldots$	basic types
		$\mid$	$T \to T$	function types
		$\mid$	α	type variables
Schemas	σ	$::=$	T	type
		$\mid$	$\forall\alpha.\sigma$	schema

A type environment Γ now maps variable to *schemas*, and typing judgement have the form $\Gamma \vdash a : \sigma$

The following types (schemas) are ok:

$$\begin{array}{l}\forall\alpha.\alpha \to \alpha \\ \forall\alpha.\forall\beta.(\alpha \times \beta) \to \alpha \\ \forall\alpha.\texttt{Bool} \to \alpha \to \alpha \to \alpha \\ \forall\alpha.(\alpha \to \alpha) \to \alpha\end{array}$$

but the following type is not longer allowed:

$$(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

Hindley-Milner System

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

$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let } x \texttt{ = } a \texttt{ in } b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

Hindley-Milner System

Notice that the rule for let is the (only) rule that introduce a polymorphic type in the type environment.

$$\frac{\Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let } x \texttt{ = } a \texttt{ in } b : \sigma_2}$$

Thanks to this we can for instance type

$$\texttt{let } f \texttt{ = } \lambda x.x \texttt{ in } (f\,f)(f\,1)$$

with $f : \forall\alpha.\alpha \to \alpha$ in the context to type $(f\,f)(f\,1)$ in order to use three times the instantiation rule for the type schema:

$$\frac{f : \forall\alpha.\alpha \to \alpha \vdash f : \forall\alpha.\alpha \to \alpha}{f : \forall\alpha.\alpha \to \alpha \vdash f : (\alpha \to \alpha)[T/\alpha]}$$

where T is respectively for each occurrence of f, $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}$, $\texttt{Int} \to \texttt{Int}$, and $\texttt{Int}$.

Hindley-Milner System

On the contrary the rule for abstractions does not introduce in the environment a schema, but just a type

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

otherwise $S \to T$ would not be well formed.

In particular,

$$\lambda x.xx$$

is no longer typeable, while

$$\texttt{let } f = \lambda x.x \texttt{ in } f f$$

is still typeable.

Outline

Hindley-Milner Algorithm

The system is not syntax directed because of the following two rules apply to any expression:

$$\frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

Hindley-Milner syntax-directed system

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

$$\frac{T \sqsubseteq \Gamma(x)}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash a : S \quad \Gamma, x : \mathrm{Gen}(S, \Gamma) \vdash b : T}{\Gamma \vdash \texttt{let } x \texttt{ = } a \texttt{ in } b : T}$$

Hindley-Milner syntax-directed system

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

$$\frac{T \sqsubseteq \Gamma(x)}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash a : S \quad \Gamma, x : \mathrm{Gen}(S,\Gamma) \vdash b : T}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b : T}$$

Where

$$T \sqsubseteq \forall\alpha_1 \ldots \forall\alpha_n.S \iff \exists S_1, \ldots, S_n \text{ such that } T = S[S_1/\alpha_1 \ldots S_n/\alpha_n]$$

and

$$\mathrm{Gen}(S,\Gamma) = \forall\alpha_1 \ldots \forall\alpha_n.S \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \mathrm{fv}(S) \setminus \mathrm{fv}(\Gamma)$$

Hindley-Milner syntax-directed system

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

$$\frac{T \sqsubseteq \Gamma(x)}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash a : S \quad \Gamma, x : \mathrm{Gen}(S,\Gamma) \vdash b : T}{\Gamma \vdash \texttt{let } x \texttt{ = } a \texttt{ in } b : T}$$

Where

$$T \sqsubseteq \forall\alpha_1 \ldots \forall\alpha_n.S \iff \exists S_1, \ldots, S_n \text{ such that } T = S[S_1/\alpha_1 \ldots S_n/\alpha_n]$$

and

$$\mathrm{Gen}(S,\Gamma) = \forall\alpha_1 \ldots \forall\alpha_n.S \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \mathrm{fv}(S) \setminus \mathrm{fv}(\Gamma)$$

Syntax directed but **Not an algorithm yet!**

State: a current substitution ϕ and an infinite set of fresh variables V

$$
\begin{array}{rcl}
\texttt{fresh} & = & \texttt{do}\ \alpha \in V \\
 & & \texttt{do}\ V := V \setminus \{\alpha\} \\
 & & \texttt{return}\ \alpha \\
 & & \\
W(\Gamma \vdash x) & = & \texttt{let}\ \forall\alpha_1 \ldots \alpha_n.T \leftarrow \Gamma(x) \\
 & & \texttt{do}\ \beta_1,\ldots,\beta_n \leftarrow \texttt{fresh},\ldots,\texttt{fresh} \\
 & & \texttt{return}\ T[\beta_1/\alpha_1,\ldots,\beta_n/\alpha_n] \\
W(\Gamma \vdash \lambda x.a) & = & \texttt{do}\ \alpha \leftarrow \texttt{fresh} \\
 & & \texttt{do}\ T \leftarrow W(\Gamma, x:\alpha \vdash a) \\
 & & \texttt{return}\ \alpha \to T \\
W(\Gamma \vdash ab) & = & \texttt{do}\ T \leftarrow W(\Gamma \vdash a) \\
 & & \texttt{do}\ S \leftarrow W(\Gamma \vdash b) \\
 & & \texttt{do}\ \alpha \leftarrow \texttt{fresh} \\
 & & \texttt{do}\ \phi\ := \texttt{mgu}(\phi(T), \phi(S \to \alpha)) \circ \phi \\
 & & \texttt{return}\ \alpha \\
W(\Gamma \vdash \texttt{let}\ x = a\ \texttt{in}\ b) & = & \texttt{do}\ S \leftarrow W(\Gamma \vdash a) \\
 & & \texttt{do}\ \sigma \leftarrow \mathrm{Gen}(\phi(S), \phi(\Gamma)) \\
 & & \texttt{return}\ W(\Gamma, x:\sigma \vdash b)
\end{array}
$$

Most General Unifier

$$
\begin{array}{rcl}
\texttt{mgu}(\varnothing) & = & \texttt{id} \\
\texttt{mgu}(\{(\alpha,\alpha)\} \cup C) & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{(\alpha,T)\} \cup C) & = & \texttt{mgu}(C[T/\alpha]) \circ [T/\alpha] \ \texttt{if}\ \alpha\ \texttt{not free in}\ T \\
\texttt{mgu}(\{(T,\alpha)\} \cup C) & = & \texttt{mgu}(C[T/\alpha]) \circ [T/\alpha] \ \texttt{if}\ \alpha\ \texttt{not free in}\ T \\
\texttt{mgu}(\{(S_1 \to S_2, T_1 \to T_2)\} \cup C) & = & \texttt{mgu}(\{(S_1,T_1),(S_2,T_2)\} \cup C)
\end{array}
$$

In all the other cases `mgu` fails

Ad-Hoc Polymorphism

Outline

Outline

Set-theoretic types

We consider the following possibly recursive types:

$$\texttt{T} ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Any} \mid \texttt{(T,T)} \mid \texttt{T} \vee \texttt{T} \mid \texttt{T \& T} \mid \texttt{not(T)} \mid \texttt{T-->T}$$

Useful for:

1. XML types
2. Precise typing of pattern matching
3. Overloaded functions
4. Mixins
5. General programming paradigms

Let us see each point more in detail

Note: henceforward I will sometimes use $\mathbf{T}_1 | \mathbf{T}_2$ to denote $\mathbf{T}_1 \vee \mathbf{T}_2$

1. XML types

```
<?xml version="1.0"?>
  <!DOCTYPE biblio [
  <!ELEMENT biblio (book*)>
  <!ELEMENT book (title, (author+)|(editor+), price?)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT editor (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
```

Can be encoded with union and recursive types

```
type Biblio = (‘biblio,X)
type      X = (Book,X)∨‘nil

type Book = (‘book,(Title, Y∨Z))
type    Y = (Author,Y∨(Price,‘nil)∨‘nil)
type    Z = (Editor,Z∨(Price,‘nil)∨‘nil)

type Title  = (‘title,String)
type Author = (‘author,String)
type Editor = (‘editor,String)
type Price  = (‘price,String)
```

2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

`match` e `with` p_1 `->` e_1 `|` p_2 `->` e_2

where patterns are defined as follows:

$$p ::= x \mid (p\,,p) \mid p|p \mid p\&p$$

2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

$$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

where patterns are defined as follows:

$$p ::= x \mid (p\,,p) \mid p|p \mid p\&p$$

If we interpret types as set of values

$$t = \{v \mid v \text{ is a value of type } t\}$$

then the set of all values that match a pattern is a type

$$\{\!\!\{p\}\!\!\} = \{v \mid v \text{ is a value that matches } p\}$$

$$\begin{array}{rcl} \{\!\!\{x\}\!\!\} & = & \texttt{Any} \\ \{\!\!\{(p_1\,,p_2)\}\!\!\} & = & (\{\!\!\{p_1\}\!\!\}\,,\{\!\!\{p_2\}\!\!\}) \\ \{\!\!\{p_1|p_2\}\!\!\} & = & \{\!\!\{p_1\}\!\!\} \vee \{\!\!\{p_2\}\!\!\} \\ \{\!\!\{p_1\&p_2\}\!\!\} & = & \{\!\!\{p_1\}\!\!\}\ \&\ \{\!\!\{p_2\}\!\!\} \end{array}$$

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

`match` e `with` p_1 `->` e_1 `|` p_2 `->` e_2

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

`match` e `with` p_1 `->` e_1 `|` p_2 `->` e_2

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

Suppose that $e : \texttt{T}$ and let us write $\texttt{T}_1 \setminus \texttt{T}_2$ for $\texttt{T}_1 \,\&\, \texttt{not}(\texttt{T}_2)$

- To infer the type $\texttt{T}_1$ of e_1 we need $\texttt{T} \,\&\, \wr p_1 \int$;

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

`match` e `with` p_1 `->` e_1 `|` p_2 `->` e_2

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \{\!\!\{ p_1 \}\!\!\}$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \setminus \{\!\!\{ p_1 \}\!\!\}) \,\&\, \{\!\!\{ p_2 \}\!\!\}$;

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

match e with p_1 -> e_1 | p_2 -> e_2

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \langle\!\langle p_1 \rangle\!\rangle$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \setminus \langle\!\langle p_1 \rangle\!\rangle) \,\&\, \langle\!\langle p_2 \rangle\!\rangle$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

```
match e with p1 -> e1 | p2 -> e2
```

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \setminus \wr p_1 \int) \,\&\, \wr p_2 \int$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leq \wr p_1 \int \vee \wr p_2 \int$;

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

```
match e with p1 -> e1 | p2 -> e2
```

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \langle\!\langle p_1 \rangle\!\rangle$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \setminus \langle\!\langle p_1 \rangle\!\rangle) \,\&\, \langle\!\langle p_2 \rangle\!\rangle$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leq \langle\!\langle p_1 \rangle\!\rangle \vee \langle\!\langle p_2 \rangle\!\rangle$;

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching:*

$\mathtt{match}\ e\ \mathtt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \setminus \wr p_1 \int) \,\&\, \wr p_2 \int$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leq \wr p_1 \int \vee \wr p_2 \int$;

Formally:

[Match]

$$\frac{\Gamma \vdash e : \mathtt{T} \qquad \Gamma, \mathtt{T} \,\&\, \wr p_1 \int / p_1 \vdash e_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{T} \setminus \wr p_1 \int / p_2 \vdash e_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ p_1 \texttt{->} e_1\ \texttt{|}\ p_1 \texttt{->} e_2 : \mathtt{T}_1 \vee \mathtt{T}_2} (\mathbf{T} \leq \wr p_1 \int \vee \wr p_2 \int)$$

where $\mathtt{T}/p$ is the type environment for the capture variables in p when the pattern is matched against values in $\mathtt{T}$.
(e.g., $((\mathtt{Int}\,,\mathtt{Int}) \vee (\mathtt{Bool}\,,\mathtt{Char}))/(x\,,y)$ is $x : \mathtt{Int} \vee \mathtt{Bool}, y : \mathtt{Int} \vee \mathtt{Char}$)

3. Overloaded functions

Intersection types are useful to type overloaded functions (in the Go language):

```
package main
import "fmt"
func Opposite (x interface{}) interface{} {
  var res interface{}
  switch value := x.(type) {
    case bool:
      res = (!value)            // x has type bool
    case int:
      res = (-value)            // x has type int
  }
  return res
}

func main() { fmt.Println(Opposite(3) , Opposite(true)) }
```

In Go `Opposite` has type `Any-->Any` (every value has type `interface{}`).

Better type with intersections `Opposite:  (Int-->Int) & (Bool-->Bool)`

3. Overloaded functions

Intersection types are useful to type overloaded functions (in the Go language):

```
package main
import "fmt"
func Opposite (x interface{}) interface{} {
  var res interface{}
  switch value := x.(type) {
    case bool:
      res = (!value)              // x has type bool
    case int:
      res = (-value)              // x has type int
  }
  return res
}

func main() { fmt.Println(Opposite(3) , Opposite(true)) }
```

In Go `Opposite` has type `Any-->Any` (every value has type `interface{}`).
Better type with intersections `Opposite: (Int-->Int) & (Bool-->Bool)`

Intersections can also to give a more refined description of standard functions:

```
func Successor(x int) { return(x+1) }
```

which could be typed as `Successor:(Odd-->Even) & (Even-->Odd)`

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

```
let foo = function
  | (‘A,‘B) -> true
  | (‘B,‘A) -> false
```

   and what is the problem ?

2. Which type could we give if we had full-fledged union types?

3. Give an intersection type that refines the previous type

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

```
let foo = function
  | (‘A,‘B) -> true
  | (‘B,‘A) -> false
```

and what is the problem ?

`[< ‘A | ‘B ] * [< ‘A | ‘B ] -> bool` thus `foo( ‘A , ‘A)` fails

2. Which type could we give if we had full-fledged union types?

3. Give an intersection type that refines the previous type

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

   ```
   let foo = function
     | (‘A,‘B) -> true
     | (‘B,‘A) -> false
   ```

   and what is the problem ?

   `[< ‘A | ‘B ] * [< ‘A | ‘B ] -> bool` thus `foo( ‘A , ‘A)` fails

2. Which type could we give if we had full-fledged union types?

   `(‘A * ‘B )| ( ‘B * ‘A)  -> bool`

3. Give an intersection type that refines the previous type

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

```
let foo = function
  | (‘A,‘B) -> true
  | (‘B,‘A) -> false
```

   and what is the problem ?

   `[< ‘A | ‘B ] * [< ‘A | ‘B ] -> bool` thus `foo( ‘A , ‘A)` fails

2. Which type could we give if we had full-fledged union types?

   `(‘A * ‘B )| ( ‘B * ‘A)  -> bool`

3. Give an intersection type that refines the previous type

   `((‘A * ‘B ) -> true) & (( ‘B * ‘A)  -> false)`

You can try it on `http://www.cduce.org/ocaml/bi`

4. Typing of Mixins

Intersection types are used in Microsoft's Typescript to type mixins.

```
function extend<T, U>(first: T, second: U): T & U {
    /* <T> exp is a type cast (equivalent: exp as T) */
    let result = <T & U>{};
    for (let id in first) {
            (<any>result)[id] = (<any>first)[id]; }
    for (let id in second) { if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id]; } }
    return result;
}
class Person {
    constructor(public name: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() { ... }
}

var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

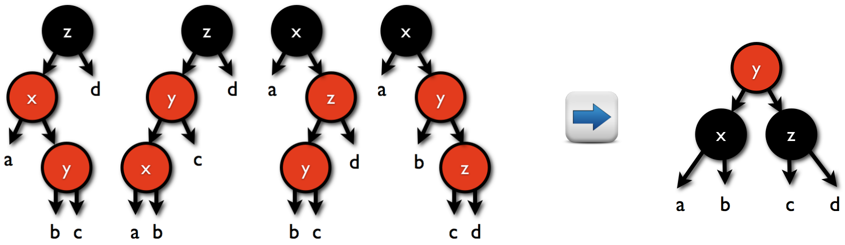
1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key of Okasaki's insertion is the function `balance` which transforms an *unbalanced tree*, into a *valid red-black tree* (as long as a, b, c, and d are valid):

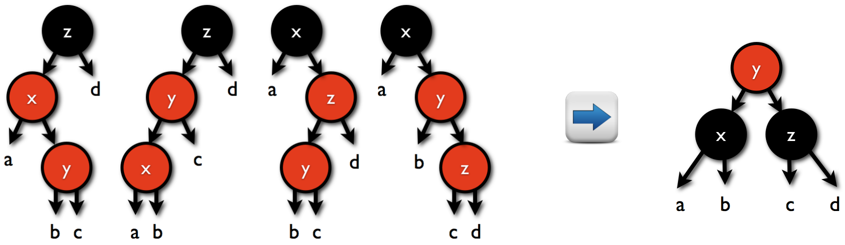


5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key of Okasaki's insertion is the function `balance` which transforms an *unbalanced tree*, into a *valid red-black tree* (as long as a, b, c, and d are valid):



In ML we need GADTs to enforce the invariants.

```
type α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

① Write the correct definitions

```
type αRBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

```
type α RBtree =
    | Leaf
    | Red( α , RBtree , RBtree)
    | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

① Write the correct definitions

① Write the correct definitions
② Add Type annotations to Function definitions

```
type α RBtree =
  | Leaf
  | Red( α , RBtree , RBtree)
  | Blk( α , RBtree , RBtree)

let balance =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
 function ( x , t ) ->
  let ins =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree)|(RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree)|(RBtree,Wrong) )

let balance: (Unbal→Rtree) & ( (β\Unbal)→(β\Unbal) ) =
 function
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree)→Btree =
 function ( x , t ) ->
  let ins: (Leaf→Rtree) & (Btree→RBtree\Leaf) & (Rtree→Rtree|Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins: (Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

Constraints are respected

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
     -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins: (Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

Result of insert satisfies constraints statically by typing

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins: (Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
         if x < y  then balance c( y, (ins a), b ) else
         if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

Use of overloading and full-fledged set-theoretic types

```
type RBtree = Btree | Rtree
type Rtree  = Red(α, Btree , Btree )
type Btree  = Blk(α, RBtree, RBtree) | Leaf

type Wrong = Red( α, (Rtree,RBtree) | (RBtree,Rtree) )
type Unbal = Blk( α, (Wrong,RBtree) | (RBtree,Wrong) )

let balance: (Unbal → Rtree) & ( (β\Unbal) → (β\Unbal) ) =
 function
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert: (α, Btree) → Btree =
 function ( x , t ) ->
  let ins: (Leaf → Rtree) & (Btree → RBtree\Leaf) & (Rtree → Rtree|Wrong) =
   function
     | Leaf -> Red(x,Leaf,Leaf)
     | c(y,a,b) as z ->
        if x < y  then balance c( y, (ins a), b ) else
        if x > y  then balance c( y, a, (ins b) ) else z
  in let _(y,a,b) = ins t in Blk(y,a,b)
```

A form of bounded polymorphism

$\forall(\alpha \leq \neg\text{Unbal}).\alpha \to \alpha$

Cutting edge research

Type checking the previous definitions is not so difficult.
The hard part is to type partial applications:

$$\texttt{map} : (\alpha \to \beta) \to [\ \alpha\ ] \to [\ \beta\ ]$$

$$\texttt{balance} : (\texttt{Unbal} \to \texttt{Rtree})\ \&\ ((\beta \backslash \texttt{Unbal}) \to (\beta \backslash \texttt{Unbal}))$$

$$\begin{aligned}\texttt{map balance} : & ([\ \texttt{Unbal}\ ] \to [\ \texttt{Rtree}\ ])\\ \&\ & ([\ \alpha \backslash \texttt{Unbal}\ ] \to [\ \alpha \backslash \texttt{Unbal}\ ])\\ \&\ & ([\ \alpha | \texttt{Unbal}\ ] \to [(\alpha \backslash \texttt{Unbal}) | \texttt{Rtree}\ ])\end{aligned}$$

Fortunately, programmers (and you) are spared from these gory details.

New languages use union and intersections

Facebook's Flow:

```
// @flow
function toStringPrimitives(val: number | boolean | string) {
  return String(val);
}



type One = { foo: number };
type Two = { bar: boolean };

type Both = One & Two;

var value: Both = {
  foo: 1,
  bar: true
};
```

New languages use union and intersections

Typed-Racket

```
(let ([a-number 37])
    (if (even? a-number)
        'yes
        'no))
- : Symbol [more precisely: (U 'no 'yes)]
'no
```

```
(: f : (case-> (-> True Integer Integer)
               (-> False Boolean Boolean)))
  (define (f condition x)
    (if condition
       (add1 x)
       (not x)))
```

New languages using negation

Typescript

Negation types are proposed in a merge request for TypeScript:

```
function asValid<T extends not null>
  (value: T, isValid: (value: T) => boolean) : T | null
    return isValid(value) ? value : null;


declare const x: number;
declare const y: number | null;
asValid(x, n => n >= 0);    // OK
asValid(y, n => n >= 0);    // Error
```

Full-fledged connectives for novel type expressivity

The recursive `flatten` function:

Full-fledged connectives for novel type expressivity

The recursive `flatten` function:

```
let flatten
  | [] -> []
  | [h ; t] -> (flatten h)@(flatten t)
  | x -> [x]
```

Full-fledged connectives for novel type expressivity

The recursive `flatten` function:

```
(* recursive type with union intersection and negation *)

 type Tree('a) = ('a\[Any*]) | [ (Tree('a))* ]

 let flatten ( (Tree('a)) -> ['a*] )
   | [] -> []
   | [h ; t] -> (flatten h)@(flatten t)
   | x -> [x]
```

Full-fledged connectives for novel type expressivity

The recursive `flatten` function:

```
(* recursive type with union intersection and negation *)
 type Tree('a) = ('a\[Any*]) | [ (Tree('a))* ]

 let flatten ( (Tree('a)) -> ['a*] )
   | [] -> []
   | [h ; t] -> (flatten h)@(flatten t)
   | x -> [x]
```

The function `flatten` can be applied to any expression since `Tree('a)` unifies with every type.
It returns a list whose element type is the union of the types of all the leaves:

```
# flatten [ 3 'r' [4 [`true 5]] [ "quo" [[`false] "stop"] ] ];;
- : [ (Bool | 3--5 | 'o'--'u')* ]
  = [ 3 'r' 4 true 5 'quo' false 'stop' ]
```

Encoding of bounded polymorphism

When combined with polymorphic types, set-theoretic types can encode a limited form of bounded polymorphism:

$$\forall(\mathtt{T}_1 \leq \alpha \leq \mathtt{T}_2).\mathtt{T}$$

is encoded as

$$\mathtt{T}\{\alpha := (\alpha \vee \mathtt{T}_1) \wedge \mathtt{T}_2\}$$

For instance:

$$\mathtt{balance}\ :\ \ (\mathtt{Unbal} \to \mathtt{Rtree})\ \&\ (\beta\backslash\mathtt{Unbal} \to \beta\backslash\mathtt{Unbal})$$

can be read as:

$$\mathtt{balance}\ : \forall(\beta \leq \mathtt{not(Unbal)})\,.\,(\mathtt{Unbal} \to \mathtt{Rtree})\ \&\ (\beta \to \beta)$$

Limited form since you can compare just types with equal bounds

How to understand/explain set-theoretic type connectives?

- The type connectives union, intersection, and negation are completely defined by the subtyping relation:
  - $\mathbf{T}_1 \vee \mathbf{T}_2$ is the least upper bound of $\mathbf{T}_1$ and $\mathbf{T}_2$
  - $\mathbf{T}_1 \,\&\, \mathbf{T}_2$ is the greatest lower bound of $\mathbf{T}_1$ and $\mathbf{T}_2$
  - **not(*T*)** is the only type whose union and intersection with **T** yield the **Any** and **Empty** types, respectively.
- Defining (and deciding) subtyping for *type connectives* (i.e., $\vee$, &, not()) is far more difficult than for *type constructors* (i.e., -->, $\times$, $\{...\}$, $\ldots$). [examples later on]
- Understanding connectives in terms of subtyping is out of reach of simple programmers

How to understand/explain set-theoretic type connectives?

- The type connectives union, intersection, and negation are completely defined by the subtyping relation:
  - $\mathbf{T}_1 \vee \mathbf{T}_2$ is the least upper bound of $\mathbf{T}_1$ and $\mathbf{T}_2$
  - $\mathbf{T}_1 \mathbin{\&} \mathbf{T}_2$ is the greatest lower bound of $\mathbf{T}_1$ and $\mathbf{T}_2$
  - **not**(T) is the only type whose union and intersection with **T** yield the **Any** and **Empty** types, respectively.
- Defining (and deciding) subtyping for *type connectives* (i.e., $\vee$, &, not()) is far more difficult than for *type constructors* (i.e., -->, $\times$, $\{...\}, \ldots$). [examples later on]
- Understanding connectives in terms of subtyping is out of reach of simple programmers

Give a set-theoretic semantics to types
define subtyping semantically

Types as sets of values and semantic subtyping

$\mathtt{T} ::= \mathtt{Bool} \mid \mathtt{Int} \mid \mathtt{Any} \mid (\mathtt{T}\,,\mathtt{T}) \mid \mathtt{T} \vee \mathtt{T} \mid \mathtt{T}\,\&\,\mathtt{T} \mid \mathtt{not(T)} \mid \mathtt{T\text{-->}T}$

Each type *denotes* a set of values:

$\mathtt{Bool}$ is the set that contains just two values $\{\mathtt{true}\,,\mathtt{false}\}$

$\mathtt{Int}$ is the set of all the numeric constants: $\{\mathtt{0}\,,\,\mathtt{-1}\,,\,\mathtt{1}\,,\,\mathtt{-2}\,,\,\mathtt{2}\,,\,\mathtt{-3}\,,\,\ldots\}$.

$\mathtt{Any}$ is the set of *all* values.

$(\mathtt{T}_1\,,\mathtt{T}_2)$ is the set of all the pairs (v_1, v_2) where v_1 is a value in $\mathtt{T}_1$ and v_2 a value in $\mathtt{T}_2$, that is $\{(v_1, v_2) \mid v_1 \in \mathtt{T}_1\,,\, v_2 \in \mathtt{T}_2\}$.

$\mathtt{T}_1 \vee \mathtt{T}_2$ is the *union* of the sets $\mathtt{T}_1$ and $\mathtt{T}_2$, that is $\{v \mid v \in \mathtt{T}_1 \text{ or } v \in \mathtt{T}_2\}$

$\mathtt{T}_1\,\&\,\mathtt{T}_2$ is the *intersection* of the sets $\mathtt{T}_1$ and $\mathtt{T}_2$, i.e. $\{v \mid v \in \mathtt{T}_1 \text{ and } v \in \mathtt{T}_2\}$.

$\mathtt{not(T)}$ is the set of all the values not in $\mathtt{T}$, that is $\{v \mid v \notin \mathtt{T}\}$.
In particular $\mathtt{not(Any)}$ is the empty set (written $\mathtt{Empty}$).

$\mathtt{T}_1\mathtt{\text{-->}}\mathtt{T}_2$ is the set of all function values that when applied to a value in $\mathtt{T}_1$, if they return a value, then this value is in $\mathtt{T}_2$.

Types as sets of values and semantic subtyping

$T ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Any} \mid (T\,,T) \mid T \vee T \mid T \,\&\, T \mid \texttt{not}(T) \mid T\texttt{-->}T$

Each type *denotes* a set of values:

`Bool` is the set that contains just two values $\{\texttt{true}, \texttt{false}\}$

`Int` is the set of all the numeric constants: $\{0, -1, 1, -2, 2, -3, \ldots\}$.

`Any` is the set of *all* values.

$(T_1\,, T_2)$ is the set of all the pairs (v_1, v_2) where v_1 is a value in T_1 and v_2 a value in T_2, that is $\{(v_1, v_2) \mid v_1 \in T_1\,,\, v_2 \in T_2\}$.

$T_1 \vee T_2$ is the *union* of the sets T_1 and T_2, that is $\{v \mid v \in T_1 \text{ or } v \in T_2\}$

$T_1 \,\&\, T_2$ is the *intersection* of the sets T_1 and T_2, i.e. $\{v \mid v \in T_1 \text{ and } v \in T_2\}$.

$\texttt{not}(T)$ is the set of all the values not in T, that is $\{v \mid v \notin T\}$.
In particular `not(Any)` is the empty set (written `Empty`).

$T_1\texttt{-->}T_2$ is the set of all function values that when applied to a value in T_1, if they return a value, then this value is in T_2.

Semantic subtyping

Subtyping is set-containment

Semantic Subtyping in a nutshell

Semantic subtyping

$t ::= B \mid t{\times}t \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$

Semantic subtyping

$t ::= B \mid t{\times}t \mid t{\rightarrow}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$

- Constructor subtyping is *easy*:
  constructors do not mix, *eg.*:

$$\frac{s_2 \leq s_1 \qquad t_1 \leq t_2}{s_1{\rightarrow}t_1 \leq s_2{\rightarrow}t_2}$$

Semantic subtyping

$t ::= B \mid t\times t \mid t\rightarrow t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$

- Constructor subtyping is *easy*:
  constructors do not mix, *eg.*:

$$\frac{s_2 \leq s_1 \qquad t_1 \leq t_2}{s_1\rightarrow t_1 \leq s_2\rightarrow t_2}$$

- Connective subtyping is *harder*:
  connectives distribute over *constructors*, *eg.*

$$(s_1\vee s_2)\rightarrow t \quad \gtreqless \quad (s_1\rightarrow t)\wedge(s_2\rightarrow t)$$

Semantic subtyping

$$t ::= B \mid t{\times}t \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

- Constructor subtyping is *easy*:
  constructors do not mix, *eg.*:

$$\frac{s_2 \leq s_1 \qquad t_1 \leq t_2}{s_1{\to}t_1 \leq s_2{\to}t_2}$$

- Connective subtyping is *harder*:
  connectives distribute over *constructors*, *eg.*

$$(s_1{\vee}s_2){\to}t \quad \gtreqless \quad (s_1{\to}t){\wedge}(s_2{\to}t)$$

Define subtyping semantically: [Hosoya, Pierce]

1. Interpret types as sets (of values)
2. *Define* subtyping as set containment.

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] \quad = \quad [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] \quad = \quad \{f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!]\}$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$\begin{array}{ll} [\![\mathbb{0}]\!] = \varnothing & [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \\ [\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] & [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!] \end{array}$$

- **Constructors** have their natural interpretation:

$$\begin{array}{lcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] & = & \{f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!]\} \end{array}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \quad \stackrel{def}{\Longleftrightarrow} \quad [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad \mathcal{D}^2 \subseteq \mathcal{D}$$
$$[\![t_1 \to t_2]\!] = \{ f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!] \} \qquad \mathcal{D}^{\mathcal{D}} \subseteq \mathcal{D}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \stackrel{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1$$

- **Constructors** have their natura

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$

$$[\![t_1 \to t_2]\!] = \{ f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!] \}$$

cardinality problem

$\mathcal{D}^{\mathcal{D}} \subseteq \mathcal{D}$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$[\![\mathbb{0}]\!] = \varnothing$ $\quad$ $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$

$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!]$ $\quad$ $[\![t_1 \wedge t_2]\!] = [\![t_1$

- **Constructors** have their natura

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \{ f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!] \}$$

cardinality problem

$\mathcal{D}^{\mathcal{D}} \subseteq \mathcal{D}$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*
define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \{ f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!] \}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*
define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \{ f \subseteq \mathcal{D}^2 \mid (d_1, d_2) \in f, d_1 \in [\![t_1]\!] \Rightarrow d_2 \in [\![t_2]\!] \}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*
define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*
define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have **their natural interpretation**:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*

define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\;]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

  such that

  - **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

  - **Constructors** have **the same** $\subseteq$ **as** their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \stackrel{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*

define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have **the same $\subseteq$ as** their natural interpretation:

$$[\![s_1 \times s_2]\!] \subseteq [\![t_1 \times t_2]\!] \iff \underline{[\![s_1]\!] \times [\![s_2]\!] \subseteq [\![t_1]\!] \times [\![t_2]\!]}$$
$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types *are*
define *how they are related*

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have **the same** $\subseteq$ **as** their natural interpretation:

$$[\![s_1 \times s_2]\!] \subseteq [\![t_1 \times t_2]\!] \iff [\![s_1]\!] \times [\![s_2]\!] \subseteq [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \overline{\mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}})} \subseteq \overline{\mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping [Benzaken, Castagna, Frisch]

1. Gives an interpretation satisfying the above constraints;
2. Gives an algorithm to decide the induced subtyping relation.

1: An interpretation that satisfies the previous constraints.

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1{\to}s_2]\!] \subseteq [\![t_1{\to}t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\,]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $[\![\,]\!]_{\mathcal{D}}$ is defined as:

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $[\![\ ]\!]_{\mathcal{D}}$ is defined as:

$$[\![\mathbb{0}]\!]_{\mathcal{D}} = \varnothing \qquad [\![\mathbb{1}]\!]_{\mathcal{D}} = \mathcal{D} \qquad [\![\neg t]\!]_{\mathcal{D}} = \mathcal{D} \setminus [\![t]\!]_{\mathcal{D}}$$

$$[\![s \vee t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cup [\![t]\!]_{\mathcal{D}} \qquad [\![s \wedge t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cap [\![t]\!]_{\mathcal{D}}$$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $[\![\ ]\!]_{\mathcal{D}}$ is defined as:

$$[\![\mathbb{0}]\!]_{\mathcal{D}} = \varnothing \qquad [\![\mathbb{1}]\!]_{\mathcal{D}} = \mathcal{D} \qquad [\![\neg t]\!]_{\mathcal{D}} = \mathcal{D} \setminus [\![t]\!]_{\mathcal{D}}$$

$$[\![s \vee t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cup [\![t]\!]_{\mathcal{D}} \qquad [\![s \wedge t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cap [\![t]\!]_{\mathcal{D}}$$

$$[\![s \times t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \times [\![t]\!]_{\mathcal{D}} \qquad [\![t \to s]\!]_{\mathcal{D}} = \mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{D}} \times \overline{[\![s]\!]_{\mathcal{D}}}})$$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $[\![\ ]\!]_{\mathcal{D}}$ is defined as:

$$[\![\mathbb{0}]\!]_{\mathcal{D}} = \varnothing \qquad [\![\mathbb{1}]\!]_{\mathcal{D}} = \mathcal{D} \qquad [\![\neg t]\!]_{\mathcal{D}} = \mathcal{D} \backslash [\![t]\!]_{\mathcal{D}}$$

$$[\![s \vee t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cup [\![t]\!]_{\mathcal{D}} \qquad [\![s \wedge t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cap [\![t]\!]_{\mathcal{D}}$$

$$[\![s \times t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \times [\![t]\!]_{\mathcal{D}} \qquad [\![t \to s]\!]_{\mathcal{D}} = \mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{D}} \times \overline{[\![s]\!]_{\mathcal{D}}}})$$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1{\to}s_2]\!] \subseteq [\![t_1{\to}t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $[\![\ ]\!]_{\mathcal{D}}$ is defined as:

$$[\![\mathbb{0}]\!]_{\mathcal{D}} = \varnothing \qquad [\![\mathbb{1}]\!]_{\mathcal{D}} = \mathcal{D} \qquad [\![\neg t]\!]_{\mathcal{D}} = \mathcal{D}\setminus[\![t]\!]_{\mathcal{D}}$$

$$[\![s{\vee}t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cup [\![t]\!]_{\mathcal{D}} \qquad [\![s{\wedge}t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cap [\![t]\!]_{\mathcal{D}}$$

$$[\![s{\times}t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \times [\![t]\!]_{\mathcal{D}} \qquad [\![t{\to}s]\!]_{\mathcal{D}} = \mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{D}} \times \overline{[\![s]\!]_{\mathcal{D}}}})$$

It is a model:

$$\mathcal{P}_f(X) \subseteq \mathcal{P}_f(Y) \iff X \subseteq Y \iff \mathcal{P}(X) \subseteq \mathcal{P}(Y)$$

1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\![\ ]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ such that:

$$[\![s_1{\to}s_2]\!] \subseteq [\![t_1{\to}t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

1. $\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $[\![\ ]\!]_{\mathcal{D}}$ is defined as:

$$[\![\mathbb{0}]\!]_{\mathcal{D}} = \varnothing \qquad [\![\mathbb{1}]\!]_{\mathcal{D}} = \mathcal{D} \qquad [\![\neg t]\!]_{\mathcal{D}} = \mathcal{D} \setminus [\![t]\!]_{\mathcal{D}}$$

$$[\![s{\vee}t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cup [\![t]\!]_{\mathcal{D}} \qquad [\![s{\wedge}t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \cap [\![t]\!]_{\mathcal{D}}$$

$$[\![s{\times}t]\!]_{\mathcal{D}} = [\![s]\!]_{\mathcal{D}} \times [\![t]\!]_{\mathcal{D}} \qquad [\![t{\to}s]\!]_{\mathcal{D}} = \mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{D}} \times \overline{[\![s]\!]_{\mathcal{D}}}})$$

It is a model:

$$\mathcal{P}_f(X) \subseteq \mathcal{P}_f(Y) \iff X \subseteq Y \iff \mathcal{P}(X) \subseteq \mathcal{P}(Y)$$

It is the **best** model: for any other model $[\![\ ]\!]_{\mathcal{D}'}$

$$t_1 \leq_{\mathcal{D}'} t_2 \quad \Rightarrow \quad t_1 \leq_{\mathcal{D}} t_2$$

2: An algorithm to decide $t_1 \leq t_2$.

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!] \Leftrightarrow [\![t_1 \wedge \neg t_2]\!] = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

2: An algorithm to decide $t_1 \leq t_2$.

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!] \Leftrightarrow [\![t_1 \wedge \neg t_2]\!] = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

Step 2: ***Put the type whose emptiness is to be decided in disjunctive normal form.***

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

where $a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1}$ and $\ell ::= a \mid \neg a$

2: An algorithm to decide $t_1 \leq t_2$.

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!] \Leftrightarrow [\![t_1 \wedge \neg t_2]\!] = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

Step 2: ***Put the type whose emptiness is to be decided in disjunctive normal form.***

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

where $a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1}$ and $\ell ::= a \mid \neg a$

Step 3: ***Simplify mixed intersections:***

Mixed summands of the union can be simplified. For instance:

- $(t_1 \times t_2) \wedge (t_1 \to t_2) \leq \mathbb{0}$ is always true
- $(t_1 \times t_2) \wedge \neg(t_1 \to t_2) \leq \mathbb{0}$ holds iff $t_1 \times t_2 \leq \mathbb{0}$.

2: An algorithm to decide $t_1 \leq t_2$.

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!] \Leftrightarrow [\![t_1 \wedge \neg t_2]\!] = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

Step 2: ***Put the type whose emptiness is to be decided in disjunctive normal form.***

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

where $a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1}$ and $\ell ::= a \mid \neg a$

Step 3: ***Simplify mixed intersections:***

Mixed summands of the union can be simplified. For instance:

- $(t_1 \times t_2) \wedge (t_1 \to t_2) \leq \mathbb{0}$ is always true
- $(t_1 \times t_2) \wedge \neg(t_1 \to t_2) \leq \mathbb{0}$ holds iff $t_1 \times t_2 \leq \mathbb{0}$.

The problem is reduced to deciding:

$$\bigwedge_{i \in I} s_i \times t_i \bigwedge_{j \in J} \neg(s_j \times t_j) \leq \mathbb{0} \quad \text{and} \quad \bigwedge_{i \in I} s_i \to t_i \bigwedge_{j \in J} \neg(s_j \to t_j) \leq \mathbb{0}$$

(similarly for basic types)

Step 4: ***Use the set-theoretic interpretation to simplify the intersections:***

Decomposition law for products:

$$\bigwedge_{i\in I} t_i{\times}s_i \leq \bigvee_{i\in J} t_i{\times}s_i \iff \forall J' \subset J.\ \left(\bigwedge_{i\in I} t_i \leq \bigvee_{i\in J'} t_i\right) \text{ or } \left(\bigwedge_{i\in I} s_i \leq \bigvee_{i\in J\setminus J'} s_i\right)$$

Decomposition law for arrows:

$$\bigwedge_{i\in I} t_i{\to}s_i \leq \bigvee_{i\in J} t_i{\to}s_i \iff \exists j \in J. \forall I' \subset I.\ \left(t_j \leq \bigvee_{i\in I'} t_i\right) \text{ or } \left(I' \neq I \text{ et } \bigwedge_{i\in I\setminus I'} s_i \leq s_j\right)$$

Step 5: ***Memoize (for recursive types) and recurse.***

Application to a language.

Language

Syntax

$$
\begin{array}{llll}
\textbf{Exprs} & e ::= & x & \text{variables} \\
 & \mid & \lambda^{\wedge_{i\in I} s_i \to t_i} x.e & \text{abstractions} \\
 & \mid & ee & \text{applications} \\
 & \mid & (e,e) & \text{pairs} \\
 & \mid & \pi_i e & \text{projections, } i = 1,2 \\
 & \mid & (x = e \in t)\,?\,e : e & \text{binding type case}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{Values} & v ::= & (v,v) \\
 & \mid & \lambda^{\wedge_{i\in I} s_i \to t_i} x.e
\end{array}
$$

Language

Syntax

$$
\begin{array}{llll}
\textbf{Exprs} & e ::= & x & \text{variables} \\
& \mid & \lambda^{\wedge_{i\in I}s_i\to t_i}x.e & \text{abstractions} \\
& \mid & ee & \text{applications} \\
& \mid & (e,e) & \text{pairs} \\
& \mid & \pi_i e & \text{projections, } i=1,2 \\
& \mid & (x=e\in t)\,?\,e:e & \text{binding type case}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{Values} & v ::= & (v,v) \\
& \mid & \lambda^{\wedge_{i\in I}s_i\to t_i}x.e
\end{array}
$$

Semantics

$$
\begin{array}{rcll}
(\lambda^{\wedge_{i\in I}s_i\to t_i}x.e)v & \longrightarrow & e[v/x] & \\
\pi_i(v_1,v_2) & \longrightarrow & v_i & i=1,2 \\
(x=v\in t)\,?\,e_1:e_2 & \longrightarrow & e_1[v/x] & v\in t \\
(x=v\in t)\,?\,e_1:e_2 & \longrightarrow & e_2[v/x] & v\notin t
\end{array}
$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i{\in}I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i{\in}I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i{\in}I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i{\in}I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\textsc{TypeCase}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma, x : s_1 \vdash e_1 : t_1 \qquad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)\,?\,e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i} \quad \begin{array}{l} s_1 \equiv t_0 \wedge t \\ s_2 \equiv t_0 \wedge \neg t \end{array}$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i \in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\textsc{TypeCase}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma, x : s_1 \vdash e_1 : t_1 \qquad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)\,?\,e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i} \qquad \begin{array}{l} s_1 \equiv t_0 \wedge t \\ s_2 \equiv t_0 \wedge \neg t \end{array}$$

Typing

$$[\text{SUBSUMPTION}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\text{APP}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\text{ABS}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\text{SEL}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\text{PAIR}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\text{TYPECASE}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma, x : s_1 \vdash e_1 : t_1 \qquad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)\,?\,e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i} \qquad \begin{array}{l} s_1 \equiv t_0 \wedge t \\ s_2 \equiv t_0 \wedge \neg t \end{array}$$

A form of occurrence typing

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i \in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\textsc{TypeCase}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma, x : s_1 \vdash e_1 : t_1 \qquad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i} \quad \begin{array}{l} s_1 \equiv t_0 \wedge t \\ s_2 \equiv t_0 \wedge \neg t \end{array}$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\textsc{TypeCase}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma, x : s_1 \vdash e_1 : t_1 \qquad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \not\simeq \mathbb{0}\}} t_i} \quad \begin{array}{l} s_1 \equiv t_0 \wedge t \\ s_2 \equiv t_0 \wedge \neg t \end{array}$$

Necessary for typing overloaded functions:

$$\lambda^{(\mathrm{Int}\to\mathrm{Int})\wedge(\mathrm{Bool}\to\mathrm{Bool})} x.(y = x \in \mathrm{Int})?(y+1) : \mathrm{not}(y)$$

Typing

$$[\textsc{Subsumption}]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$[\textsc{App}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\textsc{Abs}]\ \frac{\forall i{\in}I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : \bigwedge_{i\in I} s_i \to t_i}$$

$$[\textsc{Sel}]\ \frac{\Gamma \vdash e : (t_1\ ,\ t_2)}{\Gamma \vdash \pi_i e : t_i} \qquad [\textsc{Pair}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

$$[\textsc{TypeCase}]\ \frac{\Gamma \vdash e : t_0 \qquad \Gamma, x : s_1 \vdash e_1 : t_1 \qquad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)\,?\,e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i} \quad \begin{array}{l} s_1 \equiv t_0 \wedge t \\ s_2 \equiv t_0 \wedge \neg t \end{array}$$

The type system is sound

Back to the initial example

```
function double (x) {
   (typeof(x) === "number") ?  2*x  :  x.concat(x)
}
```

Back to the initial example

```
function double (x) {
   (typeof(x) === "number") ?  2*x  :  x.concat(x)
}
```

$$\lambda^{\mathbf{t}} x.(y = x \in \mathtt{Int})\,\mathbf{?}\,(2 * y) : (y.concat(y)) \tag{1}$$

Back to the initial example

```
function double (x) {
    (typeof(x) === "number") ? 2*x : x.concat(x)
}
```

$$\lambda^{\mathbf{t}}x.(y = x \in \mathtt{Int})\,\mathbf{?}\,(2 * y) : (y.concat(y)) \qquad (1)$$

Exercise

Use the previous rules to check that (1) is well-typed for:

- $\mathbf{t} = (\mathtt{Int} \vee \mathtt{String}) \to (\mathtt{Int} \vee \mathtt{String})$
- $\mathbf{t} = (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{String} \to \mathtt{String})$

where $\mathtt{String} = \mu X.\{\text{concat} : X \to X\}$

Closing the circle

What about the interpretation of types as set of "values"?

Closing the circle

What about the interpretation of types as set of "values"?

I interpreted types into subsets of $\mathcal{D}$ rather than into sets of:

$$\textbf{Values} \qquad v ::= (v,v) \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e$$

Closing the circle

What about the interpretation of types as set of "values"?

I interpreted types into subsets of $\mathcal{D}$ rather than into sets of:

$$\textbf{Values} \qquad v \quad ::= \quad (v, v) \ \mid\ \lambda^{\wedge_{i \in I} s_i \to t_i} x.e$$

Define a new interpretation of types:

$$[\![t]\!]_{\mathcal{V}} = \{ v \mid\ \vdash v : t \}$$

Closing the circle

What about the interpretation of types as set of "values"?

I interpreted types into subsets of $\mathcal{D}$ rather than into sets of:

$$\textbf{Values} \qquad v \quad ::= \quad (v, v) \mid \lambda^{\wedge_{i \in I} s_i \to t_i} x.e$$

Define a new interpretation of types:

$$[\![t]\!]_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

This induces a new subtyping relation:

$$t \leq_{\mathcal{V}} s \quad \stackrel{def}{\Longleftrightarrow} \quad [\![t]\!]_{\mathcal{V}} \subset [\![s]\!]_{\mathcal{V}}$$

Closing the circle

What about the interpretation of types as set of "values"?

I interpreted types into subsets of $\mathcal{D}$ rather than into sets of:

$$\textbf{Values} \quad v \ ::= \ (v, v) \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e$$

Define a new interpretation of types:

$$[\![t]\!]_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

This induces a new subtyping relation:

$$t \leq_{\mathcal{V}} s \quad \overset{def}{\iff} \quad [\![t]\!]_{\mathcal{V}} \subset [\![s]\!]_{\mathcal{V}}$$

Actually, it is not a new one ... it is the old one:

Theorem [Frisch, Castagna, Benzaken 2002&2008]

$$t \leq_{\mathcal{V}} s \quad \iff \quad t \leq_{\mathcal{D}} s$$

where $\leq_{\mathcal{D}}$ is the subtyping via $\mathcal{D}$ and used to define $\vdash v : t$

Closing the circle

Was then $\mathcal{D}$ really necessary?

Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

$$[\![t]\!]_{\mathcal{V}}$$

Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

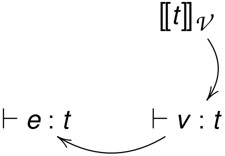
$$[\![t]\!]_{\mathcal{V}} \longrightarrow\ \vdash v : t$$

Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

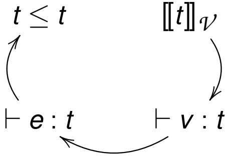
λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition



Was then $\mathcal{D}$ really necessary?

YES!

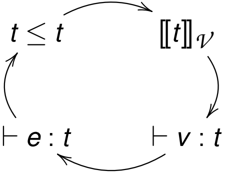
λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition



Was then $\mathcal{D}$ really necessary?

YES!

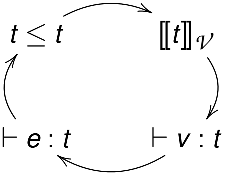
λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition



Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition



Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

$$t \leq t \qquad [\![t]\!]_{\mathcal{V}}$$

$$\vdash e : t \qquad \vdash v : t$$

Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

$$[\![t]\!]_{\mathcal{D}}$$

$$t \leq t \qquad [\![t]\!]_{\mathcal{V}}$$

$$\vdash e : t \qquad \vdash v : t$$

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

$$[\![t]\!]_{\mathcal{D}}$$

$$t \leq t \qquad [\![t]\!]_{\mathcal{V}}$$

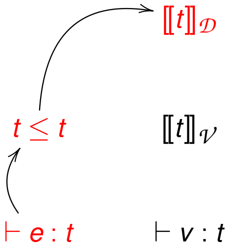
$$\vdash e : t \qquad \vdash v : t$$

Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition



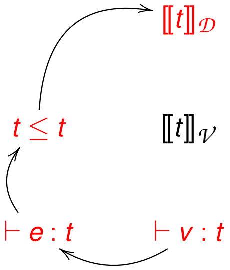
Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.

We are in a circular definition

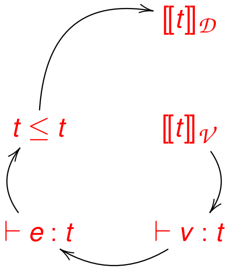


Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

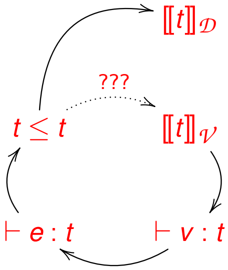


Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

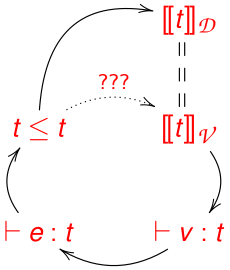


Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.
~~We are in a circular definition~~



Closing the circle

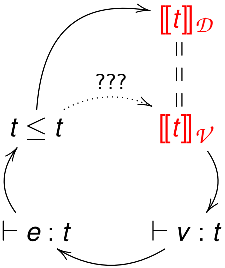
Was then $\mathcal{D}$ really necessary?

YES!

λ-abstractions are values and need (sub)typing to be defined.

~~We are in a circular definition~~

Theorem 5.5 [Frisch, Castagna, Benzaken JACM 2008]



Outline

Motivating examples: reminder 1

The recursive `flatten` function:

Motivating examples: reminder 1

The recursive `flatten` function:

```
(* recursive type with union intersection and negation *)

 type Tree(α) = (α\[Any*]) | [ (Tree(α))* ]


(* recursive flatten written in polymorphic CDuce        *)

let flatten ( (Tree(α)) -> [α*] )
   | [] -> []
   | [h ; t] -> (flatten h)@(flatten t)
   | x -> [x]
```

Motivating examples: reminder 1

The recursive `flatten` function:

```
(* recursive type with union intersection and negation *)

 type Tree(α) = (α\[Any*]) | [ (Tree(α))* ]


(* recursive flatten written in polymorphic CDuce        *)

let flatten ( (Tree(α)) -> [α*] )
   | [] -> []
   | [h ; t] -> (flatten h)@(flatten t)
   | x -> [x]
```

Rationale

The language does not changes apart from the fact that type variables such as α may occur in type annontations.

Motivating examples: reminder 2

Type refinement of `balance` for red-black trees

Motivating examples: reminder 2

Type refinement of `balance` for red-black trees

```
let balance: (Unbal→Rtree) & ((β\Unbal)→(β\Unbal)) =
 function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
      -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x
```

Naive solution

$$t ::= B \mid t\times t \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

Naive solution

$$t ::= B \mid t\times t \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

Naive solution

$$t ::= B \mid t \times t \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

Idea: Use the previous relation since is defined for "ground types"

Let σ : **Vars** $\to$ **ClosedTypes** denote ground substitutions. Define:

$$s \leq t \stackrel{def}{\iff} \forall \sigma \,.\, s\sigma \leq t\sigma$$

Naive solution

$$t ::= B \mid t{\times}t \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

Idea: Use the previous relation since is defined for “ground types”

Let σ : **Vars** $\to$ **ClosedTypes** denote ground substitutions. Define:

$$s \leq t \overset{def}{\iff} \forall\sigma\,.\, s\sigma \leq t\sigma$$

or equivalently

$$s \leq t \overset{def}{\iff} \forall\sigma.[\![s\sigma]\!] \subseteq [\![t\sigma]\!]$$

Naive solution

$$t ::= B \mid t\times t \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

Idea: Use the previous relation since is defined for "ground types"

Let σ : **Vars** $\to$ **ClosedTypes** denote ground substitutions. Define:

$$s \leq t \overset{def}{\iff} \forall\sigma.\, s\sigma \leq t\sigma$$

or equivalently

$$s \leq t \overset{def}{\iff} \forall\sigma.[\![s\sigma]\!] \subseteq [\![t\sigma]\!]$$

THIS IS A WRONG WAY: TOO MANY PROBLEMS

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma.\ s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It *breaks* parametricity:**

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma.\ s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It *breaks* parametricity:**

$$(t\times\alpha) \ \leq\ (t\times\neg t)\vee(\alpha\times t) \qquad (2)$$

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma.\, s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It *breaks* parametricity:**

$$(t\times\alpha) \;\leq\; (t\times\neg t)\vee(\alpha\times t) \qquad (2)$$

This inclusion holds if and only if t is an *indivisible* type (*eg*., a singleton or a basic type):

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\; s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It *breaks* parametricity:**

$$(t\times\alpha) \;\leq\; (t\times\neg t)\vee(\alpha\times t) \tag{2}$$

This inclusion holds if and only if t is an *indivisible* type (*eg*., a singleton or a basic type):

Property of indivisible types

If t is an *indivisible type*, then for all possible interpretations of α

$$t\leq\alpha \quad\text{or}\quad \alpha\leq\neg t$$

holds.

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\; s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It *breaks* parametricity:**

$$(t\times\alpha) \;\; \leq \;\; (t\times\neg t)\vee(\alpha\times t) \qquad (2)$$

This inclusion holds if and only if t is an *indivisible* type (*eg*., a singleton or a basic type):

> **Property of indivisible types**
>
> If t is an *indivisible type*, then for all possible interpretations of α
>
> $$t\leq\alpha \quad \text{or} \quad \alpha\leq\neg t$$
>
> holds.

- If $\alpha\leq\neg t$ then the left element of the union in (2) suffices;
- If $t\leq\alpha$, then $\alpha=(\alpha\setminus t)\vee t$. Thus $(t\times\alpha)=(t\times(\alpha\setminus t))\vee(t\times t)$. This union is contained component-wise in the one in (2).

Problems with the naive solution

The fact that

$$(t\times\alpha) \;\leq\; (t\times\neg t)\vee(\alpha\times t)$$

holds if and only if t is *indivisible* is really catastrophic:

Problems with the naive solution

The fact that

$$(t \times \alpha) \;\leq\; (t \times \neg t) \vee (\alpha \times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.

Problems with the naive solution

The fact that

$$(t \times \alpha) \leq (t \times \neg t) \vee (\alpha \times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- **This subtyping relation breaks parametricity**:
  by subsumption a function generic in its first argument,
  becomes generic on its second argument.

Problems with the naive solution

The fact that

$$(t \times \alpha) \leq (t \times \neg t) \vee (\alpha \times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- **This subtyping relation breaks parametricity**:
  by subsumption a function generic in its first argument,
  becomes generic on its second argument.

- A semantic solution was deemed unfeasible (even w/o arrows)
- Problem eschewed by resorting to syntactic solutions: [Hosoya, Frisch, Castagna: POPL 05], [Vouillon: POPL 06].

Problems with the naive solution

The fact that

$$(t\times\alpha) \leq (t\times\neg t)\vee(\alpha\times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- **This subtyping relation breaks parametricity**:
  by subsumption a function generic in its first argument,
  becomes generic on its second argument.

- A semantic solution was deemed unfeasible (even w/o arrows)
- Problem eschewed by resorting to syntactic solutions: [Hosoya, Frisch, Castagna: POPL 05], [Vouillon: POPL 06].

A SEMANTIC SOLUTION IS POSSIBLE

A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly

A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly

The crux of the problem is that for an indivisible type $\boldsymbol{i}$

$$\boldsymbol{i} \leq \alpha \quad \text{or} \quad \alpha \leq \neg \boldsymbol{i}$$

validity can **stutter** from one formula to another, missing in this way the uniformity typical of parametricity

A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly

The crux of the problem is that for an indivisible type i

$$i \leq \alpha \quad \text{or} \quad \alpha \leq \neg i$$

validity can **stutter** from one formula to another, missing in this way the uniformity typical of parametricity

The *leitmotiv* of this work

A semantic characterization of models where *stuttering* is absent, should yield a subtyping relation that is:

1. Semantic
2. Intuitive for the programmer
3. Decidable

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})^{\textbf{Vars}} \to \mathcal{P}(\mathcal{D})$$

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})^{\textbf{Vars}} \to \mathcal{P}(\mathcal{D})$$

with

$$\begin{array}{lcl lcl}
[\![\alpha]\!]\eta & = & \eta(\alpha) & [\![\neg t]\!]\eta & = & \mathcal{D} \setminus [\![t]\!]\eta \\
[\![t_1 \vee t_2]\!]\eta & = & [\![t_1]\!]\eta \cup [\![t_2]\!]\eta & [\![t_1 \wedge t_2]\!]\eta & = & [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \\
[\![\mathbb{0}]\!]\eta & = & \varnothing & [\![\mathbb{1}]\!]\eta & = & \mathcal{D}
\end{array}$$

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})^{\textbf{Vars}} \to \mathcal{P}(\mathcal{D})$$

with

$$\begin{array}{lcllcl}
[\![\alpha]\!]\eta & = & \eta(\alpha) & [\![\neg t]\!]\eta & = & \mathcal{D}\backslash[\![t]\!]\eta \\
[\![t_1 \vee t_2]\!]\eta & = & [\![t_1]\!]\eta \cup [\![t_2]\!]\eta & [\![t_1 \wedge t_2]\!]\eta & = & [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \\
[\![\mathbb{0}]\!]\eta & = & \varnothing & [\![\mathbb{1}]\!]\eta & = & \mathcal{D}
\end{array}$$

and such that it satisfies:

$$[\![t_1 \to s_1]\!]\eta \subseteq [\![t_2 \to s_2]\!]\eta \iff \mathcal{P}(\overline{[\![t_1]\!]\eta \times \overline{[\![s_1]\!]\eta}}) \subseteq \mathcal{P}(\overline{[\![t_2]\!]\eta \times \overline{[\![s_2]\!]\eta}})$$

Subtyping relation

In this framework the natural definition of subtyping is

$$s \leq t \stackrel{def}{\iff} \forall \eta . [\![s]\!]\eta \subseteq [\![t]\!]\eta$$

It "just" remains to find the uniformity condition to avoid stuttering and recover parametricity.

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t{\wedge}\neg\alpha]\!]\eta{=}\varnothing \text{ or } [\![t{\wedge}\alpha]\!]\eta{=}\varnothing)$ —that is, $(t \leq \alpha \text{ or } \alpha \leq \neg t)$— holds if and only if t is empty.

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t\wedge\neg\boldsymbol{\alpha}]\!]\eta{=}\varnothing \text{ or } [\![t\wedge\boldsymbol{\alpha}]\!]\eta{=}\varnothing)$ —that is, $(t \leq \boldsymbol{\alpha}$ or $\boldsymbol{\alpha} \leq \neg t)$— holds if and only if t is empty.
- There are natural models:all models that map all non-empty types into infinite sets satisfy it [our initial intuition].

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t{\wedge}\neg\alpha]\!]\eta{=}\varnothing \text{ or } [\![t{\wedge}\alpha]\!]\eta{=}\varnothing)$ —that is, ($t \leq \alpha$ or $\alpha \leq \neg t$)— holds if and only if t is empty.
- There are natural models:all models that map all non-empty types into infinite sets satisfy it [our initial intuition].
- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t{\wedge}\neg\alpha]\!]\eta{=}\varnothing \text{ or } [\![t{\wedge}\alpha]\!]\eta{=}\varnothing)$ —that is, ($t \leq \alpha$ or $\alpha \leq \neg t$)— holds if and only if t is empty.
- There are natural models:all models that map all non-empty types into infinite sets satisfy it [our initial intuition].
- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.
- An intuitive relation: the algorithm returns intuitive results (actually, it helps to better understand twisted examples)

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t{\wedge}\neg\boldsymbol{\alpha}]\!]\eta{=}\varnothing \text{ or } [\![t{\wedge}\boldsymbol{\alpha}]\!]\eta{=}\varnothing)$ —that is, ($t \leq \boldsymbol{\alpha}$ or $\boldsymbol{\alpha} \leq \neg t$)— holds if and only if t is empty.
- There are natural models:all models that map all non-empty types into infinite sets satisfy it [our initial intuition].
- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.
- An intuitive relation: the algorithm returns intuitive results (actually, it helps to better understand twisted examples)

Examples of subtyping relations

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

or distributivity laws:

$$(\alpha \vee \beta \times \gamma) \sim (\alpha \times \gamma) \vee (\beta \times \gamma)$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

or distributivity laws:

$$(\alpha \vee \beta \times \gamma) \sim (\alpha \times \gamma) \vee (\beta \times \gamma)$$

and combining them deduce:

$$(\alpha \times \gamma \to \delta_1) \wedge (\beta \times \gamma \to \delta_2) \leq (\alpha \vee \beta \times \gamma) \to \delta_1 \vee \delta_2$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

or distributivity laws:

$$(\alpha \vee \beta \times \gamma) \sim (\alpha \times \gamma) \vee (\beta \times \gamma)$$

and combining them deduce:

$$(\alpha \times \gamma \to \delta_1) \wedge (\beta \times \gamma \to \delta_2) \leq (\alpha \vee \beta \times \gamma) \to \delta_1 \vee \delta_2$$

Of course the problematic relation never holds, whatever the t:

$$(t \times \alpha) \not\leq (t \times \neg t) \vee (\alpha \times t)$$

We can prove relevant relations on infinite types, *eg.*, for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \text{nil}$$

We can prove relevant relations on infinite types, *eg.*, for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \text{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

We can prove relevant relations on infinite types, *eg.*, for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \text{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}) \vee (\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil}))$$

We can prove relevant relations on infinite types, *eg.*, for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \text{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}) \vee (\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil}))$$

And we can prove far more complicated relations (see paper).

Subtyping algorithm

Subtyping Algorithm: $t_1 \leq t_2$

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff \forall\eta.[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \forall\eta.[\![t_1 \wedge \neg t_2]\!]\eta = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

Subtyping Algorithm: $t_1 \leq t_2$

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff \forall\eta.[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \forall\eta.[\![t_1 \wedge \neg t_2]\!]\eta = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

Step 2: ***Put the type whose emptiness is to be decided in disjunctive normal form.***

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

where $a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1} \mid \boldsymbol{\alpha}$ and $\ell ::= a \mid \neg a$

Subtyping Algorithm: $t_1 \leq t_2$

Step 1: ***Transform the subtyping problem into an emptiness decision problem:***

$$t_1 \leq t_2 \iff \forall\eta.[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \forall\eta.[\![t_1 \wedge \neg t_2]\!]\eta = \varnothing \iff t_1 \wedge \neg t_2 \leq \mathbb{0}$$

Step 2: ***Put the type whose emptiness is to be decided in disjunctive normal form.***

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

where $a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$ and $\ell ::= a \mid \neg a$

Step 3: ***Simplify mixed intersections:***

Solve: $$\bigwedge_{i \in I} a_i \bigwedge_{j \in J} \neg a'_j \bigwedge_{h \in H} \alpha_h \bigwedge_{k \in K} \neg \beta_k$$

where all a have the same toplevel constructor.

Step 4: Eliminate toplevel negative variables.,

$$\forall\eta.[\![t]\!]\eta = \varnothing \iff \forall\eta.[\![t[\neg\alpha/\alpha]]\!]\eta = \varnothing$$

so replace $\neg\beta_k$ for β_k (forall $k \in K$)

Solve: $$\bigwedge_{i\in I} a_i \bigwedge_{j\in J} \neg a'_j \bigwedge_{h\in H} \alpha_h$$

Step 4: ***Eliminate toplevel negative variables.***,

$$\forall\eta.[\![t]\!]\eta = \varnothing \iff \forall\eta.[\![t[\neg\alpha/\alpha]]\!]\eta = \varnothing$$

so replace $\neg\beta_k$ for β_k (forall $k \in K$)

Solve: $$\bigwedge_{i\in I} a_i \bigwedge_{j\in J} \neg a'_j \bigwedge_{h\in H} \alpha_h$$

Step 5: ***Eliminate toplevel variables.***

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \bigwedge_{h\in H} \alpha_h \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2$$

holds if and only if

$$\bigwedge_{t_1\times t_2\in P} t_1\sigma\times t_2\sigma \bigwedge_{h\in H} \gamma_h^1\times\gamma_h^2 \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\sigma\times t'_2\sigma$$

where $\sigma = [(\gamma_h^1\times\gamma_h^2)\vee\alpha_h\,/\,\alpha_h]_{h\in H}$ (similarly for arrows)

Step 6: ***Eliminate toplevel constructors, memoize, and recurse***.

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \quad \leq \quad \bigvee_{t_1'\times t_2'\in N} t_1'\times t_2' \tag{3}$$

Equation (3) holds if and only if for all $N'\subseteq N$,

$$\forall\eta.\left([\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1'\times t_2'\in N'} \neg t_1']\!]\eta=\varnothing \text{ or } [\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1'\times t_2'\in N\setminus N'} \neg t_2']\!]\eta=\varnothing \right)$$

Apply *convexity* to distribute the quantification over the or's:

$$\forall\eta.\left([\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1'\times t_2'\in N'} \neg t_1']\!]\eta=\varnothing \right) \text{ or } \forall\eta.\left([\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1'\times t_2'\in N\setminus N'} \neg t_2']\!]\eta=\varnothing \right)$$

Yielding the following simplification: (similarly for arrows)

$$\forall N'\subseteq N.\left(\bigwedge_{t_1\times t_2\in P} t_1 \leq \bigvee_{t_1'\times t_2'\in N'} t_1' \right) \text{ or } \left(\bigwedge_{t_1\times t_2\in P} t_2 \leq \bigvee_{t_1'\times t_2'\in N\setminus N'} t_2' \right)$$

Outline

A motivating example in Haskell [cf. typing of `balance`]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

A motivating example in Haskell [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |    _ -> x
```

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |    _ -> x
```

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→ Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument

A motivating example in Haskell (almost) [cf. typing of `balance`]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |    _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int → Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          | _ -> x
```

type case

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int → Bool) ∧ ((α \ Int) → (α \ Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          | _ -> x
```

type case

Boolean connectives

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [cf. typing of `balance`]

```
map :: (α→β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

type variables

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          | _ -> x
```

type case

Boolean connectives

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |    _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

> Common pattern for functional data structures: red-black trees balancing; ZDD operations; XML nodes modification

A motivating example in Haskell (almost) [cf. typing of `balance`]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

The combination of type-case and intersections yields statically typed dynamic overloading.

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |    _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both map and even

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |    _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature

A motivating example in Haskell (almost) [cf. typing of `balance`]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |    _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature
3. Can *deduce* the type of the partial application `map even`

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature
3. **Can *deduce* the type of the partial application `map even`**

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both map ar
2. Can *check* the types sp he signature
3. **Can *deduce* the type of the partial application map even**

Tough!

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
          | [] -> []
          | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
         | Int -> (x `mod` 2) == 0
         |   _ -> x
```

We expect **map even** to have the following type:

$$
\begin{array}{l}
([\mathtt{Int}] \to [\mathtt{Bool}]) \wedge \\
([\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]) \wedge \\
([\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}])
\end{array}
$$

A motivating example in Haskell (almost) [cf. typing of balance]

```
map :: (α → β) → [α] → [β]
map f l = case l of
          | [] -> []
          | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
         | Int -> (x `mod` 2) == 0
         |    _ -> x
```

We expect **map even** to have the following type:

$([\mathtt{Int}] \to [\mathtt{Bool}]) \wedge$	int lists are transformed into bool lists
$([\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]) \wedge$	lists w/o ints return the same type
$([\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}])$	ints in the arg. are replaced by bools

A motivating example in Haskell (almost) [cf. typing of `balance`]

```
map :: (α → β) → [α] → [β]
map f l = case l of
          | [] -> []
          | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
         | Int -> (x `mod` 2) == 0
         |   _ -> x
```

We expect **map even** to have the following type:

$([\texttt{Int}] \to [\texttt{Bool}]) \wedge$	int lists are transformed into bool lists
$([\alpha\backslash\texttt{Int}] \to [\alpha\backslash\texttt{Int}]) \wedge$	lists w/o ints return the same type
$([\alpha\vee\texttt{Int}] \to [(\alpha\backslash\texttt{Int})\vee\texttt{Bool}])$	ints in the arg. are replaced by bools

Difficult because of expansion: needs *a set of type substitutions* —rather than just one— to unify the domain and the argument types.

The rule for applications

1. In the type system:

$$\text{(APPL)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

The rule for applications

1. In the type system:

$$\textsc{(Appl)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

$$\textsc{(Appl-algorithm)}\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

1. In the type system:

$$
\textsc{(Appl)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}
$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

$$
\textsc{(Appl-algorithm)}\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad
\begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}
$$

conditions for Typeability

The rule for applications

1. In the type system:

$$(\textsc{Appl})\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

$$(\textsc{Appl-algorithm})\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

The rule for applications

1. In the type system:

$$\text{(APPL)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

$$\text{(APPL-ALGORITHM)}\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

3. Inference of type substitutions

[where $t[\sigma_i]_{i\in I} \stackrel{def}{=} \bigwedge_{i\in I} t\sigma_i$]

$$\text{(APPL-INFERENCE)}\quad \frac{\exists [\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J} \quad \Gamma \vdash_{\mathcal{I}} e_1 : t \qquad \Gamma \vdash_{\mathcal{I}} e_2 : s}{\Gamma \vdash_{\mathcal{I}} e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j\in J} \leq s[\sigma_i]_{i\in I} \to u\}} \quad \begin{array}{l} t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \\ s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J}) \end{array}$$

The rule for applications

1. In the type system:

(APPL)

$$\frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

(APPL-ALGORITHM)

$$\frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

3. Inference of type substitutions

[where $t[\sigma_i]_{i\in I} \stackrel{def}{=} \bigwedge_{i\in I} t\sigma_i$]

conditions for Typeability

(APPL-INFERENCE)

$$\frac{\exists[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J} \quad \Gamma \vdash_{\mathcal{I}} e_1 : t \qquad \Gamma \vdash_{\mathcal{I}} e_2 : s}{\Gamma \vdash_{\mathcal{I}} e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j\in J} \leq s[\sigma_i]_{i\in I} \to u\}} \quad \begin{array}{l} t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \\ s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J}) \end{array}$$

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J})$$

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \text{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let s and t be two types. A type-substitution σ is a solution for the *tallying* of (s,t) iff $s\sigma \leq t\sigma$.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let s and t be two types. A type-substitution σ is a solution for the *tallying* of (s,t) iff $s\sigma \leq t\sigma$.

Generally: let $C = \{(s_1 \leq t_1), ..., (s_n \leq t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s \leq t) \in C$.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, two sets $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of *tallying problems*:

Definition (Type tallying)

Let s and t be two types. A type-substitution σ is a solution for the *tallying* of (s,t) iff $s\sigma \leq t\sigma$.

Generally: let $C = \{(s_1 \leq t_1), \ldots, (s_n \leq t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s \leq t) \in C$.

Type tallying is decidable and a sound and complete set of solutions for every tallying problem can be effectively found in **three** simple **steps**.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\mathtt{Int} \leq \alpha), (\mathtt{Bool} \leq \alpha)\} \quad \rightsquigarrow \quad \{(\mathtt{Int} \vee \mathtt{Bool} \leq \alpha)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha{\leq}t_1$ and $\alpha{\leq}t_2$ are in C, then replace them by $\alpha \leq t_1{\wedge}t_2$;
- if $s_1{\leq}\alpha$ and $s_2{\leq}\alpha$ are in C, then replace them by $s_1{\vee}s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i{\leq}\alpha_i{\leq}t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s{\vee}\beta){\wedge}t$ with β fresh.
2. substitute $(s{\vee}\beta){\wedge}t$ for all α in the other constraints of C
3. repeat with another constraint

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1),(t_1 \leq t_2)\}$
2. $\{(\texttt{Int} \leq \alpha),(\texttt{Bool} \leq \alpha)\} \quad \rightsquigarrow \quad \{(\texttt{Int} \vee \texttt{Bool} \leq \alpha)\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. substitute $(s \vee \beta) \wedge t$ for all α in the other constraints of C
3. repeat with another constraint

Example:

1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \quad \rightsquigarrow \quad \{(s_2 \leq \mathbb{0})\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\texttt{Int} \leq \alpha), (\texttt{Bool} \leq \alpha)\} \quad \rightsquigarrow \quad \{(\texttt{Int} \vee \texttt{Bool} \leq \alpha)\}$
3. $\{(\texttt{Int} \leq \alpha_1 \leq \text{Real}), (\alpha_2 \leq \alpha_1 \wedge \texttt{Int})\}$
$\rightsquigarrow \quad \{\alpha_1 = (\texttt{Int} \vee \beta) \wedge \text{Real}), (\alpha_2 = \texttt{Int})\}$

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form $\alpha \leq t$ or $t \leq \alpha$.

Step 2: Merge constraints on the same variable.

- if $\alpha \leq t_1$ and $\alpha \leq t_2$ are in C, then replace them by $\alpha \leq t_1 \wedge t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in C, then replace them by $s_1 \vee s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. substitute $(s \vee \beta) \wedge t$ for all α in the other constraints of C
3. repeat with another constraint

At the end we have a sets of equations $\{\alpha_i = u_i \mid i \in [1..n]\}$ that (with some care) are *contractive*. By Courcelle there exists a solution, *ie*, a substitution for $\alpha_1, \ldots, \alpha_n$ into (possibly recursive regular) types $t_1, \ldots, t_n$ (in which the fresh β's are free variables).

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of even

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\mathtt{Int} \to \mathtt{Bool}) \wedge (\alpha \setminus \mathtt{Int} \to \alpha \setminus \mathtt{Int})$ is the type of even

- The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$, $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathtt{Int}\ ,\ \mathtt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \mathtt{Int}\ ,\ \alpha \setminus \mathtt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \mathtt{Int}\ ,\ (\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool} \leq \beta_1\}$;

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1]\to[\beta_1] \leq s \to \gamma$$

where $s = (\texttt{Int} \to \texttt{Bool})\wedge(\alpha\setminus\texttt{Int} \to \alpha\setminus\texttt{Int})$ is the type of even

- The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$, $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1 \leq \alpha\setminus\texttt{Int}\ ,\ \alpha\setminus\texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1 \leq \alpha\vee\texttt{Int}\ ,\ (\alpha\setminus\texttt{Int})\vee\texttt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [] \to []\}$,
  $\{\gamma = [\texttt{Int}]\to[\texttt{Bool}]\}$,
  $\{\gamma = [\alpha\setminus\texttt{Int}] \to [\alpha\setminus\texttt{Int}]\}$,
  $\{\gamma = [\alpha\vee\texttt{Int}] \to [(\alpha\setminus\texttt{Int})\vee\texttt{Bool}]\}$.

Example: map even

Start with the following tallying problem:

$$(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma$$

where $s = (\texttt{Int} \to \texttt{Bool}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$ is the type of `even`

- The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$, $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \texttt{Int}\ ,\ \texttt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \texttt{Int}\ ,\ \alpha \setminus \texttt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \texttt{Int}\ ,\ (\alpha \setminus \texttt{Int}) \vee \texttt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [\,] \to [\,]\}$,
  $\{\gamma = [\texttt{Int}] \to [\texttt{Bool}]\}$,
  $\{\gamma = [\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]\}$,
  $\{\gamma = [\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}]\}$.
- The last two are minimal and we take their intersection:
  $\{\gamma = ([\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]) \wedge ([\alpha \vee \texttt{Int}] \to [(\alpha \setminus \texttt{Int}) \vee \texttt{Bool}])\}$

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type- reconstruction for intersection types since we have *recursive types*).

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type- reconstruction for intersection types since we have *recursive types*).

Completeness: For every solution of the inference problem, our algorithm finds an equivalent or more general solution. However, this solution is not necessary the first solution found.

In a dully execution of the algorithm on `map even` the good solution is the second one.

On completeness and decidability

The algorithm produces a set of solutions that is **sound** (it finds only correct solutions) and **complete** (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type- reconstruction for intersection types since we have *recursive types*).

Completeness: For every solution of the inference problem, our algorithm finds an equivalent or more general solution. However, this solution is not necessary the first solution found.

In a dully execution of the algorithm on `map even` the good solution is the second one.

Principality: This raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist?

References

- Frisch et al: *Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types*. JACM, vol. 55, n. 4, 2008.
  Reference publication for monomorphic semantic subtyping.
- G. Castagna: *Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers)*. Logical Methods in Computer Science. 2019 (To appear).
  A simple introduction to semantic subtyping and a detailed description of the implementation of subtyping and type-checking algorithms.
- G. Castagna and Z. Xu: *Set-theoretic foundation of parametric polymorphism and subtyping*. In ICFP 11.
  Subtyping for polymorphic set-theoretic types
- Castagna et al.: *Polymorphic Functions with Set-Theoretic Types*. Part 1 (POPL 14) and Part 2 (POPL 15).
  Languages with polymorphic set-theoretic types
- T. Petrucciani: *Polymorphic Set-Theoretic Types for Functional Languages.* PhD thesis, March 2019.
  Type reconstruction for polymorphic set-theoretic types

To try it out

- CDuce: `http://www.cduce.org`.
- For polymorphism use the development branch available at `https://gitlab.math.univ-paris-diderot.fr/cduce`)
- For a flavor of type reconstruction try the interactive interpreter at `http://www.cduce.org/ocaml/bi`

Gradual Typing

Outline

Outline

Motivating example: reminder

```
function double (x    ) {
  (<condition>) ? 2*x : x.concat(x)
}
```

Cannot give a type to `x` that works with both `2*x` and `x.concat(x)`

Motivating example: reminder

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)
}
```

Cannot give a type to `x` that works with both `2*x` and `x.concat(x)`

Solution

Add an unknown/type "?"

Motivating example: reminder

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)
}
```

Cannot give a type to `x` that works with both `2*x` and `x.concat(x)`

Solution

Add an unknown/type "?"

Develop a type theory for "?" such that:

- No solution for **?** for some execution ⇒ statically reject
- No problem for any solution for **?** ⇒ statically accept, do nothing
- For each possible execution there exists some solution for **?** ⇒ statically accept and add run-time checks

Reject at compile time:

```
function wrong (x : ?) {
  return (2*x + x(2));  //cannot be a number and a function
}
```

Reject at compile time:

```
function wrong (x : ?) {
  return (2*x + x(2));  //cannot be a number and a function
}
```

Accept as is:

```
function ok (x : ?) {
  if (typeof(x) === "number"){ return 42 } else { return x }
}
```

Intuitively the function has type: $? \to (\texttt{number} \mid ?)$

Reject at compile time:

```
function wrong (x : ?) {
  return (2*x + x(2));  //cannot be a number and a function
}
```

Accept as is:

```
function ok (x : ?) {
  if (typeof(x) === "number"){ return 42 } else { return x }
}
```

Intuitively the function has type: $? \to$ (number | ?)

Accept and insert checks:

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)
}
```

Compile as

```
function double (x : ?) {
  (<condition>) ? 2*(x⟨number⟩) : (x⟨string⟩).concat(x⟨string⟩)
}
```

Rationale

Mix static and dynamic typing

Rationale

Mix static and dynamic typing

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)

function apply (f : number --> number, x : number) {
   return (f x);
}

apply (double , (double 42))
```

Rationale

Mix static and dynamic typing

Dynamically typed:

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)
```

Statically typed:

```
function apply (f : number --> number, x : number) {
   return (f x);
}
```

Mixed typing:

```
apply (double , (double 42))
```

Rationale

Mix static and dynamic typing

Dynamically typed:

```
function double (x : ?) {
  (<condition>) ? 2*x : x.concat(x)
```

Statically typed:

```
function apply (f : number --> number, x : number) {
   return (f x);
}
```

Mixed typing:

```
apply (double , (double 42))
```

Add checks at the boundaries:

```
apply (double , (double 42))
```

must be compiled as

$\texttt{apply (double}\langle\texttt{number}\rightarrow\texttt{number}\rangle\texttt{ , (double 42)}\langle\texttt{number}\rangle\texttt{)}$

A hot topic

Prominent Languages with Gradual Typing:

- Typed Racket
- Reticulated Python
- TypeScript (Microsoft)
- Flow (Facebook)
- Hack (Facebook)
- Dart (Google)
- Thorn
- Safe Typescript

A hot topic

Prominent Languages with Gradual Typing:

- Typed Racket
- Reticulated Python
- TypeScript (Microsoft)
- Flow (Facebook)
- Hack (Facebook)
- Dart (Google)
- Thorn
- Safe Typescript

- Retrofitted on existing languages
- New languages

A hot topic

Prominent Languages with Gradual Typing:

- Typed Racket
- Reticulated Python
- TypeScript (Microsoft)
- Flow (Facebook)
- Hack (Facebook)
- Dart (Google)
- Thorn
- Safe Typescript

- Retrofitted on existing languages
- New languages

- Insert checks at run-time (a.k.a. sound gradual typing)
- Permissive typing (no checks inserted)
- Strict typing
- Occurrence typing

Roadmap

1. Add "**?**" to types
2. Define a typing discipline for programs with "**?**"
   - A well-typed program must still be well-typed with less-precise annotations
   - Less-precise annotations may make a program to become well-typed
3. Use the typing derivation to add dynamic type-checks at the boundaries between statically-type and dynamically-typed parts
   - Using less precise annotations in a well-typed program must not yield failures of dynamic checks (preserve semantics)
   - Failures of dynamic checks are due only to the dynamically-typed parts

Type precision: the lesser the "**?**", the more precise the type.

Outline

Simply-typed λ-calculus types:

$$\textit{Types} \qquad T ::= \texttt{Bool} \mid \texttt{Int} \mid T \to T$$

Gradual Typing [Siek&Taha 2006]

Simply-typed λ-calculus types:

$$\textit{Types} \qquad T \;::=\; \texttt{Bool} \;\mid\; \texttt{Int} \;\mid\; T \to T \;\mid\; ?$$

Simply-typed λ-calculus types:

$$\textit{Types} \qquad T \quad ::= \quad \texttt{Bool} \quad | \quad \texttt{Int} \quad | \quad T \to T \quad | \quad ?$$

A new **consistency** relation "∼" governs implicit casts involving "**?**":

$$\frac{}{\texttt{Bool} \sim \texttt{Bool}} \qquad \frac{}{\texttt{Int} \sim \texttt{Int}} \qquad \frac{}{T \sim ?} \qquad \frac{}{? \sim T} \qquad \frac{S_1 \sim T_1 \qquad S_2 \sim T_2}{S_1 \to S_2 \sim T_1 \to T_2}$$

Gradual Typing [Siek&Taha 2006]

Simply-typed λ-calculus types:

$$\textit{Types} \qquad T \; ::= \; \texttt{Bool} \;\mid\; \texttt{Int} \;\mid\; T \to T \;\mid\; \textbf{?}$$

A new **consistency** relation "$\sim$" governs implicit casts involving "**?**":

$$\frac{}{\texttt{Bool}\sim\texttt{Bool}} \qquad \frac{}{\texttt{Int}\sim\texttt{Int}} \qquad \frac{}{T\sim\textbf{?}} \qquad \frac{}{\textbf{?}\sim T} \qquad \frac{S_1\sim T_1 \qquad S_2\sim T_2}{S_1 \to S_2\sim T_1 \to T_2}$$

Relax application for consistent types:

$$[\to\textsc{Elim}_\sim]\; \frac{\Gamma \vdash a : S\to T \qquad \Gamma \vdash b : U \qquad U\sim S}{\Gamma \vdash ab : T}$$

Simply-typed λ-calculus types:

$$\textit{Types} \quad T ::= \texttt{Bool} \mid \texttt{Int} \mid T \to T \mid \;?$$

A new **consistency** relation "$\sim$" governs implicit casts involving "**?**":

$$\frac{}{\texttt{Bool}\sim\texttt{Bool}} \qquad \frac{}{\texttt{Int}\sim\texttt{Int}} \qquad \frac{}{T\sim ?} \qquad \frac{}{?\sim T} \qquad \frac{S_1\sim T_1 \qquad S_2\sim T_2}{S_1\to S_2\sim T_1\to T_2}$$

Relax application for consistent types:

$$[\to\textsc{Elim}_\sim]\ \frac{\Gamma\vdash a : S\to T \qquad \Gamma\vdash b : U \qquad U\sim S}{\Gamma\vdash ab : T}$$

Use the type derivation to insert casts

$$[\to\textsc{Elim}_\sim]\ \frac{\Gamma\vdash a : S\to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma\vdash b : U \overset{\text{compiles}}{\dashrightarrow} b' \qquad U\sim S}{\Gamma\vdash ab : T \overset{\text{compiles}}{\dashrightarrow} a(b\langle S\rangle)}\ (U\not\equiv S)$$

Simply-typed λ-calculus types:

$$\textit{Types} \quad T \ ::= \ \texttt{Bool} \ \mid \ \texttt{Int} \ \mid \ T \to T \ \mid \ ?$$

A new **consistency** relation "$\sim$" governs implicit casts involving "**?**":

$$\frac{}{\texttt{Bool}\sim\texttt{Bool}} \qquad \frac{}{\texttt{Int}\sim\texttt{Int}} \qquad \frac{}{T\sim ?} \qquad \frac{}{?\sim T} \qquad \frac{S_1\sim T_1 \qquad S_2\sim T_2}{S_1 \to S_2\sim T_1 \to T_2}$$

Relax application for consistent types:

$$[\to\text{ELIM}_\sim]\ \frac{\Gamma\vdash a : S\to T \qquad \Gamma\vdash b : U \qquad U\sim S}{\Gamma\vdash ab : T}$$

Use the type derivation to insert casts

$$[\to\text{ELIM}_\sim]\ \frac{\Gamma\vdash a : S\to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma\vdash b : U \overset{\text{compiles}}{\dashrightarrow} b' \qquad U\sim S}{\Gamma\vdash ab : T \overset{\text{compiles}}{\dashrightarrow} a(b\langle S\rangle)}\ (U \not\equiv S)$$

Gradual Typing

[Siek&Taha 2006]

Simply-typed λ-calculus types:

$$\textit{Types} \qquad T \;::=\; \texttt{Bool} \;\mid\; \texttt{Int} \;\mid\; T \to T \;\mid\; ?$$

A new **consistency** relation "$\sim$" governs implicit casts involving "**?**":

$$\frac{}{\texttt{Bool}\sim\texttt{Bool}} \qquad \frac{}{\texttt{Int}\sim\texttt{Int}} \qquad \frac{}{T\sim ?} \qquad \frac{}{?\sim T} \qquad \frac{S_1\sim T_1 \qquad S_2\sim T_2}{S_1 \to S_2\sim T_1 \to T_2}$$

Relax application for consistent types:

$$[\to\text{ELIM}_\sim]\ \frac{\Gamma \vdash a : S\to T \qquad \Gamma \vdash b : U}{\Gamma \vdash ab : T}$$

The remaining compilation rules implement the identity (they do not modify the compiled term)

Use the type derivation to insert casts

$$[\to\text{ELIM}_\sim]\ \frac{\Gamma \vdash a : S\to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma \vdash b : U \overset{\text{compiles}}{\dashrightarrow} b' \qquad U\sim S}{\Gamma \vdash ab : T \overset{\text{compiles}}{\dashrightarrow} a(b\langle S\rangle)}\ (U \not\equiv S)$$

Problems

- **The consistency relation *must not* be transitive:**

Since $\texttt{Int}\sim ?$ and $?\sim\texttt{Bool}$, then transitivity would imply $\texttt{Int}\sim\texttt{Bool}$:

$$\frac{\vdash \lambda x:\texttt{Int}.x+1:\texttt{Int}\to\texttt{Int} \qquad \vdash \mathsf{true}:\texttt{Bool} \qquad \texttt{Int}\sim\texttt{Bool}}{\vdash (\lambda x:\texttt{Int}.x+1)\mathsf{true}:\texttt{Int}}$$

it is hard to work with a non-transitive relation.

Problems

- **The consistency relation *must not* be transitive:**

Since $\texttt{Int}\sim ?$ and $?\sim\texttt{Bool}$, then transitivity would imply $\texttt{Int}\sim\texttt{Bool}$:

$$\frac{\vdash \lambda x:\texttt{Int}.x+1:\texttt{Int}\to\texttt{Int} \qquad \vdash \text{true}:\texttt{Bool} \qquad \texttt{Int}\sim\texttt{Bool}}{\vdash (\lambda x:\texttt{Int}.x+1)\text{true}:\texttt{Int}}$$

it is hard to work with a non-transitive relation.

- **It has a flavor of substitutivity ... but not always:**

```
function double (x : ?) { (<condition>) ? 2*x : x.concat(x) }
function apply (f : number --> number, x : number) { return (f x) }
apply (double , (double 42))
```

It compiles as $\texttt{apply}\,(\,\texttt{double}\langle\texttt{Int}\to\texttt{Int}\rangle\,,\,(\texttt{double}(42\langle ?\rangle))\langle\texttt{Int}\rangle\,)$

Problems

- **The consistency relation *must not* be transitive:**

Since $\texttt{Int}\sim ?$ and $?\sim\texttt{Bool}$, then transitivity would imply $\texttt{Int}\sim\texttt{Bool}$:

$$\frac{\vdash \lambda x:\texttt{Int}.x+1 : \texttt{Int}\to\texttt{Int} \qquad \vdash \mathsf{true} : \texttt{Bool} \qquad \texttt{Int}\sim\texttt{Bool}}{\vdash (\lambda x:\texttt{Int}.x+1)\mathsf{true} : \texttt{Int}}$$

it is hard to work with a non-transitive relation.

- **It has a flavor of substitutivity ... but not always:**

```
function double (x : ?) { (<condition>) ? 2*x : x.concat(x) }
function apply (f : number --> number, x : number) { return (f x) }
apply (double , (double 42))
```

It compiles as $\texttt{apply}\,(\,\texttt{double}\langle\texttt{Int}\to\texttt{Int}\rangle\,,\,(\texttt{double}(42\langle ?\rangle))\langle\texttt{Int}\rangle\,)$

- Casting $?\to ?$ to $\texttt{Int}\to\texttt{Int}$ is ok.

Problems

- **The consistency relation *must not* be transitive:**

Since $\texttt{Int}\sim ?$ and $?\sim\texttt{Bool}$, then transitivity would imply $\texttt{Int}\sim\texttt{Bool}$:

$$\frac{\vdash \lambda x:\texttt{Int}.x+1 : \texttt{Int}\to\texttt{Int} \qquad \vdash \text{true} : \texttt{Bool} \qquad \texttt{Int}\sim\texttt{Bool}}{\vdash (\lambda x:\texttt{Int}.x+1)\text{true} : \texttt{Int}}$$

it is hard to work with a non-transitive relation.

- **It has a flavor of substitutivity ... but not always:**

```
function double (x : ?) { (<condition>) ? 2*x : x.concat(x) }
function apply (f : number --> number, x : number) { return (f x) }
apply (double , (double 42))
```

It compiles as $\texttt{apply}\,(\,\texttt{double}\langle\texttt{Int}\to\texttt{Int}\rangle\,,\,(\texttt{double}(42\langle ?\rangle))\langle\texttt{Int}\rangle\,)$

- Casting $?\to ?$ to $\texttt{Int}\to\texttt{Int}$ is ok.
- Casting $?$ to $\texttt{Int}$ is ok.

Problems

- **The consistency relation *must not* be transitive:**

Since $\texttt{Int}\sim?$ and $?\sim\texttt{Bool}$, then transitivity would imply $\texttt{Int}\sim\texttt{Bool}$:

$$\frac{\vdash \lambda x:\texttt{Int}.x+1:\texttt{Int}\to\texttt{Int} \qquad \vdash \text{true}:\texttt{Bool} \qquad \texttt{Int}\sim\texttt{Bool}}{\vdash (\lambda x:\texttt{Int}.x+1)\text{true}:\texttt{Int}}$$

it is hard to work with a non-transitive relation.

- **It has a flavor of substitutivity ... but not always:**

```
function double (x : ?) { (<condition>) ? 2*x : x.concat(x) }
function apply (f : number --> number, x : number) { return (f x) }
apply (double , (double 42))
```

It compiles as $\texttt{apply}\,(\,\texttt{double}\langle\texttt{Int}\to\texttt{Int}\rangle\,,\,(\texttt{double}(42\langle ? \rangle))\langle\texttt{Int}\rangle\,)$

- Casting $?\to?$ to $\texttt{Int}\to\texttt{Int}$ is ok.
- Casting $?$ to $\texttt{Int}$ is ok.
- Casting an $\texttt{Int}$ to $?$ looks weird

Problems

- **The [$\rightarrow$ELIM$_\sim$] rule looks more an algorithic step than a typing rule:**

$$[\rightarrow\text{ELIM}_\sim]\quad \frac{\Gamma \vdash a : S\rightarrow T \qquad \Gamma \vdash b : U \qquad U\sim S}{\Gamma \vdash ab : T}$$

$$[\rightarrow\text{ELIM}_\leq]\quad \frac{\Gamma \vdash_{\mathcal{A}} a : S\rightarrow T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U\leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

Problems

- **The [$\to$Elim$_\sim$] rule looks more an algorithic step than a typing rule:**

$$[\to\text{Elim}_\sim]\ \frac{\Gamma \vdash a : S\to T \qquad \Gamma \vdash b : U \qquad U \sim S}{\Gamma \vdash ab : T} \qquad [\to\text{Elim}_\leq]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S\to T \qquad \Gamma \vdash_{\mathcal{A}} b : U \qquad U \leq S}{\Gamma \vdash_{\mathcal{A}} ab : T}$$

We need a more principled methodology

Problems

- **The [$\to$Elim$_\sim$] rule looks more an algorithic step than a typing rule:**

$$[\to\textsc{Elim}_\sim]\ \frac{\Gamma\vdash a:S\to T \qquad \Gamma\vdash b:U \qquad U\sim S}{\Gamma\vdash ab:T}$$

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

We need a more principled methodology

Let's take inspiration from what we did for subtyping

Precision and Materialization

The precision relation "$\sqsubseteq$":

Precision relates a type with unknown "**?**" components to the types it *may* dynamically become at run time.

Precision and Materialization

The precision relation "$\sqsubseteq$":

Precision relates a type with unknown "**?**" components to the types it *may* dynamically become at run time.

Informally

The less "**?**" it uses, the more *precise* a type is.

Precision and Materialization

The precision relation "$\sqsubseteq$":

Precision relates a type with unknown "**?**" components to the types it *may* dynamically become at run time.

Informally

The less "**?**" it uses, the more *precise* a type is.

Can be defined by induction for simple types:

$$\frac{}{? \sqsubseteq T} \qquad \frac{S_1 \sqsubseteq T_1 \qquad S_2 \sqsubseteq T_2}{S_1 \to S_2 \sqsubseteq T_1 \to T_2} \qquad \frac{}{T \sqsubseteq T} \qquad \frac{T_1 \sqsubseteq T_2 \qquad T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3}$$

Precision and Materialization

The precision relation "$\sqsubseteq$":

Precision relates a type with unknown "**?**" components to the types it *may* dynamically become at run time.

Informally

The less "**?**" it uses, the more *precise* a type is.

Can be defined by induction for simple types:

$$\frac{}{? \sqsubseteq T} \qquad \frac{S_1 \sqsubseteq T_1 \qquad S_2 \sqsubseteq T_2}{S_1 \to S_2 \sqsubseteq T_1 \to T_2} \qquad \frac{}{T \sqsubseteq T} \qquad \frac{T_1 \sqsubseteq T_2 \qquad T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3}$$

- It is *not* subtyping

Precision and Materialization

The precision relation "$\sqsubseteq$":

Precision relates a type with unknown "**?**" components to the types it *may* dynamically become at run time.

Informally

The less "**?**" it uses, the more *precise* a type is.

Can be defined by induction for simple types:

$$\frac{}{? \sqsubseteq T} \qquad \frac{S_1 \sqsubseteq T_1 \qquad S_2 \sqsubseteq T_2}{S_1 \to S_2 \sqsubseteq T_1 \to T_2} \qquad \frac{}{T \sqsubseteq T} \qquad \frac{T_1 \sqsubseteq T_2 \qquad T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3}$$

- It is *not* subtyping
- It is a pre-order

Precision and Materialization

The precision relation "$\sqsubseteq$":

Precision relates a type with unknown "**?**" components to the types it *may* dynamically become at run time.

Informally

The less "**?**" it uses, the more *precise* a type is.

Can be defined by induction for simple types:

$$\frac{}{? \sqsubseteq T} \qquad \frac{S_1 \sqsubseteq T_1 \qquad S_2 \sqsubseteq T_2}{S_1 \to S_2 \sqsubseteq T_1 \to T_2} \qquad \frac{}{T \sqsubseteq T} \qquad \frac{T_1 \sqsubseteq T_2 \qquad T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3}$$

- It is *not* subtyping
- It is a pre-order

Intuition

$T \sqsubseteq T'$ means that at run-time type T may turn out to be the type T'

we say that T ***may materialize into*** T'

Precision and Materialization

The precision relation is a pre-order thus, in particular, it is *transitive*:

$$? \quad \sqsubseteq \quad ? \to ? \quad \sqsubseteq \quad ? \to \mathtt{Int} \quad \sqsubseteq \quad \mathtt{Int} \to \mathtt{Int}$$

Precision and Materialization

The precision relation is a pre-order thus, in particular, it is *transitive*:

$$? \quad \sqsubseteq \quad ? \to ? \quad \sqsubseteq \quad ? \to \mathtt{Int} \quad \sqsubseteq \quad \mathtt{Int} \to \mathtt{Int}$$

but:

$$? \quad \sqsubseteq \quad \mathtt{Int} \quad \not\sqsubseteq \quad ?$$

Precision and Materialization

The precision relation is a pre-order thus, in particular, it is *transitive*:

$$? \quad \sqsubseteq \quad ? \to ? \quad \sqsubseteq \quad ? \to \texttt{Int} \quad \sqsubseteq \quad \texttt{Int} \to \texttt{Int}$$

but:

$$? \quad \sqsubseteq \quad \texttt{Int} \quad \not\sqsubseteq \quad ?$$

This means that it can be used in a subsumption-like rule:

$$[\text{MATERIALIZE}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

Precision and Materialization

The precision relation is a pre-order thus, in particular, it is *transitive*:

$$? \quad \sqsubseteq \quad ? \to ? \quad \sqsubseteq \quad ? \to \texttt{Int} \quad \sqsubseteq \quad \texttt{Int} \to \texttt{Int}$$

but:

$$? \quad \sqsubseteq \quad \texttt{Int} \quad \not\sqsubseteq \quad ?$$

This means that it can be used in a subsumption-like rule:

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

We can add it to any type system to embed gradual typing in it.

Precision and Materialization

The precision relation is a pre-order thus, in particular, it is *transitive*:

$$? \sqsubseteq ?\to ? \sqsubseteq ?\to \texttt{Int} \sqsubseteq \texttt{Int}\to\texttt{Int}$$

but:

$$? \sqsubseteq \texttt{Int} \not\sqsubseteq ?$$

This means that it can be used in a subsumption-like rule:

$$[\textsc{Materialize}]\ \frac{\Gamma\vdash a : S \qquad S\sqsubseteq T}{\Gamma\vdash a : T}$$

We can add it to any type system to embed gradual typing in it.

Rationale

As *subtyping* caputures "*safe replacement*",
so *precision* captures "*potential materialization*".

Precision and Materialization

Since *potential materialization* does not mean *assured* materialization, then we have to check it at run-time:

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Precision and Materialization

Since *potential materialization* does not mean *assured* materialization, then we have to check it at run-time:

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Rationale

- *Subtyping* = assured materialization (cast always works)
- *Precision* = possible materialization (cast may fail)

Precision and Materialization

Since *potential materialization* does not mean *assured* materialization, then we have to check it at run-time:

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Rationale

- *Subtyping* = assured materialization (cast always works)
- *Precision* = possible materialization (cast may fail)

From a logical viewpoint:

[Subsumption]

$$\frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \leq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'(\!|T|\!)}$$

Subsumption as implicit coercions (subtyping)

[Materialize]

$$\frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Materialization as explicit casts (precision)

Summing up

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

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

$$[\text{VAR}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\to\text{INTRO}]\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad [\to\text{ELIM}]\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

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

$$[\text{VAR}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\to\text{INTRO}]\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad [\to\text{ELIM}]\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing

$$
\begin{array}{llll}
\textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \quad \mid \quad \textbf{?} \\
\textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots
\end{array}
\qquad (\lambda x{:}T.a)b \longrightarrow a[b/x]
$$

$$
[\text{VAR}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}
\qquad
[\to\text{INTRO}]\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T}
\qquad
[\to\text{ELIM}]\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}
$$

$$
[\text{MATERIALIZE}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}
$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \quad \mid \quad ? \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

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

$$\text{[VAR]}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{[}{\to}\text{INTRO]}\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad \text{[}{\to}\text{ELIM]}\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$\text{[MATERIALIZE]}\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad \text{[MATERIALIZE}_{\text{COMPIL}}\text{]}\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. **Et voila: you have added gradual typing**

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \ ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

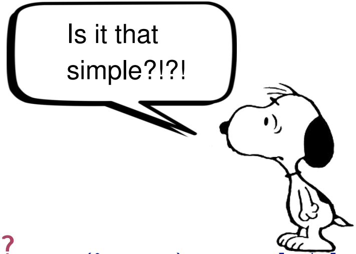
$$(\lambda x{:}T.a)b \longrightarrow a[b/x]$$

$$\text{[VAR]}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{[}{\to}\text{INTRO]}\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad \text{[}{\to}\text{ELIM]}\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$\text{[MATERIALIZE]}\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad \text{[MATERIALIZE}_{\text{COMPIL}}\text{]}\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Summing up

1. Take your favorite typed language
2. Add “**?**” to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing



$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \; ? \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

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

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

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

$$[\text{VAR}]\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad [\to\text{INTRO}]\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T} \qquad [\to\text{ELIM}]\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}$$

$$[\text{MATERIALIZE}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\text{MATERIALIZE}_{\text{COMPIL}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \; ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

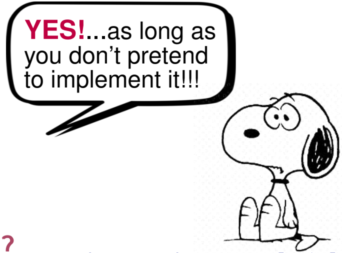
$$(\lambda x{:}T.a)b \longrightarrow a[b/x]$$

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Summing up

1. Take your favorite typed language
2. Add "**?**" to types
3. Add the materialization rule (with suitable $\sqsubseteq$)
4. Compile to insert casts
5. Et voila: you have added gradual typing



$$
\begin{array}{llll}
\textit{Types} & T & ::= & \mathtt{Int} \mid \mathtt{Bool} \mid T \to T \mid \; ? \\
\textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots
\end{array}
$$

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

$$
\text{[VAR]}\ \frac{}{\Gamma \vdash x : \Gamma(x)}
\qquad
\text{[}\to\text{INTRO]}\ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x{:}S.a : S \to T}
\qquad
\text{[}\to\text{ELIM]}\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T}
$$

$$
\text{[MATERIALIZE]}\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}
\qquad
\text{[MATERIALIZE}_{\text{COMPIL}}\text{]}\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}
$$

Algorithmic aspects

From more theoretical to more practical ones:

Algorithmic aspects

From more theoretical to more practical ones:

- Materialization elimination: as we had to eliminate subsumption to get a type-checking algorithm so we have to do the same for [MATERIALIZE].

Algorithmic aspects

From more theoretical to more practical ones:

- Materialization elimination: as we had to eliminate subsumption to get a type-checking algorithm so we have to do the same for [MATERIALIZE].

- Implementation of casts: the implementation of the cast calculus is not trivial. How do we check casts? In particular, how do we handle functional casts:

$$(\mathtt{double}\langle\mathtt{Int}\to\mathtt{Int}\rangle)(42) \quad\longrightarrow\quad ????$$

Algorithmic aspects

From more theoretical to more practical ones:

- Materialization elimination: as we had to eliminate subsumption to get a type-checking algorithm so we have to do the same for [MATERIALIZE].

- Implementation of casts: the implementation of the cast calculus is not trivial. How do we check casts? In particular, how do we handle functional casts:

$$(\mathtt{double}\langle \mathtt{Int}\to\mathtt{Int}\rangle)(42) \quad\longrightarrow\quad ????$$

- Error messages: when a cast fails which part of the program is to blame?

Algorithmic aspects

From more theoretical to more practical ones:

- Materialization elimination: as we had to eliminate subsumption to get a type-checking algorithm so we have to do the same for [MATERIALIZE].

- Implementation of casts: the implementation of the cast calculus is not trivial. How do we check casts? In particular, how do we handle functional casts:

$$(\texttt{double}\langle\texttt{Int}\to\texttt{Int}\rangle)(42) \quad\longrightarrow\quad \texttt{????}$$

- Error messages: when a cast fails which part of the program is to blame?

- Efficient implementation: how to avoid accumulation of cast compositions (i.e., stack overflow) and how to implement efficiently tail recursion for functions with casts?

Algorithmic aspects

From more theoretical to more practical ones:

- Materialization elimination: as we had to eliminate subsumption to get a type-checking algorithm so we have to do the same for [MATERIALIZE].
- Implementation of casts: the implementation of the cast calculus is not trivial. How do we check casts? In particular, how do we handle functional casts:

$$(\texttt{double}\langle\texttt{Int}\to\texttt{Int}\rangle)(42) \quad\longrightarrow\quad ????$$

- Error messages: when a cast fails which part of the program is to blame?
- Efficient implementation: how to avoid accumulation of cast compositions (i.e., stack overflow) and how to implement efficiently tail recursion for functions with casts?

But before that, let me show you that the approach works and it is pretty general

A principled approach

Simply Typed Lambda Calculus

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

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

Typing

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

A principled approach

Simply Typed Lambda Calculus

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \quad \mid \quad ? \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

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

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

A principled approach

Simply Typed Lambda Calculus

Syntax:

$$\begin{array}{llcl} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \; ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

semantics must be given by compilation

Semantics:

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

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

A principled approach

Simply Typed Lambda Calculus

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \quad \mid \quad \textbf{?} \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

A principled approach

Simply Typed Lambda Calculus + Gradual Typing

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \quad \mid \quad ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

A principled approach

Simply Typed Lambda Calculus + Gradual Typing + Subtyping

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

$$[\text{MATERIALIZE}_{\text{COMPIL}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Typing

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

$$[\text{MATERIALIZE}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\text{SUBSUM}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

Soundness

If the reduction semantics of the cast calculus is reasonably defined (see later) then:

Theorem (Soundness)

If $\Gamma \vdash a : T$, then $\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'$ and

- either a' reduces to a value of type T
- or a' diverges
- or a' fails for a cast on a dynamic type

Soundness

If the reduction semantics of the cast calculus is reasonably defined (see later) then:

Theorem (Soundness)

If $\Gamma \vdash a : T$, then $\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'$ and

- either a' reduces to a value of type T
- or a' diverges
- or a' fails for a cast on a dynamic type

HM Polymorphism

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \alpha \\ \textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x.a \mid \texttt{let}\ x = a\ \texttt{in}\ b \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

$$(\beta) \qquad (\lambda x.a)b \longrightarrow a[b/x]$$

Typing

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

$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

HM Polymorphism + Gradual Typing

Syntax:

$$
\begin{array}{llll}
\textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \alpha \mid ? \\
\textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \\
\textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x.a \mid \texttt{let}\ x = a\ \texttt{in}\ b \mid 1 \mid 2 \mid \ldots
\end{array}
$$

Semantics:

$$
[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}
$$

Typing

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

$$
\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x\ =\ a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}
$$

$$
[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}
$$

HM Polymorphism + Gradual Typing + Subtyping

Syntax:

$$\begin{array}{llcl} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \alpha \mid ? \\ \textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x.a \mid \texttt{let}\ x = a\ \texttt{in}\ b \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Typing

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

$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x = a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

HM Polymorphism + Gradual Typing + Subtyping

Syntax:

Types	T	$::=$	$\texttt{Int} \mid \texttt{Bool} \mid T \to T$
Schemas	σ	$::=$	$T \mid \forall\alpha.\sigma$
Terms	a, b	$::=$	$x \mid ab \mid \lambda x.a \mid \texttt{let}\ x$

Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Typing

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

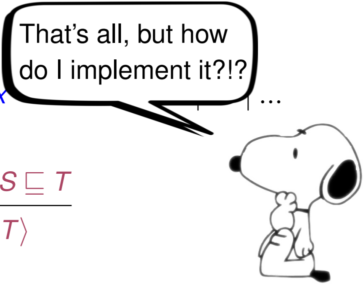
$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

HM Polymorphism + Gradual Typing + Subtyping

Syntax:

$$
\begin{array}{llll}
\textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \\
\textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \\
\textit{Terms} & a,b & ::= & x \mid ab \mid \lambda x.a \mid \texttt{let}\ x \ \ldots \mid \ldots
\end{array}
$$



Semantics:

$$
[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}
$$

Typing

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

$$
\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}
$$

$$
[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}
$$

Outline

1. Type-checking algorithm

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

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T} \qquad \text{[MATERIALIZE]}\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

1. Type-checking algorithm

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

$$\frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : S}{\Gamma \vdash ab : T} \qquad \text{[MATERIALIZE]}\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

1. Type-checking algorithm

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

$$[\to\textsc{Elim}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} ab : T}\ \exists V. S \sqsubseteq V, U \sqsubseteq V$$

1. Type-checking algorithm

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

$$[\to\text{ELIM}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} ab : T}\ \exists V.S \sqsubseteq V, U \sqsubseteq V$$

It is a sound and complete algorithm:

$$\Gamma \vdash a : T \quad \iff \quad \Gamma \vdash_{\mathcal{A}} a : S \text{ and } S \sqsubseteq T$$

1. Type-checking algorithm

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

$$[\to\text{ELIM}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} ab : T}\ \exists V. S \sqsubseteq V, U \sqsubseteq V$$

It is a sound and complete algorithm:

$$\Gamma \vdash a : T \quad \iff \quad \Gamma \vdash_{\mathcal{A}} a : S \text{ and } S \sqsubseteq T$$

Actually this is the good old $[\to\text{ELIM}_{\sim}]$ rule of Siek&Taha (but defined for a sensible relation):

$$[\to\text{ELIM}_{\sim}]\ \frac{\Gamma \vdash a : S \to T \qquad \Gamma \vdash b : U \qquad U \sim S}{\Gamma \vdash ab : T}$$

since $U \sim S \iff \exists V. S \sqsubseteq V, U \sqsubseteq V$

2. Compilation

Thanks to the algorithm every well-typed term is a associated to a unique typing derivation: we know *where* to put casts.

2. Compilation

Thanks to the algorithm every well-typed term is a associated to a unique typing derivation: we know *where* to put casts. Indeed:

$$[\to\text{ELIM}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} a(b) : T}\ \exists V. S \sqsubseteq V, U \sqsubseteq V$$

2. Compilation

Thanks to the algorithm every well-typed term is a associated to a unique typing derivation: we know *where* to put casts. Indeed:

$$[\to\textsc{Elim}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} a(b) : T}\ \exists V. S \sqsubseteq V, U \sqsubseteq V$$

corresponds to the derivation

$$\to\textsc{Elim}\ \frac{\textsc{Mater}\ \dfrac{\Gamma \vdash a : S\to T \qquad \dfrac{S \sqsubseteq V \qquad T \sqsubseteq T}{S\to T \sqsubseteq V\to T}}{\Gamma \vdash a : V\to T} \qquad \dfrac{\Gamma \vdash b : U \qquad U \sqsubseteq V}{\Gamma \vdash b : V}\ \textsc{Mater}}{\Gamma \vdash_{\mathcal{A}} a(b) : T}$$

2. Compilation

Thanks to the algorithm every well-typed term is a associated to a unique typing derivation: we know *where* to put casts. Indeed:

$$[\to\textsc{Elim}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} a(b) : T}\ \exists V. S \sqsubseteq V, U \sqsubseteq V$$

corresponds to the derivation which tells us where to put cast:

$$\to\textsc{Elim}\ \frac{\textsc{Mater}\ \dfrac{\Gamma \vdash a : S{\to}T \qquad \dfrac{S \sqsubseteq V \qquad T \sqsubseteq T}{S{\to}T \sqsubseteq V{\to}T}}{\Gamma \vdash a\langle V{\to}T\rangle : V{\to}T} \qquad \dfrac{\Gamma \vdash b : U \qquad U \sqsubseteq V}{\Gamma \vdash b\langle V\rangle : V}\ \textsc{Mater}}{\Gamma \vdash_{\mathcal{A}} a\langle V{\to}T\rangle\,(b\langle V\rangle) : T}$$

2. Compilation

Thanks to the algorithm every well-typed term is a associated to a unique typing derivation: we know *where* to put casts. Indeed:

$$[\to\text{ELIM}_{\sqsubseteq}]\ \frac{\Gamma \vdash_{\mathcal{A}} a : S \to T \qquad \Gamma \vdash_{\mathcal{A}} b : U}{\Gamma \vdash_{\mathcal{A}} a(b) : T}\ \exists V. S \sqsubseteq V, U \sqsubseteq V$$

corresponds to the derivation which tells us where to put cast:

$$\to\text{ELIM}\ \dfrac{\text{MATER}\ \dfrac{\Gamma \vdash a : S \to T \qquad \dfrac{S \sqsubseteq V \qquad T \sqsubseteq T}{S \to T \sqsubseteq V \to T}}{\Gamma \vdash a\langle V \to T\rangle : V \to T} \qquad \dfrac{\Gamma \vdash b : U \qquad U \sqsubseteq V}{\Gamma \vdash b\langle V\rangle : V}\ \text{MATER}}{\Gamma \vdash_{\mathcal{A}} a\langle V \to T\rangle(b\langle V\rangle) : T}$$

Which V shall we use? well, obviously:

$$V = \min_{\sqsubseteq}\{W \mid S \sqsubseteq W, U \sqsubseteq W\}$$

2. Compilation

This yields the following compilation rule:

$$[\to\textsc{Elim}_{\sqsubseteq}\textsc{Compil}]\quad \frac{\Gamma \vdash a : S \to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma \vdash b : U \overset{\text{compiles}}{\dashrightarrow} b'}{\Gamma \vdash_{\mathcal{A}} ab : T \overset{\text{compiles}}{\dashrightarrow} a'\langle V \to T\rangle(b'\langle V\rangle)}\,(V = \min_{\sqsubseteq}\{W \mid S \sqsubseteq W, U \sqsubseteq W\})$$

This yields the following compilation rule:

$$[\rightarrow\text{ELIM}_{\sqsubseteq\text{COMPIL}}]\quad \frac{\Gamma \vdash a : S \rightarrow T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma \vdash b : U \overset{\text{compiles}}{\dashrightarrow} b'}{\Gamma \vdash_{\mathcal{A}} ab : T \overset{\text{compiles}}{\dashrightarrow} a'\langle V \rightarrow T\rangle(b'\langle V\rangle)}\,(V = \min_{\sqsubseteq}\{W \mid S \sqsubseteq W, U \sqsubseteq W\})$$

Of course we do not insert the corresponding cast when $V = S$ or $V = U$.

2. Compilation

This yields the following compilation rule:

$$[\to\textsc{Elim}_{\sqsubseteq}\textsc{Compil}]\quad \frac{\Gamma\vdash a:S\to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma\vdash b:U \overset{\text{compiles}}{\dashrightarrow} b'}{\Gamma\vdash_{\mathcal{A}} ab:T \overset{\text{compiles}}{\dashrightarrow} a'\langle V\to T\rangle(b'\langle V\rangle)}\,(V{=}\min_{\sqsubseteq}\{W \mid S{\sqsubseteq}W, U{\sqsubseteq}W\})$$

Of course we do not insert the corresponding cast when $V = S$ or $V = U$.

Cast insertion different from Siek&Taha: we cast both the function and the arguement:

We only use "upcast", that is cast from less precise to more precise types. This is formalized by the [MATERIALIZE] rule for *the language with casts* (all the other rules are as before)

$$[\textsc{Materialize}]\ \frac{\Gamma\vdash a:S \qquad S\sqsubseteq T}{\Gamma\vdash a\langle T\rangle : T}$$

This yields the following compilation rule:

$$[\to\text{ELIM}_{\sqsubseteq\text{COMPIL}}]\ \frac{\Gamma \vdash a : S \to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma \vdash b : U \overset{\text{compiles}}{\dashrightarrow} b'}{\Gamma \vdash_{\mathcal{A}} ab : T \overset{\text{compiles}}{\dashrightarrow} a'\langle V \to T\rangle(b'\langle V\rangle)}\ (V = \min_{\sqsubseteq}\{W \mid S \sqsubseteq W, U \sqsubseteq W\})$$

Of course we do not insert the corresponding cast when $V = S$ or $V = U$.

Cast insertion different from Siek&Taha: we cast both the function and the arguement:

We only use “upcast”, that is cast from less precise to more precise types. This is formalized by the [MATERIALIZE] rule for *the language with casts* (all the other rules are as before)

$$[\text{MATERIALIZE}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T\rangle : T}$$

The compilation rules map well-typed terms into well-typed terms: terms are cast to types *more precise* than their static type.

This yields the following compilation rule:

$$[\to\textsc{Elim}_{\sqsubseteq}\textsc{Compil}]\quad \frac{\Gamma \vdash a : S \to T \overset{\text{compiles}}{\dashrightarrow} a' \qquad \Gamma \vdash b : U \overset{\text{compiles}}{\dashrightarrow} b'}{\Gamma \vdash_{\mathcal{A}} ab : T \overset{\text{compiles}}{\dashrightarrow} a'\langle V{\to}T\rangle(b'\langle V\rangle)}\,(V{=}\min_{\sqsubseteq}\{W \mid S{\sqsubseteq}W, U{\sqsubseteq}W\})$$

Of course we do not insert the corresponding cast when $V = S$ or $V = U$.

Cast insertion different from Siek&Taha: we cast both the function and the arguement:

We only use “upcast”, that is cast from less precise
This is formalized by the [MATERIALIZE] rule for *the language with casts* (all the other rules are as before)

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T\rangle : T}$$

The compilation rules map well-typed terms into well-typed terms: terms are cast to types *more precise* than their static type.

Gradually Typed Language

Syntax:

$$\begin{array}{llcl} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid 1 \mid 2 \mid \ldots \end{array}$$

Typing

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

The cast language

Gradually Typed Language

Syntax:

$$
\begin{array}{llllllllllllll}
\textit{Types} & T & ::= & \texttt{Int} & \mid & \texttt{Bool} & \mid & T \to T & \mid & ? & & & & \\
\textit{Terms} & a, b & ::= & x & \mid & a\,b & \mid & \lambda x{:}T.a & \mid & a\langle T\rangle & \mid & 1 & \mid & 2 \mid \ldots
\end{array}
$$

Typing

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

Gradually Typed Language

Syntax:

$$
\begin{array}{llllllllllllll}
\textit{Types} & T & ::= & \texttt{Int} & \mid & \texttt{Bool} & \mid & T \to T & \mid & ? \\
\textit{Terms} & a, b & ::= & x & \mid & a\,b & \mid & \lambda x{:}T.a & \mid & a\langle T\rangle & \mid & 1 & \mid & 2 \mid \ldots
\end{array}
$$

Typing

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

$$
[\text{MATERIALIZE}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T\rangle : T}
$$

The cast language

Gradually Typed Language

Syntax:

$$\begin{array}{llcl} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \; ? \\ \textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x{:}T.a \mid a\langle T\rangle \mid 1 \mid 2 \mid \ldots \end{array}$$

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T\rangle : T}$$

Semantics:

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

The cast language

Gradually Typed Language with Casts

Syntax:

$$\begin{array}{llllllllllll} \textit{Types} & T & ::= & \texttt{Int} & \mid & \texttt{Bool} & \mid & T \to T & \mid & ? \\ \textit{Terms} & a, b & ::= & x & \mid & a\,b & \mid & \lambda x{:}T.a & \mid & a\langle T \rangle & \mid & 1 \mid 2 \mid \ldots \end{array}$$

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T \rangle : T}$$

Semantics:

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

The cast language

Gradually Typed Language with Casts

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid ? \\ \textit{Terms} & a, b & ::= & x \mid a\,b \mid \lambda x{:}T.a \mid a\langle T\rangle \mid 1 \mid 2 \mid \ldots \end{array}$$

Typing

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

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T\rangle : T}$$

Semantics:

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

Still missing the semantics for casts

The cast language

What is the dynamic semantics of casts?

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$\begin{array}{lcl} 3\langle\texttt{Int}\rangle & \longrightarrow & 3 \\ 3\langle\texttt{Bool}\rangle & \longrightarrow & \text{Fail} \end{array}$$

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$3\langle \texttt{Int} \rangle \longrightarrow 3$$
$$3\langle \texttt{Bool} \rangle \longrightarrow \text{Fail}$$

If T is not an arrow type, then for $a\langle T \rangle$ check whether the result of a is of type T

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$
\begin{array}{lcl}
3\langle\texttt{Int}\rangle & \longrightarrow & 3 \\
3\langle\texttt{Bool}\rangle & \longrightarrow & \text{Fail}
\end{array}
$$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}\!\to\!\texttt{Int}\rangle$.

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$3\langle\texttt{Int}\rangle \longrightarrow 3$

$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle$. Function foo *is not* of type $\texttt{Int}\to\texttt{Int}$

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$3\langle\texttt{Int}\rangle \longrightarrow 3$$
$$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle$. Function `foo` *is not* of type $\texttt{Int}\to\texttt{Int}$, nevertheless $(\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle)(42)$ *must not* fail: it's applied to an `Int` and returns an `Int`.

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$3\langle\texttt{Int}\rangle \longrightarrow 3$

$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else {
}
```

Consider $\texttt{foo}\langle\texttt{Int}{\to}\texttt{Int}\rangle$. Function `foo` *is not* ... ss $(\texttt{foo}\langle\texttt{Int}{\to}\texttt{Int}\rangle)(42)$ *must not* fail: it's applied to an `Int` an... ...rns an `Int`.

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$3\langle\texttt{Int}\rangle \longrightarrow 3$$
$$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle$. Function `foo` *is not* of type $\texttt{Int}\to\texttt{Int}$, nevertheless $(\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle)(42)$ *must not* fail: it's applied to an `Int` and returns an `Int`.

Delay the dynamic check of a type until you get to non-functional values

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$3\langle\texttt{Int}\rangle \longrightarrow 3$$
$$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}{\to}\texttt{Int}\rangle$. Function `foo` *is not* of type $\texttt{Int}{\to}\texttt{Int}$, nevertheless $(\texttt{foo}\langle\texttt{Int}{\to}\texttt{Int}\rangle)(42)$ *must not* fail: it's applied to an `Int` and returns an `Int`.

Delay the dynamic check of a type until you get to non-functional values

$$(\texttt{foo}\langle\texttt{Int} \to \texttt{Int}\rangle)(\mathit{exp})$$

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$3\langle\texttt{Int}\rangle \longrightarrow 3$

$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle$. Function `foo` *is not* of type $\texttt{Int}\to\texttt{Int}$, nevertheless $(\texttt{foo}\langle\texttt{Int}\to\texttt{Int}\rangle)(42)$ *must not* fail: it's applied to an `Int` and returns an `Int`.

Delay the dynamic check of a type until you get to non-functional values

$(\texttt{foo}\langle\texttt{Int} \to \texttt{Int}\rangle)(\ 42\ )$

The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$$3\langle\texttt{Int}\rangle \longrightarrow 3$$
$$3\langle\texttt{Bool}\rangle \longrightarrow \text{Fail}$$

If T is not an arrow type, then for $a\langle T\rangle$ check whether the result of a is of type T

Not so trivial for functions:

```
function foo (x : ?) {
  if (x == 42) { return (2*x)} else { true }
}
```

Consider $\texttt{foo}\langle\texttt{Int}\rightarrow\texttt{Int}\rangle$. Function `foo` *is not* of type $\texttt{Int}\rightarrow\texttt{Int}$, nevertheless $(\texttt{foo}\langle\texttt{Int}\rightarrow\texttt{Int}\rangle)(42)$ *must not* fail: it's applied to an `Int` and returns an `Int`.

Delay the dynamic check of a type until you get to non-functional values

$$(\texttt{foo}\langle\texttt{Int}\rightarrow\texttt{Int}\rangle)(\ 42\ ) \longrightarrow (\texttt{foo}(42\langle\texttt{Int}\rangle))\langle\texttt{Int}\rangle$$

The cast language

Syntax:

$$
\begin{array}{llllllllllllll}
\textit{Types} & T & ::= & \texttt{Int} & \mid & \texttt{Bool} & \mid & T \to T & \mid & ? & & & & \\
\textit{Terms} & a, b & ::= & x & \mid & a\,b & \mid & \lambda x{:}T.a & \mid & a\langle T\rangle & \mid & 1 & \mid & 2 \mid \ldots \\
\textit{Values} & v & ::= & \lambda x{:}T.a & \mid & 1 & \mid & 2 & \mid & \ldots & & & &
\end{array}
$$

Typing

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

$$
[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a\langle T\rangle : T}
$$

Semantics:

$$
\begin{array}{rcll}
(\lambda x{:}T.a)v & \longrightarrow & a[v/x] & \\
v\langle T\rangle & \longrightarrow & v & \text{if } T \neq S_1 \to S_2 \text{ and } \vdash v : T \\
v\langle T\rangle & \longrightarrow & \text{Fail} & \text{if } T \neq S_1 \to S_2 \text{ and } \not\vdash v : T \\
(v_1\langle S \to T\rangle)v_2 & \longrightarrow & (v_1(v_2\langle S\rangle))\langle T\rangle &
\end{array}
$$

The cast language

The cast language is sound:

Theorem (Soundness)

For every term a of the cast language, if $\Gamma \vdash a : T$, then

- either a reduces to a value of type T
- or a diverges
- or a reduces to Fail

[no stuck term]

What are the consenquences of this theorem on our initial language?
How does it fit our framework? Let me first add a further bit

Tracking errors

The message Fail is not very useful for debugging

Tracking errors

The message Fail is not very useful for debugging

We can modify compilation to track the origine of failures:

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle^{\ell}}$$

where ℓ is a pointer to the source code of a

Tracking errors

The message Fail is not very useful for debugging

We can modify compilation to track the origine of failures:

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle^{\ell}}$$

where ℓ is a pointer to the source code of a

Then it suffices to change the semantics of the cast language to return this pointer:

Semantics:

$$\begin{array}{rcll}
(\lambda x{:}T.a)v & \longrightarrow & a[v/x] & \\
v\langle T\rangle^{\ell} & \longrightarrow & v & \text{if } T \neq S_1 \to S_2 \text{ and } \vdash v : T \\
v\langle T\rangle^{\ell} & \longrightarrow & \text{blame } \ell & \text{if } T \neq S_1 \to S_2 \text{ and } \not\vdash v : T \\
(v_1\langle S \to T\rangle^{\ell})v_2 & \longrightarrow & (v_1(v_2\langle S\rangle^{\ell}))\langle T\rangle^{\ell} &
\end{array}$$

Outline

Criterion: Type Soundness

Every expression must only result in values whose type agrees with the static type of the expression.

Criterion: Type Soundness

Every expression must only result in values whose type agrees with the static type of the expression.

Theorem (Soundness)

If $\Gamma \vdash a : T$, then $\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'$ and

- either a' reduces to a value of type T
- or a' diverges
- or a' fails for a cast on a dynamic type

Criterion: Type Soundness

Every expression must only result in values whose type agrees with the static type of the expression.

Theorem (Soundness)

If $\Gamma \vdash a : T$, then $\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'$ and

- either a' reduces to a value of type T
- or a' diverges
- or a' fails for a cast on a dynamic type

A Corollary of the soundness of the cast calculus and of the following lemma of type preservation.

Lemma. If $\Gamma \vdash a : T$ then then $\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'$ and $\Gamma \vdash a' : S \sqsubseteq T$

Criterion: Blame Tracking

When a runtime type error occurs, it is never the fault of a statically typed region of code.

Criterion: Blame Tracking

When a runtime type error occurs, it is never the fault of a statically typed region of code.

Theorem (Blame Theorem)

Let $C[a]$ be a program such that **?** does not occur in a.
If $\Gamma \vdash C[a] : T \overset{\text{compiles}}{\dashrightarrow} b$ and $b \longrightarrow \text{blame } \ell$, then $\ell \in C[]$ and $\ell \notin a$.

Criterion: Gradual Guarantee

Using less precise types must not change the outcome of type checking or of running a program.

Criterion: Gradual Guarantee

Using less precise types must not change the outcome of type checking or of running a program.

An expression a is *less precise* than b, written $a \sqsubseteq b$, if a is b but with less precise annotations.

Note: a dynamically typed version of a is where all annotations are $?$: it is a minimal element in the precision lattice.

Criterion: Gradual Guarantee

Using less precise types must not change the outcome of type checking or of running a program.

An expression a is *less precise* than b, written $a \sqsubseteq b$, if a is b but with less precise annotations.

Note: a dynamically typed version of a is where all annotations are **?**: it is a minimal element in the precision lattice.

Theorem (Gradual Guarantee)

If $\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'$ and $b \sqsubseteq a$, then:

- $\Gamma \vdash b : T' \overset{\text{compiles}}{\dashrightarrow} b'$ and $T' \sqsubseteq T$
- if $a' \longrightarrow v$, then $b' \longrightarrow v'$ and $v' \sqsubseteq v$.

Outline

A hint to efficient implementation

A gradually typed tail-recursive function:

```
let rec odd : Int -> ? = fun n ->
    if n = 0 then false
    else (even (n-1))
and even : Int -> Bool = fun n ->
    if n = 0 then true
    else (odd (n-1))
```

A hint to efficient implementation

A gradually typed tail-recursive function: In Siek&Taha it is compiled into:

```
let rec odd : Int -> ? = fun n ->
    if n = 0 then false<?>
    else (even (n-1))<?>
and even : Int -> Bool = fun n ->
    if n = 0 then true
    else (odd (n-1))<Bool>
```

A hint to efficient implementation

A gradually typed tail-recursive function:

```
let rec odd : Int -> ? = fun n ->
    if n = 0 then false<?>
    else (even (n-1))<?>
and even : Int -> Bool = fun n ->
    if n = 0 then true
    else (odd (n-1))<Bool>
```

It produces accumulation of casts:

```
odd 5  ⟶  (even 4)<?>
       ⟶  (odd 3)<Bool><?>
       ⟶  (even 2)<?><Bool><?>
       ⟶  (odd 1)<Bool><?><Bool><?>
       ⟶  (even 0)<?><Bool><?><Bool><?>
```

A hint to efficient implementation

A gradually typed tail-recursive function:

```
let rec odd : Int -> ? = fun n ->
    if n = 0 then false<?>
    else (even (n-1))<?>
and even : Int -> Bool = fun n ->
    if n = 0 then true
    else (odd (n-1))<Bool>
```

It produces accumulation of casts:

```
odd 5  ⟶  (even 4)<?>
       ⟶  (odd 3)<Bool><?>
       ⟶  (even 2)<?><Bool><?>
       ⟶  (odd 1)<Bool><?><Bool><?>
       ⟶  (even 0)<?><Bool><?><Bool><?>
```

Solution: specific implementation of tail-recursion combine with cast compression via intersection types:

$E\langle\tau\rangle\langle\tau'\rangle$ **can be "compressed" to** $E\langle\tau\wedge\tau'\rangle$**.**

HM Polymorphism + Gradual Typing

Syntax:

Types	T	$::=$	$\texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \alpha \mid ?$
Schemas	σ	$::=$	$T \mid \forall\alpha.\sigma$
Terms	a, b	$::=$	$x \mid ab \mid \lambda x.a \mid \texttt{let}\ x = a\ \texttt{in}\ b \mid 1 \mid 2 \mid \ldots$

Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Typing

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

$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x\ =\ a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T}$$

HM Polymorphism + Gradual Typing + Subtyping

Syntax:

$$\begin{array}{llll} \textit{Types} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \alpha \mid ? \\ \textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \\ \textit{Terms} & a,b & ::= & x \mid ab \mid \lambda x.a \mid \texttt{let } x = a \texttt{ in } b \mid 1 \mid 2 \mid \ldots \end{array}$$

Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T \rangle}$$

Typing

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

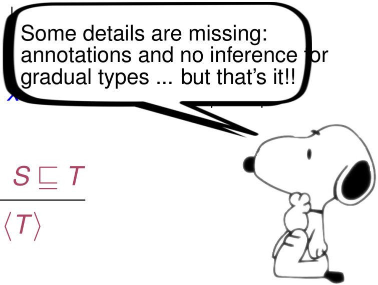
$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let } x = a \texttt{ in } b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

HM Polymorphism + Gradual Typing + Subtyping

Syntax:

Types	T	$::=$	$\mathtt{Int} \mid \mathtt{Bool} \mid T \to T \mid$
Schemas	σ	$::=$	$T \mid \forall\alpha.\sigma$
Terms	a, b	$::=$	$x \mid ab \mid \lambda x.a \mid \mathtt{let}\ x$



Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Typing

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

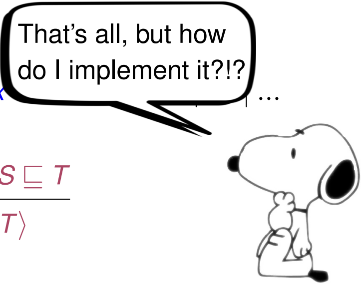
$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \mathtt{let}\ x\ \mathtt{=}\ a\ \mathtt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

HM Polymorphism + Gradual Typing + Subtyping

Syntax:

Types	T	$::=$	$\texttt{Int} \mid \texttt{Bool} \mid T \to T \mid$
Schemas	σ	$::=$	$T \mid \forall\alpha.\sigma$
Terms	a, b	$::=$	$x \mid ab \mid \lambda x.a \mid \texttt{let}\ x$ ... $\mid \ldots$



Semantics:

$$[\textsc{Materialize}_{\textsc{Compil}}]\ \frac{\Gamma \vdash a : S \overset{\text{compiles}}{\dashrightarrow} a' \qquad S \sqsubseteq T}{\Gamma \vdash a : T \overset{\text{compiles}}{\dashrightarrow} a'\langle T\rangle}$$

Typing

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

$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : T \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.T} \qquad \frac{\Gamma \vdash a : \forall\alpha.T}{\Gamma \vdash a : T[S/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : S \qquad S \sqsubseteq T}{\Gamma \vdash a : T} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : S \qquad S \leq T}{\Gamma \vdash a : T}$$

The missing details

Syntax:

$$
\begin{array}{llcl}
\textit{StaticTypes} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid \alpha \\
\textit{GradualTypes} & \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau \to \tau \mid \alpha \mid ? \\
\textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \\
\textit{Terms} & a, b & ::= & x \mid ab \mid \lambda x.a \mid \lambda x:\tau.a \mid \texttt{let}\ x = a\ \texttt{in}\ b \mid 1 \mid 2
\end{array}
$$

Typing

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash a : \tau' \to \tau \qquad \Gamma \vdash b : \tau'}{\Gamma \vdash ab : \tau}$$

$$\frac{\Gamma, x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda x : \tau.a : \tau \to \tau'} \qquad \frac{\Gamma, x : S \vdash a : \tau}{\Gamma \vdash \lambda x.a : S \to \tau}$$

$$\frac{\Gamma \vdash a : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \texttt{let}\ x = a\ \texttt{in}\ b : \sigma_2} \qquad \frac{\Gamma \vdash a : \tau \quad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash a : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash a : \forall\alpha.\tau}{\Gamma \vdash a : \tau[\tau'/\alpha]}$$

$$[\textsc{Materialize}]\ \frac{\Gamma \vdash a : \tau' \qquad \tau' \sqsubseteq \tau}{\Gamma \vdash a : \tau} \qquad [\textsc{Subsum}]\ \frac{\Gamma \vdash a : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash a : \tau}$$

We generate sets D of *type constraints*

$$D ::= \varnothing \mid (t_1 \mathrel{\dot{\leq}} t_2) \cup D \mid (\tau \mathrel{\dot{\sqsubseteq}} \alpha) \cup D$$

Then we find a type substitution θ that *solves* D that is

- for all $(t_1 \mathrel{\dot{\leq}} t_2)$ we have $t_1\theta = t_2\theta$
- for all $(\tau \mathrel{\dot{\sqsubseteq}} \alpha)$ we have $\tau\theta \sqsubseteq \alpha\theta$ and $\tau\theta$ is a static type

Constraint generation

We do not directly generate *type constraint*.
We first *generate structured constraints* of the form[1]:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (\tau \mathrel{\dot{\sqsubseteq}} \alpha) \mid (x \mathrel{\dot{\sqsubseteq}} \alpha) \mid \textbf{def}\ \ x : \tau\ \textbf{in}\ C \mid \exists\vec{\alpha}.C \mid C \wedge C$$

[1] Let constraints are omitted for the sake of simplicity

Constraint generation

We do not directly generate *type constraint*.
We first *generate structured constraints* of the form[1]:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (\tau \mathrel{\dot{\sqsubseteq}} \alpha) \mid (x \mathrel{\dot{\sqsubseteq}} \alpha) \mid \textbf{def}\ \ x : \tau\ \textbf{in}\ C \mid \exists\vec{\alpha}.C \mid C \wedge C$$

$$\begin{array}{rcl}
\langle\langle x : t\rangle\rangle & = & \exists\alpha.\,(x \mathrel{\dot{\sqsubseteq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t)\\
\langle\langle(\lambda x.e) : t\rangle\rangle & = & \exists\alpha_1,\alpha_2.\,(\textbf{def}\ \ x : \alpha_1\ \textbf{in}\ \langle\langle e : \alpha_2\rangle\rangle) \wedge (\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\\
(\lambda x : \tau.e) : t\rangle\rangle & = & \exists\alpha_1,\alpha_2.\,(\textbf{def}\ \ x : \tau\ \textbf{in}\ \langle\langle e : \alpha_2\rangle\rangle) \wedge (\tau \mathrel{\dot{\sqsubseteq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t)\\
\langle\langle e_1 e_2 : t\rangle\rangle & = & \exists\alpha.\langle\langle e_1 : \alpha \to t\rangle\rangle \wedge \langle\langle e_2 : \alpha\rangle\rangle
\end{array}$$

[1] Let constraints are omitted for the sake of simplicity

Constraint generation

We do not directly generate *type constraint*.
We first *generate structured constraints* of the form[1]:

$$C ::= (t \mathrel{\dot{\leq}} t) \mid (\tau \mathrel{\dot{\sqsubseteq}} \alpha) \mid (x \mathrel{\dot{\sqsubseteq}} \alpha) \mid \textbf{def}\ \ x : \tau \ \textbf{in}\ C \mid \exists\vec{\alpha}.C \mid C \wedge C$$

$$\begin{array}{rcl}
\langle\!\langle x : t \rangle\!\rangle & = & \exists\alpha.\,(x \mathrel{\dot{\sqsubseteq}} \alpha) \wedge (\alpha \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x.e) : t \rangle\!\rangle & = & \exists\alpha_1,\alpha_2.\,(\textbf{def}\ \ x : \alpha_1 \ \textbf{in}\ \langle\!\langle e : \alpha_2 \rangle\!\rangle) \wedge (\alpha_1 \mathrel{\dot{\sqsubseteq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) \\
\langle\!\langle (\lambda x : \tau.e) : t \rangle\!\rangle & = & \exists\alpha_1,\alpha_2.\,(\textbf{def}\ \ x : \tau \ \textbf{in}\ \langle\!\langle e : \alpha_2 \rangle\!\rangle) \wedge (\tau \mathrel{\dot{\sqsubseteq}} \alpha_1) \wedge (\alpha_1 \to \alpha_2 \mathrel{\dot{\leq}} t) \\
\langle\!\langle e_1 e_2 : t \rangle\!\rangle & = & \exists\alpha.\langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 : \alpha \rangle\!\rangle
\end{array}$$

Note that $\langle\!\langle (\lambda x : ?.x) : \texttt{Int} \to \texttt{Int} \rangle\!\rangle$ *can be solved*,
whereas $\langle\!\langle (\lambda x.x) : ? \to ? \rangle\!\rangle$ *cannot*.

[1] Let constraints are omitted for the sake of simplicity

Rewriting constraints

We then *rewrite the structured constraints* to obtain a set D of *type constraints*:

Rewriting constraints

We then *rewrite the structured constraints* to obtain a set D of *type constraints*:

$$\frac{}{\Gamma \vdash (x \mathrel{\dot{\sqsubseteq}} \alpha) \rightsquigarrow \{\tau[\vec{\alpha}{:=}\vec{\beta}] \mathrel{\dot{\sqsubseteq}} \alpha\}} \quad \begin{array}{l}\Gamma(x) = \forall\vec{\alpha}.\tau \\ \vec{\beta} \text{ FRESH}\end{array}$$

Rewriting constraints

We then *rewrite the structured constraints* to obtain a set D of *type constraints*:

$$\frac{}{\Gamma \vdash (x \mathrel{\dot\sqsubseteq} \alpha) \rightsquigarrow \{\tau[\vec\alpha{:=}\vec\beta] \mathrel{\dot\sqsubseteq} \alpha\}} \quad \begin{array}{l}\Gamma(x) = \forall\vec\alpha.\tau \\ \vec\beta \text{ FRESH}\end{array}$$

$$\frac{(\Gamma, x : \tau) \vdash C \rightsquigarrow D}{\Gamma \vdash \textbf{def } x : \tau \textbf{ in } C \rightsquigarrow D}$$

$$\frac{\Gamma \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma \vdash C_2 \rightsquigarrow D_2}{\Gamma \vdash C_1 \wedge C_2 \rightsquigarrow D_1 \cup D_2}$$

Solving constraints

Everything is finally solved using **standard unification**:

(1) we *replace every occurence* of **?** in materialization constraints by a *distinct fresh type variable*;

(2) we **unify**;

(3) we replace every *residual* fresh type variable *back* to **?**.

Solving constraints

Everything is finally solved using **standard unification**:

(1) we *replace every occurence* of ? in materialization constraints by a *distinct fresh type variable*;

(2) we **unify**;

(3) we replace every *residual* fresh type variable *back* to ?.

For example, the constraint

$$? \to ? \to ? \mathrel{\dot\sqsubseteq} \texttt{Bool} \to \alpha$$

Solving constraints

Everything is finally solved using **standard unification**:

(1) we *replace every occurence* of ? in materialization constraints by a *distinct fresh type variable*;

(2) we **unify**;

(3) we replace every *residual* fresh type variable *back* to ?.

For example, the constraint

$$? \to ? \to ? \mathrel{\dot{\sqsubseteq}} \texttt{Bool} \to \alpha$$

will become

$$X_1 \to X_2 \to X_3 \mathrel{\dot{\sqsubseteq}} \texttt{Bool} \to \alpha$$

Solving constraints

Everything is finally solved using **standard unification**:

(1) we *replace every occurence* of ? in materialization constraints by a *distinct fresh type variable*;

(2) we **unify**;

(3) we replace every *residual* fresh type variable *back* to ?.

For example, the constraint

$$? \to ? \to ? \mathrel{\dot{\sqsubseteq}} \texttt{Bool} \to \alpha$$

will become

$$X_1 \to X_2 \to X_3 \mathrel{\dot{\sqsubseteq}} \texttt{Bool} \to \alpha$$

and solving it will return the unifier

$$\theta : X_1 \mapsto \texttt{Bool}; X_2 \mapsto \beta; X_3 \mapsto \gamma; \alpha \mapsto (\beta \to \gamma)$$

Solving constraints

Everything is finally solved using **standard unification**:

(1) we *replace every occurence* of **?** in materialization constraints by a *distinct fresh type variable*;

(2) we **unify**;

(3) we replace every *residual* fresh type variable *back* to **?**.

For example, the constraint

$$? \to ? \to ? \mathrel{\dot{\sqsubseteq}} \mathtt{Bool} \to \alpha$$

will become

$$X_1 \to X_2 \to X_3 \mathrel{\dot{\sqsubseteq}} \mathtt{Bool} \to \alpha$$

and solving it will return the unifier

$$\theta : X_1 \mapsto \mathtt{Bool}; X_2 \mapsto \beta; X_3 \mapsto \gamma; \alpha \mapsto (\beta \to \gamma)$$

The application of $e_1 : (\mathtt{Bool} \to \alpha) \to \alpha$ to $e_2 : ? \to ? \to ?$ has thus type $? \to ?$

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma \vdash \langle\langle e : t \rangle\rangle \rightsquigarrow D$, we can *compute a unifier* θ satisfying $\mathcal{D}$.

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma \vdash \langle\langle e : t \rangle\rangle \rightsquigarrow D$, we can *compute a unifier* θ satisfying $\mathcal{D}$.

This derivation and the associated unifier *can be used to compile e* in a straightforward way: **to *every materialization constraint* introduced in $\mathcal{D}$ *corresponds a cast*.**

For instance

if $\mathcal{D} = \Gamma; \vdash \langle\langle x : t \rangle\rangle \rightsquigarrow \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$ and θ is a solution for $\{(\tau \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$ then

Compilation and Results

To summarize, given an expression e, and a constraint derivation $\mathcal{D}$ of $\Gamma \vdash \langle\langle e : t \rangle\rangle \rightsquigarrow D$, we can *compute a unifier* θ satisfying $\mathcal{D}$.

This derivation and the associated unifier *can be used to compile e* in a straightforward way: **to *every materialization constraint* introduced in $\mathcal{D}$ *corresponds a cast*.**

For instance

if $\mathcal{D} = \Gamma; \vdash \langle\langle x : t \rangle\rangle \rightsquigarrow \{(\tau \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$ and θ is a solution for $\{(\tau \mathrel{\dot{\sqsubseteq}} \alpha), (\alpha \mathrel{\dot{\leq}} t)\}$ then

$$\mathcal{D}; \theta \vdash x \overset{\text{compiles}}{\dashrightarrow} x\langle \alpha\theta \rangle$$

Inference (and compilation) for this system is *sound*, *type-preserving* and *complete* w.r.t. the declarative system.

Part 2: Adding subtyping

We saw that, declaratively, *adding subtyping* is just a matter of adding *one subsumption rule*.

Part 2: Adding subtyping

We saw that, declaratively, *adding subtyping* is just a matter of adding *one subsumption rule*.

Constraint generation is also unchanged, unification constraints just become *subtyping constraints.*

Part 2: Adding subtyping

We saw that, declaratively, *adding subtyping* is just a matter of adding *one subsumption rule*.

Constraint generation is also unchanged, unification constraints just become *subtyping constraints.*

However, to *solve constraints* such as $\{(\alpha \dot{\leq} t_1), (\alpha \dot{\leq} t_2)\}$ we have to compute *greatest lower bounds*.

Part 2: Adding subtyping

We saw that, declaratively, *adding subtyping* is just a matter of adding *one subsumption rule*.

Constraint generation is also unchanged, unification constraints just become *subtyping constraints.*

However, to *solve constraints* such as $\{(\alpha \dot{\leq} t_1), (\alpha \dot{\leq} t_2)\}$ we have to compute *greatest lower bounds*.

For example,

```
fun x -> if (fst x) then (1 + snd x) else x
```

should be of type $(\texttt{Bool}\times\texttt{Int}) \rightarrow (\ \texttt{Int} \mid (\texttt{Bool}\times\texttt{Int})\ )$

Part 3: Adding Set-Theoretic Types

The types become:

$$
\begin{array}{llll}
\textit{StaticTypes} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid T \vee T \mid \neg T \mid \texttt{Any} \mid \alpha \\
\textit{GradualTypes} & \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau \to \tau \mid \alpha \mid ? \\
\textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma
\end{array}
$$

Constraints are *unchanged*. However, the inference algorithm is now based on the *tallying algorithm* of Castagna et al. [2015], rather than unification (but the principle is the same).

$$\{(\alpha \mathrel{\dot{\leq}} t_1), (\alpha \mathrel{\dot{\leq}} t_2)\} \rightsquigarrow \{(\alpha \mathrel{\dot{\leq}} t_1 \wedge t_2)\}$$

Part 3: Adding Set-Theoretic Types

The types become:

$$\begin{array}{llll} \textit{StaticTypes} & T & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid T \vee T \mid \neg T \mid \texttt{Any} \mid \alpha \\ \textit{GradualTypes} & \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau \to \tau \mid \alpha \mid ? \\ \textit{Schemas} & \sigma & ::= & T \mid \forall\alpha.\sigma \end{array}$$

Constraints are *unchanged*. However, the inference algorithm is now based on the *tallying algorithm* of Castagna et al. [2015], rather than unification (but the principle is the same).

$$\{(\alpha \dot{\leq} t_1), (\alpha \dot{\leq} t_2)\} \rightsquigarrow \{(\alpha \dot{\leq} t_1 \wedge t_2)\}$$

Soundness still holds for the inference algorithm, but *completeness no longer holds*.

Outline

To go further

Some starting points:

- **Objects:** Siek & Taha (ECOOP 2007)
- **Type inference:** Siek & Vachharajani (DLS 2008), Garcia & Cimini (POPL 2015) [both superseded by Castagna & al (POPL 2019)]
- **Occurrence Typing:** Tobin-Hochstadt & Felleisen (POPL 2008)
- **Foundational approach:** Garcia & Clark & Tanter (POPL 2016)
- **Gradual Guarantees:** Siek& Vitousek & Cimini & Boyland (SNAPL 2015)
- **Second order parametric polymorphism:** Igarashi et al. (ICFP 2017), Xie & Bi & Oliveira (ESOP 2018)
- **Union and intersection types:** Castagna & Lanvin (ICFP 2017)
- **Implementation aspects:** Takikawa et al. (POPL 2016), Bauman et al. (OOPSLA 2017), Kuhlenschmidt et al. (PLDI 2019), Castagna & Duboc & Lanvin & Siek (IFL 2019)
- **Type inference, subtyping, union and intersection types:** Castagna & Lanvin & Petrucciani & Siek (POPL 2019) **The full monty!**