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**

- **Gradual Typing (dynamic type polymorphism)**
  Main ideas - Formal system - Algorithmic Aspects - Criteria for Gradual Typing - Implementation issues - References
Background and Motivations
1. Polymorphism

2. Motivating Examples

3. A Refresher Course on Operational Semantics
Outline

1. Polymorphism
2. Motivating Examples
3. A Refresher Course on Operational Semantics
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 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 `+` for string concatenation...)

In this course I focus on functions.

G. Castagna (CNRS)
What is polymorphism?

<table>
<thead>
<tr>
<th>Merriam-Webster Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>The quality or state of existing in or assuming different forms</td>
</tr>
</tbody>
</table>

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 \( \text{int} \rightarrow \text{int} \) and of type \( \text{bool} \rightarrow \text{bool} \))
- 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 \( \text{int} \) and stacks of \( \text{bool} \))
- 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 \(\text{int} \rightarrow \text{int}\) and of type \(\text{bool} \rightarrow \text{bool}\))
- 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 \(\text{int}\) and stacks of \(\text{bool}\))
- 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

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
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

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.

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/Unknown 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.
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:

G. Castagna (CNRS)
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 forms 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/Unknown 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/Unknown 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. Polymorphism
2. Motivating Examples
3. A Refresher Course on Operational Semantics
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 $S \times T \to S$ and returns a result of type $S$, whatever types $S$ and $T$ 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

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

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

Intuition

Add type variables and quantify them universally:

$$\forall \alpha, \beta . \, \alpha \times \beta \rightarrow \alpha$$
2. Subtyping polymorphism

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

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

It can be applied to objects with the property `length` 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

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

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

Intuition

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

```
{ length: number } → number
```

Accepts arguments of any type \( T \leq \{ \text{length: number} \} \)

(e.g. \( \{ \text{length: number, concat: string} \rightarrow \text{string} \} \))
function size (x) {
    return x.length;
}

Subtyping + Parametric

Possibility two combine the two form of polymorphism

∀α.{ length : α } → α
function size (x) {
    return x.length;
}

Subtyping + Parametric

Possibility two combine the two form of polymorphism

\forall \alpha. \{ \text{length : } \alpha \} \to \alpha

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

Bounded parametric

\forall \alpha \leq \{ \text{length : 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

```javascript
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

```javascript
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

```javascript
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

  \((\text{number}|\text{string})\rightarrow(\text{number}|\text{string})\)
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

```javascript
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

  \((\text{number} | \text{string}) \rightarrow (\text{number} | \text{string})\)
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

```javascript
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
  
  \[(\text{number}|\text{string}) \rightarrow (\text{number}|\text{string})\]

- Better solution: intersection types
  
  \[(\text{number} \rightarrow \text{number}) \& (\text{string} \rightarrow \text{string})\]
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

```javascript
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**
  
  \[(\text{number}|\text{string}) \rightarrow (\text{number}|\text{string})\]

- **Better solution: intersection types**
  
  \[(\text{number} \rightarrow \text{number}) \& (\text{string} \rightarrow \text{string})\]

needs some form of occurrence typing
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

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

Set-theoretic + Subtyping

\[
\begin{align*}
\text{(number} \rightarrow \text{number}) & \quad \& \\
\text{(not(number)} & \quad \& \quad \{\text{concat: string} \rightarrow \text{string}\}) \rightarrow \text{string}
\end{align*}
\]

Actually, set-theoretic types are defined by subtyping

Set-theoretic + Parametric

\[
\forall \alpha, \beta. \quad \begin{align*}
\text{(number} \rightarrow \text{number}) & \quad \& \\
\text{(\alpha} & \quad \& \quad \text{not(number)} & \quad \& \quad \{\text{concat: } \alpha \rightarrow \beta\}) \rightarrow \beta
\end{align*}
\]
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)

a sophisticated way to write bounded polymorphism and recursive types:

∀β, ∀(γ ≤ not(number) & µX.{concat: X → β}).
( number → number ) & ( γ → β)
4. Dynamic types

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

```javascript
function double (x) {
    ( typeof(x) === "number" ) ? 2*x : x.concat(x)
}
```
Functions that *for some specific arguments* delay the check of types at run-time

```javascript
function double (x) {
    (<some twisted condition>) ? 2*x : x.concat(x)
}
```
Functions that *for some specific arguments* delay the check of types at run-time

```javascript
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

```javascript
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

```javascript
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:

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

Accept as is:

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

Intuitively the function has type:

```
???
→ (?, number)
```

Accept and insert checks:

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

Compile as

```javascript
function double (x : ?) {
    (<condition>) ? 2*(x⟨number⟩) : (x⟨string⟩).concat(x⟨string⟩)
}
```
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: ? → (number | ?)
Reject at compile time:

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

Accept as is:

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

Intuitively the function has type:  

```
? → (number | ?)
```

Accept and insert checks:

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

Compile as

```javascript
function double (x : ?) {
    (<condition>) ? 2*(x<number>) : (x<string>).concat(x<string>)
}
```
Combined usage: all 4 together! (OCaml style)

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

Type:
\[
\text{Type: } \text{bool} \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha^\text{array} | \alpha^\text{list} & ???) \rightarrow (\beta^\text{array} | \beta^\text{list})
\]

Compiled as:
```ocaml
let mymap (condition) (f) (x : (\alpha^\text{array} | \alpha^\text{list} & ?)) =
  if condition then Array.map f (x \langle \alpha^\text{array} \rangle)
  else List.map f (x \langle \alpha^\text{list} \rangle)
```

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

G. Castagna (CNRS)
Four Forms of Polymorphism
let mymap (condition) (f) (x : ?) =
  if condition then Array.map f x else List.map f x

Type: bool \rightarrow (\alpha \rightarrow \beta) \rightarrow ? \rightarrow ?
Combined usage: all 4 together! (OCaml style)

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

Type: bool → (α → β) → ? → ?
  • x can be bound to anything (though only αlist or αarray work)
  • no information on the type of the result (though only βlist or βarray are possible)

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⟩)

Type: bool → (α → β) → ( (α array | α list) & ? ) → (β array | β list)
```
Combined usage: all 4 together! (OCaml style)

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

Type: `bool → (α → β) → ? → ?`
- `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)

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

Type: `bool → (α → β) → ((α array | α list) & ?) → (β array | β list)`

Compiled as:

```ocaml
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)

```ocaml
define mymap (condition) f x =
  if condition then Array.map f x else List.map f x

Type: bool → (α → β) → ? → ?
  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)
```

```ocaml
define mymap (condition) f x =
  if condition then Array.map f x else List.map f x

Type: bool → (α → β) → (α array | α list) & ? → (β array | β list)
```

Compiled as:

```ocaml
define mymap (condition) f x =
  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

1. Polymorphism
2. Motivating Examples
3. A Refresher Course on Operational Semantics
Syntax and small-step semantics

### Syntax

**Terms**

\[ a, b ::=} N \quad \text{Numeric constant} \\
| \quad x \quad \text{Variable} \\
| \quad ab \quad \text{Application} \\
| \quad \lambda x.a \quad \text{Abstraction} \\

**Values**

\[ v ::=} \lambda x.a \mid N \]
Syntax and small-step semantics

**Syntax**

Terms:  
\[ a, b ::= N \quad \text{Numeric constant} \]
\[ \mid x \quad \text{Variable} \]
\[ \mid ab \quad \text{Application} \]
\[ \mid \lambda x.a \quad \text{Abstraction} \]

Values:  
\[ v ::= \lambda x.a \mid N \]

**Small step semantics for strict functional languages**

Evaluation Contexts:  
\[ E ::= \[ \] \mid E a \mid v E \]

\[ \text{Beta}_v \]
\[ (\lambda x.a) v \rightarrow a[v/x] \]

\[ \text{Context} \]
\[ a \rightarrow b \]
\[ E[a] \rightarrow E[b] \]
Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under $\lambda$-abstractions;

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

Left-most reduction: In an application $ab$, 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 $a \rightarrow b$. 
Strategy and big-step semantics

Characteristics of the reduction strategy

Weak reduction: We cannot reduce under \( \lambda \)-abstractions;

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

Left-most reduction: In an application \( ab \), 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 \( a \rightarrow b \).

Big step semantics for strict functional languages

\[
N \Rightarrow N \quad \lambda x. a \Rightarrow \lambda x. a \quad \begin{align*}
    a &\Rightarrow \lambda x. c & b &\Rightarrow v & c[v_/x] &\Rightarrow v \\
    &\Rightarrow v
\end{align*}
\]
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* $e$

\[
\begin{align*}
\frac{e(x) = v}{e \vdash x \Rightarrow v} & & e \vdash N \Rightarrow N & & e \vdash \lambda x.a \Rightarrow \lambda x.a \\
\frac{e \vdash a \Rightarrow \lambda x.c & & e \vdash b \Rightarrow v & & e; x \mapsto v \vdash c \Rightarrow v}{e \vdash ab \Rightarrow v}
\end{align*}
\]
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* $e$

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

$$
e \vdash a \Rightarrow \lambda x.c \quad e \vdash b \Rightarrow v_0 \quad e; x \mapsto v_0 \vdash c \Rightarrow v
$$

$$
e \vdash ab \Rightarrow v
$$

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

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

$$
e \vdash a \Rightarrow \lambda x. c \quad e \vdash b \Rightarrow v \circ \quad e; x \mapsto v \circ \vdash c \Rightarrow v$$

$$\frac{\quad}{e \vdash ab \Rightarrow v}$$

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

In the environment used to evaluate $f \ 2$ the variable $x$ is bound to 1.
Try to evaluate

```plaintext
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 `(λx.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 \( (\lambda x. a)[e] \) (i.e., the function and the *environment of its definition*).

### Big step semantics with environments and closures

<table>
<thead>
<tr>
<th>Values</th>
<th>( v ::= N \mid (\lambda x. a)[e] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environments</td>
<td>( e ::= x_1 \mapsto v_1 ; \ldots ; x_n \mapsto v_n )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
  e(x) &= v \\
  e \vdash N &\Rightarrow N \\
  e \vdash \lambda x. a &\Rightarrow (\lambda x. a)[e]
\end{align*}
\]

\[
\begin{align*}
  e \vdash a &\Rightarrow (\lambda x. c)[e_\circ] \\
  e \vdash b &\Rightarrow v_\circ \\
  e_\circ ; x \mapsto v_\circ \vdash c &\Rightarrow v \\
  e \vdash ab &\Rightarrow v
\end{align*}
\]
De Bruijn indexes

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

$$\lambda x. (\lambda y. y x) x \quad \text{is} \quad \lambda. (\lambda.01)0$$

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

Terms

$$a, b ::= N \mid n \mid \lambda.a \mid ab$$

Values

$$v ::= N \mid (\lambda.a)[e]$$

Environments

$$e ::= v_0; v_1; \ldots; v_n$$

$$\begin{ aligned} 
 e &= v_0; \ldots; v_n; \ldots; v_m \\
 e &\vdash N \Rightarrow N \\
 e &\vdash \lambda.a \Rightarrow (\lambda.a)[e] \\
 e &\vdash a \Rightarrow (\lambda.c)[e_\circ] \\
 e &\vdash b \Rightarrow v_\circ \\
 v_\circ; e_\circ &\vdash c \Rightarrow v \\
 e &\vdash ab \Rightarrow v 
\end{aligned}$$
The canonical, efficient interpreter

```ocaml
# 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
    | 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

4  Simple Types

5  Recursive Types

6  Bibliography
Outline

4 Simple Types

5 Recursive Types

6 Bibliography
Simply Typed $\lambda$-calculus

Syntax

Types $T ::= T \rightarrow T$ \quad function types

\quad $\text{Bool} | \text{Int} | \text{Real} | \ldots$ \quad basic types

Terms $a, b ::= \text{true} | \text{false} | 1 | 2 | \ldots$ \quad constants

\quad $x$ \quad variable

\quad $ab$ \quad application

\quad $\lambda x:T.a$ \quad abstraction

Reduction

Contexts $C[] ::= [] | a[] | []a | \lambda x:T.[[]$

$\beta$-ETA

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

$C[a] \rightarrow C[b]$
### Type system

#### Typing

\[
\begin{align*}
\text{VAR} & : & \Gamma \vdash x : \Gamma(x) \\
\text{\textsc{intro}} & : & \Gamma, x : S \vdash a : T \\
& & \Gamma \vdash \lambda x : S. a : S \rightarrow T \\
\text{\textsc{elim}} & : & \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
& & \Gamma \vdash ab : T
\end{align*}
\]

(plus the typing rules for constants.)
Typing

\[\text{VAR} \quad \Gamma \vdash x : \Gamma(x)\]

\[\text{\textbf{\textsc{Intro}}} \quad \Gamma, x : S \vdash a : T \quad \Rightarrow \quad \Gamma \vdash \lambda x : S. a : S \to T\]

\[\text{\textbf{\textsc{Elim}}} \quad \Gamma \vdash a : S \to T \quad \Gamma \vdash b : S \quad \Rightarrow \quad \Gamma \vdash ab : T\]

(plus the typing rules for constants).

Theorem (Subject Reduction)

If \(\Gamma \vdash a : T\) and \(a \rightarrow^* b\), then \(\Gamma \vdash b : T\).
Type system

Typing

\[ \text{VAR} \]
\[ \Gamma \vdash x : \Gamma(x) \]

\[ \rightarrow \text{INTRO} \]
\[ \Gamma, x : S \vdash a : T \]
\[ \Gamma \vdash \lambda x : S. a : S \rightarrow T \]

\[ \rightarrow \text{ELIM} \]
\[ \Gamma \vdash a : S \rightarrow T \]
\[ \Gamma \vdash b : S \]
\[ \Gamma \vdash ab : T \]

(plus the typing rules for constants).

Theorem (Subject Reduction)

*If* \( \Gamma \vdash a : T \) *and* \( a \rightarrow^* b \), *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* \( \emptyset \vdash a : T \) *and* \( a \not\rightarrow \), *then* \( a \) *is a value*

where a value is either a constant or a lambda abstraction

\[ v ::= \lambda x : T. a \mid \text{true} \mid \text{false} \mid 1 \mid 2 \mid ... \]
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
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 \textit{syntax directed} and satisfies the \textit{subformula property}. As such it describes a deterministic algorithm.

\begin{verbatim}
let rec typecheck gamma = function
  | x -> gamma(x)         (* Var rule *)
  | \x:T.a -> T \rightarrow (typecheck (gamma, x:T) a) (* Intro rule *)
  | ab -> let T_1\rightarrow T_2 = typecheck gamma a in
            let T_3 = typecheck gamma b in
            if T_1==T_3 then T_2 else fail
\end{verbatim}
Type checking algorithm

The deduction system is \textit{syntax directed} and satisfies the \textit{subformula property}. As such it describes a deterministic algorithm.

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

\textbf{Exercise.} \textit{Write the \texttt{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

type exception Error

Use \texttt{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:

\[
\begin{align*}
\frac{\array{c} 
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
\rightarrow \text{ELIM}
}{\Gamma \vdash ab : T}
\end{align*}
\]

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

- Apply a function of type `Int → Int` to an argument of type `Odd even` even though every odd number is an integer number, too.
- If we have records, apply the function `λ x: {ℓ : Int}`. (`3 + x.ℓ`) to a record of type `{ℓ : Int, ℓ′ : Bool}`.
- If we are in OOP, send a message defined for objects of the class `Persons` to an instance of the subclass `Students`.
The rule for application requires the argument of the function to be exactly of the same type as the domain of the function:

\[ \rightarrow_{\text{ELIM}} \]

\[ \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \]

\[ \Gamma \vdash ab : T \]

So, for instance, we cannot:

- Apply a function of type \( \text{Int} \rightarrow \text{Int} \) to an argument of type \( \text{Odd} \) even though every odd number is an integer number, too.
Subtyping

The rule for application requires the argument of the function to be \emph{exactly of the same type} as the domain of the function:

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

So, for instance, we \textbf{cannot}:

- Apply a function of type \texttt{Int \rightarrow 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 \( \{\ell : \texttt{Int}, \ell' : \texttt{Bool}\} \)
The rule for application requires the argument of the function to be *exactly of the same type* as the domain of the function:

\[
\begin{array}{c}
\text{→E} \\
\Gamma \vdash a : S \to T \quad \Gamma \vdash b : S \\
\hline
\Gamma \vdash ab : T
\end{array}
\]

So, for instance, we *cannot*:

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

The rule for application requires the argument of the function to be \textit{exactly of the same type} as the domain of the function:

\[
\rightarrow \text{ELIM} \\
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
\hline \\
\Gamma \vdash ab : T 
\]

So, for instance, we \textbf{cannot}:

- Apply a function of type $\text{Int} \rightarrow \text{Int}$ to an argument of type $\text{Odd}$ even though every odd number is an integer number, too.
- If we have records, apply the function $\lambda x:\{\ell : \text{Int}\}.(3 + x.\ell)$ to a record of type $\{\ell : \text{Int}, \ell' : \text{Bool}\}$
- If we are in OOP, send a message defined for objects 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 (i.e., a reflexive and transitive binary relation) \( \leq \) on types: \( \leq \subset \mathbf{Types} \times \mathbf{Types} \) (some literature uses the notation \( <: \))

- 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 relation

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

  - *Containment*: If $S \leq T$, then every value of type $S$ is also of type $T$.
    For instance, an odd number is also an integer, a student is also a person.
    Sometimes called a "is_a" relation.
  - *Substitutability*: If $S \leq T$, then every value of type $S$ can be safely used where a value of type $T$ is expected.
    Where "safely" means, without disrupting type preservation and progress.

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

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

This *subtyping relation* has two possible interpretations:

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

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

Sometimes called a “*is_a*” relation.

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

Where “safely” means, without disrupting type preservation and progress.
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.
We suppose to have a predefined preorder \( \mathcal{B} \subseteq \text{Basic} \times \text{Basic} \) for basic types (given by the language designer).

For instance take the reflexive and transitive closure of \( \{(\text{Odd}, \text{Int}), (\text{Even}, \text{Int}), (\text{Int}, \text{Real})\} \)
We suppose to have a predefined preorder $\mathcal{B} \subset \text{Basic} \times \text{Basic}$ for basic types (given by the language designer).

For instance take the reflexive and transitive closure of

\{(Odd, Int), (Even, Int), (Int, Real)\}

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

Problem

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

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

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

2. If $a : T_1$, then $f(a)$ is well typed. If $S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$, then also $g(a)$ is well-typed. $g$ expects arguments of type $S_1$ but $a$ is of type $T_1$, so we can safely use $T_1$ where $S_1$ is expected, i.e. $T_1 \leq S_1$.

3. $f(a) : T_2$, but since $g$ returns results in $S_2$, then $g(a) : S_2$. If I use $g$ where $f$ is expected, then it must be safe to use $S_2$ results where $T_2$ results are expected $\Rightarrow S_2 \leq T_2$ must hold.

Solution

$S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \Leftrightarrow T_1 \leq S_1$ and $S_2 \leq T_2$. 

G. Castagna (CNRS)

Four Forms of Polymorphism
Subtyping of arrows: intuition

Problem

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

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

1. If $a : T_1$, then we can apply $f$ to $a$. If $S \leq T_1 \rightarrow T_2$, then we can apply $g$ to $a$, as well.

   $\Rightarrow g$ is a function, therefore $S = S_1 \rightarrow S_2$
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 $T_1 \leq S_1$
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 \textit{substitutability interpretation}:

1. If \( a : T_1 \), then we can apply \( f \) to \( a \). If \( S \leq T_1 \rightarrow T_2 \), then we can apply \( g \) to \( a \), as well.
   \[ \Rightarrow g \text{ is a function, therefore } S = S_1 \rightarrow S_2 \]

2. If \( a : T_1 \), then \( f(a) \) is well typed. If \( S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \), then also \( g(a) \) is well-typed. \( g \) expects arguments of type \( S_1 \) but \( a \) is of type \( T_1 \)
   \[ \Rightarrow \text{we can safely use } T_1 \text{ where } S_1 \text{ is expected, ie } T_1 \leq S_1 \]

3. \( f(a) : T_2 \), but since \( g \) returns results in \( S_2 \), then \( g(a) : S_2 \). If \( f \) is expected, then it must be safe to use \( S_2 \) results where \( T_2 \) results are expected.
   \[ \Rightarrow S_2 \leq T_2 \text{ must hold.} \]
**Problem**

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

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

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

2. If $a : T_1$, then $f(a)$ is well typed. If $S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$, then also $g(a)$ is well-typed. $g$ expects arguments of type $S_1$ but $a$ is of type $T_1$
   $\Rightarrow$ we can safely use $T_1$ where $S_1$ is expected, ie $T_1 \leq S_1$

3. $f(a) : T_2$, but since $g$ returns results in $S_2$, then $g(a) : S_2$. If I use $g$ where $f$ is expected, then it must be safe to use $S_2$ results where $T_2$ results are expected
   $\Rightarrow$ $S_2 \leq T_2$ must hold.

**Solution**

$S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \iff T_1 \leq S_1$ and $S_2 \leq T_2$
Covariance and contravariance

\[ S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \iff T_1 \leq S_1 \text{ and } S_2 \leq T_2 \]

Notice the different orientation of containment on domains and co-domains. We say that the type constructor \( \rightarrow \) 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

\[ S_1 \to S_2 \leq T_1 \to T_2 \iff T_1 \leq S_1 \text{ and } S_2 \leq T_2 \]

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 ...

\[ \text{Int} \to \text{Int} \leq \text{Int} \to \text{Real} \] (covariance of the codomains)

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

\[ \text{Int} \to \text{Int} \leq \text{Odd} \to \text{Int} \] (contravariance of the codomains)
Covariance and contravariance

\[ S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \iff T_1 \leq S_1 \text{ and } S_2 \leq T_2 \]

Notice the different orientation of containment on domains and co-domains. We say that the type constructor \( \rightarrow \) is

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

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

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

\[ \text{Int} \rightarrow \text{Int} \leq \text{Int} \rightarrow \text{Real} \] (covariance of the codomains)
Covariance and contravariance

\[ S_1 \to S_2 \leq T_1 \to T_2 \iff T_1 \leq S_1 \text{ and } S_2 \leq T_2 \]

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 \( \text{Int} \) so they will be also in \( \text{Real} \).
  \[ \text{Int} \to \text{Int} \leq \text{Int} \to \text{Real} \] (covariance of the codomains)
- **is also** a function that maps odds to integers: when fed with integers it returns integers, so will do the same when fed with odd numbers.
  \[ \text{Int} \to \text{Int} \leq \text{Odd} \to \text{Int} \] (contravariance of the codomains)
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) $T_1 \leq S_1$ and $S_2 \leq T_2$ are provable, so is $T_1 \leq T_2$.

The rules Refl and Trans are admissible.
Subtyping deduction system

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

\[ (B_1, B_2) \in \mathcal{B} \implies B_1 \leq B_2 \]

\[ T_1 \leq S_1 \quad S_2 \leq T_2 \implies S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \]

\[ T_1 \leq T_2 \quad T_2 \leq T_3 \implies T_1 \leq T_3 \]

This system is neither \textit{syntax directed} nor satisfies the \textit{subformula} property

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

**Subtyping deduction system**

\[
\text{BASIC} \quad \frac{(B_1, B_2) \in \mathcal{B}}{B_1 \leq B_2}
\]

\[
\text{ARROW} \quad \frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}
\]
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?
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?
We defined the subtyping relation and we know how to decide it. How do we use it for typing our programs?

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash x : \Gamma(x) \\
\text{\rightarrow INTRO} & \quad \Gamma, x : S \vdash a : T \\
& \quad \Gamma \vdash \lambda x : S. a : S \rightarrow T \\
\text{\rightarrow ELIM} & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
& \quad \Gamma \vdash ab : T
\end{align*}
\]
Type system

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

\[ \text{VAR} \]
\[ \Gamma \vdash x : \Gamma(x) \]

\[ \text{→INTRO} \]
\[ \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x : S. a : S \rightarrow T} \]

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

\[ \text{SUBSUMPTION} \]
\[ \frac{\Gamma \vdash a : S \quad S \leq T}{\Gamma \vdash a : T} \]
Type system

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

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash x : \Gamma(x) \\
\text{→INTRO} & \quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x : S. a : S \rightarrow T} \\
\text{→ELIM} & \quad \frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T}
\end{align*}
\]

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

This corresponds to the \textit{containment relation}:

If \( S \leq T \) and \( a \) is of type \( S \) then \( a \) is also of type \( T \)
We defined the subtyping relation and we know how to decide it. How do we use it for typing our programs?

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash x : \Gamma(x) \\
\text{→INTRO} & \quad \Gamma, x : S \vdash a : T \\
& \quad \Gamma \vdash \lambda x : S. a : S \rightarrow T \\
\text{→ELIM} & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
& \quad \Gamma \vdash ab : T \\
\text{SUBSUMPTION} & \quad \Gamma \vdash a : S \quad S \leq T \\
& \quad \Gamma \vdash a : T
\end{align*}
\]

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 \( a \rightarrow^* b \), then \( \Gamma \vdash b : T \).

**Progress property:** If \( \emptyset \vdash a : T \) and \( a \not\rightarrow^* \), then \( a \) is a value.
Typing algorithm

\[
\begin{align*}
\text{VAR} & : \quad \Gamma \vdash x : \Gamma(x) \\
\text{\rightarrow INTRO} & : \quad \Gamma, x : S \vdash a : T \\
& \quad \quad \Rightarrow \\
& \quad \quad \quad \Gamma \vdash \lambda x : S. a : S \rightarrow T \\
\text{\rightarrow ELIM} & : \quad \Gamma \vdash a : S \rightarrow T \\
& \quad \quad \Gamma \vdash b : S \\
& \quad \quad \Rightarrow \\
& \quad \quad \quad \Gamma \vdash ab : T \\
\text{SUBSUMPTION} & : \quad \Gamma \vdash a : S \\
& \quad \quad S \leq T \\
& \quad \quad \Rightarrow \\
& \quad \quad \quad \Gamma \vdash a : T
\end{align*}
\]
Typing algorithm

\[ \Gamma \vdash \begin{array}{c} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array} \rightarrow \text{INTRO} \]

\[ \begin{array}{c} \Gamma, x : S \vdash a : T \\ \Gamma \vdash \lambda x:S.a : S \rightarrow T \end{array} \]

\[ \rightarrow \text{ELIM} \]

\[ \begin{array}{c} \Gamma \vdash a : S \rightarrow T \\ \Gamma \vdash b : S \end{array} \]

\[ \Gamma \vdash a b : T \]

\[ \text{SUBSUMPTION} \]

\[ \begin{array}{c} \Gamma \vdash a : S \\ S \leq T \end{array} \]

\[ \Gamma \vdash a : T \]

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

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?
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?**
The system is algorithmic: it describes a typing algorithm (exercise: program typecheck and subtype by using the previous structures).

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

The system is algorithmic: it describes a typing algorithm (exercise: program type check and subtype by using the previous structures).

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?
The system is algorithmic: it describes a typing algorithm (exercise: program typecheck and subtype by using the previous structures).

The system conforms the substitutability interpretation: we use an expression of a subtype $U$ where a supertype $S$ is expected (note “use” = elimination rule).

How do we relate the two systems?

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

\[ \emptyset \vdash \lambda x: \text{Int}. x : \text{Odd} \rightarrow \text{Real} \quad \text{but} \quad \emptyset \not\vdash \forall \lambda x: \text{Int}. x : \text{Odd} \rightarrow \text{Real}. \]
Typing algorithm

\[ \text{VAR} \]
\[ \Gamma \vdash_A x : \Gamma(x) \]

\[ \rightarrow \text{INTRO} \]
\[ \Gamma, x : S \vdash_A a : T \]
\[ \Gamma \vdash_A \lambda x : S. a : S \rightarrow T \]

\[ \rightarrow \text{ELIM} \leq \]
\[ \Gamma \vdash_A a : S \rightarrow T \]
\[ \Gamma \vdash_A b : U \]
\[ U \leq S \]
\[ \Gamma \vdash_A ab : T \]

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:

\[ \emptyset \vdash A \lambda x : \text{Int}. x : \text{Odd} \rightarrow \text{Real} \]

but

\[ \emptyset \not\vdash_A \lambda x : \text{Int}. x : \text{Odd} \rightarrow \text{Real}. \]

This is expected: Algorithm = one type returned for each typable term.
Soundness and completeness of the typing algorithm

\[ a \text{ is typable by } \vdash \iff a \text{ is typable by } \vdash_A \]

\[ \iff = \text{ soundness} \]
\[ \Rightarrow = \text{ completeness} \]
Soundness and completeness of the typing algorithm

\[ a \text{ is typable by } \vdash \iff a \text{ is typable by } \vdash A \]

\( \iff \) = soundness
\( \implies \) = completeness

Theorem (Soundness)

If \( \Gamma \vdash_A a : T \), then \( \Gamma \vdash a : T \)

Theorem (Completeness)

If \( \Gamma \vdash a : T \), then \( \Gamma \vdash_A a : S \) with \( S \leq T \)
Corollary (Minimum type)

\[ \text{If } \Gamma \vdash \text{a} : T \text{ then } T = \min \{ S \mid \Gamma \vdash \text{a} : S \} \]

Proof. Let \( S = \{ S \mid \Gamma \vdash \text{a} : S \} \). Soundness ensures that \( S \) is not empty. Completeness states that \( T \) is a lower bound of \( S \). Minimality follows by using soundness once more.
Corollary (Minimum type)

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

Proof. Let $S = \{S \mid \Gamma \vdash a : S\}$. Soundness ensures that $S$ is not empty. Completeness states that $T$ is a lower bound of $S$. 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.
Corollary (Minimum type)

If $\Gamma \vdash a : T$ then $T = \min\{S \mid \Gamma \vdash a : S\}$

Proof. Let $S = \{S \mid \Gamma \vdash a : S\}$. Soundness ensures that $S$ is not empty. Completeness states that $T$ is a lower bound of $S$. 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 $\Gamma \vdash a : T$ and $a \rightarrow^* b$, then $\Gamma \vdash a : S$ 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 $\lambda$-calculus + $\leq$

- 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.
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 *containment* interpretation of the subtyping relation corresponds to the “logical” view of the type system embodied by subsumption.

The *substitutability* interpretation of the subtyping relation corresponds to the “algorithmic” view of the type system.

To *define* the type system one usually starts from the “logical” system, which is simpler since subtyping is concentrated in the subsumption rule.

To *implement* the type system one passes to the substitutability view. Subsumption is eliminated and the check of the subtyping relation is distributed in the places where values are used/consumed. This in general corresponds to embed subtype checking into elimination rules.

The obtained algorithm works on the *minimum types* of the logical system.

Computation reduces the (algorithmic) type thus increasing type information (the result of a computation represents the best possible type information: it is the *singleton type* containing the result).

The last point makes *dynamic dispatch* (aka, dynamic binding) meaningful.
Products I

Syntax

Types  $T ::= ... \mid T \times T$  \text{product types}

Terms  $a, b ::= ...$

| $(a, a)$  \text{pair} \\
| $\pi_i(a)$  \text{projection } (i=1,2)

Reduction

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

Typing

$$\times \text{INTRO}$$

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

$$\times \text{ELIM}_i$$

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

\[ \text{PROD} \]

\[
\begin{align*}
S_1 & \leq T_1 \\
S_2 & \leq T_2 \\
S_1 \times S_2 & \leq T_1 \times T_2
\end{align*}
\]

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

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

Of course subject reduction and progress still hold.

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

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

Syntax

\[
\begin{align*}
Types & \quad T ::= \ldots | \{\ell : T, \ldots, \ell : T\} & \text{record types} \\
Terms & \quad a, b ::= \ldots | \{\ell = a, \ldots, \ell = a\} & \text{record} \\
& \quad | a.\ell & \text{field selection}
\end{align*}
\]

Reduction

\[
\{\ldots, \ell = a, \ldots\}.\ell \rightarrow a
\]

Typing

\[
\begin{align*}
\{\} \text{INTRO} & \\
\Gamma \vdash a_1 : T_1 & \ldots & \Gamma \vdash a_n : T_n \\
\Gamma \vdash \{\ell_1 = a_1, \ldots, \ell_n = a_n\} : \{\ell_1 : T_1, \ldots, \ell_n : T_n\} \\
\{\} \text{ELIM} & \\
\Gamma \vdash a : \{\ldots, \ell : T, \ldots\} & \Gamma \vdash a.\ell : T
\end{align*}
\]
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.

\[
\text{RECORD} \quad S_1 \leq T_1 \ldots S_n \leq T_n \\
\{ \ell_1 : S_1, \ldots, \ell_n : S_n, \ldots, \ell_{n+k} : S_{n+k} \} \leq \{ \ell_1 : T_1, \ldots, \ell_n : T_n \}
\]

Exercise. Which are the algorithmic typing rules?
Outline

4  Simple Types

5  Recursive Types

6  Bibliography
Iso-recursive and Equi-recursive types

Lists are a classic example of recursive types:

\[ X \approx (\text{Int} \times X) \lor \text{Nil} \]

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

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

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

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

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

\[ X \approx (\text{Int} \times X) \lor \text{Nil} \]

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

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

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

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

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

Subtyping for recursive types generalizes the equi-recursive approach. The \( \approx \) relation corresponds to subtyping in both directions:

\[
\mu X. T \leq T[\mu X. T/X] \quad T[\mu X. T/X] \leq \mu X. T
\]
Recursive types are weird

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

\[
\mu X. ((\text{Int} \times X) \lor \text{Nil})
\]
interpret the type above as the finite lists of integers.

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

Actually if you have recursive terms and allow infinite values you can easily jeopardize decidability of the subtyping relation (which resorts to checking type emptiness).

This contrasts with their intuition which looks simple: we always informally applied a rule such as:

\[
A, X \leq Y \vdash S \leq T
\]
\[
A \vdash \mu X. S \leq \mu Y. T
\]
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:
  \[
  \mu X.((\text{Int} \times X) \lor \text{Nil})
  \]
  interpret the type above as the \textit{finite} lists of integers.

Then \(\mu X.(\text{Int} \times X)\) 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:
  \[ \mu X.((\text{Int} \times X) \lor \text{Nil}) \]

interpret the type above as the \textit{finite} lists of integers.

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

- Actually if you have recursive terms and allow infinite values you can easily jeopardize decidability of the subtyping relation (which resorts to checking type emptiness).

- This contrasts with their intuition which looks simple: we always informally applied a rule such as:

\[
A, X \leq Y \vdash S \leq T \\
\frac{}{A \vdash \mu X.S \leq \mu Y.T}
\]
Subtyping recursive types

Syntax

Types \( T \) ::= Any top type
|   \( T \rightarrow T \) function types
|   \( T \times T \) product types
|   \( X \) type variables
|   \( \mu X . T \) recursive types

where \( T \) is \textit{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 \( S \not\equiv X \).
2. \( T \) is contractive iff every type variable \( X \) occurring in it is separated from its binder by a \( \rightarrow \) or a \( \times \).
Subtyping recursive types

The subtyping relation is defined **COINDUCTIVELY** by the rules:

- **TOP** \( T \leq \text{Any} \)
- **PROD** \[
\frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2}
\]
- **ARROW** \[
\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}
\]
- **UNFOLD LEFT** \[
\frac{S[\mu X. S/X] \leq T}{\mu X. S \leq T}
\]
- **UNFOLD RIGHT** \[
\frac{S \leq T[\mu X. T/X]}{S \leq \mu X. T}
\]
The subtyping relation is defined \textit{COINDUCTIVELY} by the rules

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

Coinductive definition

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

The subtyping relation is defined **COINDUCTIVELY** by the rules

\[
\begin{align*}
\text{TOP} & \quad \frac{}{T \leq \text{Any}} \\
\text{P\_\text{ROD}} & \quad \frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2} \\
\text{ARROW} & \quad \frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}
\end{align*}
\]

\[
\begin{align*}
\text{U\_\text{NFOLD\_LEFT}} & \quad \frac{S[\mu X. S/X] \leq T}{\mu X. S \leq T} \\
\text{U\_\text{NFOLD\_RIGHT}} & \quad \frac{S \leq T[\mu X. T/X]}{S \leq \mu X. T}
\end{align*}
\]

**Coinductive definition**

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

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

1. Because we compare infinite expansions
2. Because it would be unsound
3. Useless since obtained by coinduction and unfold
Example of coinductive derivation

\[
\begin{align*}
\text{Arrow} & \quad \text{Even} \leq \text{Int} \quad \mu X. \text{Int} \rightarrow X \leq \mu Y. \text{Even} \rightarrow Y \\
\text{Unfold Right} & \quad \text{Int} \rightarrow (\mu X. \text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y. \text{Even} \rightarrow Y) \\
\text{Unfold Left} & \quad \text{Int} \rightarrow (\mu X. \text{Int} \rightarrow X) \leq \mu Y. \text{Even} \rightarrow Y \\
& \quad \mu X. \text{Int} \rightarrow X \leq \mu Y. \text{Even} \rightarrow Y
\end{align*}
\]
Example of coinductive derivation

\[\begin{align*}
\text{Arrow} & : \quad \text{Even} \leq \text{Int} \quad \mu X. \text{Int} \rightarrow X \leq \mu Y. \text{Even} \rightarrow Y \\
\text{Unfold Right} & : \quad \text{Int} \rightarrow (\mu X. \text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y. \text{Even} \rightarrow Y) \\
\text{Unfold Left} & : \quad \text{Int} \rightarrow (\mu X. \text{Int} \rightarrow X) \leq \mu Y. \text{Even} \rightarrow Y \\
\mu X. \text{Int} \rightarrow X & \leq \mu Y. \text{Even} \rightarrow Y
\end{align*}\]

Notice the use of coinduction
Amadio and Cardelli’s subtyping algorithm

Let \( A \subset \text{Types} \times \text{Types} \)

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

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

\[
\frac{A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \times S_2 \leq T_1 \times T_2}
\]

\[
A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A'
\]

\[
\frac{A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}
\]

\[
A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A'
\]

\[
\frac{A' \vdash S[\mu X . S/X] \leq T}{A \vdash \mu X . S \leq T}
\]

\[
A' = A \cup (\mu X . S, T); A \neq A'; T \neq \text{Any}
\]

\[
\frac{A' \vdash S \leq T[\mu X . T/X]}{A \vdash S \leq \mu X . T}
\]

\[
A' = A \cup (S, \mu X . T); A \neq A'; S \neq \mu Y . U
\]
Amadio and Cardelli’s subtyping algorithm

Determinization of the rules

\[
\frac{}{A \vdash S \leq T} \quad (S, T) \in A
\]

\[
\frac{}{A \vdash S \leq \text{Any}} \quad (S, \text{Any}) \notin A
\]

\[
A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2
\]

\[
A \vdash S_1 \times S_2 \leq T_1 \times T_2 \quad A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A'
\]

\[
A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2
\]

\[
A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \quad A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A'
\]

\[
A' \vdash S[\mu X.S/X] \leq T
\]

\[
A \vdash \mu X.S \leq T \quad A' = A \cup (\mu X.S, T); A \neq A'; T \neq \text{Any}
\]

\[
A' \vdash S \leq T[\mu X.T/X]
\]

\[
A \vdash S \leq \mu X.T \quad A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U
\]
Amadio and Cardelli’s subtyping algorithm

Store the type to implement coinduction

\[
A \vdash S \leq T \quad (S, T) \in A
\]

\[
A \vdash S \leq \text{Any} \quad (S, \text{Any}) \notin A
\]

\[
A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2 \\
A \vdash S_1 \times S_2 \leq T_1 \times T_2
\]

\[
A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A'
\]

\[
A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2 \\
A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2
\]

\[
A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A'
\]

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

\[
A' \vdash S \leq T[\mu X.T/X] \\
A \vdash S \leq \mu X.T
\]

\[
A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U
\]
Amadio and Cardelli’s subtyping algorithm

Determinization of the rules

\[ A \vdash S \leq T \quad \text{if} \quad (S, T) \in A \]

\[ A \vdash S \leq \text{Any} \quad \text{if} \quad (S, \text{Any}) \not\in A \]

\[ A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2 \]
\[ A \vdash S_1 \times S_2 \leq T_1 \times T_2 \quad \text{where} \quad A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A' \]

\[ A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2 \]
\[ A \vdash S_1 \to S_2 \leq T_1 \to T_2 \quad \text{where} \quad A' = A \cup (S_1 \to S_2, T_1 \to T_2); A \neq A' \]

\[ A' \vdash S[\mu X.S/X] \leq T \]
\[ A \vdash \mu X.S \leq T \quad \text{where} \quad A' = A \cup (\mu X.S, T); A \neq A'; T \neq \text{Any} \]

\[ A' \vdash S \leq T[\mu X.T/X] \]
\[ A \vdash S \leq \mu X.T \quad \text{where} \quad A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U \]
Amadio and Cardelli’s subtyping algorithm

Store the type to implement coinduction

\[ A \vdash S \leq T \] \[ (S, T) \in A \]

\[ A \vdash S \leq \text{Any} \] \[ (S, \text{Any}) \notin A \]

\[ A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2 \]
\[ A \vdash S_1 \times S_2 \leq T_1 \times T_2 \]
\[ A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A' \]

\[ A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2 \]
\[ A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2 \]
\[ A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A' \]

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

\[ A' \vdash S \leq T[\mu X.T/X] \]
\[ A \vdash S \leq \mu X.T \]
\[ A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U \]
Amadio and Cardelli’s subtyping algorithm

The rest is similar

\[ A \vdash S \leq T \quad (S, T) \in A \]

\[ A \vdash S \leq \text{Any} \quad (S, \text{Any}) \notin A \]

\[
\frac{A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \times S_2 \leq T_1 \times T_2}
\]

\[ A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A' \]

\[
\frac{A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2}{A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}
\]

\[ A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A' \]

\[
\frac{A' \vdash S[\mu X.S/X] \leq T}{A \vdash \mu X.S \leq T}
\]

\[ A' = A \cup (\mu X.S, T); A \neq A'; T \neq \text{Any} \]

\[
\frac{A' \vdash S \leq T[\mu X.T/X]}{A \vdash S \leq \mu X.T}
\]

\[ A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U \]
Amadio and Cardelli’s subtyping algorithm

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

$$A \vdash S \leq T$$

$$A \vdash S \leq \text{Any} \quad (S, \text{Any}) \not\in A$$

$$A' \vdash S_1 \leq T_1 \quad A' \vdash S_2 \leq T_2$$

$$A \vdash S_1 \times S_2 \leq T_1 \times T_2$$

$$A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A'$$

$$A' \vdash T_1 \leq S_1 \quad A' \vdash S_2 \leq T_2$$

$$A \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2$$

$$A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A'$$

$$A' \vdash S[\mu X.S/X] \leq T$$

$$A \vdash \mu X.S \leq T$$

$$A' = A \cup (\mu X.S, T); A \neq A'; T \neq \text{Any}$$

$$A' \vdash S \leq T[\mu X.T/X]$$

$$A \vdash S \leq \mu X.T$$

$$A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U$$
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 $\emptyset \vdash S \leq T$ is provable.

To see the proof of the above theorem you can refer to the following reference:

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 $S \leq T$. 
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 $\emptyset \vdash S \leq T$ is provable.

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 $\emptyset \vdash S \leq T$ is provable.


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.

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

- $F$-closed if it contains all the elements that can be deduced by $F$ with hypothesis in $X$.
- $F$-consistent if every element of $X$ can be deduced by $F$ 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 $X$. 

Four Forms of Polymorphism

G. Castagna (CNRS)
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 $F$ be a deduction system on a universe $U$ (i.e. a monotone function from $P(U)$ to $P(U)$). A set $X \in P(U)$ is:

- $F$-closed if it contains all the elements that can be deduced by $F$ with hypothesis in $X$.
- $F$-consistent if every element of $X$ can be deduced by $F$ from other elements in $X$.

**Induction and coinduction**

A deduction system

- *inductively* defines the least $F$-closed set
- *coinductively* defines the greatest $F$-consistent set
**Induction and coinduction**

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

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

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

Example:

$U = \{a, b, c, d, e, f, g\}$
Induction and coinduction

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

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

**Observation**

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

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

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

**Observation**

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

**Example:**

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Induction and coinduction**

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

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

**Observation**

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

**Example:**

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

\[
\begin{array}{ccccccc}
  a & b & c & d & e & f \\
  \hline
  b & c & a & d & e & g \\
\end{array}
\]

Inductively:

$\emptyset$
**Induction and coinduction**

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

**Coinduction:** start from $U$, remove all elements that are not consequence of other elements, and iterate.

**Observation**

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

**Example:**

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

Inductively:

$\{d\}$
**Induction and coinduction**

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

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

**Observation**

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

**Example:**

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

\[
\begin{array}{cccccc}
  \hline
  \quad & \quad & \quad & a & b & c \\
  b & c & a & d & e & g \\
  \hline
\end{array}
\]

Inductively:

\[ \{d, e\} \]
**Induction and coinduction**

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

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

**Observation**

In all the (algorithmic, i.e., without refl and trans) subtyping systems 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:**

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

Inductively:

$$\{ d, e \}$$
Induction and coinduction

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

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

**Observation**

In all the (algorithmic, i.e., 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:**

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inductively: $\{d, e\}$

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

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

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

**Observation**

In all the (algorithmic, i.e., 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:**

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>a</td>
<td></td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>d</td>
<td></td>
<td></td>
<td>e</td>
<td>g</td>
</tr>
</tbody>
</table>

Inductively: $\{d, e\}$

Coinductively: $\{a, b, c, d, e, f, g\}$
**Induction and Coinduction**

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

**Coinduction:** start from $U$, remove all elements that are not consequence of other elements, and iterate.

**Observation**

In all the (algorithmic, i.e., 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:**

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

Inductively: $\{d, e\}$

Coinductively: $\{a, b, c, d, e, g\}$
**Induction and coinduction**

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

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

**Observation**

In all the (algorithmic, i.e., 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:**

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

Inductively: $\{d, e\}$

Coinductively: $\{a, b, c, d, e, g\}$
**Induction and coinduction**

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

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

**Observation**

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

**Example:**

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

\[
\begin{array}{ccccccc}
 a & b & c & d & e & f \\
 b & c & a & d & e & g \\
\end{array}
\]

Inductively:  \{d, e\}  Coinductively:  \{a, b, c, d, e\}
Induction and coinduction

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

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

**Observation**

In all the (algorithmic, i.e., 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:**

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

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

$$
\begin{array}{c}
\text{n} \\
\hline
\text{n + 1}
\end{array}
$$

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

$$
\begin{array}{c}
(m, n) \\
\hline
(n, o)
\end{array}
\rightarrow
\begin{array}{c}
(m, o)
\end{array}
$$

for $m, n, o \in \mathbb{N}$.
Why Coinduction for Recursive types?

We want to use $S = \mu X. \text{Int} \rightarrow X$ where $T = \mu Y. \text{Even} \rightarrow Y$ is expected.
Why Coinduction for Recursive types?

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

Use the substitutability interpretation.
Let $e : T$ then $e$:

1. waits for an $\text{Even}$ number,
2. fed by an $\text{Even}$ number returns a function that behaves similarly: (1) wait for an $\text{Even}$ ...
Why Coinduction for Recursive types?

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

Use the substitutability interpretation.

Let $e : T$ then $e$:

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

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

1. waits for an $\text{Int}$ number,
2. fed by an $\text{Int}$ (or a $\text{Even}$) number returns a function that behaves similarly: (1) wait for ...
Why Coinduction for Recursive types?

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

Use the substitutability interpretation.

Let $e : T$ then $e$:

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

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

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

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

$S \leq T \implies \text{Int} \to S \leq \text{Even} \to T \implies S \leq T \land \text{Even} \leq \text{Int}$
This is exactly the proof we saw at the beginning:

\[
\begin{align*}
\text{Arrow} & \quad \text{Even} \leq \text{Int} & \quad \mu X.\text{Int} \rightarrow X \leq \mu Y.\text{Even} \rightarrow Y \\
\text{Unfold Right} & \quad \text{Int} \rightarrow (\mu X.\text{Int} \rightarrow X) \leq \text{Even} \rightarrow (\mu Y.\text{Even} \rightarrow Y) \\
\text{Unfold Left} & \quad \mu X.\text{Int} \rightarrow X \leq \mu Y.\text{Even} \rightarrow Y
\end{align*}
\]
This is exactly the proof we saw at the beginning:

\[
\begin{align*}
\text{Even} & \leq \text{Int} & \Rightarrow & \mu X. \text{Int} \rightarrow X \leq \mu Y. \text{Even} \rightarrow Y \\
\text{Int} & \rightarrow (\mu X. \text{Int} \rightarrow X) & \leq & \text{Even} & \rightarrow (\mu Y. \text{Even} \rightarrow Y) \\
\text{Int} & \rightarrow (\mu X. \text{Int} \rightarrow X) & \leq & \mu Y. \text{Even} \rightarrow Y \\
\mu X. \text{Int} & \rightarrow X & \leq & \mu Y. \text{Even} \rightarrow Y
\end{align*}
\]

Coinduction

\( S \leq T \) is not an axiom but \( \{ S \leq T, \text{Even} \leq \text{Int} \} \) is a self-justifying set.
This is exactly the proof we saw at the beginning:

\[
\begin{align*}
\text{ARROW} & \quad \text{Even} \leq \text{Int} \quad \mu X.\text{Int} \to X \leq \mu Y.\text{Even} \to Y \\
\text{UNFOLD RIGHT} & \quad \text{Int} \to (\mu X.\text{Int} \to X) \leq \text{Even} \to (\mu Y.\text{Even} \to Y) \\
\text{UNFOLD LEFT} & \quad \mu X.\text{Int} \to X \leq \mu Y.\text{Even} \to Y
\end{align*}
\]

Coinduction

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

Observation:

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

\[
\text{subtype}(A, S, T) = \text{if } (S, T) \in A \text{ then } A \text{ else }
\]
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:

\[
\text{subtype}(A, S, T) = \begin{cases} 
\text{if } (S, T) \in A \text{ then } A \text{ else} \\
\text{let } A_0 = A \cup \{(S, T)\} \text{ in}
\end{cases}
\]
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:

\[
\text{subtype}(A, S, T) = \begin{cases} 
\text{if } (S, T) \in A \text{ then } A \text{ else} \\
\text{let } A_0 = A \cup \{(S, T)\} \text{ in} \\
\text{if } T = \text{Any} \text{ then } A_0
\end{cases}
\]
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:

\[
\text{subtype}(A, S, T) = \begin{cases} 
\text{if } (S, T) \in A \text{ then } A \text{ else} \\
\text{let } A_0 = A \cup \{(S, T)\} \text{ in} \\
\text{if } T = \text{Any} \text{ then } A_0 \\
\text{else if } S = S_1 \times S_2 \text{ and } T = T_1 \times T_2 \text{ then} \\
\text{subtype}(\text{subtype}(A_0, S_1, T_1), S_2, T_2)
\end{cases}
\]
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:

\[
\text{subtype}(A, S, T) = \begin{cases} 
\text{if } (S, T) \in A \text{ then } A \text{ else} \\
\text{let } A_0 = A \cup \{(S, T)\} \text{ in} \\
\text{if } T = \text{Any} \text{ then } A_0 \\
\text{else if } S = S_1 \times S_2 \text{ and } T = T_1 \times T_2 \text{ then} \\
\text{subtype(subtype}(A_0, S_1, T_1), S_2, T_2) \\
\text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \text{ then} \\
\text{subtype}(\text{subtype}(A_0, T_1, S_1), S_2, T_2) \\
\text{else} \text{ fail}
\end{cases}
\]
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:

\[ \text{subtype}(A, S, T) = \text{if } (S, T) \in A \text{ then } A \text{ else } \]

\[ \text{let } A_0 = A \cup \{(S, T)\} \text{ in } \]

\[ \text{if } T = \text{Any} \text{ then } A_0 \]

\[ \text{else if } S = S_1 \times S_2 \text{ and } T = T_1 \times T_2 \text{ then } \]

\[ \text{subtype}(\text{subtype}(A_0, S_1, T_1), S_2, T_2) \]

\[ \text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \text{ then } \]

\[ \text{subtype}(\text{subtype}(A_0, T_1, S_1), S_2, T_2) \]

\[ \text{else if } T = \mu X . T_1 \text{ then } \]

\[ \text{subtype}(A_0, S, T_1[\mu X . T_1 / X]) \]
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:

\[
\text{subtype}(A, S, T) = \begin{cases} 
\text{if } (S, T) \in A \text{ then } A \text{ else} \\
\text{let } A_0 = A \cup \{(S, T)\} \text{ in} \\
\text{if } T = \text{Any} \text{ then } A_0 \\
\text{else if } S = S_1 \times S_2 \text{ and } T = T_1 \times T_2 \text{ then} \\
\text{ subtype( subtype}(A_0, S_1, T_1), S_2, T_2) \\
\text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \text{ then} \\
\text{ subtype( subtype}(A_0, T_1, S_1), S_2, T_2) \\
\text{else if } T = \mu X. T_1 \text{ then} \\
\text{ subtype}(A_0, S, T_1[\mu X. T_1/X]) \\
\text{else if } S = \mu X. S_1 \text{ then} \\
\text{ subtype}(A_0, S_1[\mu X. S_1/X], T)
\end{cases}
\]
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:

\[
\text{subtype}(A, S, T) = \begin{align*}
\text{if } (S, T) \in A \text{ then } A \text{ else} \\
\text{let } A_0 = A \cup \{(S, T)\} \text{ in} \\
\text{if } T = \text{Any} \text{ then } A_0 \\
\text{else if } S = S_1 \times S_2 \text{ and } T = T_1 \times T_2 \text{ then} \\
\quad \text{subtype(subtype}(A_0, S_1, T_1), S_2, T_2) \\
\text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2 \text{ then} \\
\quad \text{subtype(subtype}(A_0, T_1, S_1), S_2, T_2) \\
\text{else if } T = \mu X. T_1 \text{ then} \\
\quad \text{subtype}(A_0, S, T_1[\mu X. T_1/X]) \\
\text{else if } S = \mu X. S_1 \text{ then} \\
\quad \text{subtype}(A_0, S_1[\mu X. S_1/X], T) \\
\text{else } \text{fail}
\end{align*}
\]
Compare the previous algorithm with the Amadio-Cardelli algorithm:

\[ A \vdash S \leq T \quad (S, T) \in A \]

\[ A \vdash S \leq \text{Any} \quad (S, \text{Any}) \notin A \]

\[
\begin{align*}
A' & \vdash S_1 \leq T_1 & A' & \vdash S_2 \leq T_2 \\
A & \vdash S_1 \times S_2 \leq T_1 \times T_2
\end{align*}
\]

\[ A' = A \cup (S_1 \times S_2, T_1 \times T_2); A \neq A' \]

\[
\begin{align*}
A' & \vdash T_1 \leq S_1 & A' & \vdash S_2 \leq T_2 \\
A & \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2
\end{align*}
\]

\[ A' = A \cup (S_1 \rightarrow S_2, T_1 \rightarrow T_2); A \neq A' \]

\[ A' \vdash S[\mu X.S/X] \leq T \]

\[ A \vdash \mu X.S \leq T \]

\[ A' = A \cup (\mu X.S, T); A \neq A'; T \neq \text{Any} \]

\[ A' \vdash S \leq T[\mu X.T/X] \]

\[ A \vdash S \leq \mu X.T \]

\[ A' = A \cup (S, \mu X.T); A \neq A'; S \neq \mu Y.U \]
They both check containment in the relation coinductively defined by:

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

But the former is far more efficient.

Parametric polymorphism
Outline

7 Introduction

8 Hindley-Milner System

9 Inference algorithm
Monomorphic calculus

**Types** \( T \) ::= \( \text{Bool} | \text{Int} | \text{Real} | \ldots \) basic types
\[ \mid T \rightarrow T \] function types

**Terms** \( a, b \) ::= \( \text{true} | \text{false} | 1 | 2 | \ldots \) constants
\[ \mid x \] variable
\[ \mid ab \] application
\[ \mid \lambda x:T.a \] abstraction
\[ \mid \text{let } x:T = a \text{ in } b \] let
Parametric polymorphism

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

\[
\text{let } f : \text{Int} \rightarrow \text{Int} = \lambda x : \text{Int}. x \text{ in } b
\]

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

\[
\begin{align*}
\Gamma & \vdash x : \Gamma(x) \\
\Gamma, x : S & \vdash a : T \\
\Gamma & \vdash \lambda x. a : S \rightarrow T \\
\Gamma & \vdash a : S \rightarrow T \\
\Gamma & \vdash b : S \\
\Gamma & \vdash b : S \\
\Gamma & \vdash ab : T
\end{align*}
\]

In particular, \( \lambda x. x \) can be given all the types of the form \( T \rightarrow T \) for every \( T \).
Parametric polymorphism

We extend the syntax of types

\[ Types \quad T ::= \quad \text{Bool} \mid \text{Int} \mid \text{Real} \mid \ldots \quad \text{basic types} \]
\[ \quad \quad \mid \quad T \to T \quad \text{function types} \]
\[ \quad \mid \quad \alpha \quad \text{type variables} \]
\[ \quad \mid \quad \forall \alpha. T \quad \text{polymorphic types} \]

We add to the previous rules these two rules

\[
\frac{\Gamma \vdash a : T \quad \alpha \not\in \text{fv}(\Gamma)}{\Gamma \vdash a : \forall \alpha. T} \quad \frac{\Gamma \vdash a : \forall \alpha. T}{\Gamma \vdash a : T[S/\alpha]}
\]

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:

let \( f = \lambda x. x \) in \((f3, f\text{true}) : \text{Int} \times \text{Bool}\)
Remark

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

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

is crucial ... without it we can derive

$$
\frac{
\vdash \lambda x.x : \alpha \rightarrow (\forall \alpha. \alpha)
}{
\vdash \lambda x.x : \alpha \rightarrow (\forall \alpha. \alpha)
}$$

and therefore type, for instance, $(\lambda x.x)12$ with any type we wish
For terms without type annotations the problems:

- **type inference**: given an expression \(a\) find if there exists a type \(T\) such that \(a : T\)

- **type checking**: given an 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 annotations 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 annotations 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 ...)
The quantification can only be prenex:

\[ \text{Types} \quad T \::=\quad \text{Bool} \mid \text{Int} \mid \text{Real} \mid \ldots \quad \text{basic types} \\
\quad \mid \quad T \rightarrow T \quad \text{function types} \\
\quad \mid \quad \alpha \quad \text{type variables} \]

\[ \text{Schemas} \quad \sigma \::=\quad T \quad \text{type} \\
\quad \mid \quad \forall \alpha. \sigma \quad \text{schema} \]

A type environment $\Gamma$ now maps variable to schemas, and typing judgement have the form $\Gamma \vdash a : \sigma$
The following types (schemas) are ok:

\[
\forall \alpha. \alpha \to \alpha \\
\forall \alpha. \forall \beta. (\alpha \times \beta) \to \alpha \\
\forall \alpha. \text{Bool} \to \alpha \to \alpha \to \alpha \\
\forall \alpha. (\alpha \to \alpha) \to \alpha
\]

but the following type is not longer allowed:

\[
(\forall \alpha. \alpha \to \alpha) \to (\forall \alpha. \alpha \to \alpha)
\]
Hindley-Milner System

\[ \Gamma \vdash x : \Gamma(x) \quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x. a : S \rightarrow T} \quad \frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T} \]

\[ \frac{\Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2}{\Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2} \quad \frac{\Gamma \vdash a : T \quad \alpha \notin \text{fv}(\Gamma)}{\Gamma \vdash a : \forall \alpha. T} \quad \frac{\Gamma \vdash a : \forall \alpha. T}{\Gamma \vdash a : T[S/\alpha]} \]
Hindley-Milner System

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

\[ \Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 \]
\[ \Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 \]

Thanks to this we can for instance type

\[ \text{let } f = \lambda x. x \text{ in } (f f)(f 1) \]

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:

\[ f : \forall \alpha. \alpha \to \alpha \vdash f : \forall \alpha. \alpha \to \alpha \]
\[ f : \forall \alpha. \alpha \to \alpha \vdash f : (\alpha \to \alpha)[T/\alpha] \]

where \( T \) is respectively for each occurrence of \( f \), \((\text{Int } \to \text{Int}) \to \text{Int} \to \text{Int}, \text{Int } \to \text{Int}, \) and \( \text{Int} \).
On the contrary the rule for abstractions does not introduce in the environment a schema, but just a type

\[
\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x. a : S \rightarrow T}
\]

otherwise \( S \rightarrow T \) would not be well formed.

In particular,

\[
\lambda x. xx
\]

is no longer typeable, while

\[
\text{let } f = \lambda x. x \text{ in } ff
\]

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

\[
\Gamma \vdash a : T \quad \alpha \not\in \text{fv}(\Gamma) \quad \Rightarrow \quad \Gamma \vdash a : \forall \alpha . T
\]

\[
\Gamma \vdash a : \forall \alpha . T \quad \Rightarrow \quad \Gamma \vdash a : T[S/\alpha]
\]
Hindley-Milner syntax-directed system

\[
\begin{align*}
\Gamma, x : S &\vdash a : T \\
\Gamma &\vdash \lambda x. a : S \rightarrow T \\
\Gamma \vdash a : S \rightarrow T &\quad \Gamma \vdash b : S \\
\Gamma &\vdash ab : T \\
T &\sqsubseteq \Gamma(x) \\
\Gamma &\vdash x : T \\
\Gamma \vdash a : S &\quad \Gamma, x : \text{Gen}(S, \Gamma) \vdash b : T \\
\Gamma &\vdash \text{let } x = a \text{ in } b : T
\end{align*}
\]
Hindley-Milner syntax-directed system

\[ \Gamma, x : S \vdash a : T \]
\[ \Gamma \vdash \lambda x. a : S \rightarrow T \]
\[ \Gamma \vdash a : S \rightarrow T \]
\[ \Gamma \vdash b : S \]
\[ \Gamma \vdash ab : T \]

\[ T \sqsubseteq \Gamma(x) \]
\[ \Gamma \vdash x : T \]
\[ \Gamma, x : \text{Gen}(S, \Gamma) \vdash b : T \]
\[ \Gamma \vdash \text{let } x = a \text{ in } b : T \]

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

\[ \text{Gen}(S, \Gamma) = \forall \alpha_1 \ldots \forall \alpha_n . S \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \text{fv}(S) \setminus \text{fv}(\Gamma) \]
Hindley-Milner syntax-directed system

\[ \begin{align*}
\Gamma, x : S &\vdash a : T \\
\Gamma &\vdash \lambda x. a : S \rightarrow T \\
\Gamma &\vdash ab : T \\
\Gamma &\vdash a : S \rightarrow T \\
\Gamma, x : \text{Gen}(S, \Gamma) &\vdash b : T
\end{align*} \]

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

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

Syntax directed but \textbf{Not an algorithm yet!}
State: a current substitution $\phi$ and an infinite set of fresh variables $V$

```
fresh = do \(\alpha \in V\)
do \(V := V \setminus \{\alpha\}\)
return \(\alpha\)
```

```
W(\(\Gamma \vdash x\)) = let \(\forall \alpha_1, \ldots, \alpha_n. T \leftarrow \Gamma(x)\)
do \(\beta_1, \ldots, \beta_n \leftarrow \text{fresh}, \ldots, \text{fresh}\)
return \(T[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\)
```

```
W(\(\Gamma \vdash \lambda x. a\)) = do \(\alpha \leftarrow \text{fresh}\)
do \(T \leftarrow W(\Gamma, x : \alpha \vdash a)\)
return \(\alpha \rightarrow T\)
```

```
W(\(\Gamma \vdash ab\)) = do \(T \leftarrow W(\Gamma \vdash a)\)
do \(S \leftarrow W(\Gamma \vdash b)\)
do \(\alpha \leftarrow \text{fresh}\)
do \(\phi := \text{mgu}(\phi(T), \phi(S \rightarrow \alpha)) \circ \phi\)
return \(\alpha\)
```

```
W(\(\Gamma \vdash \text{let } x = a \text{ in } b\)) = do \(S \leftarrow W(\Gamma \vdash a)\)
do \(\sigma \leftarrow \text{Gen}(\phi(S), \phi(\Gamma))\)
return \(W(\Gamma, x : \sigma \vdash b)\)
```
Most General Unifier

\[ \text{mg u}(\emptyset) = \text{id} \]

\[ \text{mg u}(\{(\alpha, \alpha)\} \cup C) = \text{mg u}(C) \]

\[ \text{mg u}(\{(\alpha, T)\} \cup C) = \text{mg u}(C[T/\alpha] \circ [T/\alpha] \text{ if } \alpha \text{ not free in } T) \]

\[ \text{mg u}(\{(T, \alpha)\} \cup C) = \text{mg u}(C[T/\alpha] \circ [T/\alpha] \text{ if } \alpha \text{ not free in } T) \]

\[ \text{mg u}(\{(S_1 \rightarrow S_2, T_1 \rightarrow T_2)\} \cup C) = \text{mg u}(\{(S_1, T_1), (S_2, T_2)\} \cup C) \]

In all the other cases \text{mg u} fails
Ad-Hoc Polymorphism
Outline

10 Set-theoretic types

11 Semantic Subtyping

12 Application to a language.

13 Adding Parametric Polymorphism: the Types

14 Adding Parametric Polymorphism: the Language
Outline

10 Set-theoretic types

11 Semantic Subtyping

12 Application to a language.

13 Adding Parametric Polymorphism: the Types

14 Adding Parametric Polymorphism: the Language
Set-theoretic types

We consider the following possibly recursive types:

\[ T ::= \text{Bool} \mid \text{Int} \mid \text{Any} \mid (T,T) \mid T \lor T \mid T \land T \mid \text{not}(T) \mid T \rightarrow T \]

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 \( T_1 \mid T_2 \) to denote \( T_1 \lor T_2 \)
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:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

where patterns are defined as follows:

\[
p ::= x \mid (p, p) \mid p \mid p \mid p\&p
\]

Let us see how to type pattern matching.
2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

where patterns are defined as follows:

\[
p ::= x \mid (p_1, p_2) \mid p_1 \mid p_2 \mid p_1 \lor p_2 \mid p_1 \land p_2
\]

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

\[
\llbracket p \rrbracket = \{ v \mid v \text{ is a value that matches } p \}
\]

\[
\begin{align*}
\llbracket x \rrbracket &= \text{Any} \\
\llbracket (p_1, p_2) \rrbracket &= (\llbracket p_1 \rrbracket, \llbracket p_2 \rrbracket) \\
\llbracket p_1 \mid p_2 \rrbracket &= \llbracket p_1 \rrbracket \lor \llbracket p_2 \rrbracket \\
\llbracket p_1 \land p_2 \rrbracket &= \llbracket p_1 \rrbracket \land \llbracket p_2 \rrbracket
\end{align*}
\]
Boolean type connectives are needed to *type pattern matching*:

- To infer the type $T_1$ of $e_1$ we need $T & p_1$;
- To infer the type $T_2$ of $e_2$ we need $(T \backslash p_1) \& p_2$;
- The type of the match expression is $T_1 \lor T_2$.

Pattern matching is exhaustive if

- Formally:

\[
\begin{align*}
\textit{MATCH} & \quad \Gamma \vdash e : T \\
& \quad \Gamma, T \& p_1 \vdash e_1 : T_1 \\
& \quad \Gamma, T \backslash p_1 \vdash e_2 : T_2 \\
& \quad \Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \lor T_2 \\
& \quad (T \leq p_1 \lor p_2) \\
\end{align*}
\]

where $T/p$ is the type environment for the capture variables in $p$ when the pattern is matched against values in $T$. (e.g., $(\text{Int,Int} \lor \text{Bool,Char})/ (x,y)$ is $x : \text{Int} \lor \text{Bool}, y : \text{Int} \lor \text{Char}$.)
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

match \( e \) with \( p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \)

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg (T_2) \)
Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \text{not}(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{ p_1 \} \);

- Pattern matching is exhaustive if \( T \leq \{ p_1 \} \vee \{ p_2 \} \);

Formally:

\[
\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \vee T_2
\]

where \( T \setminus \{ p \} \) is the type environment for the capture variables in \( p \) when the pattern is matched against values in \( T \). (e.g., \( (\text{Int,Int}) \setminus \{ x, y \} \) is \( x : \text{Int} \vee \text{Bool}, y : \text{Int} \vee \text{Char} \)).
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& {\downarrow p_1\uparrow} \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus {\downarrow p_1\uparrow}) \& {\downarrow p_2\uparrow} \);
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{p_1\} \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \{p_1\}) \& \{p_2\} \);
- The type of the match expression is \( T_1 \lor T_2 \).
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \text{not}(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{p_1\} \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \{p_1\}) \& \{p_2\} \);
- The type of the match expression is \( T_1 \lor T_2 \).
- Pattern matching is exhaustive if \( T \leq \{p_1\} \lor \{p_2\} \);
Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \text{not}(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \downarrow p_1 \downarrow \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \downarrow p_1 \downarrow) \& \downarrow p_2 \downarrow \);
- The type of the match expression is \( T_1 \lor T_2 \).
- Pattern matching is exhaustive if \( T \leq \downarrow p_1 \downarrow \lor \downarrow p_2 \downarrow \);
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to \textit{type pattern matching}:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \text{not}(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \not p_1 \);  
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus p_1) \& p_2 \);  
- The type of the match expression is \( T_1 \lor T_2 \).  
- Pattern matching is exhaustive if \( T \leq \not p_1 \lor \not p_2 \);

Formally:

\[
\begin{array}{c}
\Gamma \vdash e : T \\
\Gamma, T \& \not p_1 \not p_1 \vdash e_1 : T_1 \\
\Gamma, T \not p_1 \not p_2 \vdash e_2 : T_2 \\
\end{array}
\Rightarrow
\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \lor T_2
\]

where \( T/p \) is the type environment for the capture variables in \( p \) when the pattern is matched against values in \( T \).

(e.g., \((\text{Int, Int}) \lor (\text{Bool, Char})\)/(x, y) is \( x : \text{Int} \lor \text{Bool}, y : \text{Int} \lor \text{Char})\)
3. Overloaded functions

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

```go
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):

```go
code
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:

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

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

1. What is the type returned by

   ```ml
   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.
Exercise:

1. What is the type returned by

   ```ocaml
   let foo = function
     | ('A,'B) -> true
     | ('B,'A) -> false
   and what is the problem?
   ```

   ```ocaml
   [< '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
Exercise:

1. What is the type returned by
   
   ```ocaml
   let foo = function
   | ('A,'B) -> true
   | ('B,'A) -> false
   ```

   and what is the problem?

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

2. Which type could we give if we had full-fledged union types?
   ```ocaml
   ( 'A * 'B )| ( 'B * 'A ) -> bool
   ```

3. Give an intersection type that refines the previous type
Exercise:

1. What is the type returned by
   
   ```cpp
   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](http://www.cduce.org/ocaml/bi)
4. Typing of Mixins

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

```typescript
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 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 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 \( \alpha \) RBtree =
| Leaf
| Red( \( \alpha \), RBtree , RBtree)
| Blk( \( \alpha \), 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 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 RBtree = Btree | Rtree
type Rtree = Red( α, Btree , Btree )
type Btree = Blk( α, Rtree , Rtree ) | Leaf
type Wrong = Red( α, (Rtree,Rtree) | (Rtree,Rtree) )
type Unbal = Blk( α, (Wrong,Rtree) | (Rtree,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 )

G. Castagna (CNRS)
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)
### Four Forms of Polymorphism

**Type Definitions**

- `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))`

**Functions**

- `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 ) ->` (function body as in the original image)

- `let ins : (Leaf → Rtree) & (Btree → RBtree\ Leaf) & (Rtree → Rtree | Wrong) = function
  | Leaf -> Red(x,Leaf,Leaf)
  | c(y,a,b) as z ->` (function body as in the original image)
type RBtree = Btree | Rtree

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))) |
                x -> x

        -> Red(y, Blk(x,a,b), Blk(z,c,d))

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

∀(α ≤ Unbal). α → α
Type checking the previous definitions is not so difficult.
The hard part is to type partial applications:

\[
\text{map} : (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
\]

\[
\text{balance} : (\text{Unbal} \rightarrow \text{Rtree}) \& ((\beta \\text{Unbal}) \rightarrow (\beta \\text{Unbal}))
\]

\[
\text{map} \ \text{balance} : (\left[\text{Unbal}\right] \rightarrow \left[\text{Rtree}\right])
\& (\left[\alpha \\text{Unbal}\right] \rightarrow \left[\alpha \\text{Unbal}\right])
\& (\left[\alpha \\text{Unbal}\right] \rightarrow [(\alpha \\text{Unbal}) \rightarrow \text{Rtree}])
\]

Fortunately, programmers (and you) are spared from these gory details.
New languages use union and intersections

Facebook’s Flow:

```javascript
// @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
      (if condition
          (add1 x)
          (not x)))))

G. Castagna (CNRS)
New languages using negation

Typescript

Negation types are proposed in a merge request for TypeScript:

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

```typescript
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:

```plaintext
# flatten [ 3 'r' [4 ['true 5'] ] [ "quo" ['false'] "stop" ] ] ;
- : [ (Bool | 3--5 | 'o'--'u')* ]
  = [ 3 'r' 4 true 5 'quo' false 'stop' ]
```
Full-fledged connectives for novel type expressivity

The recursive `flatten` function:

```plaintext
let flatten
  | [] -> []
  | [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:

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

``` ML

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:

``` ML
# flatten [ 3 'r' [4 ['true 5]] [ "quo" [['false] "stop"] ] ];;
- : [ (Bool | 3--5 | 'o'--'u')* ]
= [ 3 'r' 4 true 5 'quo' false 'stop' ]
```
When combined with polymorphic types, set-theoretic types can encode a limited form of bounded polymorphism:

\[ \forall (T_1 \leq \alpha \leq T_2).T \]

is encoded as

\[ T\{\alpha := (\alpha \lor T_1) \land T_2}\} \]

For instance:

\[ \text{balance} : (\text{Unbal} \rightarrow \text{Rtree}) \land (\beta \setminus \text{Unbal} \rightarrow \beta \setminus \text{Unbal}) \]

can be read as:

\[ \text{balance} : \forall (\beta \leq \text{not(\text{Unbal})}).(\text{Unbal} \rightarrow \text{Rtree}) \land (\beta \rightarrow \beta) \]

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:
  - $T_1 \lor T_2$ is the least upper bound of $T_1$ and $T_2$
  - $T_1 \land T_2$ is the greatest lower bound of $T_1$ and $T_2$
  - $\text{not}(T)$ 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., $\lor$, $\land$, $\text{not}()$) is far more difficult than for type constructors (i.e., $\rightarrow$, $\times$, $\{\ldots\}$, $\ldots$). [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:
  - \( T_1 \lor T_2 \) is the least upper bound of \( T_1 \) and \( T_2 \)
  - \( T_1 \land T_2 \) is the greatest lower bound of \( T_1 \) and \( T_2 \)
  - \( \neg T \) is the only type whose union and intersection with \( T \) yield the \( \text{Any} \) and \( \text{Empty} \) types, respectively.

- Defining (and deciding) subtyping for \textit{type connectives} (i.e., \( \lor \), \( \land \), \( \neg() \)) is far more difficult than for \textit{type constructors} (i.e., \( \rightarrow \), \( \times \), \{...\}, ...).
  [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
Each type denotes a set of values:

**Bool** is the set that contains just two values \{true, false\}.

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

**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 \(\{(v_1, v_2) \mid v_1 \in T_1, v_2 \in T_2\}\).

**\(T_1 \lor T_2\)** is the union of the sets \(T_1\) and \(T_2\), that is \(\{v \mid v \in T_1 \text{ or } v \in T_2\}\).

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

**\(\neg(T)\)** is the set of all the values not in \(T\), that is \(\{v \mid v \notin T\}\).

In particular **\(\neg(\text{Any})\)** is the empty set (written **Empty**).

**\(T_1 \rightarrow T_2\)** 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 \(T_2\).
Types as sets of values and semantic subtyping

\[ T ::= \text{Bool} | \text{Int} | \text{Any} | (T, T) | T \lor T | T \land T | \text{not}(T) | T \rightarrow T \]

Each type *denotes* a set of values:

- **Bool** is the set that contains just two values \{true, 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\}\.
- \(T_1 \lor T_2\) is the *union* of the sets \(T_1\) and \(T_2\), that is \{\(v \mid v \in T_1\) or \(v \in T_2\}\.
- \(T_1 \land T_2\) is the *intersection* of the sets \(T_1\) and \(T_2\), i.e. \{\(v \mid v \in T_1\) and \(v \in T_2\}\.
- \(\text{not}(T)\) is the set of all the values not in \(T\), that is \{\(v \mid v \notin T\}\.

In particular \(\text{not}(\text{Any})\) is the empty set (written \(\text{Empty}\)).

\(T_1 \rightarrow T_2\) 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

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

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

\[ s_2 \leq s_1 t_1 \leq t_2 \quad s_1 \rightarrow\rightarrow t_1 \leq s_2 \rightarrow\rightarrow t_2 \]

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

\[ (s_1 \lor\lor s_2) \rightarrow\rightarrow t \triangleright (s_1 \rightarrow\rightarrow t) \land\land (s_2 \rightarrow\rightarrow t) \]

Define subtyping semantically: [Hosoya, Pierce]

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

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

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

\[
\frac{s_2 \leq s_1 \quad t_1 \leq t_2}{s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2}
\]

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

\[
(s_1 \lor s_2) \rightarrow t \leq (s_1 \rightarrow t) \land (s_2 \rightarrow t)
\]
Semantic subtyping

\[ t ::= B \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

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

  \[
  \frac{s_2 \leq s_1 \quad t_1 \leq t_2}{s_1 \to s_1 \leq s_2 \to t_2}
  \]

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

  \[
  (s_1 \lor s_2) \to t \quad \supseteq \quad (s_1 \to t) \land (s_2 \to t)
  \]
Semantic subtyping

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

- **Constructor subtyping** is easy:
  - constructors do not mix, e.g.:
    \[ s_2 \leq s_1 \quad t_1 \leq t_2 \]
    \[ s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2 \]

- **Connective subtyping** is harder:
  - connectives distribute over constructors, e.g.
    \[ (s_1 \lor s_2) \rightarrow t \quad \supseteq \quad (s_1 \rightarrow t) \land (s_2 \rightarrow t) \]

Define subtyping semantically: [Hosoya, Pierce]

1. Interpret types as sets (of values)
2. Define subtyping as set containment.
**First**, define an interpretation of types into sets.

\[
\llbracket \ rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})
\]

such that

Connectives have their set-theoretic interpretation:

\[
\begin{align*}
\llbracket \neg \rrbracket &= \emptyset \\
\llbracket t_1 \lor \cdots \lor t_n \rrbracket &= \llbracket t_1 \rrbracket \cup \cdots \cup \llbracket t_n \rrbracket \\
\llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
\llbracket t_1 \land \cdots \land t_n \rrbracket &= \llbracket t_1 \rrbracket \cap \cdots \cap \llbracket t_n \rrbracket
\end{align*}
\]

Constructors have their natural interpretation:

\[
\llbracket t_1 \times \cdots \times t_n \rrbracket = \llbracket t_1 \rrbracket \times \cdots \times \llbracket t_n \rrbracket
\]

\[
\begin{align*}
\mathcal{D}_2 &\subseteq \mathcal{D} \\
\llbracket t_1 \rightarrow \cdots \rightarrow t_n \rrbracket &= \{ f | f \text{ function from } \llbracket t_1 \rrbracket \text{ to } \llbracket t_2 \rrbracket \}
\end{align*}
\]
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [[ \cdot ]] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  - \[ [[ \mathbf{0} ]] = \emptyset \]
  - \[ [[ t_1 \lor t_2 ]] = [[ t_1 ]] \cup [[ t_2 ]] \]
  - \[ [[ \neg t ]] = \mathcal{D} \setminus [[ t ]] \]
  - \[ [[ t_1 \land t_2 ]] = [[ t_1 ]] \cap [[ t_2 ]] \]
First, define an interpretation of types into sets.

\[\llbracket \cdot \rrbracket : \text{Types} \to \mathcal{P}(D)\]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  \llbracket 0 \rrbracket = \emptyset \quad \llbracket t_1 \lor t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket = D \setminus \llbracket t \rrbracket \quad \llbracket t_1 \land t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
  \]

- **Constructors** have their natural interpretation:
  \[
  \llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
  \llbracket t_1 \to t_2 \rrbracket = \{ f \mid f \text{ function from} \llbracket t_1 \rrbracket \text{ to} \llbracket t_2 \rrbracket \}\]
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [ \ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[ [ \overline{0} ] = \emptyset \]
  \[ [ t_1 \lor t_2 ] = [ t_1 ] \cup [ t_2 ] \]
  \[ [ \neg t ] = \mathcal{D} \setminus [ t ] \]
  \[ [ t_1 \land t_2 ] = [ t_1 ] \cap [ t_2 ] \]

- **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.

\[ s \leq t \overset{\text{def}}{\iff} [ s ] \subseteq [ t ] \]
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[[\ ]] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})

such that

- **Connectives** have their set-theoretic interpretation:
  \[ [\mathbf{0}] = \emptyset \quad [\mathbf{t}_1 \lor \mathbf{t}_2] = [\mathbf{t}_1] \cup [\mathbf{t}_2] \]
  \[ [\neg \mathbf{t}] = \mathcal{D}\setminus[\mathbf{t}] \quad [\mathbf{t}_1 \land \mathbf{t}_2] = [\mathbf{t}_1] \cap [\mathbf{t}_2] \]

- **Constructors** have their natural interpretation:
  \[ [\mathbf{t}_1 \times \mathbf{t}_2] = [\mathbf{t}_1] \times [\mathbf{t}_2] \]
  \[ [\mathbf{t}_1 \to \mathbf{t}_2] = \{ f \mid f \text{ function from } [\mathbf{t}_1] \text{ to } [\mathbf{t}_2] \} \]

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

\[ s \leq t \overset{\text{def}}{\iff} [s] \subseteq [t] \]
First, define an interpretation of types into sets.

\[ \llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  - \( \llbracket 0 \rrbracket = \emptyset \)
  - \( \llbracket t_1 \lor t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \)
  - \( \llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket \)
  - \( \llbracket t_1 \land t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket \)

- **Constructors** have their natural interpretation:
  - \( \llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \)
  - \( \llbracket t_1 \rightarrow t_2 \rrbracket = \{ f \mid f \text{ function from } \llbracket t_1 \rrbracket \text{ to } \llbracket t_2 \rrbracket \} \)

Then define the subtyping relation as set-containment.

\[ s \leq t \overset{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \]
First, define an interpretation of types into sets:

$$[[\ ]]: \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$$

such that:

- **Connectives** have their set-theoretic interpretation:
  
  $$[[0]] = \emptyset$$
  $$[[t_1 \lor t_2]] = [[t_1]] \cup [[t_2]]$$
  $$[[\neg t]] = \mathcal{D} \setminus [[t]]$$
  $$[[t_1 \land t_2]] = [[t_1]] \cap [[t_2]]$$

- **Constructors** have their natural interpretation:
  
  $$[[t_1 \times t_2]] = [[t_1]] \times [[t_2]]$$
  $$[[t_1 \rightarrow t_2]] = \{f \mid f \text{ function from } [[t_1]] \text{ to } [[t_2]]\}$$

Then define the subtyping relation as set-containment:

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

**Key idea**

Do not define what types *are* define *how they are related*
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [\ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  \begin{align*}
  [0] &= \emptyset \\
  [t_1 \lor t_2] &= [t_1] \cup [t_2] \\
  [-t] &= \mathcal{D} \setminus [t] \\
  [t_1 \land t_2] &= [t_1] \cap [t_2]
  \end{align*}
  \]

- **Constructors** have their natural interpretation:
  \[
  \begin{align*}
  [t_1 \times t_2] &= [t_1] \times [t_2] \\
  [t_1 \rightarrow t_2] &= \{ f \mid f \text{ function from} [t_1] \text{ to} [t_2] \}
  \end{align*}
  \]

Then define the subtyping relation as set-containment.

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

Key idea

Do not define what types *are*, define *how they are related*
Semantic subtyping: formalization

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

\[ \llbracket \cdot \rrbracket : \text{Types} \to \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[ \begin{align*}
  \llbracket 0 \rrbracket &= \emptyset \\
  \llbracket t_1 \lor t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
  \llbracket t_1 \land t_2 \rrbracket &= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
  \end{align*} \]

- **Constructors** have their natural interpretation:
  \[ \begin{align*}
  \llbracket t_1 \times t_2 \rrbracket &= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
  \llbracket t_1 \to t_2 \rrbracket &= \{ f \subseteq \mathcal{D}^2 \mid (d_1, d_2) \in f, d_1 \in \llbracket t_1 \rrbracket \Rightarrow d_2 \in \llbracket t_2 \rrbracket \}\]

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

\[ s \leq t \overset{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \]

**Key idea**

Do not define what types *are* define *how they are related*
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [\ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  \begin{align*}
  [0] &= \emptyset \\
  [t_1 \lor t_2] &= [t_1] \cup [t_2] \\
  [\neg t] &= \mathcal{D} \setminus [t] \\
  [t_1 \land t_2] &= [t_1] \cap [t_2]
  \end{align*}
  \]

- **Constructors** have their natural interpretation:
  \[
  \begin{align*}
  [t_1 \times t_2] &= [t_1] \times [t_2] \\
  [t_1 \rightarrow t_2] &= \mathcal{P}( [t_1] \times [t_2] )
  \end{align*}
  \]

Then define the subtyping relation as set-containment.

\[
 s \leq t \quad \overset{\text{def}}{\iff} \quad [s] \subseteq [t]
\]

Key idea

Do not define what types are defined how they are related
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ \text{[[ ]]} : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[ \text{[[0]]} = \emptyset \]
  \[ \text{[[} t_1 \lor t_2 ]] = \text{[[} t_1 ]] \cup \text{[[} t_2 ]] \]
  \[ \text{[[} \neg t ]] = \mathcal{D} \setminus \text{[[} t ]] \]
  \[ \text{[[} t_1 \land t_2 ]] = \text{[[} t_1 ]] \cap \text{[[} t_2 ]] \]

- ** Constructors** have their natural interpretation:
  \[ \text{[[} t_1 \times t_2 ]] = \text{[[} t_1 ]] \times \text{[[} t_2 ]] \]
  \[ \text{[[} t_1 \rightarrow t_2 ]] = \mathcal{P}(\text{[[} t_1 ]] \times \text{[[} t_2 ]]) \]

Then define the subtyping relation as set-containment.

\[ s \leq t \quad \overset{\text{def}}{\iff} \quad \text{[[} s ]] \subseteq \text{[[} t ]] \]

**Key idea**

Do not define what types *are*, define *how they are related*
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[[\_\_\_] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})\]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  [0] = \emptyset \quad [t_1 \lor t_2] = [t_1] \cup [t_2] \\
  [-t] = \mathcal{D} \setminus [t] \quad [t_1 \land t_2] = [t_1] \cap [t_2]
  \]

- **Constructors** have the same \(\subseteq\) as their natural interpretation:
  \[
  [t_1 \times t_2] = [t_1] \times [t_2] \\
  [t_1 \to t_2] = \mathcal{P}([t_1] \times [t_2])
  \]

Then define the subtyping relation as set-containment.

\[
s \leq t \quad \overset{\text{def}}{\iff} \quad [s] \subseteq [t]
\]

**Key idea**

Do not define what types *are* define *how they are related*
First, define an interpretation of types into sets.

\[ \llbracket \ \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[ \llbracket 0 \rrbracket = \emptyset \quad \llbracket \top \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \]
  \[ \llbracket \bot \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket \quad \llbracket t_1 \land t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket \]

- **Constructors** have the same \( \subseteq \) as their natural interpretation:
  \[ \llbracket s_1 \times s_2 \rrbracket \subseteq \llbracket t_1 \times t_2 \rrbracket \iff \llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket \subseteq \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \]
  \[ \llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket) \]

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

\[ s \leq t \overset{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \]

**Key idea**

Do not define what types *are* define *how they are related*
First, define an interpretation of types into sets.

\[ \llbracket \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[ \llbracket 0 \rrbracket = \emptyset \quad \llbracket t_1 \lor t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \quad \llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket \quad \llbracket t_1 \land t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket \]

- **Constructors** have **the same** \( \subseteq \) as their natural interpretation:
  \[ \llbracket s_1 \times s_2 \rrbracket \subseteq \llbracket t_1 \times t_2 \rrbracket \iff \llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket \subseteq \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \]
  \[ \llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket) \]

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

\[ s \leq t \overset{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \]

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.
Looking for $\mathcal{D}$ and $\llbracket \mathit{\cdot} \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that:

$$
\llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)
$$
1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\cdot] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that:

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

$\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[[\cdot]] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that:

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

$\mathcal{D}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
Looking for $\mathcal{D}$ and $\llbracket\cdot\rrbracket: \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that:

$$\llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)$$

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

2. $\llbracket\cdot\rrbracket_\mathcal{D}$ is defined as:
1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $\left[ \left[ \cdot \right] \right] : \text{Types} \to \mathcal{P}(\mathcal{D})$ such that:

$$\left[ s_1 \rightarrow s_2 \right] \subseteq \left[ t_1 \rightarrow t_2 \right] \iff \mathcal{P}(\left[ s_1 \right] \times \left[ s_2 \right]) \subseteq \mathcal{P}(\left[ t_1 \right] \times \left[ t_2 \right])$$

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

2. $\left[ \left[ \cdot \right] \right]_{\mathcal{D}}$ is defined as:

$$\begin{align*}
\left[0\right]_{\mathcal{D}} &= \emptyset \\
\left[1\right]_{\mathcal{D}} &= \mathcal{D} \\
\left[\neg t\right]_{\mathcal{D}} &= \mathcal{D}\setminus\left[ t \right]_{\mathcal{D}} \\
\left[ s \lor t \right]_{\mathcal{D}} &= \left[ s \right]_{\mathcal{D}} \cup \left[ t \right]_{\mathcal{D}} \\
\left[ s \land t \right]_{\mathcal{D}} &= \left[ s \right]_{\mathcal{D}} \cap \left[ t \right]_{\mathcal{D}}
\end{align*}$$
1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $\llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that:

$$
\llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)
$$

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

2. $\llbracket \cdot \rrbracket_\mathcal{D}$ is defined as:

$$
\begin{align*}
\llbracket 0 \rrbracket_\mathcal{D} &= \emptyset \\
\llbracket 1 \rrbracket_\mathcal{D} &= \mathcal{D} \\
\llbracket s \lor t \rrbracket_\mathcal{D} &= \llbracket s \rrbracket_\mathcal{D} \cup \llbracket t \rrbracket_\mathcal{D} \\
\llbracket s \land t \rrbracket_\mathcal{D} &= \llbracket s \rrbracket_\mathcal{D} \cap \llbracket t \rrbracket_\mathcal{D} \\
\llbracket s \times t \rrbracket_\mathcal{D} &= \llbracket s \rrbracket_\mathcal{D} \times \llbracket t \rrbracket_\mathcal{D} \\
\llbracket t \rightarrow s \rrbracket_\mathcal{D} &= \mathcal{P}_f(\llbracket t \rrbracket_\mathcal{D} \times \llbracket s \rrbracket_\mathcal{D})
\end{align*}
$$
1: An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\cdot]:$ Types $\rightarrow \mathcal{P}(\mathcal{D})$ such that:

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

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

2. $[[\cdot]]_\mathcal{D}$ is defined as:

$$[[0]]_\mathcal{D} = \emptyset \quad [[1]]_\mathcal{D} = \mathcal{D} \quad [[\neg t]]_\mathcal{D} = \mathcal{D} \setminus [[t]]_\mathcal{D}$$

$$[[s \lor t]]_\mathcal{D} = [[s]]_\mathcal{D} \cup [[t]]_\mathcal{D} \quad [[s \land t]]_\mathcal{D} = [[s]]_\mathcal{D} \cap [[t]]_\mathcal{D}$$

$$[[s \times t]]_\mathcal{D} = [[s]]_\mathcal{D} \times [[t]]_\mathcal{D} \quad [[t \rightarrow s]]_\mathcal{D} = \mathcal{P}_f([[t]]_\mathcal{D} \times [[s]]_\mathcal{D})$$
An interpretation that satisfies the previous constraints.

Looking for $\mathcal{D}$ and $[\ ] : \textbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that:

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

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

2. $[\ ]_\mathcal{D}$ is defined as:

$$[0]_\mathcal{D} = \emptyset \quad [1]_\mathcal{D} = \mathcal{D} \quad [\neg t]_\mathcal{D} = \mathcal{D} \setminus [t]_\mathcal{D}$$

$$[s \lor t]_\mathcal{D} = [s]_\mathcal{D} \cup [t]_\mathcal{D} \quad [s \land t]_\mathcal{D} = [s]_\mathcal{D} \cap [t]_\mathcal{D}$$

$$[s \times t]_\mathcal{D} = [s]_\mathcal{D} \times [t]_\mathcal{D} \quad [t \rightarrow s]_\mathcal{D} = \mathcal{P}f([t]_\mathcal{D} \times [s]_\mathcal{D})$$

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 \( D \) and \( \llbracket \ ] : \text{Types} \rightarrow \mathcal{P}(D) \) such that:

\[
\llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)
\]

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

2. \( \llbracket \ ]_D \) is defined as:

\[
\begin{align*}
\llbracket 0 \rrbracket_D &= \emptyset & \llbracket 1 \rrbracket_D &= D & \llbracket \neg t \rrbracket_D &= D \setminus \llbracket t \rrbracket_D \\
\llbracket s \lor t \rrbracket_D &= \llbracket s \rrbracket_D \cup \llbracket t \rrbracket_D & \llbracket s \land t \rrbracket_D &= \llbracket s \rrbracket_D \cap \llbracket t \rrbracket_D & \llbracket s \times t \rrbracket_D &= \llbracket s \rrbracket_D \times \llbracket t \rrbracket_D \\
\llbracket s \rightarrow t \rrbracket_D &= \mathcal{P}(\llbracket s \rrbracket_D \times \llbracket t \rrbracket_D)
\end{align*}
\]

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 \( \llbracket \ ]_{D'} \)

\[
t_1 \leq_{D'} t_2 \implies t_1 \leq_D t_2
\]
2: An algorithm to decide \( t_1 \leq t_2 \).

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

\[
t_1 \leq t_2 \iff \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket \iff \llbracket t_1 \land \neg t_2 \rrbracket = \emptyset \iff t_1 \land \neg t_2 \leq 0
\]
2: An algorithm to decide $t_1 \leq t_2$.

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

$t_1 \leq t_2 \iff [[t_1]] \subseteq [[t_2]] \iff [[t_1 \wedge \neg t_2]] = \emptyset \iff t_1 \wedge \neg t_2 \leq 0$

**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 $a ::= b \mid t \times t \mid t \rightarrow t \mid 0 \mid 1$ 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:

\[
t_1 \leq t_2 \iff \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket \iff \llbracket t_1 \land \neg t_2 \rrbracket = \emptyset \iff t_1 \land \neg t_2 \leq 0
\]

**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 $a ::= b \mid t \times t \mid t \rightarrow t \mid 0 \mid 1$ and $\ell ::= a \mid \neg a$

**Step 3:** Simplify mixed intersections:

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

- $(t_1 \times t_2) \land (t_1 \rightarrow t_2) \leq 0$ is always true
- $(t_1 \times t_2) \land \neg (t_1 \rightarrow t_2) \leq 0$ holds iff $t_1 \times t_2 \leq 0$. 
2: An algorithm to decide \( t_1 \leq t_2 \).

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

\[
\begin{align*}
t_1 \leq t_2 & \iff [t_1] \subseteq [t_2] \iff [t_1 \land \neg t_2] = \emptyset \iff t_1 \land \neg t_2 \leq 0
\end{align*}
\]

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

\[
\bigvee \bigwedge \ell_{ij}
\]

where \( a ::= b \mid t \times t \mid t \rightarrow t \mid 0 \mid 1 \) and \( \ell ::= a \mid \neg a \)

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

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

- \( (t_1 \times t_2) \land (t_1 \rightarrow t_2) \leq 0 \) is always true
- \( (t_1 \times t_2) \land \neg(t_1 \rightarrow t_2) \leq 0 \) holds iff \( t_1 \times t_2 \leq 0 \).

The problem is reduced to deciding:

\[
\bigwedge_{i \in I} s_i \times t_i \bigwedge_{j \in J} \neg(s_j \times t_j) \leq 0 \quad \text{and} \quad \bigwedge_{i \in I} s_i \rightarrow t_i \bigwedge_{j \in J} \neg(s_j \rightarrow t_j) \leq 0
\]

(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:

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

Step 5: Memoize (for recursive types) and recurse.
Application to a language.
Language

Syntax

**Exprs**

\[ e ::= x \quad \text{variables} \]
\[ \lambda^{i \in I : s_i \rightarrow t_i} x.e \quad \text{abstractions} \]
\[ e e \quad \text{applications} \]
\[ (e, e) \quad \text{pairs} \]
\[ \pi_i e \quad \text{projections, } i = 1, 2 \]
\[ (x = e \in t) ? e : e \quad \text{binding type case} \]

**Values**

\[ v ::= (v, v) \]
\[ \lambda^{i \in I : s_i \rightarrow t_i} x.e \]
Language

Syntax

Exprs  \[ e ::= x \text{ variables} \]
| \[ \lambda^{i \in I, s_i \rightarrow t_i} x.e \text{ abstractions} \]
| \[ e e \text{ applications} \]
| \[ (e, e) \text{ pairs} \]
| \[ \pi_i e \text{ projections, } i = 1, 2 \]
| \[ (x = e \in t) ? e : e \text{ binding type case} \]

Values  \[ v ::= (v, v) \]
| \[ \lambda^{i \in I, s_i \rightarrow t_i} x.e \]

Semantics

\[ (\lambda^{i \in I, s_i \rightarrow t_i} x.e) v \rightarrow e[v/x] \]
\[ \pi_i (v_1, v_2) \rightarrow v_i \quad i = 1, 2 \]
\[ (x = v \in t) ? e_1 : e_2 \rightarrow e_1[v/x] \quad v \in t \]
\[ (x = v \in t) ? e_1 : e_2 \rightarrow e_2[v/x] \quad v \not\in t \]
Typing

\[
\frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'}
\]

[SUBSUMPTION]
Typing

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

\[ \text{[APP]} \quad \frac { \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1 } { \Gamma \vdash e_1 e_2 : t_2 } \]

\[ \text{[ABS]} \quad \frac { \forall i \in I \quad \Gamma, x : s_i \vdash e : t_i } { \Gamma \vdash \lambda^\wedge_{i \in I} x. e : \wedge_{i \in I} s_i \rightarrow t_i } \]
Typing

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

\[\text{[APP]} \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}\]

\[\text{[ABS]} \quad \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge i \in I s_i \rightarrow t_i} x. e : \wedge i \in I s_i \rightarrow t_i}\]
Typing

[SUBSUMPTION] \[
\Gamma \vdash e : t \\
\Gamma \vdash e : t' \\
\Gamma \vdash e : t \leq t'
\]

[APP] \[
\Gamma \vdash e_1 : t_1 \rightarrow t_2 \\
\Gamma \vdash e_2 : t_1 \\
\Gamma \vdash e_1 e_2 : t_2
\]

[ABS] \[
\forall i \in I \\
\Gamma, x : s_i \vdash e : t_i \\
\Gamma \vdash \lambda \wedge_{i \in I} s_i \rightarrow t_i \ x.e : \wedge_{i \in I} s_i \rightarrow t_i
\]

[SEL] \[
\Gamma \vdash e : (t_1, t_2) \\
\Gamma \vdash \pi_i e : t_i
\]

[PAIR] \[
\Gamma \vdash e_1 : t_1 \\
\Gamma \vdash e_2 : t_2 \\
\Gamma \vdash (e_1, e_2) : t_1 \times t_2
\]
Typing

\[
\begin{align*}
\text{[SUBSUMPTION]} & \\
\Gamma \vdash e : t & \quad t \leq t' & \quad \Gamma \vdash e : t'
\end{align*}
\]

\[
\begin{align*}
\text{[APP]} & \\
\Gamma \vdash e_1 : t_1 \rightarrow t_2 & \quad \Gamma \vdash e_2 : t_1 & \quad \Gamma \vdash e_1 e_2 : t_2
\end{align*}
\]

\[
\begin{align*}
\text{[ABS]} & \\
\forall i \in I & \quad \Gamma, x : s_i \vdash e : t_i & \quad \Gamma \vdash \lambda \bigwedge_{i \in I} s_i \rightarrow t_i . e \bigwedge_{i \in I} s_i \rightarrow t_i
\end{align*}
\]

\[
\begin{align*}
\text{[SEL]} & \\
\Gamma \vdash e : (t_1, t_2) & \quad \Gamma \vdash \pi_i e : t_i
\end{align*}
\]

\[
\begin{align*}
\text{[PAIR]} & \\
\Gamma \vdash e_1 : t_1 & \quad \Gamma \vdash e_2 : t_2 & \quad \Gamma \vdash (e_1, e_2) : t_1 \times t_2
\end{align*}
\]

\[
\begin{align*}
\text{[TYPECASE]} & \\
\Gamma \vdash e : t_0 & \quad \Gamma, x : s_1 \vdash e_1 : t_1 & \quad \Gamma, x : s_2 \vdash e_2 : t_2 & \quad s_1 \equiv t_0 \land t & \quad s_2 \equiv t_0 \land \neg t
\end{align*}
\]

\[
\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i|s_i \neq \emptyset\}} t_i
\]
Typing

\[
\begin{align*}
\text{[SUBSUMPTION]} & \quad \frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'} \\
\text{[APP]} & \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 \ e_2 : t_2} \\
\text{[ABS]} & \quad \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda \ (\land_{i \in I} s_i \rightarrow t_i \ x \ e) : \land_{i \in I} s_i \rightarrow t_i} \\
\text{[SEL]} & \quad \frac{\Gamma \vdash e : (t_1 , t_2)}{\Gamma \vdash \pi_i \ e : t_i} \\
\text{[PAIR]} & \quad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \\
\text{[TYPECASE]} & \quad \frac{\Gamma \vdash e : t_0 \quad \Gamma, x : s_1 \vdash e_1 : t_1 \quad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t) \ ? e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \\
& \quad s_1 \equiv t_0 \land t \\
& \quad s_2 \equiv t_0 \land \neg t
\end{align*}
\]
Typing

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

\[\text{[APP]} \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}\]

\[\text{[ABS]} \quad \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda \land_{i \in I} s_i \rightarrow t_i \cdot x.e : \land_{i \in I} s_i \rightarrow t_i}\]

\[\text{[SEL]} \quad \frac{\Gamma \vdash e : (t_1, t_2)}{\Gamma \vdash \pi_i e : t_i}\]

\[\text{[PAIR]} \quad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\]

\[\text{[TYPECASE]} \quad \frac{\Gamma \vdash e : t_0 \quad \Gamma, x : s_1 \vdash e_1 : t_1 \quad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t) \cdot ?e_1 : e_2 : \bigvee_{\{i \mid s_i \neq 0\}} t_i} \quad s_1 \equiv t_0 \land t \quad s_2 \equiv t_0 \land \neg t\]

A form of occurrence typing
Typing

\[\Gamma \vdash e : t \quad t \leq t' \quad \Gamma \vdash e : t'\]

\[\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1 \quad \Gamma \vdash e_1 e_2 : t_2\]

\[\Gamma \vdash e : (t_1, t_2) \quad \Gamma \vdash \pi_i e : t_i\]

\[\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i\]

\[\forall i \in I \quad \Gamma \vdash \lambda^{\land i \in I s_i \rightarrow t_i} x. e : \land i \in I s_i \rightarrow t_i\]

\[\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash (e_1, e_2) : t_1 \times t_2\]

\[\Gamma \vdash e : t_0 \quad \Gamma, x : s_1 \vdash e_1 : t_1 \quad \Gamma, x : s_2 \vdash e_2 : t_2 \quad \Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{i \mid s_i \neq 0} t_i\]

\[s_1 \equiv t_0 \land t \quad s_2 \equiv t_0 \land \neg t\]
Typing

\[ \frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'} \]

\[ \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \]

\[ \frac{\forall i \in I \quad \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda \bigwedge_{i \in I} s_i \rightarrow t_i \; x . e : \bigwedge_{i \in I} s_i \rightarrow t_i} \]

\[ \frac{\Gamma \vdash e : (t_1, t_2)}{\Gamma \vdash \pi_i e : t_i} \]

\[ \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \]

\[ \frac{\Gamma \vdash e : t_0 \quad \Gamma, x : s_1 \vdash e_1 : t_1 \quad \Gamma, x : s_2 \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \quad s_1 \equiv t_0 \wedge t \quad s_2 \equiv t_0 \wedge \neg t \]

Necessary for typing overloaded functions:

\[ \lambda (\text{Int} \rightarrow \text{Int}) \land (\text{Bool} \rightarrow \text{Bool}) \; x . (y = x \in \text{Int})? (y + 1) : \text{not}(y) \]
Typing

\[
\begin{align*}
\text{[SUBSUMPTION]} & \quad \Gamma \vdash e : t \quad t \leq t' \\
& \Rightarrow \quad \Gamma \vdash e : t'
\end{align*}
\]

\[
\begin{align*}
\text{[APP]} & \quad \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1 \\
& \Rightarrow \quad \Gamma \vdash e_1 e_2 : t_2
\end{align*}
\]

\[
\begin{align*}
\text{[ABS]} & \quad \forall i \in I \quad \Gamma, x : s_i \vdash e : t_i \\
& \Rightarrow \quad \Gamma \vdash \lambda^{\wedge i \in I s_i \rightarrow t_i} x . e : \wedge_{i \in I} s_i \rightarrow t_i
\end{align*}
\]

\[
\begin{align*}
\text{[SEL]} & \quad \Gamma \vdash e : (t_1, t_2) \\
& \Rightarrow \quad \Gamma \vdash \pi_i e : t_i
\end{align*}
\]

\[
\begin{align*}
\text{[PAIR]} & \quad \Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \\
& \Rightarrow \quad \Gamma \vdash (e_1, e_2) : t_1 \times t_2
\end{align*}
\]

\[
\begin{align*}
\text{[TYPECASE]} & \quad \Gamma \vdash e : t_0 \\
& \Rightarrow \quad \Gamma, x : s_1 \vdash e_1 : t_1 \\
& \Rightarrow \quad \Gamma, x : s_2 \vdash e_2 : t_2 \\
& \Rightarrow \quad \Gamma \vdash (x = e \in t) ? e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i \\
\end{align*}
\]

The type system is sound
function double (x) {
    (typeof(x) === "number") ? 2*x : x.concat(x)
}
function double (x) {
  (typeof(x) === "number") ? 2*x : x.concat(x)
}

\[ \lambda^t x. (y = x \in \text{Int})? (2 \ast y): (y \cdot \text{concat}(y)) \] (1)
function double (x) {
    (typeof(x) === "number") ? 2*x : x.concat(x)
}

\[\lambda^t x. (y = x \in Int)?(2 \ast y):(y . concat(y))\] (1)

**Exercise**

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

- \( t = (\text{Int} \lor \text{String}) \rightarrow (\text{Int} \lor \text{String}) \)
- \( t = (\text{Int} \rightarrow \text{Int}) \land (\text{String} \rightarrow \text{String}) \)

where \( \text{String} = \mu X . \{ \text{concat} : X \rightarrow X \} \)
What about the interpretation of types as set of “values”?

I interpreted types into subsets of $D$ rather than into sets of:

Values $v ::= (v, v) | \lambda i \in I s_i \rightarrow t_i$.

Define a new interpretation of types:

$[[t]]_V = \{ v | \vdash v : t \}$

This induces a new subtyping relation:

$t \leq V s \iff [[t]]_V \subset [[s]]_V$

Actually, it is not a new one ... it is the old one:

Theorem [Frisch, Castagna, Benzaken 2002&2008]

$t \leq V s \iff t \leq D s$ where $\leq D$ is the subtyping via $D$ and used to define $\vdash v : t$.
What about the interpretation of types as set of “values”?

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

$$\text{Values} \quad \nu ::= (\nu, \nu) \mid \lambda_{i \in I} s_i \rightarrow t_i \ x \ . e$$
What about the interpretation of types as set of “values”?

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

$$\text{Values} \quad \nu ::= (\nu, \nu) \mid \lambda^{i \in I} s_i \to t_i \ x. e$$

Define a new interpretation of types:

$$\llbracket t \rrbracket_\nu = \{ \nu \mid \vdash \nu : t \}$$
What about the interpretation of types as set of “values”?

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

\[
\text{Values} \quad \nu \ ::= \ (\nu, \nu) \mid \lambda^{i \in I} s_i \rightarrow t_i \, x.\, e
\]

Define a new interpretation of types:

\[
\llbracket t \rrbracket_{\nu} = \{\nu \mid \vdash \nu : t\}
\]

This induces a new subtyping relation:

\[
t \leq_{\nu} s \iff \llbracket t \rrbracket_{\nu} \subseteq \llbracket s \rrbracket_{\nu}
\]
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:

$$\text{Values} \quad v ::= (v, v) \mid \lambda i \in I \rightarrow t_i x. e$$

Define a new interpretation of types:

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

This induces a new subtyping relation:

$$t \leq_{\text{V}} s \iff [[t]]_{\text{V}} \subset [[s]]_{\text{V}}$$

Actually, it is not a new one ... it is the old one:

**Theorem [Frisch, Castagna, Benzaken 2002&2008]**

$$t \leq_{\text{V}} s \iff t \leq_{\mathcal{D}} s$$

where $\leq_{\mathcal{D}}$ is the subtyping via $\mathcal{D}$ and used to define $\vdash v : t$
Closing the circle

Was then $D$ really necessary?
Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition.
Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

$$[[t]]_\nu$$
Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined.
We are in a circular definition

\[
\lambda \vdash v : t
\]
Closing the circle

Was then \( \mathcal{D} \) really necessary?

YES!

\( \lambda \)-abstractions are values and need (sub)typing to be defined. We are in a circular definition

\[
\vdash e : t \quad \vdash v : t
\]

\( \llbracket t \rrbracket \nu \)
Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

\[
\begin{align*}
\vdash e : t & \quad \vdash v : t \\
\Downarrow & \\
\mathbb{G} & \vdash (t \leq t) \\
\Downarrow & \\
\mathbb{V} & \vdash [t]_V
\end{align*}
\]
Was then $D$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition
Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

\[
\begin{align*}
t & \leq t \\
\vdash e : t & \quad \vdash v : t \\
[t] \nu & \quad t \leq [t] \nu
\end{align*}
\]
Closing the circle

Was then $\cal D$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

$t \leq t \quad [t]_\nu$

$\vdash e : t \quad \vdash v : t$
Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

$$\lambda e : t \vdash v : t$$

$$t \leq t \quad \llbracket t \rrbracket \mathcal{D}$$
Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

$$t \leq t \quad [t]_{\mathcal{D}}$$

$$\vdash e : t \quad \vdash v : t$$
Was then $\mathcal{D}$ really necessary?

YES!

\(\lambda\)-abstractions are values and need (sub)typing to be defined. We are in a circular definition

\[
\begin{align*}
\vdash e : t & \quad \vdash v : t \\
\vdash \lambda x. e : [t] & \quad [t]_{\mathcal{D}}
\end{align*}
\]
Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

$$
t \leq t \quad \vdash e : t \quad \vdash v : t
$$

$$
\lambda - abstractions \text{ are values and need (sub)typing to be defined. We are in a circular definition.}
$$

$$
\vdash e : t \quad \vdash v : t
$$

$$
\vdash e : t \quad \vdash v : t
$$
Closing the circle

Was then $\mathcal{D}$ really necessary?

**YES!**

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

\[
\begin{align*}
& t \leq t & \Rightarrow & \llbracket t \rrbracket \mathcal{D} \\
& \vdash e : t & \Rightarrow & \vdash v : t
\end{align*}
\]
Closing the circle

Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-abstractions are values and need (sub)typing to be defined. We are in a circular definition

\[
\begin{align*}
\vdash e : t & \quad \vdash v : t \\
\implies t \leq t & \quad \implies [t]_{\mathcal{D}} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad [t]_{\mathcal{V}}
\end{align*}
\]
Was then $\mathcal{D}$ really necessary?

YES!

$\lambda$-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!

$\lambda$-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

10. Set-theoretic types

11. Semantic Subtyping

12. Application to a language.

13. Adding Parametric Polymorphism: the Types

Motivating examples: reminder 1

The recursive `flatten` function:
Motivating examples: reminder 1

**The recursive flatten function:**

(* recursive type with union intersection and negation *)

\[
\text{type Tree(}\alpha\text{) = (}\alpha\setminus[\text{Any*}]) \mid \left[ (\text{Tree(}\alpha\text{)})^* \right]
\]

(* recursive flatten written in polymorphic CDuce *)

\[
\text{let flatten ( (Tree(}\alpha\text{)) -> [}\alpha^*\] )}
\]
\[
\mid [] \rightarrow []
\]
\[
\mid [h ; t] \rightarrow (\text{flatten h})(\text{flatten t})
\]
\[
\mid x \rightarrow [x]
\]
Motivating examples: reminder 1

**The recursive flatten function:**

(* recursive type with union intersection and negation *)

\[
\text{type Tree}(\alpha) = (\alpha \setminus \text{Any}*]) | [ (\text{Tree}(\alpha))* ]
\]

(* recursive flatten written in polymorphic CDuce *)

```
let flatten ( (Tree(\alpha)) -> [\alpha*] )
 | []  -> []
 | [h ; t]  -> (flatten h)@(flatten t)
 | x   -> [x]
```

**Rationale**

The language does not change apart from the fact that type variables such as \(\alpha\) may occur in type annotations.
Type refinement of balance for red-black trees
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

\[ t ::= B | t \times t | t \rightarrow t | t \lor t | t \land t | \neg t | 0 | 1 \]
Naive solution

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

Idea:
Use the previous relation since is defined for "ground types"

Let \( \sigma : \text{Vars} \rightarrow \text{ClosedTypes} \) denote ground substitutions. Define:

\[ s \leq t \text{ def } \iff \forall \sigma. \, [s]_\sigma \subseteq [t]_\sigma \]

THIS IS A WRONG WAY: TOO MANY PROBLEMS
Naive solution

\[ t ::= B \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

**Idea:** Use the previous relation since is defined for “ground types”

Let \( \sigma : \text{Vars} \to \text{ClosedTypes} \) denote ground substitutions. Define:

\[ s \leq t \overset{\text{def}}{\iff} \forall \sigma \cdot s\sigma \leq t\sigma \]

THIS IS A WRONG WAY: TOO MANY PROBLEMS
Naive solution

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

**Idea:** Use the previous relation since is defined for “ground types”

Let \( \sigma : \text{Vars} \rightarrow \text{ClosedTypes} \) denote ground substitutions. Define:

\[ s \leq t \overset{\text{def}}{\iff} \forall \sigma . s\sigma \leq t\sigma \]

or equivalently

\[ s \leq t \overset{\text{def}}{\iff} \forall \sigma . [[[s\sigma]]] \subseteq [[[t\sigma]]] \]
Naive solution

\[ t ::= B \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

**Idea:** Use the previous relation since is defined for “ground types”

Let \( \sigma : \text{Vars} \to \text{ClosedTypes} \) denote ground substitutions. Define:

\[ s \leq t \overset{\text{def}}{\iff} \forall \sigma. s\sigma \leq t\sigma \]

or equivalently

\[ s \leq t \overset{\text{def}}{\iff} \forall \sigma. [s\sigma] \subseteq [t\sigma] \]

**THIS IS A WRONG WAY: TOO MANY PROBLEMS**
Problems with the naive solution

Haruo Hosoya conjectured that deciding \( \forall \sigma. s\sigma \leq t\sigma \) is \textit{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:
Haruo Hosoya conjectured that deciding $\forall \sigma. \ s\sigma \leq t\sigma$ is \textit{at least} as hard as solving Diophantine equations.

It \textit{breaks} parametricity:

\[
(t \times \alpha) \leq (t \times \lnot t) \lor (\alpha \times t) \tag{2}
\]
Problems with the naive solution

1 Haruo Hosoya conjectured that deciding $\forall \sigma. \ s\sigma \leq t\sigma$ is \textit{at least} as hard as solving Diophantine equations

2 It \textit{breaks} parametricity:

\[
(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)
\]  

This inclusion holds if and only if $t$ is an \textit{indivisible} type (\textit{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 \textit{at least} as hard as solving Diophantine equations.

2. It \textit{breaks} parametricity:

$$\quad (t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)$$

This inclusion holds if and only if $t$ is an \textit{indivisible} type (\textit{eg.}, a singleton or a basic type):

<table>
<thead>
<tr>
<th>Property of indivisible types</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $t$ is an \textit{indivisible type}, then for all possible interpretations of $\alpha$</td>
</tr>
<tr>
<td>$t \leq \alpha$ or $\alpha \leq \neg t$</td>
</tr>
<tr>
<td>holds.</td>
</tr>
</tbody>
</table>
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:

$$ (t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t) $$

This inclusion holds if and only if $t$ is an indivisible type (e.g., a singleton or a basic type):

Property of indivisible types

If $t$ is an indivisible type, then for all possible interpretations of $\alpha$

$$ t \leq \alpha \quad \text{or} \quad \alpha \leq \neg t $$

holds.

- If $\alpha \leq \neg t$ then the left element of the union in (2) suffices;
- If $t \leq \alpha$, then $\alpha = (\alpha \setminus t) \lor t$. Thus $(t \times \alpha) = (t \times (\alpha \setminus t)) \lor (t \times t)$. This union is contained component-wise in the one in (2).
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

holds if and only if \( t \) is \textit{indivisible} is really catastrophic:
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

holds if and only if \(t\) is \textit{indivisible} is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

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

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

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

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

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

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 $i$

$$i \leq \alpha \quad \text{or} \quad \alpha \leq \neg i$$

validity can stutter from one formula to another, missing in this way the uniformity typical of parametricity.
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:

\[ \eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]
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:

\[ \eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]

and now the interpretation function takes an extra parameter

\[ \llbracket \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})^{\text{Vars}} \rightarrow \mathcal{P}(\mathcal{D}) \]
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:

\[ \eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]

and now the interpretation function takes an extra parameter

\[ \llbracket \ \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})^{\text{Vars}} \rightarrow \mathcal{P}(\mathcal{D}) \]

with

\[
\begin{align*}
\llbracket \alpha \rrbracket \eta &= \eta(\alpha) \\
\llbracket t_1 \lor t_2 \rrbracket \eta &= \llbracket t_1 \rrbracket \eta \cup \llbracket t_2 \rrbracket \eta \\
\llbracket \top \rrbracket \eta &= \emptyset \\
\llbracket \bot \rrbracket \eta &= \emptyset \\
\llbracket \neg t \rrbracket \eta &= \mathcal{D} \setminus \llbracket t \rrbracket \eta \\
\llbracket t_1 \land t_2 \rrbracket \eta &= \llbracket t_1 \rrbracket \eta \cap \llbracket t_2 \rrbracket \eta \\
\llbracket 0 \rrbracket \eta &= \mathcal{D}
\end{align*}
\]
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:

\[ \eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]

and now the interpretation function takes an extra parameter

\[ \llbracket \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})^\text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]

with

\[
\begin{align*}
\llbracket \alpha \rrbracket \eta &= \eta(\alpha) \\
\llbracket t_1 \lor t_2 \rrbracket \eta &= \llbracket t_1 \rrbracket \eta \cup \llbracket t_2 \rrbracket \eta \\
\llbracket t_1 \land t_2 \rrbracket \eta &= \llbracket t_1 \rrbracket \eta \cap \llbracket t_2 \rrbracket \eta \\
\llbracket 0 \rrbracket \eta &= \emptyset \\
\llbracket 1 \rrbracket \eta &= \mathcal{D}
\end{align*}
\]

and such that it satisfies:

\[
\llbracket t_1 \rightarrow s_1 \rrbracket \eta \subseteq \llbracket t_2 \rightarrow s_2 \rrbracket \eta \iff \mathcal{P}(\llbracket t_1 \rrbracket \eta \times \llbracket s_1 \rrbracket \eta) \subseteq \mathcal{P}(\llbracket t_2 \rrbracket \eta \times \llbracket s_2 \rrbracket \eta)
\]
In this framework the natural definition of subtyping is

\[ s \leq t \iff \forall \eta. \llbracket s \rrbracket \eta \subseteq \llbracket t \rrbracket \eta \]

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

$$\forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset)$$
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \[ \forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \] — 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

\[
\forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset)
\]

- **It avoids stuttering:** \(\forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset)\) —that is, \((t \leq \alpha \text{ or } \alpha \leq \neg t)\) — 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].
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \[ \forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \] — that is, \((t \leq \alpha \text{ or } \alpha \leq \neg t)\) 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

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \[ \forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \] — that is, \((t \leq \alpha \text{ or } \alpha \leq \neg t)\) 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

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \( \forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \) — that is, \( (t \leq \alpha \text{ or } \alpha \leq \neg t) \) 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)
Examples of subtyping relations
Examples

We can internalize properties such as:

\[(\alpha \rightarrow \gamma) \land (\beta \rightarrow \gamma) \sim \alpha \lor \beta \rightarrow \gamma\]
Examples

We can internalize properties such as:

\[(\alpha \rightarrow \gamma) \land (\beta \rightarrow \gamma) \sim \alpha \lor \beta \rightarrow \gamma\]

or distributivity laws:

\[(\alpha \lor \beta \times \gamma) \sim (\alpha \times \gamma) \lor (\beta \times \gamma)\]
Examples

We can internalize properties such as:

\[(\alpha \rightarrow \gamma) \land (\beta \rightarrow \gamma) \sim \alpha \lor \beta \rightarrow \gamma\]

or distributivity laws:

\[(\alpha \lor \beta \times \gamma) \sim (\alpha \times \gamma) \lor (\beta \times \gamma)\]

and combining them deduce:

\[(\alpha \times \gamma \rightarrow \delta_1) \land (\beta \times \gamma \rightarrow \delta_2) \leq (\alpha \lor \beta \times \gamma) \rightarrow \delta_1 \lor \delta_2\]
Examples

We can internalize properties such as:

\[(\alpha \to \gamma) \land (\beta \to \gamma) \sim \alpha \lor \beta \to \gamma\]

or distributivity laws:

\[(\alpha \lor \beta \times \gamma) \sim (\alpha \times \gamma) \lor (\beta \times \gamma)\]

and combining them deduce:

\[(\alpha \times \gamma \to \delta_1) \land (\beta \times \gamma \to \delta_2) \leq (\alpha \lor \beta \times \gamma) \to \delta_1 \lor \delta_2\]

Of course the problematic relation never holds, whatever the \( t \):

\[(t \times \alpha) \not\leq (t \times \neg t) \lor (\alpha \times t)\]
We can prove relevant relations on infinite types, *eg.*, for the type of generic \( \alpha \)-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 \(\alpha\)-lists:

\[
\alpha\text{-list} = \mu z. (\alpha \times z) \lor \text{nil}
\]

we can prove that it contains both the \(\alpha\)-lists of even length

\[
\mu z. (\alpha \times (\alpha \times z)) \lor \text{nil} \leq \mu z. (\alpha \times z) \lor \text{nil}
\]

\(\alpha\)-lists of even length \(\alpha\)-lists

and the \(\alpha\)-lists with of odd length

\[
\mu z. (\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}) \leq \mu z. (\alpha \times z) \lor \text{nil}
\]

\(\alpha\)-lists of odd length \(\alpha\)-lists
We can prove relevant relations on infinite types, \textit{eg.}, for the type of generic \( \alpha \)-lists:

\[
\alpha\text{-list} = \mu z.(\alpha \times z) \lor \text{nil}
\]

we can prove that it contains both the \( \alpha \)-lists of even length

\[
\mu z.(\alpha \times (\alpha \times z)) \lor \text{nil} \leq \mu z.(\alpha \times z) \lor \text{nil}
\]

and the \( \alpha \)-lists with of odd length

\[
\mu z.(\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}) \leq \mu z.(\alpha \times z) \lor \text{nil}
\]

and that it is itself contained in the union of the two, that is:

\[
\alpha\text{-list} \sim (\mu z.(\alpha \times (\alpha \times z)) \lor \text{nil}) \lor (\mu z.(\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}))
\]
We can prove relevant relations on infinite types, eg., for the type of generic $\alpha$-lists:

$$\alpha\text{-list} = \mu z. (\alpha \times z) \lor \text{nil}$$

we can prove that it contains both the $\alpha$-lists of even length

$$\mu z. (\alpha \times (\alpha \times z)) \lor \text{nil} \leq \mu z. (\alpha \times z) \lor \text{nil}$$

and the $\alpha$-lists with of odd length

$$\mu z. (\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}) \leq \mu z. (\alpha \times z) \lor \text{nil}$$

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z. (\alpha \times (\alpha \times z)) \lor \text{nil}) \lor (\mu z. (\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}))$$

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:*

\[ t_1 \leq t_2 \iff \forall \eta. [[t_1]]\eta \subseteq [[t_2]]\eta \iff \forall \eta. [[t_1 \land \neg t_2]]\eta = \emptyset \iff t_1 \land \neg t_2 \leq \emptyset \]

\[ \land \not\land \times \to 0 1 \alpha \alpha \]

\[ \lor \land \not\land \times \to 0 1 \alpha \alpha \]

\[ \land \not\land \times \to 0 1 \alpha \alpha \]
Subtyping Algorithm: $t_1 \leq t_2$

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

$$t_1 \leq t_2 \iff \forall \eta. [[t_1]]\eta \subseteq [[t_2]]\eta \iff \forall \eta. [[t_1 \land \neg t_2]]\eta = \emptyset \iff t_1 \land \neg t_2 \leq 0$$

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

$$\bigvee_i \bigwedge_{j \in J} \ell_{ij}$$
where $a ::= b \mid t \times t \mid t \to t \mid 0 \mid 1 \mid \alpha$ and $\ell ::= a \mid \neg a$
Subtyping Algorithm: $t_1 \leq t_2$

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

$$t_1 \leq t_2 \iff \forall \eta. [t_1] \eta \subseteq [t_2] \eta \iff \forall \eta. [t_1 \land \neg t_2] \eta = \emptyset \iff t_1 \land \neg t_2 \leq 0$$

**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 $a ::= b \mid t \times t \mid t \rightarrow t \mid 0 \mid 1 \mid \alpha$ 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.**

\[ \forall \eta. \llbracket t \rrbracket \eta = \emptyset \iff \forall \eta. \llbracket t[\neg \alpha/\alpha] \rrbracket \eta = \emptyset \]

so replace \( \neg \beta_k \) for \( \beta_k \) (for all \( k \in K \))

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.

\[ \forall \eta. [t] \eta = \emptyset \iff \forall \eta. [t[\neg \alpha / \alpha]] \eta = \emptyset \]

so replace \( \neg \beta_k \) for \( \beta_k \) (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.

\[ \bigwedge_{t_1 \times t_2 \in P} t_1 \times t_2 \bigwedge_{h \in H} \alpha_h \leq \bigvee_{t'_1 \times t'_2 \in N} t'_1 \times t'_2 \]

holds if and only if

\[ \bigwedge_{t_1 \times t_2 \in P} t_1 \sigma \times t_2 \sigma \bigwedge_{h \in H} \gamma^1_h \times \gamma^2_h \leq \bigvee_{t'_1 \times t'_2 \in N} t'_1 \sigma \times t'_2 \sigma \]

where \( \sigma = [(\gamma^1_h \times \gamma^2_h) \vee \alpha_h / \alpha_h]_{h \in H} \) (similarly for arrows)
Step 6: *Eliminate toplevel constructors, memoize, and recurse.*

\[
\bigwedge_{t_1 \times t_2 \in P} t_1 \times t_2 \leq \bigvee_{t'_1 \times t'_2 \in N} t'_1 \times t'_2
\]  

Equation (3) holds if and only if for all \( N' \subseteq N \),

\[
\forall \eta.\left( \left[ \bigwedge_{t_1 \times t_2 \in P} t_1 \wedge \bigwedge_{t'_1 \times t'_2 \in N'} \neg t'_1 \right] \eta = \emptyset \text{ or } \left[ \bigwedge_{t_1 \times t_2 \in P} t_2 \wedge \bigwedge_{t'_1 \times t'_2 \in N \setminus N'} \neg t'_2 \right] \eta = \emptyset \right)
\]

Apply *convexity* to distribute the quantification over the or’s:

\[
\forall \eta.\left( \left[ \bigwedge_{t_1 \times t_2 \in P} t_1 \wedge \bigwedge_{t'_1 \times t'_2 \in N'} \neg t'_1 \right] \eta = \emptyset \right) \text{ or } \forall \eta.\left( \left[ \bigwedge_{t_1 \times t_2 \in P} t_2 \wedge \bigwedge_{t'_1 \times t'_2 \in N \setminus N'} \neg t'_2 \right] \eta = \emptyset \right)
\]

Yielding the following simplification: (similarly for arrows)

\[
\forall N' \subseteq N.\left( \left( \bigwedge_{t_1 \times t_2 \in P} t_1 \leq \bigvee_{t'_1 \times t'_2 \in N'} t'_1 \right) \text{ or } \left( \bigwedge_{t_1 \times t_2 \in P} t_2 \leq \bigvee_{t'_1 \times t'_2 \in N \setminus N'} t'_2 \right) \right)
\]
Outline

10 Set-theoretic types

11 Semantic Subtyping

12 Application to a language.

13 Adding Parametric Polymorphism: the Types

14 Adding Parametric Polymorphism: the Language
A motivating example in Haskell

map :: (α → β) → [α] → [β]
map f l = case l of
  [] -> []
  (x : xs) -> (f x : map f xs)
A motivating example in Haskell

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]

\[
\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
\]
\[
\text{map} \ f \ l = \text{case} \ l \ \text{of}
\]
\[
| \ [] \rightarrow [] \\
| \ (x : xs) \rightarrow (f \ x : \text{map} \ f \ xs)
\]

\[
\text{even} :: (\text{Int} \rightarrow \text{Bool}) \land ((\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int}))
\]
\[
\text{even} \ x = \text{case} \ x \ \text{of}
\]
\[
| \ \text{Int} \rightarrow (x \ 'mod' \ 2) = 0 \\
| \ _ \rightarrow x
\]
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]

\[
\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
\]

\[
\text{map } f \ l \ = \ \text{case } l \ \text{of}
\]
\[
| [] \ -> \ []
\]
\[
| (x : xs) \ -> \ (f \ x : \text{map } f \ xs)
\]

\[
\text{even} :: (\text{Int} \rightarrow \text{Bool}) \wedge ((\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int}))
\]

\[
\text{even } x \ = \ \text{case } x \ \text{of}
\]
\[
| \text{Int} \ -> \ (x \ \text{`mod` } 2) == 0
\]
\[
| \_ \ -> \ x
\]

- **Expression**: if the argument is an integer then return the Boolean expression otherwise return the argument

- **Type**: when applied to an \text{Int} it returns a \text{Bool}; when applied to an argument that is not an \text{Int} it returns a result of the same type.
A motivating example in Haskell (almost) [cf. typing of balance]

```haskell
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]

```haskell
map :: (\alpha \rightarrow \beta) \rightarrow \{\alpha\} \rightarrow \{\beta\
map f l = case l of
  | [] -> []
  | (x : xs) -> (f x : map f xs)

even :: (Int \rightarrow Bool) \land ((\alpha \rightarrow Int) \rightarrow (\alpha \rightarrow 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
```

- **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 :: \((\alpha \to \beta) \to [\alpha] \to [\beta]\)
map f l = case l of
  | [] -> []
  | (x : xs) -> (f x : map f xs)

even :: (Int \to \text{Bool}) \land ((\alpha \setminus \text{Int}) \to (\alpha \setminus \text{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 \text{Int} it returns a \text{Bool}; when applied to an argument that is not an \text{Int} it returns a result of the same type.

**The combination of type-case and intersections yields statically typed dynamic overloading.**
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]

\[
\begin{align*}
\text{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\text{map } f \text{ } l &= \text{case } l \text{ of} \\
| \text{[]} &\to \text{[]} \\
| (x : \text{xs}) &\to (f \text{ } x : \text{map } f \text{ } \text{xs})
\end{align*}
\]

\[
\begin{align*}
\text{even} :: (\text{Int} \to \text{Bool}) \land ((\alpha \backslash \text{Int}) \to (\alpha \backslash \text{Int})) \\
\text{even } x &= \text{case } x \text{ of} \\
| \text{Int} &\to (x \text{ ‘mod‘ } 2) == 0 \\
| _ &\to x
\end{align*}
\]

This example as a yardstick. I want to define a language that:

1. Can define both \textit{map} and \textit{even}
2. Can \textit{check} the types specified in the signature
map :: \( (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \)

\[
\text{map } f \ l = \text{case } l \text{ of } \\
| [] \rightarrow [] \\
| (x : xs) \rightarrow (f \ x : \text{map } f \ xs)
\]

even :: (Int \rightarrow \text{Bool}) \land ((\alpha \\setminus \text{Int}) \rightarrow (\alpha \\setminus \text{Int}))

even \ x = \text{case } x \text{ of } \\
| \text{Int} \rightarrow (x \mod 2) == 0 \\
| _ \rightarrow x

This example as a yardstick. I want to define a language that:

1. Can define both \text{map} and \text{even}
2. Can check the types specified in the signature
3. Can deduce the type of the partial application \text{map even}
A motivating example in Haskell (almost) [cf. typing of balance]

```haskell
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

Tough!
A motivating example in Haskell (almost)  [cf. typing of balance]

map :: (\alpha \to \beta) \to [\alpha] \to [\beta]
map f l = case l of
  | [] -> []
  | (x : xs) -> (f x : map f xs)

even :: (Int \to Boolean) \land ((\alpha \setminus Int) \to (\alpha \setminus Int))
even x = case x of
  | Int -> (x 'mod' 2) == 0
  | _ -> x

We expect \texttt{map even} to have the following type:

\texttt{([Int] \to [Boolean]) \land ([\alpha \setminus Int] \to [\alpha \setminus Int]) \land ([\alpha \setminus Int] \to [(\alpha \setminus Int) \lor Boolean])}
A motivating example in Haskell (almost)  
[cf. typing of `balance`]

map :: \((\alpha\rightarrow\beta)\rightarrow[\alpha]\rightarrow[\beta]\)  
map f l = case l of  
| []  -> []  
| (x : xs)  -> (f x : map f xs)

even :: \((\text{Int}\rightarrow\text{Bool})\wedge((\alpha\setminus\text{Int})\rightarrow(\alpha\setminus\text{Int}))\)  
even x = case x of  
| Int  -> (x `mod` 2) == 0  
| _    -> x

We expect `map even` to have the following type:

\(([\text{Int}]\rightarrow[\text{Bool}])\wedge
([\alpha\setminus\text{Int}]\rightarrow[\alpha\setminus\text{Int}])\wedge
([\alpha\vee\text{Int}]\rightarrow[\{(\alpha\setminus\text{Int})\vee\text{Bool}\}])\)

\(\text{int lists are transformed into bool lists}\)
\(\text{lists w/o ints return the same type}\)
\(\text{ints in the arg. are replaced by bools}\)
A motivating example in Haskell (almost)  [cf. typing of balance]

\[
\begin{align*}
\text{map} & \colon (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
\text{map}\ f\ l & = \text{case}\ l\ of \\
& \mid [] \rightarrow [] \\
& \mid (x : xs) \rightarrow (f\ x : \text{map}\ f\ xs)
\end{align*}
\]

\[
\begin{align*}
\text{even} & \colon (\text{Int} \rightarrow \text{Bool}) \land ((\alpha \land \text{Int}) \rightarrow (\alpha \land \text{Int})) \\
\text{even}\ x & = \text{case}\ x\ of \\
& \mid \text{Int} \rightarrow (x \mod 2) = 0 \\
& \mid _{} \rightarrow x
\end{align*}
\]

We expect \textbf{map even} to have the following type:

\[
\begin{align*}
([\text{Int}] \rightarrow [\text{Bool}]) \land \\
([\alpha \land \text{Int}] \rightarrow [\alpha \land \text{Int}]) \land \\
([\alpha \lor \text{Int}] \rightarrow [(\alpha \land \text{Int}) \lor \text{Bool}])
\end{align*}
\]

int lists are transformed into bool lists

lists w/o ints return the same type

ints in the arg. are replaced by bools

Difficult because of expansion: needs \textit{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:

\[
\frac{\Gamma \vdash e_1 : s \rightarrow u \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}
\]

[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:

\[
\begin{align*}
\text{(APPL)} & \\
\Gamma & \vdash e_1 : s \rightarrow u \\
\Gamma & \vdash e_2 : s \\
\hline
\Gamma & \vdash e_1 e_2 : u
\end{align*}
\]

[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:

\[
\begin{align*}
\text{(APPL-ALGORITHM)} & \\
\Gamma & \vdash A e_1 : t \\
\Gamma & \vdash A e_2 : s \\
\hline
\Gamma & \vdash A e_1 e_2 : \min\{u \mid t \leq s \rightarrow u\} \\
\end{align*}
\]

\[
\begin{align*}
t \leq 0 & \rightarrow 1 \\
s \leq \text{dom}(t)
\end{align*}
\]
The rule for applications

1. In the type system:

\[ \frac{\Gamma \vdash e_1 : s \to u \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u} \]  

[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:

\[ \frac{\begin{array}{c} \Gamma \vdash \_ \, e_1 : t \quad \Gamma \vdash \_ \, e_2 : s \\ t \leq 0 \to 1 \\ s \leq \text{dom}(t) \end{array}}{\Gamma \vdash \_ \, e_1 e_2 : \min\{u \mid t \leq s \to u\}} \]
The rule for applications

1. In the type system:

\[(\text{APPL})\]

\[\Gamma \vdash e_1 : s \rightarrow u \quad \Gamma \vdash e_2 : s\]

\[\Gamma \vdash e_1 e_2 : u\]

[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:

\[(\text{APPL-ALGORITHM})\]

\[\Gamma \vdash_A e_1 : t \quad \Gamma \vdash_A e_2 : s\]

\[\Gamma \vdash_A e_1 e_2 : \min\{u \mid t \leq s \rightarrow u\}\]

\[t \leq 0 \rightarrow 1\]

\[s \leq \text{dom}(t)\]
1. In the type system:

\[
\begin{align*}
(A_{\text{APPL}}) \\
\Gamma \vdash e_1 : s \rightarrow u & \quad \Gamma \vdash e_2 : s \\
\hline
\Gamma \vdash e_1 e_2 : u
\end{align*}
\]

[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:

\[
\begin{align*}
(A_{\text{APPL-ALGORITHM}}) \\
\Gamma \vdash_\mathcal{A} e_1 : t & \quad \Gamma \vdash_\mathcal{A} e_2 : s \\
\hline
\Gamma \vdash_\mathcal{A} e_1 e_2 : \min\{u \mid t \leq s \rightarrow u\}
\end{align*}
\]

\[t \leq 0 \rightarrow 1 \quad s \leq \text{dom}(t)\]

3. Inference of type substitutions

\[
\begin{align*}
(A_{\text{APPL-INFERENCE}}) \\
\exists [\sigma_i]_{i \in I}, [\sigma'_j]_{j \in J} & \quad \Gamma \vdash_I e_1 : t \\
\hline
\Gamma \vdash_I e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j \in J} \leq s[\sigma_i]_{i \in I} \rightarrow u\}
\end{align*}
\]

\[t[\sigma'_j]_{j \in J} \leq 0 \rightarrow 1 \quad s[\sigma_i]_{i \in I} \leq \text{dom}(t[\sigma'_j]_{j \in J})\]
The rule for applications

1. In the type system:

\[
(A_{PPL})
\]

\[
\Gamma \vdash e_1 : s \rightarrow u \quad \Gamma \vdash e_2 : s
\]

\[
\Gamma \vdash e_1 e_2 : u
\]

[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:

\[
(A_{PPL-ALGORITHM})
\]

\[
\Gamma \vdash A e_1 : t \quad \Gamma \vdash A e_2 : s
\]

\[
\Gamma \vdash A e_1 e_2 : \min\{u \mid t \leq s \rightarrow u\}
\]

\[t \leq 0 \rightarrow 1\]

\[s \leq \text{dom}(t)\]

3. Inference of type substitutions

\[
(A_{PPL-INFERENCE})
\]

\[
\exists [\sigma_i]_{i \in I}, [\sigma'_j]_{j \in J} \quad \Gamma \vdash I e_1 : t \quad \Gamma \vdash I e_2 : s
\]

\[
\Gamma \vdash I e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j \in J} \leq s[\sigma_i]_{i \in I} \rightarrow u\}
\]

\[t[\sigma'_j]_{j \in J} \leq 0 \rightarrow 1\]

\[s[\sigma_i]_{i \in I} \leq \text{dom}(t[\sigma'_j]_{j \in J})\]

[where \(t[\sigma_i]_{i \in I} = \bigwedge_{i \in I} t \sigma_i\).]
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:

$$t[\sigma'_j]_{j \in J} \leq 0 \rightarrow 1 \quad \text{and} \quad s[\sigma_i]_{i \in I} \leq \text{dom}(t[\sigma'_j]_{j \in J})$$
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:

$$t[\sigma'_j]_{j \in J} \leq 0 \rightarrow 1 \quad \text{and} \quad s[\sigma_i]_{i \in I} \leq \text{dom}(t[\sigma'_j]_{j \in J})$$

This can be reduced to solving a suite of *tallying problems*:

**Definition (Type tallying)**

Let $s$ and $t$ be two types. A type-substitution $\sigma$ is a solution for the *tallying* of $(s, t)$ iff $s\sigma \leq t\sigma$. 
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:

$$t[\sigma'_j]_{j \in J} \leq 0 \rightarrow 1 \quad \text{and} \quad s[\sigma_i]_{i \in I} \leq \text{dom}(t[\sigma'_j]_{j \in J})$$

This can be reduced to solving a suite of tallying problems:

**Definition (Type tallying)**

Let $s$ and $t$ be two types. A type-substitution $\sigma$ 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 $\sigma$ is a solution for the tallying of $C$ iff $s\sigma \leq t\sigma$ for all $(s \leq t) \in C$. 
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:

\[
t[\sigma'_j]_{j \in J} \leq 0 \rightarrow 1 \quad \text{and} \quad s[\sigma_i]_{i \in I} \leq \text{dom}(t[\sigma'_j]_{j \in J})
\]

This can be reduced to solving a suite of \textit{tallying problems}:

**Definition (Type tallying)**

Let \( s \) and \( t \) be two types. A type-substitution \( \sigma \) is a solution for the \textit{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 \textit{constraint set}. A type-substitution \( \sigma \) is a solution for the \textit{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 \textbf{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 2: Merge constraints on the same variable.
- If $\alpha \leq t_1$ and $\alpha \leq t_2$ are in $C$, then replace them by $\alpha \leq t_1 \land \ldots \land t_2$;
- If $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in $C$, then replace them by $s_1 \lor \ldots \lor s_2 \leq \alpha$;

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 $\{s_i \leq \alpha_i \leq t_i | i \in \{1..n\}\}$ where $\alpha_i$ are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \lor \ldots \lor \beta) \land \ldots \land t$ with $\beta$ fresh.
2. substitute $(s \lor \ldots \lor \beta) \land \ldots \land t$ for all $\alpha$ in the other constraints of $C$.
3. repeat with another constraint.

At the end we have a sets of equations $\{\alpha_i = u_i | i \in \{1..n\}\}$ that (with some care) are contractive. By Courcelle there exists a solution, i.e., a substitution for $\alpha_1, \ldots, \alpha_n$ into (possibly recursive regular) types $t_1, \ldots, t_n$ (in which the fresh $\beta$'s are free variables).
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. \( \{ (s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2) \} \leadsto \{ (s_2 \leq \emptyset) \} \) 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 $\alpha \leq t_1 \land t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in $C$, then replace them by $s_1 \lor s_2 \leq \alpha$;

Possibly decompose the new constraints generated by transitivity.

Example:
1. $\{(s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2)\} \leadsto \{(s_2 \leq \emptyset)\}$ 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 $\alpha \leq t_1 \land t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in $C$, then replace them by $s_1 \lor s_2 \leq \alpha$;
Possibly decompose the new constraints generated by transitivity.

Example:
1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \leadsto \{(s_2 \leq 0)\} \text{ or } \{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\text{Int} \leq \alpha), (\text{Bool} \leq \alpha)\} \leadsto \{(\text{Int} \lor \text{Bool} \leq \alpha)\}$
**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 $\alpha \leq t_1 \land t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in $C$, then replace them by $s_1 \lor s_2 \leq \alpha$;
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 $\alpha_i$ are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \lor \beta) \land t$ with $\beta$ fresh.
2. substitute $(s \lor \beta) \land t$ for all $\alpha$ in the other constraints of $C$
3. repeat with another constraint

Example:
1. $\{(s_1 \to t_1 \leq s_2 \to t_2)\} \leadsto \{(s_2 \leq \emptyset)\}$ or $\{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\text{Int} \leq \alpha), (\text{Bool} \leq \alpha)\} \leadsto \{(\text{Int} \lor \text{Bool} \leq \alpha)\}$
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 $\alpha \leq t_1 \land t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in $C$, then replace them by $s_1 \lor s_2 \leq \alpha$;
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 $\alpha_i$ are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \lor \beta) \land t$ with $\beta$ fresh.
2. substitute $(s \lor \beta) \land t$ for all $\alpha$ in the other constraints of $C$
3. repeat with another constraint

Example:
1. $\{(s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2)\} \leadsto \{(s_2 \leq \emptyset)\} \lor \{(s_2 \leq s_1), (t_1 \leq t_2)\}$
2. $\{(\text{Int} \leq \alpha), (\text{Bool} \leq \alpha)\} \leadsto \{(\text{Int} \lor \text{Bool} \leq \alpha)\}$
3. $\{(\text{Int} \leq \alpha_1 \leq \text{Real}), (\alpha_2 \leq \alpha_1 \land \text{Int})\}$
   $\leadsto \\{\alpha_1 = (\text{Int} \lor \beta) \land \text{Real}, (\alpha_2 = \text{Int})\}$
**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 $\alpha \leq t_1 \land t_2$;
- if $s_1 \leq \alpha$ and $s_2 \leq \alpha$ are in $C$, then replace them by $s_1 \lor s_2 \leq \alpha$;

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 $\alpha_i$ are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \lor \beta) \land t$ with $\beta$ fresh.
2. substitute $(s \lor \beta) \land t$ for all $\alpha$ 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 $\beta$’s are free variables).
Example: $\text{map even}$

Start with the following tallying problem:

$$(\alpha_1 \rightarrow \beta_1) \rightarrow [\alpha_1] \rightarrow [\beta_1] \leq s \rightarrow \gamma$$

where $s = (\text{Int} \rightarrow \text{Bool}) \land (\alpha \downarrow \text{Int} \rightarrow \alpha \downarrow \text{Int})$ is the type of $\text{even}$
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 $s = (\text{Int} \to \text{Bool}) \land (\alpha \setminus \text{Int} \to \alpha \setminus \text{Int})$ is the type of even.

The algorithm generates 9 constraint-sets: one is unsatisfiable ($s \leq \emptyset$); four are implied by the others; remain

$\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \emptyset\}$,  $\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \text{Int}, \text{Bool} \leq \beta_1\}$,$\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \setminus \text{Int}, \alpha \setminus \text{Int} \leq \beta_1\}$,$\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \lor \text{Int}, (\alpha \setminus \text{Int}) \lor \text{Bool} \leq \beta_1\}$. 
Example: \texttt{map even}

Start with the following tallying problem:

\[(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq s \to \gamma\]

where \(s = (\text{Int} \to \text{Bool}) \land (\alpha \setminus \text{Int} \to \alpha \setminus \text{Int})\) is the type of \texttt{even}.

- The algorithm generates 9 constraint-sets: one is unsatisfiable \((s \leq 0)\); four are implied by the others; remain
  \[
  \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \text{Int}\},
  \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \text{Int}, \text{Boo}l \leq \beta_1\},
  \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \setminus \text{Int}, \alpha \setminus \text{Int} \leq \beta_1\},
  \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \lor \text{Int}, (\alpha \setminus \text{Int}) \lor \text{Boo}l \leq \beta_1\};
  \]

- Four solutions for \(\gamma\):
  \[
  \{\gamma = [\text{Int}] \to [\text{Bool}]\},
  \{\gamma = [\alpha \setminus \text{Int}] \to [\alpha \setminus \text{Int}]\},
  \{\gamma = [\alpha \lor \text{Int}] \to [(\alpha \setminus \text{Int}) \lor \text{Boo}l]\}.
  \]
Example: map even

Start with the following tallying problem:

\[(\alpha_1 \rightarrow \beta_1) \rightarrow [\alpha_1] \rightarrow [\beta_1] \leq s \rightarrow \gamma\]

where \(s = (\text{Int} \rightarrow \text{Bool}) \land (\alpha \downarrow \text{Int} \rightarrow \alpha \downarrow \text{Int})\) is the type of \text{even}.

- The algorithm generates 9 constraint-sets: one is unsatisfiable \((s \leq \emptyset)\); four are implied by the others; remain

  \[\{\gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq \emptyset\}, \{\gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq \text{Int}, \text{Bool} \leq \beta_1\}, \]
  \[\{\gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq \alpha \downarrow \text{Int}, \alpha \downarrow \text{Int} \leq \beta_1\}, \]
  \[\{\gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq \alpha \lor \text{Int}, (\alpha \downarrow \text{Int}) \lor \text{Bool} \leq \beta_1\};\]

- Four solutions for \(\gamma\):

  \[\{\gamma = [] \rightarrow []\},\]
  \[\{\gamma = [\text{Int}] \rightarrow [\text{Bool}]\},\]
  \[\{\gamma = [\alpha \downarrow \text{Int}] \rightarrow [\alpha \downarrow \text{Int}]\},\]
  \[\{\gamma = [\alpha \lor \text{Int}] \rightarrow [(\alpha \downarrow \text{Int}) \lor \text{Bool}]\}.\]

- The last two are minimal and we take their intersection:

  \[\{\gamma = ([\alpha \downarrow \text{Int}] \rightarrow [\alpha \downarrow \text{Int}]) \land ([\alpha \lor \text{Int}] \rightarrow [(\alpha \downarrow \text{Int}) \lor \text{Bool}])\}\]
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*).
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.
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?

Reference publication for monomorphic semantic subtyping.


A simple introduction to semantic subtyping and a detailed description of the implementation of subtyping and type-checking algorithms.


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


Type reconstruction for polymorphic set-theoretic types
To try it out

- 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
Outline

Main ideas

Formal system

Algorithmic Aspects

Criteria for Gradual Typing

Implementation issues

References
Outline

15 Main ideas
16 Formal system
17 Algorithmic Aspects
18 Criteria for Gradual Typing
19 Implementation issues
20 References
Motivating example: reminder

```javascript
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

```javascript
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

```javascript
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:

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

Accept as is:

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

Intuitively the function has type:

`number | ?` →

Accept and insert checks:

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

Compile as

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

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

Accept as is:

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

Intuitively the function has type: `? → (number | ?)`
Reject at compile time:

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

Accept as is:

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

Intuitively the function has type: `? → (number | ?)`

Accept and insert checks:

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

Compile as

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

Mix static and dynamic typing
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))

Add checks at the boundaries:
apply (double, (double 42)) must be compiled as
apply (double ⟨number→number⟩, (double 42) ⟨number⟩)
Rationale

Mix static and dynamic typing

Dynamically typed:

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

Statically typed:

```javascript
function apply (f : number --> number, x : number) {
    return (f x);
}
```

Mixed typing:

```javascript
apply (double , (double 42))
```

Rationale

Mix static and dynamic typing

*Dynamically typed:*

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

*Statically typed:*

```javascript
function apply (f : number --> number, x : number) {
  return (f x);
}
```

*Mixed typing:*

```javascript
apply (double , (double 42))
```

Add checks at the boundaries:

```javascript
apply (double , (double 42))
```

must be compiled as

```javascript
apply (double<number --> number> , (double 42)<number>)
```
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

15 Main ideas

16 Formal system

17 Algorithmic Aspects

18 Criteria for Gradual Typing

19 Implementation issues

20 References
Simply-typed $\lambda$-calculus types:

$$Types \quad T ::= \text{Bool} \mid \text{Int} \mid T \rightarrow T$$
Gradual Typing

Simply-typed $\lambda$-calculus types:

\[
Types \quad T ::= \text{Bool} \mid \text{Int} \mid T \rightarrow T
\]

A new consistency relation "\(\sim\)" governs implicit casts involving "??":

\[
\text{Bool} \sim \text{Bool} \quad \text{Int} \sim \text{Int} \quad T \sim ?? \quad ?? \sim T
\]

Relax application for consistent types:

\[
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \sim S \quad \Gamma \vdash ab : T
\]

Use the type derivation to insert casts:

\[
\Gamma \vdash a : S \rightarrow T \quad \text{compiles} \succ a' \quad \Gamma \vdash b : U \quad \text{compiles} \succ b' \quad U \sim S \quad \Gamma \vdash ab : T \quad \text{compiles} \succ a(b\langle S \rangle)(U \not\equiv S)\]
Gradual Typing

Simply-typed $\lambda$-calculus types:

\[
T ::= \text{Bool} \mid \text{Int} \mid T \to T \mid ?
\]

A new \textbf{consistency} relation "\(\sim\)" governs implicit casts involving "\(?\)":

\[
\begin{align*}
\text{Bool} & \sim \text{Bool} \\
\text{Int} & \sim \text{Int} \\
T & \sim ? \\
? & \sim T
\end{align*}
\]

\[
\frac{S_1 \sim T_1}{S_1 \to S_2 \sim T_1 \to T_2}
\]

\[
\frac{S_2 \sim T_2}{S_1 \to S_2 \sim T_1 \to T_2}
\]
Simply-typed λ-calculus types:

\[ \text{Types } T ::= \text{Bool} \mid \text{Int} \mid T \rightarrow T \mid ? \]

A new **consistency** relation “∼” governs implicit casts involving “?”:

\[
\begin{align*}
\text{Bool} & \sim \text{Bool} \\
\text{Int} & \sim \text{Int} \\
T & \sim ? \\
? & \sim T
\end{align*}
\]

\[
\frac{S_1 \sim T_1 \quad S_2 \sim T_2}{S_1 \rightarrow S_2 \sim T_1 \rightarrow T_2}
\]

Relax application for consistent types:

\[
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \quad U \sim S
\]

\[
[\rightarrow \text{ELIM}_\sim] \quad \frac{\Gamma \vdash a \ b : T}{\Gamma \vdash \text{ab} : T}
\]
Gradual Typing

[Siek&Taha 2006]

Simply-typed λ-calculus types:

\[ T ::= \text{Bool} \mid \text{Int} \mid T \to T \mid ? \]

A new **consistency** relation “∼” governs implicit casts involving “?”:

- \( \text{Bool} \sim \text{Bool} \)
- \( \text{Int} \sim \text{Int} \)
- \( T \sim ? \)
- \( ? \sim T \)
- \( S_1 \sim T_1 \)
- \( S_2 \sim T_2 \)
- \( S_1 \to S_2 \sim T_1 \to T_2 \)

Relax application for consistent types:

\[ \frac{\text{Γ} \vdash a : S \to T \quad \text{Γ} \vdash b : U \quad U \sim S}{\text{Γ} \vdash ab : T} \]

Use the type derivation to insert casts

\[ \frac{\text{Γ} \vdash a : S \to T \quad \text{Γ} \vdash b : U \quad U \sim S}{\text{Γ} \vdash ab : T} \quad \text{(U \neq S)} \]
Gradual Typing

Simply-typed \( \lambda \)-calculus types:

\[
Types \quad T ::= \text{Bool} \mid \text{Int} \mid T \rightarrow T \mid ?
\]

A new **consistency** relation “\( \sim \)” governs implicit casts involving “?”:

\[
\begin{align*}
\text{Bool} \sim \text{Bool} & \quad \text{Int} \sim \text{Int} & \quad T \sim ? & \quad ? \sim T \\
S_1 \sim T_1 & \quad S_2 \sim T_2 \\
S_1 \rightarrow S_2 \sim T_1 \rightarrow T_2
\end{align*}
\]

Relax application for consistent types:

\[
\frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \quad U \sim S}{\Gamma \vdash ab : T}
\]

Use the type derivation to insert casts

\[
\frac{\Gamma \vdash a : S \rightarrow T \quad \text{compiles} \quad a' \quad \Gamma \vdash b : U \quad \text{compiles} \quad b' \quad U \sim S}{\Gamma \vdash ab : T \quad \text{compiles} \quad a(b\langle S\rangle)}
\]

\((U \neq S)\)
Gradual Typing

Simply-typed $\lambda$-calculus types:

\[
Types \quad T ::= \, \text{Bool} \mid \text{Int} \mid T \rightarrow T \mid ?
\]

A new \textbf{consistency} relation "\(\sim\)" governs implicit casts involving "\(?\)"

\[
\begin{array}{c}
\text{Bool} \sim \text{Bool} \\
\text{Int} \sim \text{Int} \\
T \sim ? \\
? \sim T \\
\end{array}
\]

\[
\begin{array}{c}
S_1 \sim T_1 \\
S_2 \sim T_2 \\
S_1 \rightarrow S_2 \sim T_1 \rightarrow T_2
\end{array}
\]

Relax application for consistent types:

\[
\frac{
\Gamma \vdash a : S \rightarrow T \\
\Gamma \vdash b : U \sim S
}{
\Gamma \vdash ab : T
}
\]

Use the type derivation to insert casts

\[
\frac{
\Gamma \vdash a : S \rightarrow T \\
\Gamma \vdash a' \sim b : U \sim S
}{
\Gamma \vdash ab : T \sim a(b \langle S \rangle)
}
\]

\[
(U \neq S)
\]
Problems

The consistency relation **must not be transitive:**

Since \( \text{Int} \sim ? \) and \( ? \sim \text{Bool} \), then transitivity would imply \( \text{Int} \sim \text{Bool} \):

\[
\begin{align*}
\vdash \lambda x : \text{Int}.x + 1 : \text{Int} \to \text{Int} & \quad \vdash \text{true} : \text{Bool} & \quad \text{Int} \sim \text{Bool} \\
\hline
\vdash (\lambda x : \text{Int}.x + 1)\text{true} : \text{Int}
\end{align*}
\]

it is hard to work with a non-transitive relation.
The consistency relation **must not be transitive:**

Since \( \text{Int} \sim ? \) and \(? \sim \text{Bool} \), then transitivity would imply \( \text{Int} \sim \text{Bool} \):

\[
\vdash \lambda x : \text{Int}. x + 1 : \text{Int} \rightarrow \text{Int} \quad \vdash \text{true} : \text{Bool} \quad \text{Int} \sim \text{Bool}
\]

\[
\vdash (\lambda x : \text{Int}. x + 1) \text{true} : \text{Int}
\]

It is hard to work with a non-transitive relation.

It has a flavor of substitutivity ... but not always:

```javascript
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

\[
\text{apply} (\text{double} \langle \text{Int} \rightarrow \text{Int} \rangle, (\text{double}(42 \langle ? \rangle)) \langle \text{Int} \rangle)
\]
The consistency relation \textit{must not be transitive}:

Since \( \text{Int} \sim \_ \) and \( \_ \sim \text{Bool} \), then transitivity would imply \( \text{Int} \sim \text{Bool} \):

\[
\vdash \lambda x : \text{Int}.x + 1 : \text{Int} \rightarrow \text{Int} \quad \vdash \text{true} : \text{Bool} \quad \text{Int} \sim \text{Bool}
\]

\[
\vdash (\lambda x : \text{Int}.x + 1)\text{true} : \text{Int}
\]

it is hard to work with a non-transitive relation.

It has a flavor of substitutivity ... but not always:

\begin{verbatim}
function double (x : ?) { (<condition>) ? 2*x : x.concat(x) }
function apply (f : number --> number, x : number) { return (f x) }
apply (double, (double 42))
\end{verbatim}

It compiles as \[ \text{apply} \left( \text{double} \langle \text{Int} \rightarrow \text{Int} \rangle, (\text{double}(42\langle ? \rangle))\langle \text{Int} \rangle \right) \]

\bullet \text{Casting } ? \rightarrow ? \text{ to } \text{Int} \rightarrow \text{Int} \text{ is ok.}
The consistency relation must not be transitive:

Since \( \text{Int} \sim ? \) and \( ? \sim \text{Bool} \), then transitivity would imply \( \text{Int} \sim \text{Bool} \):

\[
\frac{\vdash \lambda x : \text{Int}.x + 1 : \text{Int} \rightarrow \text{Int} \quad \vdash \text{true} : \text{Bool} \quad \text{Int} \sim \text{Bool}}{\vdash (\lambda x : \text{Int}.x + 1)\text{true} : \text{Int}}
\]

It is hard to work with a non-transitive relation.

It has a flavor of substitutivity ... but not always:

```javascript
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

\[
\text{apply} \left( \text{double} \langle \text{Int} \rightarrow \text{Int} \rangle, (\text{double}(42\langle?\rangle))\langle\text{Int}\rangle \right)
\]

- Casting \(? \rightarrow ?\) to \text{Int} \rightarrow \text{Int} is ok.
- Casting \(? \rightarrow ?\) to \text{Int} is ok.
The consistency relation **must not be transitive:**

Since \( \text{Int} \sim ? \) and \( ? \sim \text{Bool} \), then transitivity would imply \( \text{Int} \sim \text{Bool} \):

\[
\vdash \lambda x: \text{Int}.x + 1 : \text{Int} \rightarrow \text{Int} \quad \vdash \text{true} : \text{Bool} \quad \text{Int} \sim \text{Bool}
\]

\[
\vdash (\lambda x: \text{Int}.x + 1) \text{true} : \text{Int}
\]

It is hard to work with a non-transitive relation.

It has a flavor of substitutivity ... but not always:

```javascript
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

```
apply ( double<\text{Int} \rightarrow \text{Int}>, (double(42<?>))<\text{Int}> )
```

- Casting \( ? \rightarrow ? \) to \( \text{Int} \rightarrow \text{Int} \) is ok.
- Casting \( ? \) to \( \text{Int} \) is ok.
- Casting an \( \text{Int} \) to \( ? \) looks weird
The \( \rightarrow \text{ELIM}_\sim \) rule looks more an algorithmic step than a typing rule:

\[
\begin{align*}
\text{[} \rightarrow \text{ELIM}_\sim \text{]} & \quad \\
\Gamma \vdash a : S \rightarrow T & \quad \Gamma \vdash b : U & \quad U \sim S \\
\Gamma \vdash ab : T
\end{align*}
\]

\[
\begin{align*}
\text{[} \rightarrow \text{ELIM}_\leq \text{]} & \quad \\
\Gamma \vdash \mathcal{A} a : S \rightarrow T & \quad \Gamma \vdash \mathcal{A} b : U & \quad U \leq S \\
\Gamma \vdash \mathcal{A} ab : T
\end{align*}
\]
The \([\rightarrow \text{ELIM}\sim]\) rule looks more an algorithmic step than a typing rule:

\[
\begin{align*}
\text{[\rightarrow \text{ELIM}\sim]} & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \quad U \sim S \\
& \quad \Gamma \vdash ab : T
\end{align*}
\]

\[
\begin{align*}
\text{[\rightarrow \text{ELIM}\leq]} & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \quad U \leq S \\
& \quad \Gamma \vdash ab : T
\end{align*}
\]

We need a more principled methodology
The \( [\rightarrow \text{ELIM} \sim] \) rule looks more an algorithmic step than a typing rule:

\[
\begin{array}{c}
[\rightarrow \text{ELIM} \sim] \\
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \quad U \sim S \\
\hline
\Gamma \vdash ab : T
\end{array}
\]

\[
\begin{array}{c}
[\rightarrow \text{ELIM} \leq] \\
\Gamma \vdash A a : S \rightarrow T \quad \Gamma \vdash A b : U \quad U \leq S \\
\hline
\Gamma \vdash A ab : T
\end{array}
\]

We need a more principled methodology

Let’s take inspiration from what we did for subtyping
Precision and Materialization

The precision relation “⊑”:
Precision relates a type with unknown “?” components to the types it may dynamically become at run time.
The precision relation “⊑”:
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 “⊑”: 
Precision relates a type with unknown “?” components to the types it \emph{may} dynamically become at run time.

**Informally**

The less “?” it uses, the more \emph{precise} a type is.

Can be defined by induction for simple types:

\[
\begin{align*}
? & \leq T \\
S_1 & \leq T_1 \\
S_2 & \leq T_2 \\
S_1 \rightarrow S_2 & \leq T_1 \rightarrow T_2 \\
T & \leq T \\
T_1 & \leq T_2 \\
T_2 & \leq T_3 \\
T_1 \rightarrow T_3 & \leq T_3
\end{align*}
\]
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:

\[
\begin{align*}
? & \sqsubseteq T \\
S_1 \sqsubseteq T_1 & \quad S_2 \sqsubseteq T_2 \\
S_1 \to S_2 \sqsubseteq T_1 \to T_2 & \quad T \sqsubseteq T \\
T_1 \sqsubseteq T_2 & \quad T_2 \sqsubseteq T_3
\end{align*}
\]

- It is not subtyping
**Precision and Materialization**

**The precision relation “⊑”:**
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:

- \( \text{?} \subseteq T \)  
  \( S_1 \subseteq T_1, S_2 \subseteq T_2 \)
  \( S_1 \rightarrow S_2 \subseteq T_1 \rightarrow T_2 \)

- \( T_1 \subseteq T_2, T_2 \subseteq T_3 \)
  \( T_1 \subseteq T_3 \)

- **It is *not* subtyping**
- **It is a pre-order**
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:

\[
\begin{align*}
? \sqsubseteq T \\
S_1 \sqsubseteq T_1 & \quad S_2 \sqsubseteq T_2 \\
S_1 \rightarrow S_2 \sqsubseteq T_1 \rightarrow T_2 & \quad T \sqsubseteq T \\
T_1 \sqsubseteq T_2 & \quad T_2 \sqsubseteq T_3 \\
T_1 \sqsubseteq T_3
\end{align*}
\]

- 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 \textit{transitive}:

\[
\begin{align*}
? &\sqsubseteq ? \rightarrow ? & \sqsubseteq ? \rightarrow \text{Int} & \sqsubseteq \text{Int} \rightarrow \text{Int}
\end{align*}
\]

This means that it can be used in a subsumption-like rule:

\[
\Gamma \vdash a : S \quad S \sqsubseteq T \\
\Gamma \vdash a : T
\]

We can add it to any type system to embed gradual typing in it.

Rationale

As subtyping captures "safe replacement", so precision captures "potential materialization".
Precision and Materialization

The precision relation is a pre-order thus, in particular, it is *transitive*:

\[
? \sqsubseteq ? \rightarrow ? \sqsubseteq ? \rightarrow \text{Int} \sqsubseteq \text{Int} \rightarrow \text{Int}
\]

but:

\[
? \sqsubseteq \text{Int} \not\sqsubseteq ?
\]
The precision relation is a pre-order thus, in particular, it is **transitive**: 

\[ ? \sqsubseteq ? \rightarrow ? \sqsubseteq ? \rightarrow \text{Int} \sqsubseteq \text{Int} \rightarrow \text{Int} \]

but:

\[ ? \sqsubseteq \text{Int} \not\sqsubseteq ? \]

This means that it can be used in a subsumption-like rule:

\[
[MATERIALIZE] \quad \frac{\Gamma \vdash a : S \quad S \sqsubseteq T}{\Gamma \vdash a : T}
\]
The precision relation is a pre-order thus, in particular, it is transitive:

\[ ? \sqsubseteq ? \rightarrow ? \sqsubseteq ? \rightarrow \text{Int} \sqsubseteq \text{Int} \rightarrow \text{Int} \]

but:

\[ ? \sqsubseteq \text{Int} \nmid \sqsubseteq ? \]

This means that it can be used in a subsumption-like rule:

\[
\begin{array}{c}
\text{[MATERIALIZE]}
\Gamma \vdash a : S \\
S \sqsubseteq T
\hline
\Gamma \vdash a : T
\end{array}
\]

We can add it to any type system to embed gradual typing in it.
The precision relation is a pre-order thus, in particular, it is *transitive*:

\[
? \sqsubseteq ? \rightarrow ? \sqsubseteq ? \rightarrow \text{Int} \sqsubseteq \text{Int} \rightarrow \text{Int}
\]

but:

\[
? \sqsubseteq \text{Int} \not\sqsubseteq ?
\]

This means that it can be used in a subsumption-like rule:

\[
[MATERIALIZE] \quad \Gamma \vdash a : S \quad S \sqsubseteq T \quad \Gamma \vdash a : T
\]

We can add it to any type system to embed gradual typing in it.

**Rationale**

As *subtyping* captures “*safe replacement*”, so *precision* captures “*potential materialization*”.

Since *potential materialization* does not mean *assured* materialization, then we have to check it at run-time:

\[
\begin{align*}
\text{[MATERIALIZE]} & \quad \Gamma \vdash a : S \text{~~~}(\text{compiles}) \quad a' \quad S \sqsubseteq T \\
\Gamma \vdash a : T \text{~~~}(\text{compiles}) \quad a'(\langle T \rangle)
\end{align*}
\]

Rationale

Subtyping = assured materialization (cast always works)

Precision = possible materialization (cast may fail)

From a logical viewpoint:

\[
\begin{align*}
\text{[SUBSUMPTION]} & \quad \Gamma \vdash a : S \text{~~~}(\text{compiles}) \quad a' \quad S \sqsubseteq T \\
\Gamma \vdash a : T \text{~~~}(\text{compiles}) \quad a'(\langle T \rangle)
\end{align*}
\]

Subsumption as implicit coercions (subtyping)

Materialization as explicit casts (precision)
Since potential materialization does not mean assured materialization, then we have to check it at run-time:

\[
\begin{align*}
\text{[MATERIALIZE]} & \quad \Gamma \vdash a : S \quad \text{\texttt{compiles}} \quad a' \\
\quad & \quad S \sqsubseteq T \\
\Gamma \vdash a : T \quad \text{\texttt{compiles}} \quad a'(\langle T \rangle)
\end{align*}
\]

**Rationale**

- **Subtyping** = assured materialization (cast always works)
- **Precision** = possible materialization (cast may fail)
Since *potential materialization* does not mean *assured* materialization, then we have to check it at run-time:

\[
\text{[MATERIALIZE]} \quad \frac{\Gamma \vdash a : S \quad S \subseteq T}{\Gamma \vdash a : T \quad \text{compiles} \quad a' \langle T \rangle}
\]

### Rationale

- **Subtyping** = assured materialization (cast always works)
- **Precision** = possible materialization (cast may fail)

### From a logical viewpoint:

\[
\text{[SUBSUMPTION]} \quad \frac{\Gamma \vdash a : S \quad S \subseteq T}{\Gamma \vdash a : T \quad \text{compiles} \quad a' \langle T \rangle}
\]

Subsumption as implicit coercions (subtyping)

\[
\text{[MATERIALIZE]} \quad \frac{\Gamma \vdash a : S \quad S \subseteq T}{\Gamma \vdash a : T \quad \text{compiles} \quad a' \langle T \rangle}
\]

Materialization as explicit casts (precision)
Summing up

Take your favorite typed language

Add "???

Add the materialization rule (with suitable ⊑)

Compile to insert casts

Et voila: you have added gradual typing

Types

T ::= Int | Bool | T → T | ???

Terms

a, b ::= x | ab | λ x : T. a | 1 | 2 | ... |

(λ x : T. a) b −→ a[b/x]

Γ ⊢ x : Γ(x)

Γ, x : S ⊢ a : T

Γ ⊢ λ x : S. a : S → T

Γ ⊢ a : S → T

Γ ⊢ b : S ⊢ ab : T

Γ ⊢ a : S ⊑ T

Γ ⊢ a : T

compiles ≻ a′

compiles ≻ ⟨T⟩

[VAR] [INTRO] [ELIM] [MATERIALIZE] [COMPIL]
Take your favorite typed language

Add “?” to types

Add the materialization rule (with suitable \(\sqsubseteq\))

Compile to insert casts

Et voila: you have added gradual typing

\[
\begin{align*}
\text{Types} & \quad T ::= \text{Int} \mid \text{Bool} \mid T \to T \\
\text{Terms} & \quad a, b ::= x \mid ab \mid \lambda x: T. a \mid 1 \mid 2 \mid \ldots \\
(\lambda x: T. a)b & \to a[b/x]
\end{align*}
\]

\[
\begin{align*}
\text{[VAR]} & \quad \Gamma \vdash x : \Gamma(x) \\
\text{[\to \text{INTRO}]} & \quad \Gamma, x : S \vdash a : T \\
\text{[\to \text{ELIM}]} & \quad \Gamma \vdash a : S \to T \quad \Gamma \vdash b : S \\
\Gamma \vdash \lambda x : S. a : S \to T & \quad \Gamma \vdash ab : T
\end{align*}
\]
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

\[ \text{Types } T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \]
\[ \text{Terms } a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid \ldots \]

\begin{align*}
[\text{VAR}] & \quad [\rightarrow \text{INTRO}] & \quad [\rightarrow \text{ELIM}] \\
\Gamma \vdash x : \Gamma(x) & \quad \Gamma, x : S \vdash a : T & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
& \quad \Gamma \vdash \lambda x:S.a : S \rightarrow T & \quad \Gamma \vdash ab : T
\end{align*}
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

\[
\begin{align*}
\text{Types} & \quad T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \\
\text{Terms} & \quad a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid \ldots
\end{align*}
\]

\[
\begin{align*}
& \text{[VAR]} & \Gamma \vdash x : \Gamma(x) \\
& \text{[\rightarrow \text{INTRO}]} & \Gamma, x : S \vdash a : T \\
& & \Gamma \vdash \lambda x : S.a : S \rightarrow T \\
& \text{[\rightarrow \text{ELIM}]} & \Gamma \vdash a : S \rightarrow T \\
& & \Gamma \vdash b : S \\
& & \Gamma \vdash ab : T \\
& \text{[\text{MATERIALIZE}]} & \Gamma \vdash a : S \quad S \sqsubseteq T \\
& & \Gamma \vdash a : T
\end{align*}
\]
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

Types
\[ T ::= \text{Int} \mid \text{Bool} \mid T \to T \mid ? \]

Terms
\[ a, b ::= x \mid ab \mid \lambda x : T.a \mid 1 \mid 2 \mid ... \]

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

\[ \Gamma, x : S \vdash a : T \]

\[ \Gamma \vdash \lambda x : S. a : S \to T \]

\[ \Gamma \vdash ab : T \]

\[ \Gamma \vdash a : S \quad S \sqsubseteq T \]

\[ \Gamma \vdash a : T \]

\[ \Gamma \vdash a : S \quad \text{compiles} \quad a' \quad S \sqsubseteq T \]

\[ \Gamma \vdash a : T \quad \text{compiles} \quad a'(T) \]
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

Types $T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ?$

Terms $a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ...$

$\Gamma \vdash \lambda x:T.a \ b \rightarrow a[b/x]$
Summing up

1. Take your favorite typed language
2. Add “?” to types
3. Add the materialization rule (with suitable ⊑)
4. Compile to insert casts
5. Et voila: you have added gradual typing

Types $T ::= \text{Int} | \text{Bool} | T \rightarrow T | ?$

Terms $a, b ::= x | ab | \lambda x : T.a | 1 | 2 | ...$

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

Is it that simple?!?!
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

Types $T$ := $\text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ?$

Terms $a, b$ := $x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ...$

$\Gamma \vdash x : \Gamma(x)$
$\Gamma \vdash \lambda x:S.a : S \rightarrow T$
$\Gamma \vdash ab : T$

$\Gamma \vdash a : S$ $\quad S \sqsubseteq T$
$\Gamma \vdash a : T$

$\Gamma \vdash a : T$ $\quad$ [MATERIALIZE] $\quad$ $\Gamma \vdash a : S$ $\quad$ [MATERIALIZE\text{_{COMPILED}}] $\quad$ $\Gamma \vdash a : T$ $\quad$ $\Gamma \vdash a : S$ $\quad$ $\Gamma \vdash a : T$ $\quad$ $\Gamma \vdash a : T$ $\quad$ $\Gamma \vdash a : T$

$\Gamma \vdash \lambda x:S.a : S \rightarrow T$

$(\lambda x:T.a)b \rightarrow a[b/x]$
Summing up

1. Take your favorite typed language
2. Add “?” to types
3. Add the materialization rule (with suitable ⊑)
4. Compile to insert casts
5. Et voila: you have added gradual typing

Types
\[ T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \]

Terms
\[ a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ... \]

\[
\begin{align*}
\text{[VAR]} & \quad \Gamma, x : S \vdash a : T \\
\Gamma \vdash x : \Gamma(x) & \quad \Gamma \vdash \lambda x:S.a : S \rightarrow T \\
\text{[\text{INTRO}]} & \quad \Gamma \vdash ab : T \\
\text{[\text{ELIM}]} & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
\end{align*}
\]

\[
\begin{align*}
\text{[\text{MATERIALIZE}]} & \quad \Gamma \vdash a : S \\
S \sqsubseteq T & \quad \Gamma \vdash a : T \\
\text{[\text{MATERIALIZE}_{\text{COMPIL}}]} & \quad \Gamma \vdash a : S \dashv\vdash a' \\
S \sqsubseteq T & \quad \Gamma \vdash a : T \dashv\vdash a' \langle T \rangle \\
\end{align*}
\]

YES!... as long as you don’t pretend to implement it!!!
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

Types $T ::= \text{Int} | \text{Bool} | T \to T | ?$

Terms $a, b ::= x | ab | \lambda x: T . a | 1 | 2 | ...$

\[
\begin{align*}
\text{[VAR]} & \\
\Gamma \vdash x : \Gamma(x) & \\
\hline
\text{[\rightarrow INTRO]} & \\
\Gamma, x : S \vdash a : T & \\
\hline
\text{[\rightarrow ELIM]} & \\
\Gamma \vdash a : S \to T & \Gamma \vdash b : S \\
\hline
\end{align*}
\]

$\Gamma \vdash (\lambda x : T . a) b \rightarrow a[b/x]$

\[
\begin{align*}
\text{[MATERIALIZE]} & \\
\Gamma \vdash a : S & S \sqsubseteq T \\
\hline
\text{[MATERIALIZE}_{\text{COMPIL}}] & \\
\Gamma \vdash a : S & \overset{\text{compiles}}{\rightarrow} a' & S \sqsubseteq T \\
\hline
\Gamma \vdash a : T & \overset{\text{compiles}}{\rightarrow} a' \langle T \rangle
\end{align*}
\]
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 \(\text{MATERIALIZE}\).

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:

\[
\text{double} \langle \text{Int} \rightarrow \text{Int} \rangle(42) \rightarrow ???
\]

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.
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:

  \[(\text{double}\langle\text{Int} \rightarrow \text{Int}\rangle)(42) \rightarrow ???\]
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:

  \[(\text{double} \langle \text{Int} \rightarrow \text{Int} \rangle)(42) \rightarrow ????\]

- **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:
  \[
  \text{(double} \langle \text{Int} \to \text{Int} \rangle \text{)}(42) \quad \rightarrow \quad ????
  \]

- **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?
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 \[\text{MATERIALIZE}\].

- **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:

\[
(\text{double}\langle\text{Int} \rightarrow \text{Int}\rangle)(42) \rightarrow ????
\]

- **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:

Types \( T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \)

Terms \( a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid \ldots \)

Semantics:

\[ (\beta) \quad (\lambda x:T.a)b \rightarrow a[b/x] \]

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad \Gamma, x : S \vdash a : T \\
\Gamma \vdash \lambda x:S.a : S \rightarrow T & \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
\Gamma \vdash ab : T & 
\end{align*}
\]
A principled approach

Simply Typed Lambda Calculus

Syntax:

Types \( T \) ::= Int | Bool | \( T \rightarrow T \) | ?

Terms \( a, b \) ::= \( x \) | \( ab \) | \( \lambda x: T.a \) | 1 | 2 | ...

Semantics:

\[(\beta)\] \((\lambda x: T.a)b \rightarrow a[b/x]\]

Typing

\( \Gamma, x : S \vdash a : T \)
\( \Gamma \vdash x : \Gamma(x) \)

\( \Gamma \vdash \lambda x : S.a : S \rightarrow T \)

\( \Gamma \vdash a : S \rightarrow T \)
\( \Gamma \vdash b : S \)

[Materialize] \( \Gamma \vdash a : S \quad S \sqsubseteq T \)
\( \Gamma \vdash a : T \)
A principled approach

Simply Typed Lambda Calculus

Syntax:

Types  \( T \) ::= \text{Int} \mid \text{Bool} \mid T \to T \mid ?

Terms  \( a, b \) ::= x \mid ab \mid \lambda x: T.a \mid 1 \mid 2 \mid ...

Semantics:

\[
(\beta) \quad (\lambda x: T.a)b \rightarrow a[b/x]
\]

Typing

\[
\begin{align*}
\Gamma, x : S & \vdash a : T \\
\Gamma & \vdash x : \Gamma(x) \\
\Gamma & \vdash \lambda x : S.a : S \to T \\
\Gamma & \vdash a : S \rightarrow T \\
\Gamma & \vdash b : S \\
\Gamma & \vdash ab : T \\
\end{align*}
\]

\[
\text{[MATERIALIZE]} \quad \Gamma & \vdash a : S \\
S & \sqsubseteq T \\
\Gamma & \vdash a : T
\]

semantics must be given by compilation
A principled approach

Simply Typed Lambda Calculus

Syntax:

Types \( T \) ::= \( \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \)

Terms \( a, b \) ::= \( x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ... \)

Semantics:

\[
\begin{align*}
\text{[MATERIALIZE} & \text{COMPIL]} \quad & \quad \Gamma \vdash a : S \quad \text{compiles} \quad a' \quad S \trianglelefteq T \\
\quad & \quad \Gamma \vdash a : T \quad \text{compiles} \quad a' \langle T \rangle
\end{align*}
\]

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) \\
\Gamma, x : S \vdash a : T \\
\Gamma \vdash \lambda x:S.a : S \rightarrow T \\
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
\Gamma \vdash ab : T \\
\text{[MATERIALIZE]} \quad \Gamma \vdash a : S \quad S \trianglelefteq T \\
\Gamma \vdash a : T
\end{align*}
\]
A principled approach

**Simply Typed Lambda Calculus + Gradual Typing**

**Syntax:**

\[
\begin{align*}
\text{Types} & : \quad T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \\
\text{Terms} & : \quad a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ...
\end{align*}
\]

**Semantics:**

\[
\begin{align*}
\text{Materialize} & \quad \begin{array}{c}
\Gamma \vdash a : S \quad \text{compiles} \quad a' \\
\end{array} \\
& \quad \begin{array}{c}
\quad S \subseteq T \\
\Gamma \vdash a : T \quad \text{compiles} \quad a'(T)
\end{array}
\end{align*}
\]

**Typing**

\[
\begin{align*}
\Gamma, x : S & \vdash a : T \\
\Gamma & \vdash x : \Gamma(x) \\
\Gamma & \vdash \lambda x:S.a : S \rightarrow T \\
\Gamma & \vdash a : S \rightarrow T \\
\Gamma, b : S & \vdash b \\
\Gamma & \vdash ab : T \\
\end{align*}
\]

\[
\begin{align*}
\text{Materialize} & \quad \begin{array}{c}
\Gamma \vdash a : S \\
S \subseteq T
\end{array} \\
& \quad \begin{array}{c}
\Gamma \vdash a : T
\end{array}
\end{align*}
\]
A principled approach

**Simply Typed Lambda Calculus + Gradual Typing + Subtyping**

**Syntax:**

\[
\begin{align*}
Types & \quad T ::=} \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \\
Terms & \quad a, b ::=} x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ...
\end{align*}
\]

**Semantics:**

\[
\begin{align*}
\text{[MATERIALIZE}_{\text{COMPIL}} \quad & \quad \frac{\Gamma \vdash a : S \text{ \tiny compiles}}{\Gamma \vdash a' : S \subseteq T} \\
\frac{\Gamma \vdash a : T \text{ \tiny compiles}}{\Gamma \vdash a'(T)}
\end{align*}
\]

**Typing**

\[
\begin{align*}
\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash x : \Gamma(x)} & \quad \frac{\Gamma \vdash \lambda x:S.a : S \rightarrow T}{\Gamma \vdash ab : T} \\
\frac{\Gamma \vdash a : S \rightarrow T}{\Gamma \vdash b : S} & \quad \frac{\Gamma \vdash a : T}{\Gamma \vdash a : T} \\
\frac{\Gamma \vdash a : S \text{ \tiny \subseteq T}}{\Gamma \vdash a : T} & \quad \frac{\Gamma \vdash a : S \text{ \tiny \leq T}}{\Gamma \vdash a : T}
\end{align*}
\]
If the reduction semantics of the cast calculus is reasonably defined (see later) then:

**Theorem (Soundness)**

If \( \Gamma \vdash a : T \), then \( \Gamma \vdash a : T \) \( \xrightarrow{\text{compiles}} a' \) and

- either \( a' \) reduces to a value of type \( T \)
- or \( a' \) diverges
- or \( a' \) fails for a cast on a dynamic type
If the reduction semantics of the cast calculus is reasonably defined (see later) then:

Theorem (Soundness)

If $\Gamma \vdash a : T$, then $\Gamma \vdash a : T \overset{\text{compiles}}{\rightarrow} a'$ and

- either $a'$ reduces to a value of type $T$
- or $a'$ diverges
- or $a'$ fails for a cast on a dynamic type
**Syntax:**

\[ T ::= \text{Int} \mid \text{Bool} \mid T \to T \mid \alpha \]

\[ \sigma ::= T \mid \forall \alpha. \sigma \]

\[ a, b ::= x \mid ab \mid \lambda x.a \mid \text{let } x = a \text{ in } b \mid 1 \mid 2 \mid \ldots \]

**Semantics:**

\[(\beta) \quad (\lambda x.a)b \rightarrow a[b/x] \]

**Typing**

\[ \Gamma, x : S \vdash a : T \quad \Gamma \vdash a : S \to T \quad \Gamma \vdash b : S \]

\[ \Gamma \vdash x : \Gamma(x) \quad \Gamma \vdash \lambda x.a : S \to T \quad \Gamma \vdash ab : T \]

\[ \Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 \quad \Gamma \vdash a : T \quad \alpha \notin \text{fv}(\Gamma) \quad \Gamma \vdash a : \forall \alpha. T \]

\[ \Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 \quad \Gamma \vdash a : \forall \alpha. T \quad \Gamma \vdash a : T[S/\alpha] \]
Syntax:

\[\begin{align*}
\text{Types} & \quad T ::= \text{Int} \mid \text{Bool} \mid T \to T \mid \alpha \mid \,?
\end{align*}\]

\[\begin{align*}
\text{Schemas} & \quad \sigma ::= T \mid \forall \alpha. \sigma
\end{align*}\]

\[\begin{align*}
\text{Terms} & \quad a, b ::= x \mid ab \mid \lambda x.a \mid \text{let } x = a \text{ in } b \mid 1 \mid 2 \mid ...
\end{align*}\]

Semantics:

\[\begin{align*}
\Gamma \vdash a : S \quad \text{compiles} \quad a' \quad S \sqsubseteq T
\end{align*}\]

Typing

\[\begin{align*}
\Gamma, x : S \vdash a : T & \quad \Gamma \vdash a : S \to T \\
\Gamma \vdash x : \Gamma(x) & \\
\Gamma \vdash \lambda x.a : S \to T & \\
\Gamma \vdash ab : T & \\
\Gamma \vdash a : \sigma_1 & \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 & \quad \Gamma \vdash a : T \quad \alpha \not\in \text{fv}(\Gamma) & \quad \Gamma \vdash a : \forall \alpha. T \\
\Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 & \\
\left[\text{MATERIALIZE}\right] & \\
\Gamma \vdash a : S \quad S \sqsubseteq T
\end{align*}\]
HM Polymorphism + Gradual Typing + Subtyping

Syntax:

*Types* \( T \) ::=

- Int | Bool | \( T \to T \) | \( \alpha \) | ?

*Schemas* \( \sigma \) ::=

- \( T \) | \( \forall \alpha.\sigma \)

*Terms* \( a, b \) ::=

- \( x \) | \( ab \) | \( \lambda x.a \) | \( \text{let } x = a \text{ in } b \) | 1 | 2 | ...

Semantics:

\[ \Gamma, x : S \vdash a : T \]

\[ \Gamma \vdash a : \sigma_1 \quad \text{and} \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 \]

\[ \Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 \]

Typing

\[ \Gamma \vdash a : S \to T \quad \text{and} \quad \Gamma \vdash b : S \]

\[ \Gamma, x : S \vdash a : T \quad \Gamma \vdash \lambda x.a : S \to T \]

\[ \Gamma \vdash a : \forall \alpha.T \quad \alpha \notin \text{fv}(\Gamma) \]

\[ \Gamma \vdash a : T[S/\alpha] \]

[Materialize] \[ \Gamma \vdash a : S \quad S \sqsubseteq T \]

[Subsum] \[ \Gamma \vdash a : S \quad S \leq T \]

G. Castagna (CNRS)
HM Polymorphism + Gradual Typing + Subtyping

Syntax:

Types \( T \) ::= \( \text{Int} \mid \text{Bool} \mid T \rightarrow T \)

Schemas \( \sigma \) ::= \( T \mid \forall \alpha. \sigma \)

Terms \( a, b \) ::= \( x \mid ab \mid \lambda x.a \mid \text{let} x = a \text{ in } b \mid 1 \mid 2 \mid \ldots \)

Semantics:

\[ \Gamma \vdash a : S \vdash a' \quad S \sqsubseteq T \]

Typing

\[ \Gamma, x : S \vdash a : T \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \]

\[ \Gamma \vdash \lambda x.a : S \rightarrow T \]

\[ \Gamma \vdash ab : T \]

\[ \Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 \]

\[ \Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 \]

\[ \Gamma \vdash a : T \quad \alpha \notin \text{fv}(\Gamma) \]

\[ \Gamma \vdash a : \forall \alpha. T \]

\[ \Gamma \vdash a : T[S/\alpha] \]

[\text{MATERIALIZE}]

\[ \Gamma \vdash a : S \quad S \sqsubseteq T \]

[\text{SUBSUM}]

\[ \Gamma \vdash a : S \quad S \leq T \]

Some details are missing: annotations and no inference for gradual types ... but that's it!!
HM Polymorphism + Gradual Typing + Subtyping

Syntax:

**Types**
\[ T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid \alpha \mid \text{??}
\]

**Schemas**
\[ \sigma ::= T \mid \forall \alpha.\sigma
\]

**Terms**
\[ a, b ::= x \mid ab \mid \lambda x.a \mid \text{let } x = a \text{ in } b \mid 1 \mid 2 \mid \ldots
\]

Semantics:

\[ \Gamma \vdash a : S \xrightarrow{\text{compiles}} a' \quad S \subseteq T
\]

Typing

\[ \Gamma, x : S \vdash a : T \]
\[ \Gamma \vdash \lambda x.a : S \rightarrow T \]
\[ \Gamma \vdash ab : T \]
\[ \Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 \]
\[ \Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 \]
\[ \Gamma \vdash a : T \quad \alpha \notin \text{fv}(\Gamma) \]
\[ \Gamma \vdash a : \forall \alpha. T \]
\[ \Gamma \vdash a : T[S/\alpha] \]

That's all, but how do I implement it?!?
Outline

15 Main ideas

16 Formal system

17 Algorithmic Aspects

18 Criteria for Gradual Typing

19 Implementation issues

20 References
1. Type-checking algorithm

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad \Gamma, x : S \vdash a : T \\
\Gamma \vdash a : S \rightarrow T & \quad \Gamma \vdash \lambda x : S. a : S \rightarrow T \\
\Gamma \vdash b : S & \quad [\text{MATERIALIZE}] \\
\Gamma \vdash ab : T & \quad \Gamma \vdash a : T \\
\end{align*}
\]
1. Type-checking algorithm

\[
\begin{align*}
\Gamma, x : S &\vdash a : T \\
\Gamma \vdash \lambda x : S. a : S \rightarrow T \\
\Gamma \vdash a : S \rightarrow T &\quad \Gamma \vdash b : S \\
\Gamma &\vdash ab : T \\
\Gamma \vdash a : T &\quad [\text{MATERIALIZE}] \\
\Gamma \vdash a : S &\quad S \sqsubseteq T \\
\Gamma &\vdash a : T
\end{align*}
\]
1. Type-checking algorithm

\[
\Gamma \vdash \mathcal{A} \ x : \Gamma(x) \quad \frac{\Gamma, x : S \vdash \mathcal{A} \ a : T}{\Gamma \vdash \mathcal{A} \ \lambda x : S \ a : S \rightarrow T}
\]

\[
[\rightarrow \text{ELIM}_{\subseteq}] \quad \frac{\Gamma \vdash \mathcal{A} \ a : S \rightarrow T}{\Gamma \vdash \mathcal{A} \ ab : T} \quad \frac{\Gamma \vdash \mathcal{A} \ b : U}{\exists V. S \subseteq V, U \subseteq V}
\]

It is a sound and complete algorithm:

\[
\Gamma \vdash \mathcal{A} \ a : T \quad \Gamma, x : S \vdash \mathcal{A} \ a : T \quad \text{and} \quad S \sqsubseteq T
\]

Actually this is the good old \([\rightarrow \text{ELIM}]\) rule of Siek&Taha (but defined for a sensible relation):
1. Type-checking algorithm

\[
\Gamma \vdash \lambda x:S.a : S \rightarrow T \\
\Gamma, x : S \vdash a : T
\]

\[
[\rightarrow_{\text{ELIM}}] \frac{\Gamma \vdash a : S \rightarrow T}{\Gamma \vdash ab : T}
\]

\[
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \\
\exists V. S \subseteq V, U \subseteq V
\]

It is a sound and complete algorithm:

\[
\Gamma \vdash a : T \quad \iff \quad \Gamma \vdash a : S \text{ and } S \subseteq T
\]
1. Type-checking algorithm

\[ \Gamma \vdash x : \Gamma(x) \]
\[ \Gamma \vdash \lambda x : S. a : S \rightarrow T \]
\[ \Gamma \vdash a : S \rightarrow T \]
\[ \Gamma \vdash b : U \]
\[ \exists V. S \sqsubseteq V, U \sqsubseteq V \]

It is a sound and complete algorithm:
\[ \Gamma \vdash a : T \iff \Gamma \vdash a : S \text{ and } S \sqsubseteq T \]

Actually this is the good old \([\rightarrow \text{ELIM}_\sqsubseteq]\) rule of Siek&Taha (but defined for a sensible relation): 
\[ \Gamma \vdash ab : T \]

since \[ U \sim S \iff \exists V. S \sqsubseteq V, U \sqsubseteq V \]
Thanks to the algorithm every well-typed term is associated to a unique typing derivation: we know *where* to put casts.

\[
\Gamma \vdash A \ a : S \rightarrow T \\
\Gamma \vdash A \ b : U \\
\Gamma \vdash A \ a \ (b) : T \\
\exists \ V . S \sqsubseteq V, U \sqsubseteq V \\
\Gamma \vdash a : V \rightarrow T \\
\Gamma \vdash b : U \\
\Gamma \vdash A \ a \ (b) : T \\
\end{align*}

Which \( V \) shall we use? Well, obviously:

\[
V = \min \{ W | S \sqsubseteq W, U \sqsubseteq W \} 
\]
2. Compilation

Thanks to the algorithm every well-typed term is associated to a unique typing derivation: we know \textit{where} to put casts. Indeed:

\[
\begin{array}{c}
\Gamma \vdash A \ a : S \to T & \quad & \Gamma \vdash A \ b : U \\
\[\to\text{ELIM}_{\subseteq}\] & \quad & \exists V. S \subseteq V, U \subseteq V \\
\Gamma \vdash A \ a(b) : T \\
\end{array}
\]
Thanks to the algorithm every well-typed term is associated to a unique typing derivation: we know \textit{where} to put casts. Indeed:

\[
\begin{array}{c}
\Gamma \vdash \downarrow_{\text{ELIM}} \quad \Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U \\
\Gamma \vdash a(b) : T
\end{array}
\]

\[
\Gamma \vdash a : S \rightarrow T \\
S \subseteq V
\]

\[
\Gamma \vdash b : U \\
U \subseteq V
\]

\[
\begin{array}{c}
\Gamma \vdash a : V \rightarrow T \\
\rightarrow_{\text{ELIM}}
\end{array}
\]

\[
\begin{array}{c}
\exists V. S \subseteq V, U \subseteq V
\end{array}
\]

\[
\Gamma \vdash a(b) : T
\]

Which \( V \) shall we use? Well, obviously:

\[ V = \min \{ W | S \subseteq W, U \subseteq W \} \]
Thanks to the algorithm every well-typed term is associated with a unique typing derivation: we know _where_ to put casts. Indeed:

\[
\frac{\Gamma \vdash a : S \to T \quad \Gamma \vdash b : U}{\exists V. S \sqsubseteq V, U \sqsubseteq V}
\]

\[
\frac{\Gamma \vdash a(b) : T}{\exists V. S \sqsubseteq V, U \sqsubseteq V}
\]

corresponds to the derivation _which tells us where to put cast:_

\[
\frac{S \sqsubseteq V \quad T \sqsubseteq T}{\frac{\Gamma \vdash a : S \to T \quad S \to T \sqsubseteq V \to T}{\Gamma \vdash a(V \to T) : V \to T}} \quad \frac{\Gamma \vdash b : U \quad U \sqsubseteq V}{\Gamma \vdash b(V) : V}
\]

\[
\frac{\Gamma \vdash a(V \to T)(b(V)) : T}{\exists V. S \sqsubseteq V, U \sqsubseteq V}
\]
2. Compilation

Thanks to the algorithm every well-typed term is associated to a unique typing derivation: we know *where* to put casts. Indeed:

\[
\frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : U}{\exists V. S \subseteq V, U \subseteq V}
\]

corresponds to the derivation which tells us where to put cast:

\[
\frac{\frac{\frac{\Gamma \vdash a : S \rightarrow T}{\Gamma \vdash a(V \rightarrow T) : V \rightarrow T}}{\Gamma \vdash b : U}{\Gamma \vdash b(V) : V}}{\frac{\frac{\frac{\Gamma \vdash a(V \rightarrow T) \ (b(V)) : T}{\Gamma \vdash a(V \rightarrow T) : V \rightarrow T}}{\Gamma \vdash b : U}{\Gamma \vdash b(V) : V}}}
\]

Which V shall we use? Well, obviously:

\[
V = \min \{ W \mid S \subseteq W, U \subseteq W \}
\]
This yields the following compilation rule:

\[
\begin{array}{c}
\Gamma \vdash a : S \rightarrow T \quad \text{compiles} \quad a' \\
\Gamma \vdash b : U \quad \text{compiles} \quad b' \\
\Gamma \vdash \lambda a \ b : T \\
\Gamma \vdash a' \langle V \rightarrow T \rangle (b' \langle V \rangle)
\end{array}
\]

\( (V = \min \{ W \mid S \subseteq W, U \subseteq W \} ) \)
This yields the following compilation rule:

\[
\begin{array}{c}
\Gamma \vdash a : S \rightarrow T \quad \text{compiles} \quad a' \\
\Gamma \vdash b : U \quad \text{compiles} \quad b' \\
\Gamma \vdash \lambda a b : T \quad \text{compiles} \quad a' \langle V \rightarrow T \rangle (b' \langle V \rangle) \\
(V = \min \{ W \mid S \sqsubseteq W, U \sqsubseteq W \})
\end{array}
\]

Of course we do not insert the corresponding cast when \( V = S \) or \( V = U \).
2. Compilation

This yields the following compilation rule:

\[
\begin{align*}
\frac{
\Gamma \vdash a : S \rightarrow T \quad \text{compiles} \quad \Gamma \vdash b : U \quad \text{compiles}
}{
\Gamma \vdash \lambda a b : T \quad \text{compiles} \quad a' \langle V \rightarrow T \rangle (b' \langle V \rangle)
}\end{align*}
\]

\((V = \min \subseteq \{ W \mid S \subseteq W, U \subseteq W \})\)

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 argument:

We only use “upcast”, that is cast from less precise to more precise types.

This is formalized by the [MATERIALIZE] rule for \textit{the language with casts} (all the other rules are as before)

\[
\begin{align*}
\frac{
\Gamma \vdash a : S \quad S \subseteq T
}{
\Gamma \vdash a \langle T \rangle : T
}\end{align*}
\]
2. Compilation

This yields the following compilation rule:

\[
\frac{
\Gamma \vdash a : S \rightarrow T \quad \text{compiles} \quad a' \\
\Gamma \vdash b : U \quad \text{compiles} \quad b' 
}{
\Gamma \vdash a b : T \quad \text{compiles} \quad a' \langle V \rightarrow T \rangle (b' \langle V \rangle) 
}
\]

\( (V = \min \{ W \mid S \sqsubseteq W, U \sqsubseteq W \} ) \)

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 argument:

We only use “upcast”, that is cast from less precise to more precise types. This is formalized by the \([\text{MATERIALIZE}]\) rule for the language with casts (all the other rules are as before)

\[
\frac{
\Gamma \vdash a : S \\
S \sqsubseteq T
}{
\Gamma \vdash a \langle T \rangle : T
}
\]

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:

\[
\frac{\Gamma \vdash a : S \to T \text{ compiles } a' \quad \Gamma \vdash b : U \text{ compiles } b'}{\Gamma \vdash \lambda a\, b : T \text{ compiles } a'(V \to T)(b'(V))}
\]

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 argument:

We only use “upcast”, that is cast from less precise to more precise.

This is formalized by the [MATERIALIZE] rule for the language with casts (all the other rules are as before)

\[
\frac{\Gamma \vdash a : S \quad S \sqsubseteq T}{\Gamma \vdash a(T) : T}
\]

The compilation rules map well-typed terms into well-typed terms: terms are cast to types more precise than their static type.
The cast language

Gradually Typed Language

Syntax:

\[ \text{Types} \quad T ::= \text{Int} \mid \text{Bool} \mid T \to T \mid ? \]

\[ \text{Terms} \quad a, b ::= x \mid ab \mid \lambda x:T.a \mid 1 \mid 2 \mid ... \]

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad \frac{\Gamma, x : S \vdash a : T}{\Gamma, x : S \vdash \lambda x : S.a : S \to T} & \quad \frac{\Gamma \vdash a : S \to T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T}
\end{align*}
\]
The cast language

Gradually Typed Language

Syntax:

Types \( T \) ::= \text{Int} | \text{Bool} | T \rightarrow T | ?

Terms \( a, b \) ::= x | ab | \lambda x: T. a | a(T) | 1 | 2 | ... 

Typing

\[
\Gamma \vdash x : \Gamma(x) \quad \frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash \lambda x:S. a : S \rightarrow T} \quad \frac{\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S}{\Gamma \vdash ab : T}
\]

Semantics:

\((\beta) \quad (\lambda x: T. a) b \rightarrow a[b/x]\)
### Gradually Typed Language

**Syntax:**

Types

\[ T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \]

Terms

\[ a, b ::= x \mid ab \mid \lambda x:T.a \mid a\langle T \rangle \mid 1 \mid 2 \mid \ldots \]

**Typing**

\[
\begin{align*}
\Gamma, x : S & \vdash a : T \\
\Gamma & \vdash x : \Gamma(x) \\
\Gamma, x : S & \vdash a : S \rightarrow T \\
\Gamma & \vdash \lambda x : S.a : S \rightarrow T \\
\Gamma & \vdash ab : T \\
\Gamma & \vdash a : S \\
S \sqsubseteq T & \quad \text{[MATERIALIZE]} \\
\Gamma & \vdash a\langle T \rangle : T \\
\Gamma & \vdash b : S
\end{align*}
\]

Still missing the semantics for casts
The cast language

Gradually Typed Language

Syntax:

\[
\begin{align*}
\text{Types} & \quad T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \\
\text{Terms} & \quad a, b ::= x \mid ab \mid \lambda x:T.a \mid a\langle T \rangle \mid 1 \mid 2 \mid \ldots
\end{align*}
\]

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad \Gamma \vdash \lambda x:S.a : S \rightarrow T \\
\Gamma \vdash a : S \rightarrow T & \quad \Gamma \vdash b : S \\
\Gamma \vdash ab : T & \\
[\text{Materialize}] \quad \Gamma \vdash a : S \quad S \sqsubseteq T \\
\Gamma \vdash a\langle T \rangle : T
\end{align*}
\]

Semantics:

\[
(\beta) \quad (\lambda x:T.a)b \rightarrow a[b/x]
\]
The cast language

Gradually Typed Language with Casts

Syntax:

Types  \( T ::= \text{Int} | \text{Bool} | T \rightarrow T | ? \)

Terms  \( a, b ::= x | ab | \lambda x:T.a | a\langle T \rangle | 1 | 2 | ... \)

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) \\
\Gamma, x : S \vdash a : T \\
\Gamma \vdash \lambda x:S.a : S \rightarrow T \\
\Gamma \vdash a : S \rightarrow T \quad \Gamma \vdash b : S \\
\Gamma \vdash ab : T \\
[\text{MATERIALIZE}] \quad \Gamma \vdash a : S \quad S \sqsubseteq T \\
\Gamma \vdash a\langle T \rangle : T
\end{align*}
\]

Semantics:

\((\beta)\)  \( (\lambda x:T.a)b \rightarrow a[b/x] \)
The cast language

Gradually Typed Language with Casts

Syntax:

Types  \( T \) ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ?

Terms  \( a, b \) ::= x \mid ab \mid \lambda x:T.a \mid a\langle T \rangle \mid 1 \mid 2 \mid ...

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & & \Gamma, x : S \vdash a : T & & \Gamma \vdash a : S \rightarrow T & & \Gamma \vdash b : S \\
\Gamma \vdash \lambda x:S.a : S \rightarrow T & & \Gamma \vdash ab : T \\
\end{align*}
\]

Semantics:

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

Still missing the semantics for casts
The cast language

What is the dynamic semantics of casts?

Easy for non-functional values:

\[3 \langle \text{Int} \rangle \rightarrow 3\]
\[3 \langle \text{Bool} \rangle \rightarrow \text{Fail}\]

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:

```javascript
function foo (x : ???) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \( \text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle \).

Function \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \), nevertheless \( (\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle)(42) \) must not fail: it's applied to an \( \text{Int} \) and returns an \( \text{Int} \).

Delay the dynamic check of a type until you get to non-functional values

\[\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle() \rightarrow (\text{foo}(42 \langle \text{Int} \rangle)) \langle \text{Int} \rangle\]
The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

\[ 3\langle \text{Int} \rangle \rightarrow 3 \]
\[ 3\langle \text{Bool} \rangle \rightarrow \text{Fail} \]

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 \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \), nevertheless \( (\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle)(42) \) must not fail: it's applied to an \( \text{Int} \) and returns an \( \text{Int} \).

Delay the dynamic check of a type until you get to non-functional values: \( (\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle)() \rightarrow \langle \text{Int} \rangle \).
The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

\[
\begin{align*}
3\langle\text{Int}\rangle & \rightarrow 3 \\
3\langle\text{Bool}\rangle & \rightarrow \text{Fail}
\end{align*}
\]

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:

\[3\langle \text{Int} \rangle \rightarrow 3\]
\[3\langle \text{Bool} \rangle \rightarrow \text{Fail}\]

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:

```javascript
function foo (x : ?) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \(\text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle\).
The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

\[
\begin{align*}
3\langle\text{Int}\rangle & \rightarrow 3 \\
3\langle\text{Bool}\rangle & \rightarrow \text{Fail}
\end{align*}
\]

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:

```plaintext
function foo (x : ?) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \( \text{foo}\langle\text{Int} \rightarrow \text{Int}\rangle \). Function \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \)
The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

$3\langle\text{Int}\rangle \rightarrow 3$
$3\langle\text{Bool}\rangle \rightarrow \text{Fail}$

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 $\text{foo}\langle\text{Int} \rightarrow \text{Int}\rangle$. Function $\text{foo}$ is not of type $\text{Int} \rightarrow \text{Int}$, nevertheless $(\text{foo}\langle\text{Int} \rightarrow \text{Int}\rangle)(42)$ must not fail: it's applied to an $\text{Int}$ and returns an $\text{Int}$. 
The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

\[
3\langle \text{Int} \rangle \rightarrow 3 \\
3\langle \text{Bool} \rangle \rightarrow \text{Fail}
\]

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 \( \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \). Function \( \text{foo} \) is not \( \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \)

\( \langle \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \rangle(\text{exp})? \)

That is easy, but what about \( \langle \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \rangle(42) \)?

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:

\[ 3 \langle \text{Int} \rangle \rightarrow 3 \]
\[ 3 \langle \text{Bool} \rangle \rightarrow \text{Fail} \]

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:

```javascript
function foo (x : ?) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \( \text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle \). Function \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \), nevertheless \( (\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle)(42) \) must not fail: it’s applied to an \( \text{Int} \) and returns an \( \text{Int} \).

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:

\[ 3\langle \text{Int} \rangle \rightarrow 3 \]
\[ 3\langle \text{Bool} \rangle \rightarrow \text{Fail} \]

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:

```javascript
function foo (x : ?) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \( \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \). Function \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \), nevertheless \( \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \langle 42 \rangle \) must not fail: it's applied to an \( \text{Int} \) and returns an \( \text{Int} \).

Delay the dynamic check of a type until you get to non-functional values

\[ \text{foo}\langle \text{Int} \rightarrow \text{Int} \rangle \langle \text{exp} \rangle \]
The cast language

What is the dynamic semantics of casts?

Easy for non functional values:

\[ 3\langle \text{Int}\rangle \rightarrow 3 \]
\[ 3\langle \text{Bool}\rangle \rightarrow \text{Fail} \]

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:

```javascript
function foo (x : ???) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \( \text{foo}\langle \text{Int} \rightarrow \text{Int}\rangle \). Function \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \), nevertheless \( (\text{foo}\langle \text{Int} \rightarrow \text{Int}\rangle)(42) \) must not fail: it’s applied to an \( \text{Int} \) and returns an \( \text{Int} \).

Delay the dynamic check of a type until you get to non-functional values:

\( (\text{foo}\langle \text{Int} \rightarrow \text{Int}\rangle)(42) \)
The cast language

**What is the dynamic semantics of casts?**

Easy for non functional values:

\[ 3 \langle \text{Int} \rangle \rightarrow 3 \]
\[ 3 \langle \text{Bool} \rangle \rightarrow \text{Fail} \]

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:

```javascript
function foo (x : ?) {
    if (x == 42) { return (2*x)} else { true }
}
```

Consider \( \text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle \). Function \( \text{foo} \) is not of type \( \text{Int} \rightarrow \text{Int} \), nevertheless \( (\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle)(42) \) must not fail: it's applied to an \( \text{Int} \) and returns an \( \text{Int} \).

Delay the dynamic check of a type until you get to non-functional values

\[(\text{foo} \langle \text{Int} \rightarrow \text{Int} \rangle)(42) \rightarrow (\text{foo}(42 \langle \text{Int} \rangle)) \langle \text{Int} \rangle\]
The cast language

Syntax:

Types  \( T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid ? \)

Terms  \( a, b ::= x \mid ab \mid \lambda x:T.a \mid a\langle T\rangle \mid 1 \mid 2 \mid ... \)

Values  \( v ::= \lambda x:T.a \mid 1 \mid 2 \mid ... \)

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \\
\Gamma, x : S \vdash a : T & \\
\Gamma \vdash \lambda x:S.a : S \rightarrow T & \\
\Gamma \vdash a : S \rightarrow T & \quad \Gamma \vdash b : S & \\
\Gamma \vdash ab : T & \\
\Gamma \vdash a : S \quad S \subseteq T & \\
\text{[MATERIALIZE]} & \\
\Gamma \vdash a\langle T\rangle : T & \\
\end{align*}
\]

Semantics:

\[
\begin{align*}
(\lambda x:T.a)v & \quad \rightarrow \quad a[v/x] & \\
v\langle T\rangle & \quad \rightarrow \quad v & \quad \text{if } T \neq S_1 \rightarrow S_2 \text{ and } \vdash v : T \\
v\langle T\rangle & \quad \rightarrow \quad \text{Fail} & \quad \text{if } T \neq S_1 \rightarrow S_2 \text{ and } \nvdash v : T \\
(v_1\langle S \rightarrow T\rangle)v_2 & \quad \rightarrow \quad (v_1(v_2\langle S\rangle)\langle T\rangle)
\end{align*}
\]
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 consequences of this theorem on our initial language?
How does it fit our framework? Let me first add a further bit
The message Fail is not very useful for debugging

Tracking errors
Tracking errors

The message Fail is not very useful for debugging

We can modify compilation to track the origine of failures:

\[
\text{[MATERIALIZE]} \quad \frac{\Gamma \vdash a : S \text{ compiles } a' \quad S \sqsubseteq T}{\Gamma \vdash a : T \text{ compiles } a'(T)^\ell}
\]

where \( \ell \) 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:

\[
\text{[MATERIALIZE]} \quad \frac{\Gamma \vdash a : S \text{ compiles } a' \quad S \subseteq T}{\Gamma \vdash a : T \text{ compiles } a'\langle T\rangle^\ell}
\]

where \( \ell \) 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:**

\[
\begin{align*}
(\lambda x : T . a) v & \rightarrow a[v/x] \\
v\langle T\rangle^\ell & \rightarrow v \quad \text{if } T \neq S_1 \rightarrow S_2 \text{ and } \vdash v : T \\
v\langle T\rangle^\ell & \rightarrow \text{blame } \ell \quad \text{if } T \neq S_1 \rightarrow S_2 \text{ and } \nvdash v : T \\
(v_1\langle S \rightarrow T\rangle^\ell)v_2 & \rightarrow (v_1(v_2\langle S\rangle^\ell))\langle T\rangle^\ell
\end{align*}
\]
Outline

15 Main ideas

16 Formal system

17 Algorithmic Aspects

18 Criteria for Gradual Typing

19 Implementation issues

20 References
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 $\Gamma \vdash a : T \xrightarrow{\text{compiles}} a'$ and

- either $a'$ reduces to a value of type $T$
- or $a'$ diverges
- or $a'$ fails for a cast on a dynamic type
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 $\Gamma \vdash a : T \xrightarrow{\text{compiles}} a'$ 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 $\Gamma \vdash a : T \xrightarrow{\text{compiles}} a'$ and $\Gamma \vdash a' : S \subseteq T$
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 $\mathcal{R}$ does not occur in $a$.
If $\Gamma \vdash C[a] : T \xrightarrow{\text{compiles}} b$ and $b \rightarrow\text{blame } \ell$, then $\ell \in C[]$ and $\ell \notin a$. 
Using less precise types must not change the outcome of type checking or of running a program.
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 $\square$: it is a minimal element in the precision lattice.

**Theorem (Gradual Guarantee)**

If $\Gamma \vdash a : T \xrightarrow{\text{compiles}} a'$ and $b \sqsubseteq a$, then:

- $\Gamma \vdash b : T' \xrightarrow{\text{compiles}} b'$ and $T' \sqsubseteq T$

- if $a' \rightarrow v$, then $b' \rightarrow v'$ and $v' \sqsubseteq v$. 
Outline

15 Main ideas

16 Formal system

17 Algorithmic Aspects

18 Criteria for Gradual Typing

19 Implementation issues

20 References
A hint to efficient implementation

A gradually typed tail-recursive function:

```ocaml
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))
```

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⟨τ⟩⟨τ′⟩ can be “compressed” to E⟨τ∧τ′⟩.
```

G. Castagna (CNRS)
Four Forms of Polymorphism
A hint to efficient implementation

A gradually typed tail-recursive function: In Siek&Taha it is compiled into:

```ml
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>?` → `(even 0)<Bool>?`
A hint to efficient implementation

A gradually typed tail-recursive function:

```ocaml
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:

```ocaml
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:

\[ E \langle \tau \rangle \langle \tau' \rangle \text{ can be “compressed” to } E \langle \tau \land \tau' \rangle. \]
HM Polymorphism + Gradual Typing

Syntax:

Types \( T ::= \) Int | Bool | \( T \rightarrow T \) | \( \alpha \) | ?

Schemas \( \sigma ::= T \) | \( \forall \alpha.\sigma \)

Terms \( a, b ::= x \) | \( ab \) | \( \lambda x.a \) | \( \text{let } x = a \text{ in } b \) | 1 | 2 | ...

Semantics:

\[
\begin{align*}
\text{[MATERIALIZE}_{\text{COMPIL}}] & \quad \Gamma \vdash a : S \quad \text{compiles} \quad a' \quad S \sqsubseteq T \\
\Gamma \vdash a : T & \quad \text{compiles} \quad a'\langle T\rangle
\end{align*}
\]

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \\
\Gamma, x : S \vdash a : T & \\
\Gamma \vdash \lambda x.a : S \rightarrow T & \\
\Gamma \vdash b : S & \\
\Gamma \vdash ab : T
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash a : \sigma_1 & \\
\Gamma, x : \sigma_1 \vdash b : \sigma_2 & \\
\Gamma, x : \sigma_1 \vdash b : \sigma_2 & \\
\Gamma \vdash a : T & \\
\alpha \not\in \text{fv}(\Gamma) & \\
\Gamma \vdash a : \forall \alpha. T & \\
\Gamma \vdash a : T[S/\alpha]
\end{align*}
\]

\[
\text{[MATERIALIZE]} \quad \Gamma \vdash a : S \quad S \sqsubseteq T \\
\Gamma \vdash a : T
\]
Syntax:

Types \( T \) ::= \( \text{Int} \mid \text{Bool} \mid T \to T \mid \alpha \mid ? \)

Schemas \( \sigma \) ::= \( T \mid \forall \alpha.\sigma \)

Terms \( a, b \) ::= \( x \mid ab \mid \lambda x.a \mid \text{let } x = a \text{ in } b \mid 1 \mid 2 \mid ... \)

Semantics:

\[
\begin{align*}
\text{[MATERIALIZE}] & \quad \Gamma \vdash a : S \quad \text{compiles} \quad a' \quad S \subseteq T \\
\Gamma \vdash a : T \quad \text{compiles} \quad a'\langle T \rangle
\end{align*}
\]

Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad \Gamma, x : S \vdash a : T \\
\Gamma \vdash \lambda x.a : S \to T & \quad \Gamma \vdash a : S \to T \quad \Gamma \vdash b : S \\
\Gamma \vdash ab : T & \\
\Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 & \quad \Gamma \vdash a : T \quad \alpha \notin \text{fv}(\Gamma) \\
\Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 & \\
\Gamma \vdash a : \forall \alpha. T & \quad \Gamma \vdash a : T[S/\alpha]
\end{align*}
\]

\[
\begin{align*}
\text{[MATERIALIZE]} & \quad \Gamma \vdash a : S \quad S \subseteq T \\
\Gamma \vdash a : T \\
\text{[SUBSUM]} & \quad \Gamma \vdash a : S \quad S \leq T \\
\Gamma \vdash a : T
\end{align*}
\]
HM Polymorphism + Gradual Typing + Subtyping

Syntax:

Types \( T \) ::= \text{Int} \mid \text{Bool} \mid T \to T

Schemas \( \sigma \) ::= T \mid \forall \alpha. \sigma

Terms \( a, b \) ::= x \mid ab \mid \lambda x.a \mid \text{let } x = a \text{ in } b

Semantics:

\[
\begin{array}{c}
\Gamma \vdash a : S \quad \text{compiles} \quad a' \\
S \sqsubseteq T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : T \quad \text{compiles} \quad a' \langle T \rangle
\end{array}
\]

Typing

\[
\begin{array}{c}
\Gamma \vdash x : \Gamma(x)
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x : S \vdash a : T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \lambda x.a : S \to T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash b : S
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash ab : T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : T \quad \alpha \notin \text{fv}(\Gamma)
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : \forall \alpha. T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : T[S/\alpha]
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : S \quad S \sqsubseteq T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : S \quad S \leq T
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : T
\end{array}
\]

Some details are missing: annotations and no inference for gradual types ... but that’s it!!
Syntax:

Types $T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T$  
Schemas $\sigma ::= T \mid \forall \alpha. \sigma$  
Terms $a, b ::= x \mid ab \mid \lambda x.a \mid \text{let } x = a \text{ in } b \mid \ldots$

Semantics:

$$
\frac{\Gamma \vdash a : S \text{ [MATERIALIZE]} \quad S \sqsubseteq T}{\Gamma \vdash a : T} \quad \frac{\Gamma \vdash a' \quad S \sqsubseteq T}{\Gamma \vdash a' \langle T \rangle}
$$

Typing

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

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

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

That's all, but how do I implement it?!!?
The missing details

Syntax:

**StaticTypes** \( T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \mid \alpha \)**

**GradualTypes** \( \tau ::= \text{Int} \mid \text{Bool} \mid \tau \rightarrow \tau \mid \alpha \mid ? \)**

**Schemas** \( \sigma ::= T \mid \forall \alpha.\sigma \)**

**Terms** \( a, b ::= x \mid ab \mid \lambda x.a \mid \lambda x:\tau.a \mid \text{let } x = a \text{ in } b \mid 1 \mid 2 \)**

Typing

\[
\begin{align*}
\Gamma \vdash a : \tau' & \rightarrow \tau \quad \Gamma \vdash b : \tau' \\
\Gamma \vdash x : \Gamma(x) & \\
\Gamma \vdash ab : \tau \\
\Gamma, x : \tau \vdash a : \tau' & \\
\Gamma \vdash \lambda x : \tau.a : \tau \rightarrow \tau' \\
\Gamma \vdash \lambda x.a : S \rightarrow \tau \\
\Gamma \vdash a : \sigma_1 & \quad \Gamma, x : \sigma_1 \vdash b : \sigma_2 \\
\Gamma \vdash \text{let } x = a \text{ in } b : \sigma_2 \\
\Gamma \vdash a : \tau & \alpha \notin \text{fv}(\Gamma) \\
\Gamma \vdash a : \forall \alpha.\tau & \\
\Gamma \vdash a : \tau \quad \tau' \subseteq \tau \\
\Gamma \vdash \text{Materialize} \\
\Gamma \vdash a : \tau' & \tau' \leq \tau \\
\Gamma \vdash \text{Subsum} \\
\Gamma \vdash a : \tau &
\end{align*}
\]
Part 1: Without subtyping

We generate sets $D$ of type constraints

$$D ::= \emptyset \mid (t_1 \leq t_2) \cup D \mid (\tau \subseteq \alpha) \cup D$$

Then we find a type substitution $\theta$ that solves $D$ that is

- for all $(t_1 \leq t_2)$ we have $t_1 \theta = t_2 \theta$
- for all $(\tau \subseteq \alpha)$ we have $\tau \theta \subseteq \alpha \theta$ and $\tau \theta$ is a static type
Constraint generation

We do not directly generate type constraint. We first generate structured constraints of the form:

\[ C ::= (t \dot{} \leq t) | (\tau \dot{} \sqsubseteq \alpha) | (x \dot{} \sqsubseteq \alpha) | \texttt{def } x : \tau \texttt{ in } C | \exists \vec{\alpha}. C | C \land C \]

\[ \langle\langle \lambda x : ???. x \rangle\rangle : \text{Int} \rightarrow \text{Int} \]

can be solved, whereas

\[ \langle\langle \lambda x : ???. x \rangle\rangle : ???. ???. \rightarrow ???. \]

cannot.

\[ ^1\text{Let constraints are omitted for the sake of simplicity} \]
We do not directly generate \textit{type constraint}. We first \textit{generate structured constraints} of the form$^1$:

$$C ::= (t \leq t) \mid (\tau \subseteq \alpha) \mid (x \subseteq \alpha) \mid \texttt{def } x : \tau \texttt{ in } C \mid \exists \alpha. C \mid C \land C$$

Thus

$$\langle \langle x : t \rangle \rangle = \exists \alpha. (x \subseteq \alpha) \land (\alpha \leq t)$$

$$\langle \langle \lambda x . e : t \rangle \rangle = \exists \alpha_1, \alpha_2. (\texttt{def } x : \alpha_1 \texttt{ in } \langle \langle e : \alpha_2 \rangle \langle e : \alpha_2 \rangle \rangle) \land (\tau \subseteq \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \leq t)$$

$$\langle \lambda x : \tau . e : t \rangle = \exists \alpha_1, \alpha_2. (\texttt{def } x : \tau \texttt{ in } \langle \langle e : \alpha_2 \rangle \rangle) \land (\alpha_1 \rightarrow \alpha_2 \leq t)$$

$$\langle \langle e_1 e_2 : t \rangle \rangle = \exists \alpha. \langle \langle e_1 : \alpha \rightarrow t \rangle \rangle \land \langle \langle e_2 : \alpha \rangle \rangle$$

---

$^1$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:

\[ C ::= (t \preceq t) \mid (\tau \preceq \alpha) \mid (x \preceq \alpha) \mid \text{def } x : \tau \text{ in } C \mid \exists \alpha. C \mid C \land C \]

\[ \langle \langle x : t \rangle \rangle = \exists \alpha. (x \preceq \alpha) \land (\alpha \preceq t) \]
\[ \langle \langle \lambda x.e : t \rangle \rangle = \exists \alpha_1, \alpha_2. (\text{def } x : \alpha_1 \text{ in } \langle \langle e : \alpha_2 \rangle \rangle) \land (\alpha_1 \preceq \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \preceq t) \]
\[ \langle \langle \lambda x.\tau.e : t \rangle \rangle = \exists \alpha_1, \alpha_2. (\text{def } x : \tau \text{ in } \langle \langle e : \alpha_2 \rangle \rangle) \land (\tau \preceq \alpha_1) \land (\alpha_1 \rightarrow \alpha_2 \preceq t) \]
\[ \langle \langle e_1 e_2 : t \rangle \rangle = \exists \alpha. (\langle e_1 : \alpha \rightarrow t \rangle \land \langle e_2 : \alpha \rangle) \]

Note that \[ \langle \langle \lambda x : ?.x : \text{Int} \rightarrow \text{Int} \rangle \rangle \text{ can be solved,} \]
whereas \[ \langle \langle \lambda x.x : ? \rightarrow ? \rangle \rangle \text{ cannot.} \]

---

\(^1\)Let constraints are omitted for the sake of simplicity
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:

$$\Gamma \vdash (x : \alpha) \leadsto \{ \tau[\bar{\alpha} := \bar{\beta}] : \alpha \}$$

$$\Gamma(x) = \forall \bar{\alpha}.\tau$$

$$\bar{\beta} \text{ FRESH}$$
We then **rewrite the structured constraints** to obtain a set $D$ of **type constraints**:

\[
\Gamma \vdash (x : \alpha) \leadsto \{ \tau[\vec{\alpha} := \vec{\beta}] : \alpha \} \quad \text{Γ}(x) = \forall \vec{\alpha}. \tau
\]

\[
\beta \text{ FRESH}
\]

\[
(\Gamma, x : \tau) \vdash C \leadsto D
\]

\[
\Gamma \vdash \text{def} x : \tau \text{ in } C \leadsto D
\]

\[
\Gamma \vdash C_1 \leadsto D_1 \quad \Gamma \vdash C_2 \leadsto D_2
\]

\[
\Gamma \vdash C_1 \land C_2 \leadsto D_1 \cup D_2
\]
Solving constraints

Everything is finally solved using standard unification:

1. we replace every occurrence 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 $\text{??} \rightarrow \text{??} \rightarrow \text{??} \preceq \text{Bool} \rightarrow \alpha$ will become $X_1 \rightarrow X_2 \rightarrow X_3 \preceq \text{Bool} \rightarrow \alpha$ and solving it will return the unifier $\theta: X_1 \mapsto \text{Bool}; X_2 \mapsto \beta; X_3 \mapsto \gamma; \alpha \mapsto (\beta \rightarrow \gamma)$.

The application of $e_1: (\text{Bool} \rightarrow \alpha) \rightarrow \alpha$ to $e_2: ?? \rightarrow ?? \rightarrow ??$ has thus type ?? → ??.
Solving constraints

Everything is finally solved using **standard unification**: 

1. we *replace every occurrence* 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

\[ ? \rightarrow ? \rightarrow ? \subseteq \text{Bool} \rightarrow \alpha \]
Solving constraints

Everything is finally solved using **standard unification**:

1. we *replace every occurrence* 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

```
? → ? → ? ⊑ Bool → α
```

will become

```
X_1 → X_2 → X_3 ⊑ Bool → α
```
Solving constraints

Everything is finally solved using **standard unification**:

1. we **replace every occurrence** 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

\[
? \rightarrow ? \rightarrow ? \sqsubseteq \text{Bool} \rightarrow \alpha
\]

will become

\[
X_1 \rightarrow X_2 \rightarrow X_3 \sqsubseteq \text{Bool} \rightarrow \alpha
\]

and solving it will return the unifier

\[
\theta : X_1 \mapsto \text{Bool}; X_2 \mapsto \beta; X_3 \mapsto \gamma; \alpha \mapsto (\beta \rightarrow \gamma)
\]
Solving constraints

Everything is finally solved using **standard unification**:

1. we *replace every occurrence* 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

$$ ? \rightarrow ? \rightarrow ? \sqsubseteq \text{Bool} \rightarrow \alpha $$

will become

$$ X_1 \rightarrow X_2 \rightarrow X_3 \sqsubseteq \text{Bool} \rightarrow \alpha $$

and solving it will return the unifier

$$ \theta : X_1 \mapsto \text{Bool}; X_2 \mapsto \beta; X_3 \mapsto \gamma; \alpha \mapsto (\beta \rightarrow \gamma) $$

The application of $e_1 : (\text{Bool} \rightarrow \alpha) \rightarrow \alpha$ to $e_2 : ? \rightarrow ? \rightarrow ?$ has thus type $? \rightarrow ?$
To summarize, given an expression $e$, and a constraint derivation $\mathcal{D}$ of $\Gamma \vdash \langle \langle e : t \rangle \rangle \leadsto D$, we can *compute a unifier* $\theta$ 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 $\mathcal{D} = \Gamma \vdash \langle \langle e : t \rangle \rangle \leadsto \{ (\tau \mathcal{D} \sqsubseteq \alpha) , (\alpha \mathcal{D} \leq t) \}$ and $\theta$ is a solution for $\{ (\tau \mathcal{D} \sqsubseteq \alpha) , (\alpha \mathcal{D} \leq t) \}$ then $\mathcal{D} ; \theta \vdash e \triangleright e \langle \alpha \theta \rangle$. Inference (and compilation) for this system is *sound*, *type-preserving* and *complete* w.r.t. the declarative system.
To summarize, given an expression $e$, and a constraint derivation $D$ of $\Gamma \vdash \langle\langle e : t \rangle\rangle \leadsto D$, we can compute a unifier $\theta$ satisfying $D$.

This derivation and the associated unifier can be used to compile $e$ in a straightforward way: to every materialization constraint introduced in $D$ corresponds a cast.

For instance, if $D = \Gamma; \vdash \langle\langle x : t \rangle\rangle \leadsto \{ (\tau \sqsubseteq \alpha), (\alpha \leq t) \}$ and $\theta$ is a solution for $\{ (\tau \sqsubseteq \alpha), (\alpha \leq t) \}$ then...
Compilation and Results

To summarize, given an expression $e$, and a constraint derivation $D$ of $\Gamma \vdash \langle\langle e : t \rangle\rangle \leadsto D$, we can compute a unifier $\theta$ satisfying $D$.

This derivation and the associated unifier can be used to compile $e$ in a straightforward way: to every materialization constraint introduced in $D$ corresponds a cast.

For instance if $D = \Gamma; \vdash \langle\langle x : t \rangle\rangle \leadsto \{(\tau \sqsubseteq \alpha), (\alpha \preceq t)\}$ and $\theta$ is a solution for $\{(\tau \sqsubseteq \alpha), (\alpha \preceq t)\}$ then

$$D; \theta \vdash x \xrightarrow{\text{compiles}} x\langle\alpha\theta\rangle$$

Inference (and compilation) for this system is sound, type-preserving and complete w.r.t. the declarative system.
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 \( \{ (\alpha \leq t_1), (\alpha \leq t_2) \} \) 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 $\{(\alpha \leq t_1), (\alpha \leq t_2)\}$ we have to compute greatest lower bounds.

For example,

$$\text{fun } x \rightarrow \text{if } (\text{fst } x) \text{ then } (1 + \text{snd } x) \text{ else } x$$

should be of type $(\text{Bool} \times \text{Int}) \rightarrow (\text{Int} | (\text{Bool} \times \text{Int}))$
The types become:

\[
\begin{align*}
\text{StaticTypes} & \quad T ::= \text{Int} \mid \text{Bool} \mid T \to T \mid T \lor T \mid \neg T \mid \text{Any} \mid \alpha \\
\text{GradualTypes} & \quad \tau ::= \text{Int} \mid \text{Bool} \mid \tau \to \tau \mid \alpha \mid ??? \\
\text{Schemas} & \quad \sigma ::= T \mid \forall \alpha. \sigma
\end{align*}
\]

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).

\[
\left\{ (\alpha \leq t_1), (\alpha \leq t_2) \right\} \leadsto \left\{ (\alpha \preceq t_1 \land t_2) \right\}
\]
Part 3: Adding Set-Theoretic Types

The types become:

**Static Types**
\[ T ::= \text{Int} \mid \text{Bool} \mid T \to T \mid T \lor T \mid \neg T \mid \text{Any} \mid \alpha \]

**Gradual Types**
\[ \tau ::= \text{Int} \mid \text{Bool} \mid \tau \to \tau \mid \alpha \mid ? \]

**Schemas**
\[ \sigma ::= T \mid \forall \alpha. \sigma \]

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).

\[
\{ (\alpha \leq t_1), (\alpha \leq t_2) \} \leadsto \{ (\alpha \leq t_1 \land t_2) \}
\]

Soundness still holds for the inference algorithm, but completeness no longer holds.
Outline

15 Main ideas

16 Formal system

17 Algorithmic Aspects

18 Criteria for Gradual Typing

19 Implementation issues

20 References
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!**