Theory and practice of XML processing programming languages

Giuseppe Castagna

CNRS
Université Paris 7 - Denis Diderot

MPRI Lectures on Theory of Subtyping

Outline of the lecture

1. **XML Programming in** $\mathbb{C}$Duce

2. **Theoretical Foundations**

3. **Polymorphic Subtyping**

4. **Polymorphic Language**

Outline of the lecture

1. **XML Programming in ℂDuce**
    - XML Regular Expression Types and Patterns
    - XML Programming in ℂDuce
    - Tools on top of ℂDuce
2. **Theoretical Foundations**
3. **Polymorphic Subtyping**
4. **Polymorphic Language**

Outline of the lecture

1. **XML Programming in ℂDuce**
   - XML Regular Expression Types and Patterns
   - XML Programming in ℂDuce
   - Tools on top of ℂDuce
2. **Theoretical Foundations**
   - Semantic subtyping
   - Subtyping algorithms
   - ℂDuce functional core
3. **Polymorphic Subtyping**

4. **Polymorphic Language**

Outline of the lecture

1. **XML Programming in ℂDuce**
   - XML Regular Expression Types and Patterns
   - XML Programming in ℂDuce
   - Tools on top of ℂDuce
2. **Theoretical Foundations**
   - Semantic subtyping
   - Subtyping algorithms
   - ℂDuce functional core
3. **Polymorphic Subtyping**
   - Current status
   - Semantic solution
   - Subtyping algorithm
4. **Polymorphic Language**

Outline of the lecture

1. **XML Programming in ℂDuce**
   - XML Regular Expression Types and Patterns
   - XML Programming in ℂDuce
   - Tools on top of ℂDuce
2. **Theoretical Foundations**
   - Semantic subtyping
   - Subtyping algorithms
   - ℂDuce functional core
3. **Polymorphic Subtyping**
   - Current status
   - Semantic solution
   - Subtyping algorithm
4. **Polymorphic Language**
   - Motivating example
   - Formal setting
   - Explicit substitutions
   - Inference System
   - Efficient implementation

PART 1: XML PROGRAMMING IN ℂDUCE

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .

Programming with XML

- Level 0: textual representation of XML documents
    - AWK, sed, Perl
- Level 1: abstract view provided by a parser
    - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
    - XSLT, XPath

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl
- Level 1: abstract view provided by a parser
  - SAX, DOM, ...
- Level 2: untyped XML-specific languages
  - XSLT, XPath
- Level 3: XML types taken seriously (aka: related work)
  - XDuce, Xtatic
  - XQuery
  - C_ω (Microsoft)
  - ...

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
  - XSLT, XPath
- Level 3: XML types taken seriously (aka: related work)
  - XDuce, Xtatic
  - XQuery
  - C_ω (Microsoft)
  - . . .

Presentation of ℂDuce

Presentation of ℂDuce

Features:

- Oriented to XML processing
- Type centric
- General-purpose features
- Very efficient

Presentation of ℂDuce

Features:

- Oriented to XML processing
- Type centric
- General-purpose features
- Very efficient

Intended use:

- Small "adapters" between different XML applications
- Larger applications that use XML
- Web development
- Web services

Presentation of ℂDuce

Features:

- Oriented to XML processing
- Type centric
- General-purpose features
- Very efficient

Intended use:

- Small “adapters” between different XML applications
- Larger applications that use XML
- Web development
- Web services

Status:

- Public release available (0.5.3) in all major Linux distributions.
- Integration with standards
  - Internally: Unicode, XML, Namespaces, XML Schema
  - Externally: DTD, WSDL
- Some tools: graphical queries, code embedding (*à la* php)

Presentation of ℂDuce

Features:

- Oriented to XML processing
- Type centric
- General-purpose features
- Very efficient

Intended use:

- Small "adapters" between different XML applications
- Larger applications that use XML
- Web development
- Web services

Status:

- Public release available (0.5.3) in all major Linux distributions.
- Integration with standards
  - Internally: Unicode, XML, Namespaces, XML Schema
  - Externally: DTD, WSDL
- Some tools: graphical queries, code embedding (*à la* php)

Used both for teaching and in production code.

Types, Types, Types!!!

Types are pervasive in ℂDuce:

Types, Types, Types!!!

Types are pervasive in ℂDuce:

- **Static validation**
    - E.g.: does the transformation produce valid XHTML ?

Types, Types, Types!!!

Types are pervasive in ℂDuce:

- **Static validation**
  - E.g.: does the transformation produce valid XHTML ?
- **Type-driven programming semantics**
  - At the basis of the definition of patterns
  - Dynamic dispatch
  - Overloaded functions

Types, Types, Types!!!

Types are pervasive in ℂDuce:

- **Static validation**
  - E.g.: does the transformation produce valid XHTML ?
- **Type-driven programming semantics**
  - At the basis of the definition of patterns
  - Dynamic dispatch
  - Overloaded functions
- **Type-driven compilation**
  - Optimizations made possible by static types
  - Avoids unnecessary and redundant tests at runtime
  - Allows a more declarative style

Regular Expression
Types and Patterns for XML

Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```
let x = fst(e) in
let y = snd(e) in (y,x)
```

Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write

```
let (x,y) = e in (y,x)
```

Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write

```
let (x,y) = e in (y,x)
```

which is syntactic sugar for

```
match e with (x,y) -> (y,x)
```

Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write

```
let (x,y) = e in (y,x)
```

which is syntactic sugar for

```
match e with (x,y) -> (y,x)
```

"`match`" is more interesting than "`let`", since it can test several "`|`"-separated patterns.

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil
```

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | `nil

fun length (x:(List,Int)) : Int =
  match x with
   | (`nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables,

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards,

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. $(s\,,t)$ instead of $s \times t$)

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. (s, t) **instead of** $s \times t$**)**

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | `nil

fun length (x:(List,Int)) : Int =
  match x with
   | (`nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. (s,t) **instead of** $s \times t$**)**
2. **use values to denote singleton types**
   (e.g. `` `nil `` **in the list type);**

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | `nil

fun length (x:(List,Int)) : Int =
  match x with
   | (`nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. (s,t) **instead of** $s \times t$**)**
2. **use values to denote singleton types**
   (e.g. `` `nil `` **in the list type);**

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. (s,t) **instead of** $s \times t$**)**
2. **use values to denote singleton types**
   (e.g. `‘nil` **in the list type);**
3. **consider the wildcard “_” as synonym of** `Any`

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | `nil

fun length (x:(List,Int)) : Int =
  match x with
   | (`nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. **use for types the same constructors as for values**
   (e.g. (s,t) instead of $s \times t$)
2. **use values to denote singleton types**
   (e.g. `` `nil `` in the list type);
3. **consider the wildcard "_" as synonym of `Any`**

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

~~So patterns are values with capture variables, wildcards, constants.~~

Key idea behind regular patterns

Patterns are types with capture variables

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

~~So patterns are values with capture variables, wildcards, constants.~~

Key idea behind regular patterns

Patterns are types with capture variables

Example: tail-recursive version of `length` for lists:

```
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
  match x with
   | (‘nil , n) -> n
   | ((_,t), n) -> length(t,n+1)
```

~~So patterns are values with capture variables, wildcards, constants.~~

Key idea behind regular patterns

Patterns are types with capture variables

Define types: patterns come for free.

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\):

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

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

$t = \{v \mid v \text{ value of type } t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

$t \;= \{v \mid v \;\text{ value of type } t\}$ **and** $\wr p \int = \{v \mid v \text{ matches pattern } p\}$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (`|`), intersections (`&`) and differences (`\`):

- **Boolean operators are needed to type pattern matching:**

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

$t = \{v \mid v \text{ value of type } t\}$ **and** $\wr p \int = \{v \mid v \text{ matches pattern } p\}$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \llparenthesis p_1 \rrparenthesis$ (where $e : t$);

$$t = \{v \mid v \text{ value of type } t\} \quad \textbf{and} \quad \llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \{\!\!\{ p_1 \}\!\!\}$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \{\!\!\{ p_1 \}\!\!\}) \& \{\!\!\{ p_2 \}\!\!\}$;

$t = \{v \mid v$ value of type $t\}$ **and** $\{\!\!\{p\}\!\!\} = \{v \mid v$ matches pattern $p\}$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \wr p_1 \int$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \wr p_1 \int) \& \wr p_2 \int$;
  - The type of the match is $t_1 | t_2$.

$t = \{v \mid v$ value of type $t\}$ **and** $\wr p \int = \{v \mid v$ matches pattern $p\}$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \{\!\!\{ p_1 \}\!\!\}$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \{\!\!\{ p_1 \}\!\!\}) \& \{\!\!\{ p_2 \}\!\!\}$;
  - The type of the match is $t_1 \mid t_2$.

$t = \{v \mid v$ value of type $t\}$ **and** $\{\!\!\{ p \}\!\!\} = \{v \mid v$ matches pattern $p\}$

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \lbag p_1 \rbag$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \lbag p_1 \rbag) \& \lbag p_2 \rbag$;
  - The type of the match is $t_1 | t_2$.

- **Boolean type constructors are useful for programming:**

```
map catalogue with
    x :: (Car & (Guaranteed|(Any\Used)) -> x
```

  Select in *catalogue* all cars that if used then are guaranteed.

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \llbracket p_1 \rrbracket$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \llbracket p_1 \rrbracket) \& \llbracket p_2 \rrbracket$;
  - The type of the match is $t_1 | t_2$.

- **Boolean type constructors are useful for programming:**

```
map catalogue with
    x :: (Car & (Guaranteed|(Any\Used)) -> x
```

  Select in *catalogue* all cars that if used then are guaranteed.

Roadmap to extend it to XML:

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \wr p_1 \int$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \wr p_1 \int) \& \wr p_2 \int$;
  - The type of the match is $t_1 \mid t_2$.

- **Boolean type constructors are useful for programming:**

  ```
  map catalogue with
      x :: (Car & (Guaranteed|(Any\Used)) -> x
  ```

  Select in *catalogue* all cars that if used then are guaranteed.

Roadmap to extend it to XML:

1. **Define types for XML documents,**

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \{\!\!\{ p_1 \}\!\!\}$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \{\!\!\{ p_1 \}\!\!\}) \& \{\!\!\{ p_2 \}\!\!\}$;
  - The type of the match is $t_1 | t_2$.

- **Boolean type constructors are useful for programming:**

  ```
  map catalogue with
      x :: (Car & (Guaranteed|(Any\Used)) -> x
  ```

  Select in *catalogue* all cars that if used then are guaranteed.

Roadmap to extend it to XML:

1. **Define types for XML documents,**
2. **Add boolean type constructors,**

Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\\):

- **Boolean operators are needed to type pattern matching:**

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

  - To infer the type t_1 of e_1 we need $t \& \llparenthesis p_1 \rrparenthesis$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \llparenthesis p_1 \rrparenthesis) \& \llparenthesis p_2 \rrparenthesis$;
  - The type of the match is $t_1 | t_2$.

- **Boolean type constructors are useful for programming:**

```
map catalogue with
    x :: (Car & (Guaranteed|(Any\Used)) -> x
```

  Select in *catalogue* all cars that if used then are guaranteed.

Roadmap to extend it to XML:

1. **Define types for XML documents,**
2. **Add boolean type constructors,**
3. **Define patterns as types with capture variables**

XML syntax

```
<bib>
  <book year="1997">
    <title>  Object-Oriented Programming </title>
    <author>
      <last>  Castagna </last>
      <first>  Giuseppe </first>
    </author>
    <price>  56 </price>
     Bikhäuser
  </book>
  <book year="2000">
    <title>  Regexp Types for XML </title>
    <editor>
      <last>  Hosoya </last>
      <first>  Haruo </first>
    </editor>
     UoT
  </book>
</bib>
```

XML syntax

```
<bib>[
  <book year="1997">[
    <title>['Object-Oriented Programming']
    <author>[
      <last>['Castagna']
      <first>['Giuseppe']
    ]
    <price>['56']
    'Bikhäuser'
  ]
  <book year="2000">[
    <title>['Regexp Types for XML']
    <editor>
      <last>['Hosoya']
      <first>['Haruo']
    ]
    'UoT'
  ]
]
```

XML syntax

```
type Bib = <bib>[
             <book year="1997">[
               <title>['Object-Oriented Programming']
               <author>[
                 <last>['Castagna']
                 <first>['Giuseppe']
               ]
               <price>['56']
               'Bikhäuser'
             ]
             <book year="2000">[
               <title>['Regexp Types for XML']
               <editor>
                 <last>['Hosoya']
                 <first>['Haruo']
               ]
               'UoT'
             ]
           ]
```

XML syntax

```
type Bib = <bib>[
             <book year=String>[
               <title>
               <author>[
                 <last>[PCDATA]
                 <first>[PCDATA]
               ]
               <price>[PCDATA]
               PCDATA
             ]
             <book year=String>[
               <title>[PCDATA]
               <editor>[
                 <last>[PCDATA]
                 <first>[PCDATA]
               ]
               PCDATA
             ]
           ]
```

`String = [PCDATA] = [Char*]`

XML syntax

```
type Bib = <bib>[Book Book]
type Book = <book year=String>[
                        Title
                        (Author  | Editor )
                        Price?
                        PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                        Title
                        (Author+ | Editor+)
                        Price?
                        PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                       Title
                       (Author+ | Editor+)
                       Price?
                       PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]                              Kleene star
type Book = <book year=String>[
                        Title
                        (Author+ | Editor+)
                        Price?
                        PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[          attribute types
                       Title
                       (Author+ | Editor+)
                       Price?
                       PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                        Title                 nested elements
                        (Author+ | Editor+)
                        Price?
                        PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                       Title
                       (Author+ | Editor+)        unions
                       Price?
                       PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                       Title
                       (Author+ | Editor+)
                       Price?                 optional elems
                       PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                       Title
                       (Author+ | Editor+)
                       Price?
                       PCDATA]           mixed content
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

XML syntax

```
type Bib = <bib>[Book*]
type Book = <book year=String>[
                       Title
                       (Author+ | Editor+)
                       Price?
                       PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
```

This and: singletons, intersections, differences, `Empty`, and `Any`.

Patterns

Patterns = Types + Capture variables

Patterns

Patterns = Types + Capture variables

```
type Bib = <bib>[Book*]
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
    <bib>[x::Book*]
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
    <bib>[x::Book*]
```

The pattern binds `x` to the *sequence* of all books in the bibliography

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
match bibs with
    <bib>[x::Book*] -> x
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
match bibs with
    <bib>[x::Book*] -> x
```

Returns the content of `bibs`.

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
<bib>[( x::<book year="2005">_ | y::_ )*]
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
<bib>[( x::<book year="2005">_ | y::_ )*]
```

Binds `x` to the sequence of all this year's books, and `y` to all the other books.

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
match bibs with
    <bib>[( x::<book year="2005">_ | y::_ )*] -> x@y
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
```

PATTERNS

```
match bibs with
    <bib>[( x::<book year="2005">_ | y::_ )*] -> x@y
```

Returns the concatenation (i.e., "`@`") of the two captured sequences

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

PATTERNS

```
<bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*]
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

PATTERNS

```
<bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*]
```

Binds `x` to the *sequence* of books published in 1990 from publishers others than “ACM” and discards all the others.

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

PATTERNS

```
match bibs with
  <bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*] -> x
```

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

PATTERNS

```
match bibs with
  <bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*] -> x
```

Returns all the captured books

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

PATTERNS

```
match bibs with
  <bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*] -> x
```

Returns all the captured books

Exact type inference:

E.g.: if we match the pattern `[(x::Int|_)*]` against an expression of type `[Int* String Int]` the type deduced for x is `[Int+]`

Patterns

Patterns = Types + Capture variables

TYPES

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

PATTERNS

```
match bibs with
  <bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*] -> x
```

Returns all the captured books

Exact type inference:

E.g.: if we match the pattern `[(x::Int|_)*]` against an expression of type `[Int* String Int]` the type deduced for x is `[Int+]`

XML-programming in CDuce

Functions: basic usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions: basic usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Extract subsequences (union polymorphism)

```
fun (Invited|Talk -> [Author+])
    <_>[ Title x::Author* ] -> x
```

Functions: basic usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Extract subsequences (union polymorphism)

```
fun (Invited|Talk -> [Author+])
   <_>[ Title x::Author* ] -> x
```

Extract subsequences of non-consecutive elements:

```
fun ([(Invited|Talk|Event)*] -> ([Invited*], [Talk*]))
    [ (i::Invited | t::Talk | _)* ] -> (i,t)
```

Functions: basic usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Extract subsequences (union polymorphism)

```
fun (Invited|Talk -> [Author+])
    <_>[ Title x::Author* ] -> x
```

Extract subsequences of non-consecutive elements:

```
fun ([(Invited|Talk|Event)*] -> ([Invited*], [Talk*]))
    [ (i::Invited | t::Talk | _)* ] -> (i,t)
```

Perl-like string processing (`String` = `[Char*]`)

```
fun parse_email (String -> (String,String))
    | [ local::_* '@' domain::_* ] -> (local,domain)
    | _ -> raise "Invalid email address"
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Higher-order, overloading, subtyping provide name/code sharing...

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Higher-order, overloading, subtyping provide name/code sharing...

```
let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t a _* ] -> <invited>[ t a ]
| <talk>[ t a _* ] -> <talk>[ t a ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Higher-order, overloading, subtyping provide name/code sharing...

```
let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t a _* ] -> <invited>[ t a ]
| <talk>[ t a _* ] -> <talk>[ t a ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Higher-order, overloading, subtyping provide name/code sharing...

```
let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t a _* ] -> <invited>[ t a ]
| <talk>[ t a _* ] -> <talk>[ t a ]
```

Even more compact: replace the last two branches with:

```
<(k)>[ t a _* ] -> <(k)>[ t a ]
```

Functions: advanced usage

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
```

Functions can be **higher-order** and **overloaded**

```
let patch_program
(p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program]
   = xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Higher-order, overloading, subtyping provide name/code sharing...

```
let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t a _* ] -> <invited>[ t a ]
| <talk>[ t a _* ] -> <talk>[ t a ]
```

Even more compact: replace the last two branches with:

```
<(k)>[ t a _* ] -> <(k)>[ t a ]
```

. . . it is all syntactic sugar!

Types

$$t \ ::= \ \texttt{Int} \ \mid \ v \ \mid \ (t, t) \ \mid \ t \rightarrow t \ \mid \ t \vee t \ \mid \ t \wedge t \ \mid \ \neg t \ \mid \ \texttt{Any}$$

. . . it is all syntactic sugar!

Types

$$t \;::=\; \texttt{Int} \mid v \mid (t,t) \mid t{\rightarrow}t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Any}$$

Patterns

$$p \;::=\; t \mid x \mid (p,p) \mid p \vee p \mid p \wedge p$$

. . . it is all syntactic sugar!

Types

$$t \ ::= \ \texttt{Int} \ | \ v \ | \ (t,t) \ | \ t{\rightarrow}t \ | \ t \vee t \ | \ t \wedge t \ | \ \neg t \ | \ \texttt{Any}$$

Patterns

$$p \ ::= \ t \ | \ x \ | \ (p,p) \ | \ p \vee p \ | \ p \wedge p$$

Example:

```
type Book = <book>[Title (Author+|Editor+) Price?]
```

…it is all syntactic sugar!

Types

$$t \ ::= \ \texttt{Int} \mid v \mid (t,t) \mid t{\rightarrow}t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Any}$$

Patterns

$$p \ ::= \ t \mid x \mid (p,p) \mid p \vee p \mid p \wedge p$$

Example:

```
type Book = <book>[Title (Author+|Editor+) Price?]
```

encoded as

$$\begin{aligned} Book &= (\,\text{`}book, (\,Title, X \vee Y\,)) \\ X &= (Author, X \vee (Price, \text{`}nil) \vee \text{`}nil) \\ Y &= (Editor, Y \vee (Price, \text{`}nil) \vee \text{`}nil) \end{aligned}$$

Some reasons to consider regular expression types and patterns

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact**

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact ($\neq$ simple)**

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact (≠ simple)**
- **Nine practical reasons:**

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact ($\neq$ simple)**
- **Nine practical reasons:**
  1. Classic usage
  2. Informative error messages
  3. Error mining
  4. Efficient execution
  5. Compact programs
  6. Logical optimisation of pattern-based queries
  7. Pattern matches as building blocks for iterators
  8. Type/pattern-based data pruning for memory usage optimisation
  9. Type-based query optimisation

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact ($\neq$ simple)**
- **Nine practical reasons:**
  1. Classic usage
  2. Informative error messages ←
  3. Error mining
  4. Efficient execution←
  5. Compact programs
  6. Logical optimisation of pattern-based queries
  7. Pattern matches as building blocks for iterators
  8. Type/pattern-based data pruning for memory usage optimisation
  9. Type-based query optimisation

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
  fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
```

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[(t::Title | a::Author | _)+] in books
 where int_of(y) = year
```

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[(t::Title | a::Author | _)+] in books
 where int_of(y) = year
```

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[(t::Title | a::Author | _)+] in books
 where int_of(y) = year
```

Returns the following error message:

```
Error at chars 81-83:
   select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price?  ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[ ] ]
```

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[(t::Title | a::Author | _)+] in books
 where int_of(y) = year
```

Returns the following error message:

```
Error at chars 81-83:
   select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price?  ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[ ] ]
```

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[(t::Title | a::Author | _)+] in books
 where int_of(y) = year
```

Returns the following error message:

```
Error at chars 81-83:
   select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price?  ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[ ] ]
```

2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```

List of books of a given year, stripped of the Editors and Price

```
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
 select <book year=y>(t@a) from
   <book year=y>[ t::Title   a::Author+  _* ] in books
 where int_of(y) = year
```

Returns the following error message:

```
Error at chars 81-83:
   select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price?  ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[ ] ]
```

5. Efficient execution

Use static type information to perform an optimal set of tests

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]
```

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
```

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with  A  -> 1 | B -> 0
```

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_-> 1 | _ -> 0
```

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_-> 1 | _ -> 0
```

- No backtracking.

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_-> 1 | _ -> 0
```

- No backtracking.
- Whole parts of the matched data are not checked

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_-> 1 | _ -> 0
```

- No backtracking.
- Whole parts of the matched data are not checked

Computing the optimal solution requires to fully exploit intersections and differences of types

5. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don't test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with   A  -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_-> 1 | _ -> 0
```

- No backtracking.
- Whole parts of the matched data are not checked

Specific kind of push-down tree automata

On top of ℂDuce

- Full integration with OCaml
- Embedding of ℂDuce code in XML documents
- Graphical queries
- Security (control flow analysis)
- Web-services

- Full integration with OCaml
- Embedding of ℂDuce code in XML documents
- Graphical queries
- Security (control flow analysis)
- Web-services

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...
- Implement complex algorithms

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...
- Implement complex algorithms

An OCaml application that requires ℂDuce code

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...
- Implement complex algorithms

An OCaml application that requires ℂDuce code

- ℂDuce used as an XML input/output/transformation layer

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...
- Implement complex algorithms

An OCaml application that requires ℂDuce code

- ℂDuce used as an XML input/output/transformation layer
  - Configuration files
  - XML serialization of datas
  - XHTML code production

ℂDuce↔OCaml Integration

A ℂDuce application that requires OCaml code

- Reuse existing librairies
  - Abstract data structures : hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries : databases, networks, ...
- Implement complex algorithms

An OCaml application that requires ℂDuce code

- ℂDuce used as an XML input/output/transformation layer
  - Configuration files
  - XML serialization of datas
  - XHTML code production

Need to seamlessly call OCaml code in ℂDuce and viceversa

Main Challenges

Main Challenges

1. **Seamless integration:**

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions
2. **Type safety:**

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions
2. **Type safety:**
   No explicit type cast in programs:

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions
2. **Type safety:**
   No explicit type cast in programs:
   the standard type-checkers ensure type safety

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions
2. **Type safety:**
   No explicit type cast in programs:
   the standard type-checkers ensure type safety

What we need:

A mapping between OCaml and ℂDuce types and values

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

- ⊕ **ℂDuce uses (semantic) subtyping; OCaml does not**

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

⊕ **ℂDuce uses (semantic) subtyping; OCaml does not**
If we translate ℂDuce types into OCaml ones :
- soundness requires the translation to be monotone;
- no subtyping in Ocaml implies a constant translation;

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

⊕ **ℂDuce uses (semantic) subtyping; OCaml does not**
If we translate ℂDuce types into OCaml ones :
- soundness requires the translation to be monotone;
- no subtyping in Ocaml implies a constant translation;
⇒ *ℂDuce typing would be lost.*

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

- ⊕ **ℂDuce uses (semantic) subtyping; OCaml does not**
  If we translate ℂDuce types into OCaml ones :
  - soundness requires the translation to be monotone;
  - no subtyping in Ocaml implies a constant translation;
  ⇒ *ℂDuce typing would be lost.*

- ⊕ **ℂDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

- **ℂDuce uses (semantic) subtyping; OCaml does not**
  If we translate ℂDuce types into OCaml ones :
  - soundness requires the translation to be monotone;
  - no subtyping in Ocaml implies a constant translation;
  ⇒ *ℂDuce typing would be lost.*

- **ℂDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
  ⇒ *OCaml types are not enough to translate ℂDuce types.*

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

⊕ **ℂDuce uses (semantic) subtyping; OCaml does not**
If we translate ℂDuce types into OCaml ones :
- soundness requires the translation to be monotone;
- no subtyping in Ocaml implies a constant translation;
⇒ *ℂDuce typing would be lost.*

⊕ **ℂDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
⇒ *OCaml types are not enough to translate ℂDuce types.*

⊖ **OCaml supports type polymorphism; ℂDuce does not.**

How to integrate the two type systems?

The translation can go just one way: OCaml → ℂDuce

⊕ **ℂDuce uses (semantic) subtyping; OCaml does not**
If we translate ℂDuce types into OCaml ones :
- soundness requires the translation to be monotone;
- no subtyping in Ocaml implies a constant translation;
⇒ *ℂDuce typing would be lost.*

⊕ **ℂDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
⇒ *OCaml types are not enough to translate ℂDuce types.*

⊖ **OCaml supports type polymorphism; ℂDuce does not.**
⇒ *Polymorphic OCaml libraries/functions must be first instantied to be used in ℂDuce*

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to ℂDuce types.

In practice

① Define a mapping $\mathbb{T}$ from OCaml types to ℂDuce types.

t (*OCaml*)	$\mathbb{T}(t)$ (*CDuce*)
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \to t_2$	$\mathbb{T}(t_1) \to \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(`A_1, \mathbb{T}(t_1)) \mid \ldots \mid (`A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

In practice

① Define a mapping $\mathbb{T}$ from OCaml types to ℂDuce types.

t (OCaml)	$\mathbb{T}(t)$ (CDuce)
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \rightarrow t_2$	$\mathbb{T}(t_1) \rightarrow \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(\text{`}A_1, \mathbb{T}(t_1)) \mid \ldots \mid (\text{`}A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

② Define a retraction pair between OCaml and ℂDuce values.

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to ℂDuce types.

t $(OCaml)$	$\mathbb{T}(t)$ $(CDuce)$
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \rightarrow t_2$	$\mathbb{T}(t_1) \rightarrow \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(\text{`}A_1, \mathbb{T}(t_1)) \mid \ldots \mid (\text{`}A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

2. Define a retraction pair between OCaml and ℂDuce values.

`ocaml2cduce`: $t \rightarrow \mathbb{T}(t)$

`cduce2ocaml`: $\mathbb{T}(t) \rightarrow t$

Calling OCaml from ℂDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

Calling OCaml from ℂDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

The ℂDuce compiler checks type soundness and then

Calling OCaml from ℂDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

The ℂDuce compiler checks type soundness and then

- applies `cduce2ocaml` to the arguments of the call

Calling OCaml from ℂDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

The ℂDuce compiler checks type soundness and then

- applies `cduce2ocaml` to the arguments of the call
- calls the OCaml function

Calling OCaml from ℂDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

The ℂDuce compiler checks type soundness and then

- applies `cduce2ocaml` to the arguments of the call
- calls the OCaml function
- applies `ocaml2cduce` to the result of the call

Calling OCaml from ℂDuce

Easy

Use `M.f` to call the function `f` exported by the OCaml module `M`

The ℂDuce compiler checks type soundness and then
- applies `cduce2ocaml` to the arguments of the call
- calls the OCaml function
- applies `ocaml2cduce` to the result of the call

Example: use ocaml-mysql library in ℂDuce

```
let db = Mysql.connect Mysql.defaults;;

match Mysql.list_dbs db ‘None [] with
| (‘Some,l) -> print [ 'Databases: ' !(string_of l) '\ n' ]
| ‘None -> [];;
```

Calling ℂDuce from OCaml

Needs little work

Compile a ℂDuce module as an OCaml binary module by providing a OCaml (`.mli`) interface. Use it as a standard Ocaml module.

Calling ℂDuce from OCaml

Needs little work

Compile a ℂDuce module as an OCaml binary module by providing a OCaml (`.mli`) interface. Use it as a standard Ocaml module.

The ℂDuce compiler:

Calling ℂDuce from OCaml

Needs little work

Compile a ℂDuce module as an OCaml binary module by providing a OCaml (`.mli`) interface. Use it as a standard Ocaml module.

The ℂDuce compiler:

1. Checks that if `val` $f{:}t$ in the `.mli` file, then the ℂDuce type of f is a *subtype* of $\mathbb{T}(t)$

Calling ℂDuce from OCaml

Needs little work

Compile a ℂDuce module as an OCaml binary module by providing a OCaml (`.mli`) interface. Use it as a standard Ocaml module.

The ℂDuce compiler:

1. Checks that if `val` $f{:}t$ in the `.mli` file, then the ℂDuce type of f is a *subtype* of $\mathbb{T}(t)$
2. Produces the OCaml glue code to export CDuce values as OCaml ones and bind OCaml values in the CDuce module.

Calling ℂDuce from OCaml

Needs little work

Compile a ℂDuce module as an OCaml binary module by providing a OCaml (`.mli`) interface. Use it as a standard Ocaml module.

The ℂDuce compiler:

1. Checks that if `val` $f{:}t$ in the `.mli` file, then the ℂDuce type of f is a *subtype* of $\mathbb{T}(t)$
2. Produces the OCaml glue code to export CDuce values as OCaml ones and bind OCaml values in the CDuce module.

Example: use ℂDuce to compute a factorial:

```
(* File cdnum.mli: *)
val fact: Big_int.big_int -> Big_int.big_int

(* File cdnum.cd: *)
let aux ((Int,Int) -> Int)
| (x, 0 | 1) -> x
| (x, n) -> aux (x * n, n - 1)

let fact (x : Int) : Int = aux(1,x)
```

PART 2: THEORETICAL FOUNDATIONS

Goal

The goal is to show how to take your favourite type constructors

Goal

The goal is to show how to take your favourite type constructors

$$\times,\ \to,\ \{\ldots\},\ \mathtt{chan}(),\ \ldots$$

Goal

The goal is to show how to take your favourite type constructors

$$\times,\ \rightarrow,\ \{\ldots\},\ \mathtt{chan}(),\ \ldots$$

and add boolean combinators:

$$\vee,\quad \wedge,\quad \neg$$

so that they behave set-theoretically w.r.t. $\leq$

Goal

The goal is to show how to take your favourite type constructors

$$\times,\ \rightarrow,\ \{\ldots\},\ \mathtt{chan}(),\ \ldots$$

and add boolean combinators:

$$\vee,\quad \wedge,\quad \neg$$

so that they behave set-theoretically w.r.t. $\leq$

WHY?

Short answer: YOU JUST SAW IT!

Recap:

- to encode XML types
- to define XML patterns
- to precisely type pattern matching

In details

$$t \ ::= \ B \ \mid \ t{\times}t \ \mid \ t{\rightarrow}t$$

In details

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

In details

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

- Handling subtyping without combinators is easy: constructors do not mix,

In details

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

- Handling subtyping without combinators is easy: constructors do not mix, e.g. :

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

In details

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

- Handling subtyping without combinators is easy: constructors do not mix, e.g. :

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

- With combinators is much harder: combinators distribute over constructors,

In details

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

- Handling subtyping without combinators is easy: constructors do not mix, e.g. :

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

- With combinators is much harder: combinators distribute over constructors, e.g.

$$(s_1{\vee}s_2) \to t \quad \substack{\gneqq \\ \lneqq} \quad (s_1 \to t){\wedge}(s_2 \to t)$$

In details

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

- Handling subtyping without combinators is easy: constructors do not mix, e.g. :

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

- With combinators is much harder: combinators distribute over constructors, e.g.

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

MAIN IDEA

Instead of defining the subtyping relation so that it conforms to the semantic of types, define the semantics of types and derive the subtyping relation.

In details

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

- **Not a particularly new idea**. Many attempts (e.g. Aiken&Wimmers, Damm,. . . , Hosoya&Pierce).

MAIN IDEA

Instead of defining the subtyping relation so that it conforms to the semantic of types, define the semantics of types and derive the subtyping relation.

In details

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

- **Not a particularly new idea**. Many attempts (e.g. Aiken&Wimmers, Damm,. . . , Hosoya&Pierce).
- **None fully satisfactory.** (no negation, or no function types, or restrictions on unions and intersections, . . .)

MAIN IDEA

Instead of defining the subtyping relation so that it conforms to the semantic of types, define the semantics of types and derive the subtyping relation.

In details

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

- **Not a particularly new idea**. Many attempts (e.g. Aiken&Wimmers, Damm,. . . , Hosoya&Pierce).
- **None fully satisfactory.** (no negation, or no function types, or restrictions on unions and intersections, . . .)
- **Starting point of what follows: the approach of Hosoya&Pierce.**

MAIN IDEA

Instead of defining the subtyping relation so that it conforms to the semantic of types, define the semantics of types and derive the subtyping relation.

Semantic subtyping

Semantic subtyping

Semantic subtyping

1. Define a set-theoretic semantics of the types:

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

Semantic subtyping

1. Define a set-theoretic semantics of the types:

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

2. Define the subtyping relation as follows:

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

Semantic subtyping

1. Define a set-theoretic semantics of the types:

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

2. Define the subtyping relation as follows:

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

KEY OBSERVATION 1:

The *model of types* may be independent from a *model of terms*

Semantic subtyping

1. Define a set-theoretic semantics of the types:

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

2. Define the subtyping relation as follows:

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

KEY OBSERVATION 1:

The *model of types* may be independent from a *model of terms*

Hosoya and Pierce use the model of values:

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

Ok because the only values of XDuce are XML documents (no first-class functions)

Step 1 : Model

Define when $[\![\ ]\!] : \textbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

Step 1 : Model

Define when $[\![\ ]\!] : \textbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\begin{array}{lcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\
[\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\neg t]\!] & = & \mathcal{D} \setminus [\![t]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset \\
[\![\mathbb{1}]\!] & = & \mathcal{D}
\end{array}$$

Step 1 : Model

Define when $[\![\ ]\!] : \textbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\begin{array}{lcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\
[\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\neg t]\!] & = & \mathcal{D} \backslash [\![t]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset \\
[\![\mathbb{1}]\!] & = & \mathcal{D}
\end{array}$$

- Hard for constructors:

$$\begin{array}{lcl}
[\![t_1 \times t_2]\!] & = & \\
[\![t_1 \rightarrow t_2]\!] & = &
\end{array}$$

Step 1 : Model

Define when $[\![\ ]\!] : \textbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\begin{array}{rcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\
[\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\neg t]\!] & = & \mathcal{D} \backslash [\![t]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset \\
[\![\mathbb{1}]\!] & = & \mathcal{D}
\end{array}$$

- Hard for constructors:

$$\begin{array}{rcl}
[\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\
[\![t_1 \rightarrow t_2]\!] & = &
\end{array}$$

Step 1 : Model

Define when $[\![\ ]\!] : \textbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\begin{array}{rcl} [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\neg t]\!] & = & \mathcal{D} \setminus [\![t]\!] \\ [\![\mathbb{0}]\!] & = & \emptyset \\ [\![\mathbb{1}]\!] & = & \mathcal{D} \end{array}$$

- Hard for constructors:

$$\begin{array}{rcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \rightarrow t_2]\!] & = & ??? \end{array}$$

Step 1 : Model

Define when $[\![\ ]\!] : \textbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\begin{array}{lcl} [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\neg t]\!] & = & \mathcal{D} \backslash [\![t]\!] \\ [\![\mathbb{0}]\!] & = & \emptyset \\ [\![\mathbb{1}]\!] & = & \mathcal{D} \end{array}$$

- Hard for constructors:

$$\begin{array}{lcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \rightarrow t_2]\!] & = & ??? \end{array}$$

Think semantically!

Intuition

$$[\![t{\to}s]\!] = ???$$

Intuition

$$[\![t \to s]\!] = \{\text{functions from } [\![t]\!] \text{ to } [\![s]\!]\}$$

Intuition

$$[\![t \to s]\!] = \{ f \subseteq \mathcal{D}^2 \mid \forall (d_1, d_2) \in f .\ d_1 \in [\![t]\!] \Rightarrow d_2 \in [\![s]\!] \}$$

Intuition

$$[\![t{\rightarrow}s]\!] = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}}) \qquad (\ \overline{X} \overset{\text{def}}{=} \text{complement of } X \ )$$

Intuition

$$[\![t{\rightarrow}s]\!] = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}}) \qquad (*)$$

Impossible since it requires $\mathcal{P}(\mathcal{D}^2) \subseteq \mathcal{D}$

Intuition

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

KEY OBSERVATION 2:

We need the model to state **how types are related** rather than **what the types are**

Intuition

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

KEY OBSERVATION 2:

We need the model to state **how types are related**

Accept every $[\![\ ]\!]$ that behaves w.r.t. $\subseteq$ **as if** equation $(*)$ held,

Intuition

$$[\![t \rightarrow s]\!] = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}}) \qquad (*)$$

KEY OBSERVATION 2:

We need the model to

Accept every $[\![\ ]\!]$ that behaves w.r.t. $\subseteq$ **as if** equation $(*)$ held, namely

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

Intuition

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

KEY OBSERVATION 2:

We need the model to

Accept every $[\![\ ]\!]$ that behaves w.r.t. $\subseteq$ **as if** equation $(*)$ held, namely

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

and similarly for any boolean combination of arrow types.

Technically . . .

1. **Take** $[\![\_]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ such that

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] & [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset & [\![\mathbb{1}]\!] & = & \mathcal{D} \\
[\![\neg t]\!] & = & [\![\mathbb{1}]\!] \backslash [\![t]\!] & & &
\end{array}
$$

Technically . . .

1. **Take** $[\![\_]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ such that

$$
\begin{array}{rclcrcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] & \quad & [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset & & [\![\mathbb{1}]\!] & = & \mathcal{D} \\
[\![\neg t]\!] & = & [\![\mathbb{1}]\!] \backslash [\![t]\!] & & & &
\end{array}
$$

[connective semantics]

Technically . . .

1. **Take** $[\![\_]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ such that

$$\begin{array}{rcl@{\qquad\qquad}rcl}
[\![t_1{\vee}t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] & [\![t_1{\wedge}t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset & [\![\mathbb{1}]\!] & = & \mathcal{D} \\
[\![\neg t]\!] & = & [\![\mathbb{1}]\!] \setminus [\![t]\!] & & &
\end{array}$$

[connective semantics]

2. **Define** $\mathbb{E}(\_) : \textbf{Types} \to \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\begin{array}{rcl@{\quad}l}
\mathbb{E}(t_1{\times}t_2) & \overset{\text{def}}{=} & [\![t_1]\!] \times [\![t_2]\!] & \subseteq \mathcal{D}^2 \\
\mathbb{E}(t_1{\to}t_2) & \overset{\text{def}}{=} & \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) & \subseteq \mathcal{P}(\mathcal{D}^2)
\end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbb{E}(t_1{\vee}t_2) & \overset{\text{def}}{=} & \mathbb{E}(t_1) \cup \mathbb{E}(t_2) & \mathbb{E}(t_1{\wedge}t_2) & \overset{\text{def}}{=} & \mathbb{E}(t_1) \cap \mathbb{E}(t_2) \\
\mathbb{E}(\mathbb{0}) & \overset{\text{def}}{=} & \emptyset & \mathbb{E}(\mathbb{1}) & \overset{\text{def}}{=} & \mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2) \\
\mathbb{E}(\neg t) & \overset{\text{def}}{=} & \mathbb{E}(\mathbb{1}) \setminus \mathbb{E}(t) & & &
\end{array}$$

Technically . . .

1. **Take** $[\![\_]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ such that

$$\begin{array}{rcl@{\qquad\qquad}rcl}
[\![t_1{\vee}t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] & [\![t_1{\wedge}t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset & [\![\mathbb{1}]\!] & = & \mathcal{D} \\
[\![\neg t]\!] & = & [\![\mathbb{1}]\!]\setminus[\![t]\!] & & &
\end{array}$$

[connective semantics]

2. **Define** $\mathbb{E}(\_) : \textbf{Types} \to \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\begin{array}{rcl@{\quad}l}
\mathbb{E}(t_1{\times}t_2) & \overset{\text{def}}{=} & [\![t_1]\!] \times [\![t_2]\!] & \subseteq \mathcal{D}^2 \\
\mathbb{E}(t_1{\to}t_2) & \overset{\text{def}}{=} & \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) & \subseteq \mathcal{P}(\mathcal{D}^2)
\end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbb{E}(t_1{\vee}t_2) & \overset{\text{def}}{=} & \mathbb{E}(t_1) \cup \mathbb{E}(t_2) & \mathbb{E}(t_1{\wedge}t_2) & \overset{\text{def}}{=} & \mathbb{E}(t_1) \cap \mathbb{E}(t_2) \\
\mathbb{E}(\mathbb{0}) & \overset{\text{def}}{=} & \emptyset & \mathbb{E}(\mathbb{1}) & \overset{\text{def}}{=} & \mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2) \\
\mathbb{E}(\neg t) & \overset{\text{def}}{=} & \mathbb{E}(\mathbb{1})\setminus\mathbb{E}(t) & & &
\end{array}$$

[constructor semantics]

Technically . . .

1. **Take** $\llbracket\_\rrbracket : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \qquad \llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \mathbb{0} \rrbracket = \emptyset \qquad \llbracket \mathbb{1} \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \llbracket \mathbb{1} \rrbracket \setminus \llbracket t \rrbracket$$

[connective semantics]

2. **Define** $\mathbb{E}(\_) : \textbf{Types} \to \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\mathbb{E}(t_1 \times t_2) \overset{\text{def}}{=} \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \subseteq \mathcal{D}^2$$

$$\mathbb{E}(t_1 \to t_2) \overset{\text{def}}{=} \mathcal{P}(\overline{\llbracket t_1 \rrbracket \times \overline{\llbracket t_2 \rrbracket}}) \subseteq \mathcal{P}(\mathcal{D}^2)$$

$$\mathbb{E}(t_1 \vee t_2) \overset{\text{def}}{=} \mathbb{E}(t_1) \cup \mathbb{E}(t_2) \qquad \mathbb{E}(t_1 \wedge t_2) \overset{\text{def}}{=} \mathbb{E}(t_1) \cap \mathbb{E}(t_2)$$

$$\mathbb{E}(\mathbb{0}) \overset{\text{def}}{=} \emptyset \qquad \mathbb{E}(\mathbb{1}) \overset{\text{def}}{=} \mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2)$$

$$\mathbb{E}(\neg t) \overset{\text{def}}{=} \mathbb{E}(\mathbb{1}) \setminus \mathbb{E}(t)$$

[constructor semantics]

3. **Model:** Instead of requiring $\llbracket t \rrbracket = \mathbb{E}(t)$, accept $\llbracket\ \rrbracket$ if

$$\llbracket t \rrbracket = \emptyset \iff \mathbb{E}(t) = \emptyset$$

Technically . . .

1. **Take** $[\![\_]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ such that

$$\begin{array}{rcl@{\qquad\qquad}rcl} [\![t_1 \vee t_2]\!] &=& [\![t_1]\!] \cup [\![t_2]\!] & [\![t_1 \wedge t_2]\!] &=& [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\mathbb{0}]\!] &=& \emptyset & [\![\mathbb{1}]\!] &=& \mathcal{D} \\ [\![\neg t]\!] &=& [\![\mathbb{1}]\!] \setminus [\![t]\!] \end{array}$$

[connective semantics]

2. **Define** $\mathbb{E}(\_) : \textbf{Types} \to \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\begin{array}{rcll} \mathbb{E}(t_1 \times t_2) &\overset{\text{def}}{=}& [\![t_1]\!] \times [\![t_2]\!] & \subseteq \mathcal{D}^2 \\ \mathbb{E}(t_1 \to t_2) &\overset{\text{def}}{=}& \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) & \subseteq \mathcal{P}(\mathcal{D}^2) \end{array}$$

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathbb{E}(t_1 \vee t_2) &\overset{\text{def}}{=}& \mathbb{E}(t_1) \cup \mathbb{E}(t_2) & \mathbb{E}(t_1 \wedge t_2) &\overset{\text{def}}{=}& \mathbb{E}(t_1) \cap \mathbb{E}(t_2) \\ \mathbb{E}(\mathbb{0}) &\overset{\text{def}}{=}& \emptyset & \mathbb{E}(\mathbb{1}) &\overset{\text{def}}{=}& \mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2) \\ \mathbb{E}(\neg t) &\overset{\text{def}}{=}& \mathbb{E}(\mathbb{1}) \setminus \mathbb{E}(t) \end{array}$$

[constructor semantics]

3. **Model:** Instead of requiring $[\![t]\!] = \mathbb{E}(t)$, accept $[\![\ ]\!]$ if

$$[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$$

(which is equivalent to $[\![s]\!] \subseteq [\![t]\!] \iff \mathbb{E}(s) \subseteq \mathbb{E}(t)$)

The main intuition

To characterize $\leq$ all is needed is the test of emptyness

The main intuition

To characterize $\leq$ all is needed is the test of emptyness

Indeed: $s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = \varnothing$

The main intuition

To characterize $\leq$ all is needed is the test of emptyness

Indeed: $s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = \varnothing$

Instead of $[\![t]\!] = \mathbb{E}(t)$, the weaker $[\![t]\!] = \emptyset \Leftrightarrow \mathbb{E}(t) = \emptyset$ suffices for $\leq$.

The main intuition

To characterize $\leq$ all is needed is the test of emptyness

Indeed: $s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = \varnothing$

Instead of $[\![t]\!] = \mathbb{E}(t)$, the weaker $[\![t]\!] = \emptyset \Leftrightarrow \mathbb{E}(t) = \emptyset$ suffices for $\leq$.

$[\![\ ]\!]$ and $\mathbb{E}(\ )$ must have the same zeros

The main intuition

To characterize $\leq$ all is needed is the test of emptyness

Indeed: $s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = \varnothing$

Instead of $[\![t]\!] = \mathbb{E}(t)$, the weaker $[\![t]\!] = \emptyset \Leftrightarrow \mathbb{E}(t) = \emptyset$ suffices for $\leq$.

$[\![\ ]\!]$ and $\mathbb{E}(\ )$ must have the same zeros

We relaxed our requirement but . . .

DOES A MODEL EXIST?

Is it possible to define $[\![\_]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$ that satisfies the model conditions, in particular a $[\![]\!]$ such that $[\![t]\!] = \emptyset \Leftrightarrow \mathbb{E}(t) = \emptyset$?

The main intuition

To characterize $\leq$ all is needed is the test of emptyness

Indeed: $s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = \varnothing$

Instead of $[\![t]\!] = \mathbb{E}(t)$, the weaker $[\![t]\!] = \emptyset \Leftrightarrow \mathbb{E}(t) = \emptyset$ suffices for $\leq$.

$[\![\ ]\!]$ and $\mathbb{E}(\ )$ must have the same zeros

We relaxed our requirement but . . .

DOES A MODEL EXIST?

Is it possible to define $[\![\_]\!]$: **Types** $\to \mathcal{P}(\mathcal{D})$ that satisfies the model conditions, in particular a $[\![]\!]$ such that $[\![t]\!] = \emptyset \Leftrightarrow \mathbb{E}(t) = \emptyset$?

YES: an example within two slides

The role of $\mathbb{E}()$

$\mathbb{E}()$ characterizes the behavior of types (for what it concerns $\leq$ one can consider $[\![t]\!] = \mathbb{E}(t)$): it depends on the language the types are intended for.

The role of $\mathbb{E}()$

$\mathbb{E}()$ characterizes the behavior of types (for what it concerns $\leq$ one can consider $[\![t]\!] = \mathbb{E}(t)$): it depends on the language the types are intended for.

Variations are possible. Our choice

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

accounts for languages that are:

The role of $\mathbb{E}()$

$\mathbb{E}()$ characterizes the behavior of types (for what it concerns $\leq$ one can consider $[\![t]\!] = \mathbb{E}(t)$): it depends on the language the types are intended for.

Variations are possible. Our choice

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

accounts for languages that are:

1. *Non-deterministic*:
   Admits functions in which (d, d_1) and (d, d_2) with $d_1 \neq d_2$.

The role of $\mathbb{E}()$

$\mathbb{E}()$ characterizes the behavior of types (for what it concerns $\leq$ one can consider $[\![t]\!] = \mathbb{E}(t)$): it depends on the language the types are intended for.

Variations are possible. Our choice

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

accounts for languages that are:

1. *Non-deterministic*:
   Admits functions in which (d, d_1) and (d, d_2) with $d_1 \neq d_2$.
2. *Non-terminating*:
   a function in $[\![t \to s]\!]$ may be not total on $[\![t]\!]$.

The role of $\mathbb{E}()$

$\mathbb{E}()$ characterizes the behavior of types (for what it concerns $\leq$ one can consider $\llbracket t \rrbracket = \mathbb{E}(t)$): it depends on the language the types are intended for.

Variations are possible. Our choice

$$\mathbb{E}(t_1 \to t_2) = \mathcal{P}(\overline{\llbracket t_1 \rrbracket \times \overline{\llbracket t_2 \rrbracket}})$$

accounts for languages that are:

1. *Non-deterministic*:
   Admits functions in which (d, d_1) and (d, d_2) with $d_1 \neq d_2$.
2. *Non-terminating*:
   a function in $\llbracket t \to s \rrbracket$ may be not total on $\llbracket t \rrbracket$.E.g.

$$\llbracket t \to \mathbb{0} \rrbracket = \text{functions diverging on } t$$

The role of $\mathbb{E}()$

$\mathbb{E}()$ characterizes the behavior of types (for what it concerns $\leq$ one can consider $[\![t]\!] = \mathbb{E}(t)$): it depends on the language the types are intended for.

Variations are possible. Our choice

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

accounts for languages that are:

1. *Non-deterministic*:
   Admits functions in which (d, d_1) and (d, d_2) with $d_1 \neq d_2$.
2. *Non-terminating*:
   a function in $[\![t{\to}s]\!]$ may be not total on $[\![t]\!]$.E.g.
   $$[\![t{\to}\mathbb{0}]\!] = \text{functions diverging on } t$$
3. *Overloaded*:
   $$[\![(t_1{\vee}t_2){\to}(s_1{\wedge}s_2)]\!] \subsetneq [\![(t_1{\to}s_1){\wedge}(t_2{\to}s_2)]\!]$$

Closing the circle

1. Take any model $(\mathcal{B}, [\![]\!]_{\mathcal{B}})$ to bootstrap the definition.

Closing the circle

1. Take any model $(\mathcal{B}, [\![\,]\!]_{\mathcal{B}})$ to bootstrap the definition.
2. Define

$$s \leq_{\mathcal{B}} t \qquad \Longleftrightarrow \qquad [\![s]\!]_{\mathcal{B}} \;\subseteq\; [\![t]\!]_{\mathcal{B}}$$

Closing the circle

1. Take any model $(\mathcal{B}, [\![]\!]_{\mathcal{B}})$ to bootstrap the definition.
2. Define
$$s \leq_{\mathcal{B}} t \qquad \iff \qquad [\![s]\!]_{\mathcal{B}} \subseteq [\![t]\!]_{\mathcal{B}}$$
3. Take any "appropriate" language $\mathcal{L}$ and use $\leq_{\mathcal{B}}$ to type it
$$\Gamma \vdash_{\mathcal{B}} e : t$$

Closing the circle

1. Take any model $(\mathcal{B}, [\![\,]\!]_{\mathcal{B}})$ to bootstrap the definition.
2. Define
$$s \leq_{\mathcal{B}} t \qquad \Longleftrightarrow \qquad [\![s]\!]_{\mathcal{B}} \subseteq [\![t]\!]_{\mathcal{B}}$$
3. Take any "appropriate" language $\mathcal{L}$ and use $\leq_{\mathcal{B}}$ to type it
$$\Gamma \vdash_{\mathcal{B}} e : t$$
4. Define a new interpretation $[\![t]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \ \vdash_{\mathcal{B}} v : t\}$ and
$$s \leq_{\mathcal{V}} t \qquad \Longleftrightarrow \qquad [\![s]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}}$$

Closing the circle

1. Take any model $(\mathcal{B}, [\![]\!]_{\mathcal{B}})$ to bootstrap the definition.
2. Define
$$s \leq_{\mathcal{B}} t \qquad \iff \qquad [\![s]\!]_{\mathcal{B}} \ \subseteq \ [\![t]\!]_{\mathcal{B}}$$
3. Take any "appropriate" language $\mathcal{L}$ and use $\leq_{\mathcal{B}}$ to type it
$$\Gamma \vdash_{\mathcal{B}} e : t$$
4. Define a new interpretation $[\![t]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \ \vdash_{\mathcal{B}} v : t\}$ and
$$s \leq_{\mathcal{V}} t \qquad \iff \qquad [\![s]\!]_{\mathcal{V}} \ \subseteq \ [\![t]\!]_{\mathcal{V}}$$
5. If $\mathcal{L}$ is "appropriate" $(\vdash_{\mathcal{B}} v : t \iff \not\vdash_{\mathcal{B}} v : \neg t)$ then $[\![]\!]_{\mathcal{V}}$ is a model and
$$s \leq_{\mathcal{B}} t \qquad \iff \qquad s \leq_{\mathcal{V}} t$$

Closing the circle

1. Take any model $(\mathcal{B}, [\![]\!]_{\mathcal{B}})$ to bootstrap the definition.
2. Define
$$s \leq_{\mathcal{B}} t \qquad \iff \qquad [\![s]\!]_{\mathcal{B}} \subseteq [\![t]\!]_{\mathcal{B}}$$
3. Take any "appropriate" language $\mathcal{L}$ and use $\leq_{\mathcal{B}}$ to type it
$$\Gamma \vdash_{\mathcal{B}} e : t$$
4. Define a new interpretation $[\![t]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and
$$s \leq_{\mathcal{V}} t \qquad \iff \qquad [\![s]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}}$$
5. If $\mathcal{L}$ is "appropriate" $(\vdash_{\mathcal{B}} v : t \iff \not\vdash_{\mathcal{B}} v : \neg t)$ then $[\![]\!]_{\mathcal{V}}$ is a model and
$$s \leq_{\mathcal{B}} t \qquad \iff \qquad s \leq_{\mathcal{V}} t$$

The circle is closed

Exhibit a model

Does a model exists? (i.e. a $[\![\;]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

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

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

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

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

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

$$[\![s \times t]\!]_{\mathcal{U}} = [\![s]\!]_{\mathcal{U}} \times [\![t]\!]_{\mathcal{U}} \qquad [\![t \rightarrow s]\!]_{\mathcal{U}} = \mathcal{P}_f(\underline{[\![t]\!]_{\mathcal{U}} \times \overline{[\![s]\!]_{\mathcal{U}}}})$$

Exhibit a model

Does a model exists? (i.e. a $\llbracket\ \rrbracket$ such that $\llbracket t\rrbracket = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, \llbracket\ \rrbracket_{\mathcal{U}})$ where

1. $\mathcal{U}$ least solution of $X = X^2 + \mathcal{P}_f(X^2)$
2. $\llbracket\ \rrbracket_{\mathcal{U}}$ is defined as:

$$\llbracket \mathbb{0}\rrbracket_{\mathcal{U}} = \emptyset \qquad \llbracket \mathbb{1}\rrbracket_{\mathcal{U}} = \mathcal{U} \qquad \llbracket \neg t\rrbracket_{\mathcal{U}} = \mathcal{U} \backslash \llbracket t\rrbracket_{\mathcal{U}}$$

$$\llbracket s \vee t\rrbracket_{\mathcal{U}} = \llbracket s\rrbracket_{\mathcal{U}} \cup \llbracket t\rrbracket_{\mathcal{U}} \qquad \llbracket s \wedge t\rrbracket_{\mathcal{U}} = \llbracket s\rrbracket_{\mathcal{U}} \cap \llbracket t\rrbracket_{\mathcal{U}}$$

$$\llbracket s \times t\rrbracket_{\mathcal{U}} = \llbracket s\rrbracket_{\mathcal{U}} \times \llbracket t\rrbracket_{\mathcal{U}} \qquad \llbracket t \to s\rrbracket_{\mathcal{U}} = \mathcal{P}_f(\overline{\llbracket t\rrbracket_{\mathcal{U}} \times \overline{\llbracket s\rrbracket_{\mathcal{U}}}})$$

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

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

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

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

It is a model: $\mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{U}} \times \overline{[\![s]\!]_{\mathcal{U}}}}) = \varnothing \iff \mathcal{P}(\overline{[\![t]\!]_{\mathcal{U}} \times \overline{[\![s]\!]_{\mathcal{U}}}}) = \varnothing$

Exhibit a model

Does a model exists? (i.e. a $[\![\ ]\!]$ such that $[\![t]\!] = \emptyset \iff \mathbb{E}(t) = \emptyset$)
YES: take $(\mathcal{U}, [\![\ ]\!]_{\mathcal{U}})$ where

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

$$\begin{array}{lll}
[\![\mathbb{0}]\!]_{\mathcal{U}} = \emptyset & [\![\mathbb{1}]\!]_{\mathcal{U}} = \mathcal{U} & [\![\neg t]\!]_{\mathcal{U}} = \mathcal{U} \backslash [\![t]\!]_{\mathcal{U}} \\
[\![s {\vee} t]\!]_{\mathcal{U}} = [\![s]\!]_{\mathcal{U}} \cup [\![t]\!]_{\mathcal{U}} & & [\![s {\wedge} t]\!]_{\mathcal{U}} = [\![s]\!]_{\mathcal{U}} \cap [\![t]\!]_{\mathcal{U}} \\
[\![s {\times} t]\!]_{\mathcal{U}} = [\![s]\!]_{\mathcal{U}} \times [\![t]\!]_{\mathcal{U}} & & [\![t {\to} s]\!]_{\mathcal{U}} = \mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{U}} \times \overline{[\![s]\!]_{\mathcal{U}}}})
\end{array}$$

It is a model: $\mathcal{P}_f(\overline{[\![t]\!]_{\mathcal{U}} \times \overline{[\![s]\!]_{\mathcal{U}}}}) = \varnothing \iff \mathcal{P}(\overline{[\![t]\!]_{\mathcal{U}} \times \overline{[\![s]\!]_{\mathcal{U}}}}) = \varnothing$

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

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

Subtyping Algorithms.

Canonical forms

Every (recursive) type

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

Canonical forms

Every (recursive) type

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

is equivalent (semantically, w.r.t. $\leq$) to a type of the form (I omitted base types):

$$\bigvee_{(P,N)\in\Pi}((\bigwedge_{s\times t\in P} s{\times}t){\wedge}(\bigwedge_{s\times t\in N}\neg(s{\times}t))) \ \bigvee_{(P,N)\in\Sigma}((\bigwedge_{s\to t\in P} s{\to}t){\wedge}(\bigwedge_{s\to t\in N}\neg(s{\to}t)))$$

Canonical forms

Every (recursive) type

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

is equivalent (semantically, w.r.t. $\leq$) to a type of the form (I omitted base types):

$$\bigvee_{(P,N)\in\Pi}((\bigwedge_{s\times t\in P} s\times t)\wedge(\bigwedge_{s\times t\in N}\neg(s\times t)))\ \bigvee_{(P,N)\in\Sigma}((\bigwedge_{s\to t\in P} s\to t)\wedge(\bigwedge_{s\to t\in N}\neg(s\to t)))$$

1. Put it in disjunctive normal form, e.g.

$$(a_1\wedge a_2\wedge\neg a_3)\vee(a_4\wedge\neg a_5)\vee(\neg a_6\wedge\neg a_7)\vee(a_8\wedge a_9)$$

Canonical forms

Every (recursive) type

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

is equivalent (semantically, w.r.t. $\leq$) to a type of the form (I omitted base types):

$$\bigvee_{(P,N)\in\Pi}((\bigwedge_{s\times t\in P} s\times t)\wedge(\bigwedge_{s\times t\in N}\neg(s\times t))) \bigvee_{(P,N)\in\Sigma}((\bigwedge_{s\to t\in P} s\to t)\wedge(\bigwedge_{s\to t\in N}\neg(s\to t)))$$

1. Put it in disjunctive normal form, e.g.
   $$(a_1\wedge a_2\wedge\neg a_3)\vee(a_4\wedge\neg a_5)\vee(\neg a_6\wedge\neg a_7)\vee(a_8\wedge a_9)$$
2. Transform to have only homogeneous intersections, e.g.
   $$((s_1\times t_1)\wedge\neg(s_2\times t_2))\vee(\neg(s_3\to t_3)\wedge\neg(s_4\to t_4))\vee(s_5\times t_5)$$

Canonical forms

Every (recursive) type

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

is equivalent (semantically, w.r.t. $\leq$) to a type of the form (I omitted base types):

$$\bigvee_{(P,N)\in\Pi}((\bigwedge_{s\times t\in P} s\times t)\wedge(\bigwedge_{s\times t\in N}\neg(s\times t)))\ \bigvee_{(P,N)\in\Sigma}((\bigwedge_{s\to t\in P} s\to t)\wedge(\bigwedge_{s\to t\in N}\neg(s\to t)))$$

1. Put it in disjunctive normal form, e.g.
   $$(a_1\wedge a_2\wedge\neg a_3)\vee(a_4\wedge\neg a_5)\vee(\neg a_6\wedge\neg a_7)\vee(a_8\wedge a_9)$$
2. Transform to have only homogeneous intersections, e.g.
   $$((s_1\times t_1)\wedge\neg(s_2\times t_2))\vee(\neg(s_3\to t_3)\wedge\neg(s_4\to t_4))\vee(s_5\times t_5)$$
3. Group negative and positive atoms in the intersections:

$$\bigvee_{(P,N)\in S}((\bigwedge_{a\in P} a)\wedge(\bigwedge_{a\in N}\neg a))$$

Decision procedure

$$s \leq t?$$

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \iff [\![s \wedge \neg t]\!] = \varnothing \iff s \wedge \neg t = \mathbb{0}$$

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \iff [\![s {\wedge} \neg t]\!] = \varnothing \iff s {\wedge} \neg t = \mathbb{0}$$

1. Consider $s{\wedge}\neg t$

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \iff [\![s \wedge \neg t]\!] = \varnothing \iff s \wedge \neg t = \mathbb{0}$$

1. Consider $s \wedge \neg t$
2. Put it in canonical form

$$\bigvee_{(P,N)\in\Pi} ((\bigwedge_{s\times t\in P} s\times t) \wedge (\bigwedge_{s\times t\in N} \neg(s\times t))) \bigvee_{(P,N)\in\Sigma} ((\bigwedge_{s\to t\in P} s\to t) \wedge (\bigwedge_{s\to t\in N} \neg(s\to t)))$$

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \iff [\![s \wedge \neg t]\!] = \varnothing \iff s \wedge \neg t = \mathbb{0}$$

1. Consider $s \wedge \neg t$
2. Put it in canonical form

$$\bigvee_{(P,N)\in\Pi} ((\bigwedge_{s\times t\in P} s\times t) \wedge (\bigwedge_{s\times t\in N} \neg(s\times t))) \bigvee_{(P,N)\in\Sigma} ((\bigwedge_{s\to t\in P} s\to t) \wedge (\bigwedge_{s\to t\in N} \neg(s\to t)))$$

3. Decide (coinductively) whether all the intersections occuring above are empty by applying the set theoretic properties stated in the next slide.

Subtyping decomposition

Decomposition law for products:

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

Decomposition law for arrows:

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

Exercise

Using the laws of the previous slide prove the following equivalences:

$$t_1{\times}s_1 \leq t_2{\times}s_2 \quad\Longleftrightarrow\quad t_1 \leq \emptyset \text{ or } s_1 \leq \emptyset \text{ or } (t_1 \leq t_2 \text{ and } s_1 \leq s_2)$$

$$t_1{\rightarrow}s_1 \leq t_2{\rightarrow}s_2 \quad\Longleftrightarrow\quad t_2 \leq \emptyset \text{ or } \quad \text{or } (t_2 \leq t_1 \text{ and } s_1 \leq s_2)$$

Application to a language.

Language

$$
\begin{array}{rll}
e ::= & x & \text{variable} \\
\mid & \mu f^{(s_1 \to t_1; \dots ; s_n \to t_n)}(x).e & \text{abstraction, } n \geq 1 \\
\mid & e_1 e_2 & \text{application} \\
\mid & (e_1, e_2) & \text{pair} \\
\mid & \pi_i(e) & \text{projection, } i = 1, 2 \\
\mid & (x = e \in t)\,?\,e_1 : e_2 & \text{binding type case}
\end{array}
$$

Typing

$$\frac{\Gamma \vdash e : s \leq_{\mathcal{B}} t}{\Gamma \vdash e : t}\ (\textit{subsumption})$$

Typing

$$\frac{\Gamma \vdash e : s \leq_{\mathcal{B}} t}{\Gamma \vdash e : t}\ (\mathit{subsumption})$$

Typing

$$\frac{\Gamma \vdash e : s \leq_{\mathcal{B}} t}{\Gamma \vdash e : t}\ (subsumption)$$

$$\frac{(\forall i)\ \Gamma, (f : s_1 \to t_1 \wedge \ldots \wedge s_n \to t_n), (x : s_i) \vdash e : t_i}{\Gamma \vdash \mu f^{(s_1 \to t_1; \ldots; s_n \to t_n)}(x).e : s_1 \to t_1 \wedge \ldots \wedge s_n \to t_n}\ (abstr)$$

Typing

$$\frac{\Gamma \vdash e : s \leq_{\mathcal{B}} t}{\Gamma \vdash e : t}\ (\mathit{subsumption})$$

$$\frac{(\forall i)\ \Gamma, (f : s_1{\rightarrow}t_1 \wedge \ldots \wedge s_n{\rightarrow}t_n), (x : s_i) \vdash e : t_i}{\Gamma \vdash \mu f^{(s_1\rightarrow t_1;\ldots;s_n\rightarrow t_n)}(x).e : s_1{\rightarrow}t_1 \wedge \ldots \wedge s_n{\rightarrow}t_n}\ (\mathit{abstr})$$

(for $s_1 \equiv s{\wedge}t$, $s_2 \equiv s{\wedge}\neg t$)

$$\frac{\Gamma \vdash e : s \qquad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i}\ (\mathit{typecase})$$

Typing

$$\frac{\Gamma \vdash e : s \leq_{\mathcal{B}} t}{\Gamma \vdash e : t} \ (\textit{subsumption})$$

$$\frac{(\forall i)\ \Gamma, (f : s_1{\to}t_1 \wedge \ldots \wedge s_n{\to}t_n), (x : s_i) \vdash e : t_i}{\Gamma \vdash \mu f^{(s_1\to t_1;\ldots;s_n\to t_n)}(x).e : s_1{\to}t_1 \wedge \ldots \wedge s_n{\to}t_n} \ (\textit{abstr})$$

(for $s_1 \equiv s \wedge t,\ s_2 \equiv s \wedge \neg t$)

$$\frac{\Gamma \vdash e : s \qquad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i} \ (\textit{typecase})$$

Typing

$$\frac{\Gamma \vdash e : s \leq_{\mathcal{B}} t}{\Gamma \vdash e : t}\ (\mathit{subsumption})$$

$$\frac{(\forall i)\ \Gamma, (f : s_1{\to}t_1 \wedge \ldots \wedge s_n{\to}t_n), (x : s_i) \vdash e : t_i}{\Gamma \vdash \mu f^{(s_1\to t_1;\ldots;s_n\to t_n)}(x).e : s_1{\to}t_1 \wedge \ldots \wedge s_n{\to}t_n}\ (\mathit{abstr})$$

$$(\text{for } s_1 \equiv s{\wedge}t,\ s_2 \equiv s{\wedge}\neg t)$$

$$\frac{\Gamma \vdash e : s \qquad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)\,?\,e_1 : e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbb{0}\}} t_i}\ (\mathit{typecase})$$

Consider:

$$\mu \mathrm{f}^{(\mathrm{Int}\to\mathrm{Int};\mathrm{Bool}\to\mathrm{Bool})}(x).(y = x \in \mathrm{Int})\,?\,(y+1) : \mathrm{not}(y)$$

Reduction

$$\begin{array}{rcll}
(\boldsymbol{\mu} f^{(\ldots)}\mathbf{(}x\mathbf{).}e)v & \rightarrow & e[x/v, (\boldsymbol{\mu} f^{(\ldots)}\mathbf{(}x\mathbf{).}e)/f] & \\
(x = v \in t)\mathbf{?}e_1 \mathbf{:} e_2 & \rightarrow & e_1[x/v] & \text{if } v \in [\![t]\!] \\
(x = v \in t)\mathbf{?}e_1 \mathbf{:} e_2 & \rightarrow & e_2[x/v] & \text{if } v \notin [\![t]\!]
\end{array}$$

Reduction

$$\begin{array}{rcll}
(\mu f^{(\dots)}\mathbf{(}x\mathbf{)}\mathbf{.}e)v & \rightarrow & e[x/v, (\mu f^{(\dots)}\mathbf{(}x\mathbf{)}\mathbf{.}e)/f] & \\
(x = v \in t)\mathbf{?}e_1\mathbf{:}e_2 & \rightarrow & e_1[x/v] & \text{if } v \in [\![t]\!] \\
(x = v \in t)\mathbf{?}e_1\mathbf{:}e_2 & \rightarrow & e_2[x/v] & \text{if } v \notin [\![t]\!]
\end{array}$$

where

$$v \ ::= \ \mu f^{(\dots)}\mathbf{(}x\mathbf{)}\mathbf{.}e \ \mid \ (v,v)$$

Reduction

$$\begin{array}{rcll}(\mu f^{(\dots)}(x).e)v & \rightarrow & e[x/v,(\mu f^{(\dots)}(x).e)/f] & \\ (x=v\in t)\,?\,e_1:e_2 & \rightarrow & e_1[x/v] & \text{if } v\in[\![t]\!] \\ (x=v\in t)\,?\,e_1:e_2 & \rightarrow & e_2[x/v] & \text{if } v\notin[\![t]\!]\end{array}$$

where

$$v \ ::= \ \mu f^{(\dots)}(x).e \ \mid \ (v,v)$$

And we have

$$s \leq_{\mathcal{B}} t \qquad \Longleftrightarrow \qquad s \leq_{\mathcal{V}} t$$

Reduction

$$\begin{array}{rcll} (\mu f^{(\dots)}(x).e)v & \rightarrow & e[x/v, (\mu f^{(\dots)}(x).e)/f] & \\ (x = v \in t)?e_1 : e_2 & \rightarrow & e_1[x/v] & \text{if } v \in [\![t]\!] \\ (x = v \in t)?e_1 : e_2 & \rightarrow & e_2[x/v] & \text{if } v \notin [\![t]\!] \end{array}$$

where

$$v \ ::= \ \mu f^{(\dots)}(x).e \ \mid \ (v, v)$$

And we have

$$s \leq_{\mathcal{B}} t \qquad \iff \qquad s \leq_{\mathcal{V}} t$$

The circle is closed

Why does it work?

$$s \leq_{\mathcal{B}} t \qquad \Longleftrightarrow \qquad s \leq_{\mathcal{V}} t \qquad (1)$$

Equation (1) (actually, $\Rightarrow$) states that the language is quite rich, since there always exists a value to separate two distinct types; i.e. its set of values is a model of types with "enough points"

Why does it work?

$$s \leq_{\mathcal{B}} t \qquad \Longleftrightarrow \qquad s \leq_{\mathcal{V}} t \qquad (1)$$

Equation (1) (actually, ⇒) states that the language is quite rich, since there always exists a value to separate two distinct types; i.e. its set of values is a model of types with "enough points"

For any model $\mathcal{B}$,

$s \not\leq_{\mathcal{B}} t \Longrightarrow$ there exists v such that $\vdash v : s$ and $\not\vdash v : t$

Why does it work?

$$s \leq_{\mathcal{B}} t \qquad \Longleftrightarrow \qquad s \leq_{\mathcal{V}} t \tag{1}$$

Equation (1) (actually, $\Rightarrow$) states that the language is quite rich, since there always exists a value to separate two distinct types; i.e. its set of values is a model of types with "enough points"

For any model $\mathcal{B}$,

$s \not\leq_{\mathcal{B}} t \Longrightarrow$ there exists v such that $\vdash v : s$ and $\not\vdash v : t$

In particular, thanks to multiple arrows in λ-abstractions:

$$\bigwedge_{i=1..k} s_i \rightarrow t_i \not\leq t$$

then the two types are distinguished by $\mu f^{(s_1\rightarrow t_1;\ldots;s_k\rightarrow t_k)}(x).e$

Advantages for the programmer

The programmer does not need to know the gory details. All s/he needs to retain is

Advantages for the programmer

The programmer does not need to know the gory details. All s/he needs to retain is

1. Types are the set of values of that type

Advantages for the programmer

The programmer does not need to know the gory details. All s/he needs to retain is

1. Types are the set of values of that type
2. Subtyping is set inclusion

Advantages for the programmer

The programmer does not need to know the gory details. All s/he needs to retain is

1. Types are the set of values of that type
2. Subtyping is set inclusion

Furthermore the property

$$s \not\leq t \Longrightarrow \text{there exists } v \text{ such that } \vdash v : s \text{ and } \not\vdash v : t$$

is fundamental for meaningful error messages:

Advantages for the programmer

The programmer does not need to know the gory details. All s/he needs to retain is

1. Types are the set of values of that type
2. Subtyping is set inclusion

Furthermore the property

$$s \not\leq t \Longrightarrow \text{there exists } v \text{ such that } \vdash v : s \text{ and } \not\vdash v : t$$

is fundamental for meaningful error messages:

Exibit the *v* at issue rather than pointing to the failure of some deduction rule.

Summary of the theory

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

1. Define $\mathbb{E}(\ )$ for your type constructors so that it matches your semantic intuition

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

1. Define $\mathbb{E}(\ )$ for your type constructors so that it matches your semantic intuition
2. Find a model (any model).

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

1. Define $\mathbb{E}(\ )$ for your type constructors so that it matches your semantic intuition
2. Find a model (any model).
3. Use the subtyping relation induced by the model to type your language: if the intuition was right then the set of values is also a model, otherwise tweak it.

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

1. Define $\mathbb{E}(\ )$ for your type constructors so that it matches your semantic intuition
2. Find a model (any model).
3. Use the subtyping relation induced by the model to type your language: if the intuition was right then the set of values is also a model, otherwise tweak it.
4. Use the set-theoretic properties of the model (actually of $\mathbb{E}(\ )$) to decompose the emptyness test for your type constructors, and hence derive a subtyping algorithm.

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

1. Define $\mathbb{E}(\ )$ for your type constructors so that it matches your semantic intuition
2. **Find a model** (any model). [may be not easy/possible]
3. Use the subtyping relation induced by the model to type your language: if the intuition was right then the set of values is also a model, otherwise **tweak it**. [may be not easy/possible]
4. Use the set-theoretic properties of the model (actually of $\mathbb{E}(\ )$) to decompose the emptyness test for your type constructors, and hence **derive a subtyping algorithm**. [may be not easy/possible]

La morale de l'histoire est . . .

If you have a strong semantic intuition of your favorite language and you want to add set-theoretic $\vee$, $\wedge$, $\neg$ types then:

1. Define $\mathbb{E}(\ )$ for your type constructors so that it matches your semantic intuition
2. **Find a model** (any model).
3. Use the subtyping relation induced by the model to type your language: if the intuition was right then the set of values is also a model, otherwise **tweak it**.
4. Use the set-theoretic properties of the model (actually of $\mathbb{E}(\ )$) to decompose the emptyness test for your type constructors, and hence **derive a subtyping algorithm**.
5. **Enjoy.**

PART 3: POLYMORPHIC SUBTYPING

Goal

Goal

We want to add Type variables:

$$(X\times Y\to X)\wedge((X\to Y)\to X\to Y)$$

and define for them an intuitive semantics

Goal

We want to add Type variables:

$$(X \times Y \to X) \wedge ((X \to Y) \to X \to Y)$$

and define for them an intuitive semantics

WHY?

Goal

We want to add Type variables:

$$(X\times Y\to X)\land((X\to Y)\to X\to Y)$$

and define for them an intuitive semantics

WHY?

Short answers:

- Parametric polymorphism is very useful in practice.
- It covers new needs peculiar to XML processing (*eg*, SOAP envelopes).
- It would make the interface with OCaml complete
- The extension shoud shed new light on the notion of **parametricity**

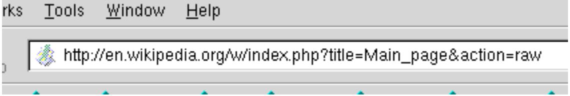
Concrete answer: an example in web development

We need parametric polymorphism to statically type service registration in the Ocsigen web server:

Concrete answer: an example in web development

We need parametric polymorphism to statically type service registration in the Ocsigen web server:

- To every page possibly with parameters



http://en.wikipedia.org/w/index.php?title=Main_page&action=raw

corresponds a function that takes the parameters (the query string) and dynamically generates the appropriate Xhtml page:

```
let wikipage (p : WikiParams) : Xhtml = ...

type WikiParams = <params>
                    <title> String </title>
                    <action> "raw"|"edit" <action>
                  </params>
```

- The binding between the URL `$WEBROOT/w/index` and the function `wikipage` is done by the Ocsigen function `register_new_service`:

```
register_new_service(wikipage,"w.index")
```

- The binding between the URL `$WEBROOT/w/index` and the function `wikipage` is done by the Ocsigen function `register_new_service`:

  ```
  register_new_service(wikipage,"w.index")
  ```

  whenever the page `$WEBROOT/w/index` is selected, Ocsigen passes the XML encoding of the query string to `wikipage` and returns its result.

- The binding between the URL `$WEBROOT/w/index` and the function `wikipage` is done by the Ocsigen function `register_new_service`:

  `register_new_service(wikipage,"w.index")`

  whenever the page `$WEBROOT/w/index` is selected, Ocsigen passes the XML encoding of the query string to `wikipage` and returns its result.
- We would like to give `register_new_service` the type

  $$\forall(\mathtt{X} \leq \mathtt{QueryString}).(\mathtt{X} \to \mathtt{Xhtml}) \times \mathtt{Path} \to \mathtt{unit}$$

  where `QueryString` is the XML type that includes all *query strings* and `Path` specifies the paths of the server.

- The binding between the URL `$WEBROOT/w/index` and the function `wikipage` is done by the Ocsigen function `register_new_service`:

  `register_new_service(wikipage,"w.index")`

  whenever the page `$WEBROOT/w/index` is selected, Ocsigen passes the XML encoding of the query string to `wikipage` and returns its result.
- We would like to give `register_new_service` the type

  $$\forall(\texttt{X} \leq \texttt{QueryString}).(\texttt{X} \to \texttt{Xhtml}) \times \texttt{Path} \to \texttt{unit}$$

  where `QueryString` is the XML type that includes all *query strings* and `Path` specifies the paths of the server.

Notice

We need both higher-order polymorphic functions

- The binding between the URL `$WEBROOT/w/index` and the function `wikipage` is done by the Ocsigen function `register_new_service`:

  ```
  register_new_service(wikipage,"w.index")
  ```

  whenever the page `$WEBROOT/w/index` is selected, Ocsigen passes the XML encoding of the query string to `wikipage` and returns its result.
- We would like to give `register_new_service` the type

  $$\forall(\mathtt{X} \leq \mathtt{QueryString}).(\mathtt{X} \to \mathtt{Xhtml}) \times \mathtt{Path} \to \mathtt{unit}$$

  where `QueryString` is the XML type that includes all *query strings* and `Path` specifies the paths of the server.

Notice

We need both higher-order polymorphic functions and bounded quantification

A very hard problem

Naive solution

$$t \;::=\; B \;\mid\; t\times t \;\mid\; t\to t$$

Naive solution

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

Naive solution

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

Naive solution

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

Now use the previous relation. This is defined for “ground types”

Let $\sigma : \textbf{Vars} \to \textbf{Types}_{\text{ground}}$ denote ground substitutions then define:

$$s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall\sigma\,.\, s\sigma \leq t\sigma$$

Naive solution

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

Now use the previous relation. This is defined for “ground types”

Let σ : **Vars** $\rightarrow$ **Types**$_{\text{ground}}$ denote ground substitutions then define:

$$s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall\sigma\,.\,s\sigma \leq t\sigma$$

or equivalently

$$s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall\sigma.[\![s\sigma]\!] \subseteq [\![t\sigma]\!]$$

Naive solution

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

Now use the previous relation. This is defined for “ground types”

Let σ : **Vars** $\rightarrow$ $\textbf{Types}_{\text{ground}}$ denote ground substitutions then define:

$$\cancel{s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall\sigma\,.\, s\sigma \leq t\sigma}$$

or equivalently

$$\cancel{s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall\sigma.[\![s\sigma]\!] \subseteq [\![t\sigma]\!]}$$

This is a wrong way

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations
2. It *breaks* parametricity:

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations
2. It *breaks* parametricity:

$$(t{\times}X) \quad\leq\quad (t{\times}\neg t)\vee(X{\times}t)$$

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations
2. It *breaks* parametricity:

$$(t\times X) \quad \leq \quad (t\times\neg t)\vee(X\times t)$$

This inclusion holds if and only if t is an *atomic* type:

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations

2. It *breaks* parametricity:

$$(t\times X) \quad\leq\quad (t\times\neg t)\vee(X\times t)$$

This inclusion holds if and only if t is an *atomic* type: Imagine that t is a singleton or a basic type (both are special cases of atomic types), then for all possible interpretation of X it holds

$$t \leq X \quad \text{or} \quad X \leq \neg t$$

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations
2. It *breaks* parametricity:

$$(t{\times}X) \quad \leq \quad (t{\times}\neg t){\vee}(X{\times}t)$$

This inclusion holds if and only if t is an *atomic* type:
Imagine that t is a singleton or a basic type (both are special cases of atomic types), then for all possible interpretation of X it holds

$$t \leq X \qquad \text{or} \qquad X \leq \neg t$$

- If $X \leq \neg t$ then the left element of the union suffices

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is equivalent to solve Diophantine equations

2. It *breaks* parametricity:

$$(t{\times}X) \quad\leq\quad (t{\times}\neg t)\vee(X{\times}t)$$

This inclusion holds if and only if t is an *atomic* type: Imagine that t is a singleton or a basic type (both are special cases of atomic types), then for all possible interpretation of X it holds

$$t \leq X \quad \text{or} \quad X \leq \neg t$$

- If $X \leq \neg t$ then the left element of the union suffices
- If $t \leq X$, then $X = (X\backslash t)\vee t$ and, therefore, $(t{\times}X) = (t{\times}(X\backslash t))\vee(t{\times}t)$. This union is contained component-wise in the one above.

Problems with the naive solution

The fact that

$$(t{\times}X) \quad\leq\quad (t{\times}\neg t)\vee(X{\times}t)$$

holds if and only if t is an *atomic* type is really catastrophic:

Problems with the naive solution

The fact that

$$(t{\times}X) \;\leq\; (t{\times}\neg t)\vee(X{\times}t)$$

holds if and only if t is an *atomic* type is really catastrophic:

- It means that to decide subtyping one has to decide atomicity of types which in general is very hard (*cf.* [Castagna, DeNicola, Varacca TCS 2008])

Problems with the naive solution

The fact that

$$(t{\times}X) \quad \leq \quad (t{\times}\neg t)\vee(X{\times}t)$$

holds if and only if t is an *atomic* type is really catastrophic:

- It means that to decide subtyping one has to decide atomicity of types which in general is very hard (*cf.* [Castagna, DeNicola, Varacca TCS 2008])
- It means that subtyping breaks parametricity since by subsumption we can consider a function generic in its first argument, as one generic on its second argument.

Problems with the naive solution

The fact that

$$(t{\times}X) \quad \leq \quad (t{\times}\neg t){\vee}(X{\times}t)$$

holds if and only if t is an *atomic* type is really catastrophic:

- It means that to decide subtyping one has to decide atomicity of types which in general is very hard (*cf.* [Castagna, DeNicola, Varacca TCS 2008])
- It means that subtyping breaks parametricity since by subsumption we can consider a function generic in its first argument, as one generic on its second argument.

We can eschew the problem by resorting to syntactic solutions:
- Castagna, Frisch, Hosoya [POPL 05]
- Vouillon [POPL 06]
It implies to give up to the underlying semantic intuition

Problems with the naive solution

The fact that

$$(t{\times}X) \leq (t{\times}\neg t)\vee(X{\times}t)$$

holds if and only if t is an *atomic* type is really catastrophic:

- It means that to decide subtyping one has to decide atomicity of types which in general is very hard (*cf.* [Castagna, DeNicola, Varacca TCS 2008])
- It means that subtyping breaks parametricity since by subsumption we can consider a function generic in its first argument, as one generic on its second argument.

We can eschew the problem by resorting to syntactic solutions:
- Castagna, Frisch, Hosoya [POPL 05]
- Vouillon [POPL 06]
It implies to give up to the underlying semantic intuition **NO!**

A semantic solution

Some faint intuition

The loss of parametricity is only due to the interpretation of atomic types, all the rest works (more or less) smoothly

A semantic solution

Some faint intuition

The loss of parametricity is only due to the interpretation of atomic types, all the rest works (more or less) smoothly

Indeed it seems that the crux of the problem is that for an atomic type a

$$a \leq X \quad \text{or} \quad X \leq \neg a$$

validity can *stutter* from one formula to another, missing in this way the uniformity typical of parametricity

A semantic solution

Some faint intuition

The loss of parametricity is only due to the interpretation of atomic types, all the rest works (more or less) smoothly

Indeed it seems that the crux of the problem is that for an atomic type a

$$a \leq X \quad \text{or} \quad X \leq \neg a$$

validity can *stutter* from one formula to another, missing in this way the uniformity typical of parametricity

If we can give a semantic characterization of models in which this stuttering is absent, then this should yield a subtyping relation that is:

- Semantic
- Intuitive for the programmer
- Decidable

Semantic solution

A semantic solution

Rough idea

We must make atomic types “splittable” so that type variables can range over strict subsets of every type, atomic types included

A semantic solution

Rough idea

We must make atomic types "splittable" so that type variables can range over strict subsets of every type, atomic types included

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignements:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

A semantic solution

Rough idea

We must make atomic types "splittable" so that type variables can range over strict subsets of every type, atomic types included

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignements:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

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

A semantic solution

Rough idea

We must make atomic types “splittable” so that type variables can range over strict subsets of every type, atomic types included

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignements:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

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

with

$$\begin{array}{lcl@{\qquad\qquad}lcl}
[\![X]\!]\eta & = & \eta(X) & [\![\neg t]\!]\eta & = & \mathcal{D}\setminus[\![t]\!]\eta \\
[\![t_1 \vee t_2]\!]\eta & = & [\![t_1]\!]\eta \cup [\![t_2]\!]\eta & [\![t_1 \wedge t_2]\!]\eta & = & [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \\
[\![\mathbb{0}]\!]\eta & = & \emptyset & [\![\mathbb{1}]\!]\eta & = & \mathcal{D}
\end{array}$$

Subtyping relation

In this framework the natural definition of subtyping is

$$s \leq t \quad \stackrel{def}{\Longleftrightarrow} \quad \forall\eta\,.\; [\![s]\!]\eta \subseteq [\![t]\!]\eta$$

It just remains to find the uniformity condition to recover parametricity.

The magic property

Consider only models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

The magic property

Consider only models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

- It avoids stuttering: $(\llbracket a{\wedge}\neg X\rrbracket\eta{=}\varnothing$ or $\llbracket a{\wedge}X\rrbracket\eta{=}\varnothing)$ holds true if and only if a is empty.

The magic property

Consider only models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $([\![a{\wedge}\neg X]\!]\eta{=}\varnothing \text{ or } [\![a{\wedge}X]\!]\eta{=}\varnothing)$ holds true if and only if a is empty.
- There is a natural model: every model in which all types are interpreted as infinite sets satisfies it (**we recover the initial faint intuition**).

The magic property

Consider only models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

- It avoids stuttering: $(\llbracket a{\wedge}\neg X\rrbracket\eta{=}\varnothing \text{ or } \llbracket a{\wedge}X\rrbracket\eta{=}\varnothing)$ holds true if and only if a is empty.
- There is a natural model: every model in which all types are interpreted as infinite sets satisfies it (**we recover the initial faint intuition**).
- A sound and complete algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm for ground types (though, decidability is an open problem, yet).

The magic property

Consider only models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

- It avoids stuttering: $(\llbracket a{\wedge}\neg X\rrbracket\eta{=}\varnothing$ or $\llbracket a{\wedge}X\rrbracket\eta{=}\varnothing)$ holds true if and only if a is empty.
- There is a natural model: every model in which all types are interpreted as infinite sets satisfies it (**we recover the initial faint intuition**).
- A sound and complete algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm for ground types (though, decidability is an open problem, yet).
- An intuitive relation: the algorithm returns intuitive results (actually, it helps to better understand twisted examples)

Examples of subtyping relations

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \;\sim\; \alpha \vee \beta \to \gamma$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \;\sim\; \alpha{\vee}\beta \to \gamma$$

or distributivity laws:

$$(\alpha{\vee}\beta \times \gamma) \;\sim\; (\alpha{\times}\gamma) \vee (\beta{\times}\gamma) \qquad (2)$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \ \sim \ \alpha{\vee}\beta \to \gamma$$

or distributivity laws:

$$(\alpha{\vee}\beta \times \gamma) \ \sim \ (\alpha{\times}\gamma) \vee (\beta{\times}\gamma) \tag{2}$$

combining them we deduce:

$$(\alpha{\times}\gamma \to \delta_1) \wedge (\beta{\times}\gamma \to \delta_2) \ \leq \ (\alpha{\vee}\beta \times \gamma) \to \delta_1 \vee \delta_2$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \ \sim \ \alpha{\vee}\beta \to \gamma$$

or distributivity laws:

$$(\alpha{\vee}\beta \times \gamma) \ \sim \ (\alpha{\times}\gamma) \vee (\beta{\times}\gamma) \tag{2}$$

combining them we deduce:

$$(\alpha{\times}\gamma \to \delta_1) \wedge (\beta{\times}\gamma \to \delta_2) \ \leq \ (\alpha{\vee}\beta \times \gamma) \to \delta_1 \vee \delta_2$$

We can prove relevant relations on infinite types. Consider generic lists:

$$\alpha \ \mathsf{list} = \mu x.(\alpha{\times}x) \vee \mathsf{nil}$$

It contains both the α-lists with an even number of elements

$$\mu x.(\alpha\times(\alpha\times x)) \vee \mathsf{nil} \;\leq\; \mu x.(\alpha\times x) \vee \mathsf{nil}$$

and the α-lists with an odd number of elements

$$\mu x.(\alpha\times(\alpha\times x)) \vee (\alpha\times\mathsf{nil}) \;\leq\; \mu x.(\alpha\times x) \vee \mathsf{nil}$$

It contains both the α-lists with an even number of elements

$$\mu x.(\alpha\times(\alpha\times x))\vee \mathsf{nil} \;\leq\; \mu x.(\alpha\times x)\vee \mathsf{nil}$$

and the α-lists with an odd number of elements

$$\mu x.(\alpha\times(\alpha\times x))\vee(\alpha\times\mathsf{nil}) \;\leq\; \mu x.(\alpha\times x)\vee \mathsf{nil}$$

and it is itself contained in the union of the two, that is:

$$\alpha\ \mathsf{list} \;\sim\; (\mu x.(\alpha\times(\alpha\times x))\vee \mathsf{nil}) \;\vee\; (\mu x.(\alpha\times(\alpha\times x))\vee(\alpha\times \mathsf{nil}))$$

It contains both the α-lists with an even number of elements

$$\mu x.(\alpha\times(\alpha\times x))\vee \mathsf{nil} \ \leq\ \mu x.(\alpha\times x)\vee \mathsf{nil}$$

and the α-lists with an odd number of elements

$$\mu x.(\alpha\times(\alpha\times x))\vee(\alpha\times\mathsf{nil}) \ \leq\ \mu x.(\alpha\times x)\vee \mathsf{nil}$$

and it is itself contained in the union of the two, that is:

$$\alpha\ \mathsf{list}\ \sim\ (\mu x.(\alpha\times(\alpha\times x))\vee \mathsf{nil})\ \vee\ (\mu x.(\alpha\times(\alpha\times x))\vee(\alpha\times \mathsf{nil}))$$

And we can prove far more complicated relations (see later).

Subtyping algorithm

Subtyping Algorithm

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

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

Subtyping Algorithm

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

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

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

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

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

Subtyping Algorithm

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

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

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

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

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

Step 3: **Simplify mixed intersections:**

Consider each summand of the union: cases such as $t_1{\times}t_2 \wedge t_1{\rightarrow}t_2$ or $t_1{\times}t_2 \wedge \neg(t_1{\rightarrow}t_2)$ are straightforward.

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 are of the same kind.

Step 4: **Eliminate toplevel negative variables.**,

$$\forall\eta.[\![t]\!]\eta=\varnothing \iff \forall\eta.[\![t\{{}^{\neg\alpha}/_{\alpha}\}]\!]\eta=\varnothing$$

so replace $\neg\beta_k$ for β_k (forall $k\in K$)

Solve: $$\bigwedge_{i\in I}a_i\bigwedge_{j\in J}\neg a'_j\bigwedge_{h\in H}\alpha_h$$

Step 4: Eliminate toplevel negative variables.,

$$\forall\eta.[\![t]\!]\eta = \varnothing \iff \forall\eta.[\![t\{{}^{\neg\alpha}/_{\alpha}\}]\!]\eta = \varnothing$$

so replace $\neg\beta_k$ for β_k (forall $k \in K$)

Solve:
$$\bigwedge_{i\in I} a_i \bigwedge_{j\in J} \neg a'_j \bigwedge_{h\in H} \alpha_h$$

Step 5: Eliminate toplevel variables.

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \bigwedge_{h\in H} \alpha_h \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2$$

holds if and only if

$$\bigwedge_{t_1\times t_2\in P} t_1\sigma \times t_2\sigma \bigwedge_{h\in H} \gamma_h^1 \times \gamma_h^2 \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\sigma \times t'_2\sigma$$

where $\sigma = \{{}^{(\gamma_h^1\times\gamma_h^2)\vee\alpha_h}/_{\alpha_h}\}_{h\in H}$ (similarly for arrows)

Step 6: **Eliminate toplevel constructors, memoize, and recurse**.
Thanks to *convexity* and the product decomposition rules

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2 \tag{3}$$

is equivalent to

$$\forall N'\subseteq N. \left(\bigwedge_{t_1\times t_2\in P} t_1 \leq \bigvee_{t'_1\times t'_2\in N'} t'_1\right) \text{ or } \left(\bigwedge_{t_1\times t_2\in P} t_2 \leq \bigvee_{t'_1\times t'_2\in N\setminus N'} t'_2\right)$$

(similarly for arrows)

PART 4: POLYMORPHIC LANGUAGE

Motivating example

A motivating example in Haskell

[no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
              | [] -> []
              | (x : xs) -> (f x : map f xs)
```

A motivating example in Haskell [no XML]

```
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) [no XML]

```
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)

[no XML]

```
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) [no XML]

```
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) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |  _  -> x
```

type-case

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
              | [] -> []
              | (x : xs) -> (f x : map f xs)
```

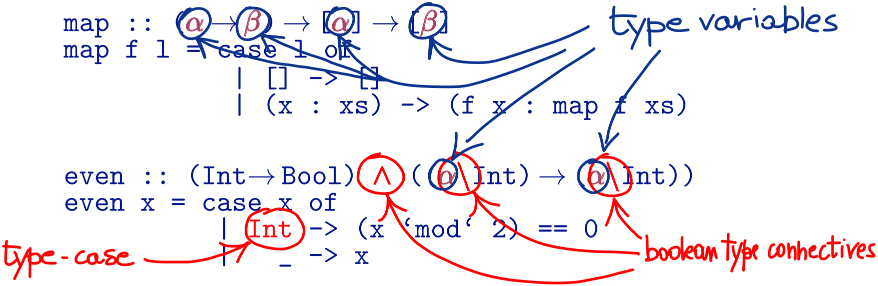
```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
           | Int -> (x `mod` 2) == 0
           | _ -> x
```

type-case

boolean type connectives

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

A motivating example in Haskell (almost) [no XML]



```
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) [no XML]

```
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*.

Typical function used to modify some nodes of an XML tree leaving the others unchanged.

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
           | []  -> []
           | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int)→(α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

- Expression: if the argument is an integer then return the Boolean expression otherwise return the argument
- Type: when applied to an `Int` it returns a `Bool`; when applied to an argument that is not an `Int` it returns a result *of the same type*.

The combination of type-case and intersections yields statically typed dynamic overloading.

A motivating example in Haskell (almost) [no XML]

```
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) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)

even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x ‘mod‘ 2) == 0
          |   _ -> x
```

This example as a yardstick. I want to define a language that:

1. Can define both `map` and `even`
2. Can *check* the types specified in the signature

A motivating example in Haskell (almost) [no XML]

```
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) [no XML]

```
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) [no XML]

```
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
```

Tough!

This example a k. I want to define a language that:

1. Can define 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) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
             | [] -> []
             | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
            | Int -> (x ‘mod‘ 2) == 0
            |   _ -> x
```

We expect **map even** to have the following type:

$(\,\mathtt{Int}\ \mathsf{list} \to \mathtt{Bool}\ \mathsf{list}\,) \wedge$
$(\,\alpha\backslash\mathtt{Int}\ \mathsf{list} \to \alpha\backslash\mathtt{Int}\ \mathsf{list}\,) \wedge$
$(\,\alpha\vee\mathtt{Int}\ \mathsf{list} \to (\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}\ \mathsf{list}\,)$

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
             | [] -> []
             | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
            | Int -> (x ‘mod‘ 2) == 0
            |   _ -> x
```

We expect **map even** to have the following type:

$(\mathtt{Int}\ \mathrm{list} \to \mathtt{Bool}\ \mathrm{list}) \wedge$	int lists are transformed into bool lists
$(\alpha\setminus\mathtt{Int}\ \mathrm{list} \to \alpha\setminus\mathtt{Int}\ \mathrm{list}) \wedge$	lists w/o ints return the same type
$(\alpha\vee\mathtt{Int}\ \mathrm{list} \to (\alpha\setminus\mathtt{Int})\vee\mathtt{Bool}\ \mathrm{list})$	ints in the arg. are replaced by bools

A motivating example in Haskell (almost) [no XML]

```
map :: (α → β) → [α] → [β]
map f l = case l of
            | [] -> []
            | (x : xs) -> (f x : map f xs)
```

```
even :: (Int→Bool) ∧ ((α\Int) → (α\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

We expect **map even** to have the following type:

$(\mathtt{Int}\ \mathsf{list} \to \mathtt{Bool}\ \mathsf{list})\ \wedge$	int lists are transformed into bool lists
$(\alpha\backslash\mathtt{Int}\ \mathsf{list} \to \alpha\backslash\mathtt{Int}\ \mathsf{list})\ \wedge$	lists w/o ints return the same type
$(\alpha\vee\mathtt{Int}\ \mathsf{list} \to (\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}\ \mathsf{list})$	ints in the arg. are replaced by bools

Difficult because of expansion: needs *a set of type substitutions* — rather than just one— to unify the domain and the argument types.

Formal framework

Formal calculus

Exprs $e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}x.e \mid e{\in}t\,?\,e : e$

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

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Expressions include:

Formal calculus

Exprs $e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Expressions include:

A type-case:

- abstracts regular type patterns
- makes dynamic overloading possible

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Expressions include:

A type-case:

- abstracts regular type patterns
- makes dynamic overloading possible

Explicitly-typed functions:

- Needed by the type-case [e.g. $\mu f.\lambda x.f{\in}(\mathbb{1}{\to}\mathtt{Int})\,?\,\mathtt{true} : 42$]
- More expressive with the result type (parameter type not enough)

$\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$: well typed if for all $i{\in}I$ from $x : s_i$ we can deduce $e : t_i$.

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I}s_i\to t_i}x.e \mid e{\in}t\,?\,e : e$

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

Types may be recursive and have a set-theoretic interpretation:

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Types may be recursive and have a set-theoretic interpretation:

Constructors: $[\![\mathtt{Int}]\!]{=}\{0, 1, -1, ...\}$. $[\![s \to t]\!] = \lambda$-abstractions that when applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

Types $t \quad ::= \quad B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \boldsymbol{\alpha}$

Types may be recursive and have a set-theoretic interpretation:

Constructors: $[\![\mathtt{Int}]\!]{=}\{0,1,-1,...\}$. $[\![s \to t]\!] = \lambda$-abstractions that when applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Connectives have the corresponding set-theoretic interpretation:

$$[\![s{\vee}t]\!] = [\![s]\!] \cup [\![t]\!] \qquad [\![s{\wedge}t]\!] = [\![s]\!] \cap [\![t]\!] \qquad [\![\neg t]\!] = [\![\mathbb{1}]\!] \setminus [\![t]\!]$$

Formal calculus

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$

Types may be recursive and have a set-theoretic interpretation:

Constructors: $[\![\mathtt{Int}]\!]{=}\{0,1,-1,...\}$. $[\![s \to t]\!] = \lambda$-abstractions that when applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Connectives have the corresponding set-theoretic interpretation:

$$[\![s{\vee}t]\!] = [\![s]\!] \cup [\![t]\!] \qquad [\![s{\wedge}t]\!] = [\![s]\!] \cap [\![t]\!] \qquad [\![\neg t]\!] = [\![\mathbb{1}]\!] \setminus [\![t]\!]$$

Subtyping:

- it is *defined* as set-containment: $s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$;

Formal calculus

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

Types $t \quad ::= \quad B \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \boldsymbol{\alpha}$

Types may be recursive and have a set-theoretic interpretation:

Constructors: $[\![\texttt{Int}]\!]{=}\{0, 1, -1, ...\}$. $[\![s \to t]\!] = \lambda$-abstractions that when applied to arguments in $[\![s]\!]$ return only results in $[\![t]\!]$.

Connectives have the corresponding set-theoretic interpretation:

$$[\![s{\vee}t]\!] = [\![s]\!] \cup [\![t]\!] \qquad [\![s{\wedge}t]\!] = [\![s]\!] \cap [\![t]\!] \qquad [\![\neg t]\!] = [\![\mathbb{1}]\!] \setminus [\![t]\!]$$

Subtyping with type variables:

- it is *defined* as set-containment: $s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$;
- it is such that forall type-substitutions σ: $s \leq t \;\Rightarrow\; s\sigma \leq t\sigma$;
- it is decidable. [ICFP2011].

Formal calculus: **new stuff**

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Polymorphic functions.

Formal calculus

$$\begin{array}{llcl}
\textbf{Exprs} & e & ::= & x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i} x.e \mid e{\in}t\,?\,e : e \\
\textbf{Types} & t & ::= & B \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha
\end{array}$$

Polymorphic functions: The novelty of this work is that type variables can occur in the *interfaces*.

Formal calculus: **new stuff**

Exprs $e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Polymorphic functions: The novelty of this work is that type variables can occur in the *interfaces*.

- $\lambda^{\alpha\to\alpha}x.x$ polymorphic identity
- $\lambda^{(\alpha\to\beta)\wedge\alpha\to\beta}x.xx$ auto-application

Formal calculus: **new stuff**

Exprs $e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Polymorphic functions: The novelty of this work is that type variables can occur in the *interfaces*.

- $\lambda^{\alpha\to\alpha}x.x$ polymorphic identity
- $\lambda^{(\alpha\to\beta)\wedge\alpha\to\beta}x.xx$ auto-application

Meaning: types obtained by subsumption *and by instantiation*

- $\lambda^{\alpha\to\alpha}x.x : \mathbb{0} \to \mathbb{1}$ subsumption
- $\lambda^{\alpha\to\alpha}x.x : \neg\mathtt{Int}$ subsumption
- $\lambda^{\alpha\to\alpha}x.x : \mathtt{Int} \to \mathtt{Int}$ instantiation New
- $\lambda^{\alpha\to\alpha}x.x : \mathtt{Bool} \to \mathtt{Bool}$ instantiation New

Formal calculus: **new stuff**

Exprs	e	$::=$	$x \mid ee \mid \lambda^{\wedge_{i\in I}s_i\to t_i}x.e \mid e{\in}t\,?\,e : e$
Types	t	$::=$	$B \mid t\to t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \boldsymbol{\alpha}$

Problem

Define an explicitly typed, polymorphic calculus
with intersection types and dynamic type-case

Formal calculus: **new stuff**

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Problem

Define an explicitly typed, polymorphic calculus
with intersection types and dynamic type-case

Formal calculus: **new stuff**

Exprs $e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$

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

Problem

Define an explicitly typed, polymorphic calculus
with intersection types and dynamic type-case

Four simple points to show why dealing with this blend is quite problematic

1. Polymorphism needs instantiation:

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\texttt{Int}/_{\alpha}\}$:

$$(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \quad \in \mathtt{Int}\to\mathtt{Int}$
$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathrm{true} \notin \mathtt{Int}\to\mathtt{Int}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \quad \in \mathtt{Int}\to\mathtt{Int}$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathtt{true} \notin \mathtt{Int}\to\mathtt{Int}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \mathtt{Int}\to\mathtt{Int}$ $\qquad\leadsto \lambda^{\mathtt{Int}\to\mathtt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathsf{true} \notin \mathtt{Int}\to\mathtt{Int}$ $\qquad\leadsto \lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.\mathsf{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

3. Relabeling must be applied also on function bodies:

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \mathtt{Int}\to\mathtt{Int}$ $\quad\rightsquigarrow \lambda^{\mathtt{Int}\to\mathtt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathsf{true} \notin \mathtt{Int}\to\mathtt{Int}$ $\quad\rightsquigarrow \lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.\mathsf{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

3. Relabeling must be applied also on function bodies:

A "daffy" definition of identity:

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

1. Polymorphism needs instantiation:

To apply $\lambda^{\alpha\to\alpha}x.x$ to 42 we must use the instance obtained by the type substitution $\{\mathtt{Int}/_{\alpha}\}$:

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.x)42$$

we *relabel* the function by instantiating its interface.

2. Type-case needs explicit relabeling:

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42 \in \mathtt{Int}\to\mathtt{Int}$ $\rightsquigarrow \lambda^{\mathtt{Int}\to\mathtt{Int}}y.42$

$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\mathsf{true} \notin \mathtt{Int}\to\mathtt{Int}$ $\rightsquigarrow \lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.\mathsf{true}$

Interfaces determine λ-abstractions's types [intrinsic semantics]

3. Relabeling must be applied also on function bodies:

A “daffy” definition of identity:

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

To apply it to 42, relabeling the outer λ by $\{\mathtt{Int}/_{\alpha}\}$ does not suffice:

$$(\lambda^{\alpha\to\alpha}y.42)42$$

is not well typed. The body must be relabeled as well, by applying the $\{\mathtt{Int}/_{\alpha}\}$ yielding: $(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.42)42$

4. Relabeling the body is not always straightforward:

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$\mathtt{Int}\to\mathtt{Int} \qquad \mathtt{Bool}\to\mathtt{Bool}$$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \land (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*
Intuitively: apply $\{\mathtt{Int}/_{\alpha}\}$ and $\{\mathtt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*
Intuitively: apply $\{{}^{\texttt{Int}}/_{\alpha}\}$ and $\{{}^{\texttt{Bool}}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}] \quad\rightsquigarrow\quad \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*
Intuitively: apply $\{{}^{\mathtt{Int}}/_{\alpha}\}$ and $\{{}^{\mathtt{Bool}}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{{}^{\mathtt{Int}}/_{\alpha}\},\{{}^{\mathtt{Bool}}/_{\alpha}\}] \quad\rightsquigarrow\quad \lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.x$$

We applied a set of type substitutions: $t[\sigma_i]_{i\in I} = \bigwedge_{i\in I} t\sigma_i$

4. Relabeling the body is not always straightforward:

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$\mathtt{Int}\to\mathtt{Int} \qquad \mathtt{Bool}\to\mathtt{Bool}$$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*
Intuitively: apply $\{\mathtt{Int}/_{\alpha}\}$ and $\{\mathtt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.
We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*
Intuitively: apply $\{\texttt{Int}/_{\alpha}\}$ and $\{\texttt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}] \quad\rightsquigarrow\quad \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

So it has their intersection.

We can feed the identity to a function which expects argument of that type. But *how do we relabel it?*

Intuitively: apply $\{{}^{\texttt{Int}}\!/_{\alpha}\}$ and $\{{}^{\texttt{Bool}}\!/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{{}^{\texttt{Int}}\!/_{\alpha}\},\{{}^{\texttt{Bool}}\!/_{\alpha}\}] \quad \rightsquigarrow \quad \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$$

We applied a set of type substitutions: $t[\sigma_i]_{i\in I} = \bigwedge_{i\in I} t\sigma_i$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

The identity function $\lambda^{\alpha\to\alpha}x.x$ has both these types:

$$\texttt{Int}\to\texttt{Int} \qquad \texttt{Bool}\to\texttt{Bool}$$

So it has their intersection.
We can feed the the identity to a function which expects argument of that type. But *how do we relabel it?*
Intuitively: apply $\{\texttt{Int}/_{\alpha}\}$ and $\{\texttt{Bool}/_{\alpha}\}$ to the interface and replace it by the intersection of the two instances:

$$(\lambda^{\alpha\to\alpha}x.x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}] \quad\rightsquigarrow\quad \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$$

We applied a *set* of type substitutions: $t[\sigma_i]_{i\in I} = \bigwedge_{i\in I} t\sigma_i$.

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

Consider again the daffy identity $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$.
It also has type

$$(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

Consider again the daffy identity $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$.
It also has type

$$(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})$$

Applying the set of substitutions $[\{\mathtt{Int}/_{\alpha}\},\{\mathtt{Bool}/_{\alpha}\}]$ both to the interface and the body yields an ill-typed term:

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

4. Relabeling the body is not always straightforward:

1. More than one type-substitution needed
2. Relabeling depends on the dynamic type of the argument

Consider again the daffy identity $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$.
It also has type

$$(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$$

Applying the set of substitutions $[\{{}^{\mathtt{Int}}/_{\alpha}\}, \{{}^{\mathtt{Bool}}/_{\alpha}\}]$ both to the interface and the body yields an ill-typed term:

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

Let us see why
it is not well typed

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

In order to type

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \texttt{Int} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Int}$
2. $x : \texttt{Bool} \vdash (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x : \texttt{Bool}$

Both fail because $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is not well typed

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \mathtt{Int} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Int}$
2. $x : \mathtt{Bool} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Bool}$

FAILED

Both fail because $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x$ is not well typed

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \mathtt{Int} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Int}$
2. $x : \mathtt{Bool} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Bool}$

FAILED

Both fail because $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \mathtt{Int} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Int}$
2. $x : \mathtt{Bool} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Bool}$

FAILED

Both fail because $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In our example with $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$ and $[\{{}^{\mathtt{Int}}/_{\alpha}\}, \{{}^{\mathtt{Bool}}/_{\alpha}\}]$:

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \mathtt{Int} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Int}$
2. $x : \mathtt{Bool} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Bool}$

FAILED

Both fail because $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In our example with $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$ and $[\{\mathtt{Int}/_{\alpha}\}, \{\mathtt{Bool}/_{\alpha}\}]$:

In order to type

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x)$$

we must check that it has both types of the interface:

1. $x : \mathtt{Int} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Int}$
2. $x : \mathtt{Bool} \vdash (\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x : \mathtt{Bool}$

FAILED

Both fail because $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x$ is not well typed

Key idea

The relabeling of the body *must change*
according to the type of the parameter

In our example with $(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$ and $[\{\mathtt{Int}/_{\alpha}\},\{\mathtt{Bool}/_{\alpha}\}]$:

- $(\lambda^{\alpha\to\alpha}y.x)$ must be relabeled as $(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)$ when $x : \mathtt{Int}$;
- $(\lambda^{\alpha\to\alpha}y.x)$ must be relabeled as $(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)$ when $x : \mathtt{Bool}$

A new technique

Observation

This “dependent relabeling” is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique

Observation

This “dependent relabeling” is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique: “lazy” relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*

A new technique

Observation

This “dependent relabeling” is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique: “lazy” relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:* To pass the daffy identity to a function that expects arguments of type $(\mathtt{Int}{\to}\mathtt{Int}) \wedge (\mathtt{Bool}{\to}\mathtt{Bool})$ first “lazily” relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{^{\mathtt{Int}}/_{\alpha}\},\{^{\mathtt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

A new technique

Observation

This “dependent relabeling” is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique: “lazy” relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:* To pass the daffy identity to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int}) \wedge (\texttt{Bool}\to\texttt{Bool})$ first “lazily” relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{^{\texttt{Int}}/_{\alpha}\},\{^{\texttt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

- The decoration indicates that the function must be relabeled

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:*
  To pass the daffy identity to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$
  first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{^{\texttt{Int}}/_{\alpha}\},\{^{\texttt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

- The decoration indicates that the function must be relabeled
- The relabeling will be actually propagated to the body of the function at the moment of the reduction (lazy relabeling)

A new technique

Observation

This "dependent relabeling" is the stumbling block for the definition of an explicitly-typed λ-calculus with intersection types.

A new technique: "lazy" relabeling of bodies.

- *Decorate λ-abstractions by sets of type-substitutions:* To pass the daffy identity to a function that expects arguments of type $(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$ first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{^{\mathtt{Int}}/_{\alpha}\},\{^{\mathtt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

- The decoration indicates that the function must be relabeled
- The relabeling will be actually propagated to the body of the function at the moment of the reduction (lazy relabeling)
- The new decoration is statically used by the type system to ensure soundness.

Details follow, but remember we want to program in this language

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e$$

No decorations: We do not want to oblige the programmer to write any explicit type substitution.

Details follow, but remember we want to program in this language

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e {\in} t \,?\, e : e$$

No decorations: We do not want to oblige the programmer to write any explicit type substitution.

The technical development will proceed as follows:

1. Define a calculus with explicit type-substitutions and decorated λ-abstractions.
2. Define an inference system that deduces where to insert explicit type-substitutions in a term of the language above
3. Define a compilation and execution technique thanks to which type substitutions are computed only when strictly necessary (in general, as efficient as a monomorphic execution).

Details follow, but remember we want to program in this language

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \,?\, e : e$$

No decorations: We do not want to oblige the programmer to write any explicit type substitution.

The technical development will proceed as follows:

1. Define a calculus with explicit type-substitutions and decorated λ-abstractions.
2. Define an inference system that deduces where to insert explicit type-substitutions in a term of the language above
3. Define a compilation and execution technique thanks to which type substitutions are computed only when strictly necessary (in general, as efficient as a monomorphic execution).

Before proceeding we can already check our first yardstick:

$$\begin{aligned}
\texttt{even} &= \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})} x \,.\, x{\in}\texttt{Int} \,?\, (x \bmod 2) = 0 : x \\
\texttt{map} &= \mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\ f \,. \\
&\qquad\qquad \lambda^{[\alpha]\to[\beta]} \ell \,.\, \ell{\in}\texttt{nil} \,?\, \texttt{nil} : (f(\pi_1 \ell), mf(\pi_2 \ell))
\end{aligned}$$

A calculus with explicit type-substitutions

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t \,?\, e : e \mid e[\sigma_i]_{i\in I}$$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha}x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/_{\alpha}\}]42$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{{}^{\texttt{Int}}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\}]} x.x)42$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\ ?\ e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/_{\alpha}\}]} x.x)42$
- ✗ $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Bool}/_{\alpha}\}]42$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\ ?\ e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{{}^{\mathtt{Int}}/_{\alpha}\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{{}^{\mathtt{Int}}/_{\alpha}\}]} x.x)42$
- ✗ $(\lambda^{\alpha\to\alpha} x.x)[\{{}^{\mathtt{Bool}}/_{\alpha}\}]42$
- ✗ $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}} y.y3)(\lambda^{\alpha\to\alpha} x.x)$
- ✓ $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}} y.y3)((\lambda^{\alpha\to\alpha} x.x)[\{{}^{\mathtt{Int}}/_{\alpha}\}])$

A calculus with explicit type-substitutions

Explicitly pinpoint where sets of type substitutions are applied:

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t \,?\, e : e \mid e[\sigma_i]_{i\in I}$$

Some examples:

- ✗ $(\lambda^{\alpha\to\alpha} x.x)42$
- ✓ $(\lambda^{\alpha\to\alpha} x.x)[\{\texttt{Int}/\alpha\}]42$
- ✓ $(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\}]} x.x)42$
- ✗ $(\lambda^{\alpha\to\alpha} x.x)[\{\texttt{Bool}/\alpha\}]42$
- ✗ $(\lambda^{(\texttt{Int}\to\texttt{Int})\to\texttt{Int}} y.y3)(\lambda^{\alpha\to\alpha} x.x)$
- ✓ $(\lambda^{(\texttt{Int}\to\texttt{Int})\to\texttt{Int}} y.y3)((\lambda^{\alpha\to\alpha} x.x)[\{\texttt{Int}/\alpha\}])$
- ✓ $(\lambda^{((\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool}))\to t} y.e)((\lambda^{\alpha\to\alpha} x.x)[\{\texttt{Int}/\alpha\}, \{\texttt{Bool}/\alpha\}])$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** *pushes the type substitutions into the decorations of the* λ*'s inside* e

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x\\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}} x.e\\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J})\\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J}\\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$: [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & e@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$
\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})
\end{array}
$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & e@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})
\end{array}$$

Notions of reduction:

$$\begin{array}{rcl}
e[\sigma_j]_{j\in J} & \rightsquigarrow & e@[\sigma_j]_{j\in J} \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)v & \rightsquigarrow & (e@[\sigma_j]_{j\in P})\{v/_x\} \qquad P=\{j{\in}J \mid \exists i{\in}I, \vdash v : t_i\sigma_j\} \\
v{\in}t\,?\,e_1 : e_2 & \rightsquigarrow & \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$\begin{array}{rcl}
x@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & x \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}} x.e \\
(e_1 e_2)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \overset{\text{def}}{=} & e@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})
\end{array}$$

Notions of reduction:

$$\begin{array}{rcl}
e[\sigma_j]_{j\in J} & \rightsquigarrow & e@[\sigma_j]_{j\in J} \\
(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)v & \rightsquigarrow & (e@[\sigma_j]_{j\in P})\{{}^{v}\!/\!{}_{x}\} \qquad P = \{j{\in}J \mid \exists i{\in}I, \vdash v : t_i\sigma_j\} \\
v{\in}t\,?\,e_1 : e_2 & \rightsquigarrow & \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}
\end{array}$$

Reduction semantics

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Relabeling operation $e@[\sigma_j]_{j\in J}$**:** [Pushes σ's down into λ's]

$$x@[\sigma_j]_{j\in J} \overset{\text{def}}{=} x$$

$(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]$ $\overset{\text{def}}{=}$ $\lambda^{\wedge_{i\in I} t_i\to s_i}$ $x.e$

$(e_1 e_2)@[$

$(e{\in}t\,?\,e_1 : e_2)@[$ $: e_2@[\sigma_j]_{j\in J}$

$(e[\sigma_k]_{k\in K})@[$

Only keep the substitutions that make the type of the argument v match at least one input type of the interface

Notions of reduction:

$$e[\sigma_j]_{j\in J} \rightsquigarrow e@[\sigma_j]_{j\in J}$$

$$(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)v \rightsquigarrow (e@[\sigma_j]_{j\in P})\{{}^{v}\!/\!_{x}\} \qquad P = \{j{\in}J \mid \exists i{\in}I, \vdash v : t_i\sigma_j\}$$

$$v{\in}t\,?\,e_1 : e_2 \rightsquigarrow \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}$$

Example

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)$$

Example

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)z$$

Example

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha} x.(\lambda^{\alpha\to\alpha} y.x)x)[\{{}^{\texttt{Int}}\!/\!_{\alpha}\}, \{{}^{\texttt{Bool}}\!/\!_{\alpha}\}]z)42$$

Example

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/_{\alpha}\},\{\texttt{Bool}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z)42$$

$$\rightsquigarrow\quad(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]42$$

$$\rightsquigarrow\quad(\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

$$\rightsquigarrow\quad(\lambda^{\texttt{Int}\to\texttt{Int}}y.42)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z)42$$

$$\rightsquigarrow\quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]42$$

$$\rightsquigarrow\quad (\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\},\underline{\underline{\{{}^{\texttt{Bool}}/_{\alpha}\}}}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

$$\rightsquigarrow\quad (\lambda^{\boxed{\texttt{Int}\to\texttt{Int}}}y.42)42$$

no **Bool** here

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z)42$$

$$\leadsto\quad(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]42$$

$$\leadsto\quad(\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

$$\leadsto\quad(\lambda^{\texttt{Int}\to\texttt{Int}}y.42)42$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

REJECTED

$$\rightsquigarrow \quad (\lambda^{\texttt{Int}\to\texttt{Int}}y.42)42 \qquad\qquad \equiv (((\lambda^{\alpha\to\alpha}y.x)x)@[\{{}^{\texttt{Int}}/_{\alpha}\}])\{{}^{42}/_{x}\}$$

Example

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}z.(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]z)42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x)[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]42$$

$$\rightsquigarrow \quad (\lambda^{\alpha\to\alpha}_{[\{{}^{\texttt{Int}}/_{\alpha}\},\{{}^{\texttt{Bool}}/_{\alpha}\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)42$$

REJECTED

$$\rightsquigarrow \quad (\lambda^{\texttt{Int}\to\texttt{Int}}y.42)42 \qquad \equiv (((\lambda^{\alpha\to\alpha}y.x)x)@[\{{}^{\texttt{Int}}/_{\alpha}\}])\{{}^{42}/_{x}\}$$

$$\rightsquigarrow \quad 42$$

Type system

$$(subsumption)\ \frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2} \qquad (inst)\ \frac{\Gamma \vdash e : t \qquad \sigma_j \sharp \Gamma}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

$$(appl)\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(abstr)\ \frac{\Gamma, x : t_i\sigma_j \vdash e@[\sigma_j] : s_i\sigma_j}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ \begin{array}{l} i \in I \\ j \in J \end{array}$$

[plus the rules for type-case and variables]

Type system

$$(subsumption)\ \frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2} \qquad (inst)\ \frac{\Gamma \vdash e : t \qquad \sigma_j \sharp\ \Gamma}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

$$(appl)\ \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(abstr)\ \frac{\Gamma, x : t_i\sigma_j \vdash e@[\sigma_j] : s_i\sigma_j}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \rightarrow s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \rightarrow s_i\sigma_j}\ \begin{array}{l} i \in I \\ j \in J \end{array}$$

[plus the rules for type-case and variables]

Type system

$$(\textit{subsumption})\quad \frac{\Gamma \vdash e : t_1 \qquad t_1 \leq t_2}{\Gamma \vdash e : t_2} \qquad\qquad (\textit{inst})\quad \frac{\Gamma \vdash e : t \qquad \sigma_j \sharp \Gamma}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

$$(\textit{appl})\quad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(\textit{abstr})\quad \frac{\Gamma, x : t_i \quad \vdash e \qquad : s_i}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e : \bigwedge_{i\in I} \quad t_i \quad \to s_i}\; i \in I$$

[plus the rules for type-case and variables]

Type system

$$(subsumption)\ \frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2} \qquad (inst)\ \frac{\Gamma \vdash e : t \qquad \sigma_j \sharp\, \Gamma}{\Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

$$(appl)\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$(abstr)\ \frac{\Gamma, x : t_i\sigma_j \vdash e@[\sigma_j] : s_i\sigma_j}{\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ \begin{array}{l} i \in I \\ j \in J \end{array}$$

[plus the rules for type-case and variables]

Properties

Theorem (Subject Reduction)

For every term e *and type* t*, if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$.

Properties

Theorem (Subject Reduction)

For every term e *and type* t, *if* $\Gamma \vdash e : t$ *and* $e \leadsto e'$, *then* $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \leadsto e'$.

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e. *If* $\vdash_{BCD} a : t$, *then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$.

Properties

Theorem (Subject Reduction)

For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ be Barendregt, Coppo, and Dezani, typing, and $\lceil e \rceil$ the type erasure of e. If $\vdash_{BCD} a : t$, then $\exists e$ s.t. $\vdash e : t$ and $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

Properties

Theorem (Subject Reduction)

For every term e *and type* t*, if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : t$*.*

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$*.*

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e*. If* $\vdash_{BCD} a : t$*, then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$*.*

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e {\in} t \,?\, e : e \mid e[\sigma_i]_{i\in I}$$

Properties

Theorem (Subject Reduction)

For every term e *and type* t*, if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : t$*.*

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$*.*

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e*. If* $\vdash_{BCD} a : t$*, then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$*.*

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \;\cancel{\mid e[\sigma_i]_{i\in I}}$$

Properties

Theorem (Subject Reduction)

For every term e *and type* t, *if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$, *then* $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e. *If* $\vdash_{BCD} a : t$, *then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t \,?\, e : e \;\cancel{\mid e[\sigma_i]_{i\in I}}$$

satisfies the above theorem and is closed by reduction.

Properties

Theorem (Subject Reduction)

For every term e *and type* t, *if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$, *then* $\Gamma \vdash e' : t$.

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$.

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e. *If* $\vdash_{BCD} a : t$, *then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$.

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid \cancel{e{\in}t\,?\,e : e} \mid \cancel{e[\sigma_i]_{i\in I}}$$

satisfies the above theorem and is closed by reduction, too.

Properties

Theorem (Subject Reduction)

For every term e *and type* t*, if* $\Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e' : t$*.*

Theorem (Progress)

Let e *be a well-typed closed term. If* e *is not a value, then there exists a term* e' *such that* $e \rightsquigarrow e'$*.*

Theorem

Let $\vdash_{BCD}$ *be Barendregt, Coppo, and Dezani, typing, and* $\lceil e \rceil$ *the type erasure of* e*. If* $\vdash_{BCD} a : t$*, then* $\exists e$ *s.t.* $\vdash e : t$ *and* $\lceil e \rceil = a$*.*

Note that

$$e \quad ::= \quad x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_i]_{i\in I}$$

The first n terms ($n = 3, 4, 5$) form an explicitly-typed λ-calculus with intersection types subsuming BCD.

Properties

The definitions we gave:

$$\begin{array}{rcl}
\texttt{even} & = & \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x \\
\texttt{map} & = & \mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\ f\,. \\
 & & \qquad\lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell))
\end{array}$$

are well typed.

Properties

The definitions we gave:

$$\begin{array}{rcl} \texttt{even} & = & \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})} x\,.\, x{\in}\texttt{Int}\,?\,(x \bmod 2) = 0 : x \\ \texttt{map} & = & \mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}\, f\,. \\ & & \qquad \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell)) \end{array}$$

are well typed.

A yardstick for the language

- ✓ Can define both `map` and `even`
- ✓ Can *check* the types specified in the signature
- ? Can *deduce* the type of the partial application `map even`

Inference of explicit type-substitutions

Two problems:

1. **Local type-substitution inference:** Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\ ?\ e : e$$

a sound & complete algorithm that, whenever possible, inserts sets of type-substitutions that make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\ ?\ e : e \mid e[\sigma_j]_{j\in J}$$

Two problems:

① **Local type-substitution inference:** Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e {\in} t \,?\, e : e$$

a sound & complete algorithm that, whenever possible, inserts sets of type-substitutions that make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e {\in} t \,?\, e : e \mid e[\sigma_j]_{j\in J}$$

(and, yes, the type inferred for `map even` is as expected)

Two problems:

1. **Local type-substitution inference:** Given a term of

   $$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t \; ? \; e : e$$

   a sound & complete algorithm that, whenever possible, inserts sets of type-substitutions that make it a well-typed term of

   $$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t \; ? \; e : e \mid e[\sigma_j]_{j\in J}$$

   (and, yes, the type inferred for `map even` is as expected)

2. **Type recostruction:** Given a term

   $$\lambda x.e$$

   find, if possible, a set of type-substitutions $[\sigma_j]_{j\in J}$ such that

   $$\lambda^{\alpha\to\beta}_{[\sigma_j]_{j\in J}} x.e$$

   is well typed

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

No inference for decorations of λ's

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

No inference for decorations of λ's

The reason is purely practical:

- $\lambda^{\alpha\to\alpha} x.3$ must return a static type error

Local Type-Substitution Inference

Given a term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e{\in}t\,?\,e : e$$

Infer whether it is possible to insert sets of type-substitutions in it to make it a well-typed term of

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\,]} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}$$

No inference for decorations of λ's

The reason is purely practical:

- $\lambda^{\alpha\to\alpha}x.3$ must return a static type error
- If we infer decorations, then it can be typed: $\lambda^{\alpha\to\alpha}_{\{\mathtt{Int}/\alpha\}}x.3$

The rule for applications

1. In the type system: [with explicit type-subst.]

$$\textsc{(Appl)}\ \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

The rule for applications

1. In the type system: [with explicit type-subst.]

$$\frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}\ (\textsc{Appl})$$

[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: [with explicit type-subst.]

$$\frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}}\ \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}\ (\textsc{Appl-algorithm})$$

The rule for applications

1. In the type system: [with explicit type-subst.]

$$(\textsc{Appl})\ \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination: [with explicit type-subst.]

$$(\textsc{Appl-algorithm})\ \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}}$$

$t \leq \mathbb{0} \to \mathbb{1}$
$s \leq \mathrm{dom}(t)$

conditions for Typeability

The rule for applications

1. In the type system: [with explicit type-subst.]

$$\textsc{(Appl)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination: [with explicit type-subst.]

$$\textsc{(Appl-algorithm)}\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

The rule for applications

1. **In the type system:** [with explicit type-subst.]

$$\textsc{(Appl)}\quad \frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. **Subsumption elimination:** [with explicit type-subst.]

$$\textsc{(Appl-algorithm)}\quad \frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

3. **Inference of type substitutions** [w/o explicit type-subst.]

$$\textsc{(Appl-inference)}\quad \frac{\exists [\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J} \quad \Gamma \vdash_{\mathcal{I}} e_1 : t \qquad \Gamma \vdash_{\mathcal{I}} e_2 : s}{\Gamma \vdash_{\mathcal{I}} e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j\in J} \leq s[\sigma_i]_{i\in I} \to u\}} \quad \begin{array}{l} t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \\ s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J}) \end{array}$$

The rule for applications

1. In the type system:

[with explicit type-subst.]

(Appl)

$$\frac{\Gamma \vdash e_1 : s \to u \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : u}$$

[The type of the function is subsumed to an arrow and the type of the argument is subsumed to the domain of this arrow].

2. Subsumption elimination:

[with explicit type-subst.]

(Appl-algorithm)

$$\frac{\Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Gamma \vdash_{\mathcal{A}} e_2 : s}{\Gamma \vdash_{\mathcal{A}} e_1 e_2 : \min\{u \mid t \leq s \to u\}} \quad \begin{array}{l} t \leq \mathbb{0} \to \mathbb{1} \\ s \leq \mathrm{dom}(t) \end{array}$$

3. Inference of type substitutions

[w/o explicit type-subst.]

conditions for typeability

(Appl-inference)

$$\frac{\exists [\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J} \qquad \Gamma \vdash_{\mathcal{I}} e_1 : t \qquad \Gamma \vdash_{\mathcal{I}} e_2 : s}{\Gamma \vdash_{\mathcal{I}} e_1 e_2 : \min\{u \mid t[\sigma'_j]_{j\in J} \leq s[\sigma_i]_{i\in I} \to u\}} \quad \begin{array}{l} t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \\ s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J}) \end{array}$$

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of tallying problems

Definition (Type tallying)

Let $C = \{(s_1, t_1), ..., (s_n, t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s, t) \in C$.

Tallying problem

The problem of inferring the type of an application is thus to find for s and t given, $[\sigma_i]_{i\in I}, [\sigma'_j]_{j\in J}$ such that:

$$t[\sigma'_j]_{j\in J} \leq \mathbb{0} \to \mathbb{1} \quad \text{and} \quad s[\sigma_i]_{i\in I} \leq \mathrm{dom}(t[\sigma'_j]_{j\in J})$$

This can be reduced to solving a suite of tallying problems

Definition (Type tallying)

Let $C = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ a *constraint set*. A type-substitution σ is a solution for the *tallying* of C iff $s\sigma \leq t\sigma$ for all $(s, t) \in C$.

A sound and complete set of solutions for every tallying problem can be effectively found in three simple steps.

Step 1: Decompose constraints.

Use the set-theoretic decomposition rules to transform C into a set of constraint sets whose constraints are of the form (α, t) or (t, α).

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 (α, t) or (t, α).

Step 2: Merge constraints on the same variable.

- if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
- if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1 \vee s_2, \alpha)$;

Possibly decompose the new constraints generated by transitivity.

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 (α, t) or (t, α).

Step 2: Merge constraints on the same variable.

- if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
- if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1 \vee s_2, \alpha)$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. in all other constraints in replace every α by $(s \vee \beta) \wedge t$
3. repeat with another constraint

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 (α, t) or (t, α).

Step 2: Merge constraints on the same variable.

- if (α, t_1) and (α, t_2) are in C, then replace them by $(\alpha, t_1 \wedge t_2)$;
- if (s_1, α) and (s_2, α) are in C, then replace them by $(s_1 \vee s_2, \alpha)$;

Possibly decompose the new constraints generated by transitivity.

Step 3: Transform into a set of equations.

After Step 2 we have constraint-sets of the form $\{s_i \leq \alpha_i \leq t_i \mid i \in [1..n]\}$ where α_i are pairwise distinct.

1. select $s \leq \alpha \leq t$ and replace it by $\alpha = (s \vee \beta) \wedge t$ with β fresh.
2. in all other constraints in replace every α by $(s \vee \beta) \wedge t$
3. repeat with another constraint

At the end we have a sets of equations $\{\alpha_i = u_i \mid i \in [1..n]\}$ that (with some care) are *contractive*. By Courcelle there exists a solution, *ie*, a substitution for $\alpha_1, \ldots, \alpha_n$ into (possibly recursive regular) types $t_1, \ldots, t_n$ (in which the fresh β's are free variables).

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}\Big(\bigwedge_{i\in I} t\sigma_i\Big)$$

1. Fix the cardinalities of I and J (at the beginning both 1);

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:

   $\bigwedge_{i\in I}(t\rho_i)\sigma'_i \leq \mathbb{0}\to\mathbb{1}$ and $\bigwedge_{j\in J}(s\rho_j)\sigma'_j \leq \mathrm{dom}(\bigwedge_{i\in I}(t\rho_i)\sigma'_i)$;

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \;\leq\; \mathbb{0}{\to}\mathbb{1} \quad\text{and}\quad \bigwedge_{j\in J} s\sigma_j \;\leq\; \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k{\in}I{\cup}J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable: $(\bigwedge_{i\in I}\ t\rho_i)\sigma \ \leq \mathbb{0}{\to}\mathbb{1}$ and $(\bigwedge_{j\in J}\ s\rho_j)\sigma \ \leq \mathrm{dom}((\bigwedge_{i\in I}\ t\rho_i)\sigma\ )$;

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \;\leq\; \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \;\leq\; \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable: $(\bigwedge_{i\in I}\ t\rho_i)\sigma \;\leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J}\ s\rho_j)\sigma \;\leq \mathrm{dom}((\bigwedge_{i\in I}\ t\rho_i)\sigma\ )$;
3. Solve the tallying problem for
$$\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$$
with $t_1 = \bigwedge_{i\in I} t\rho_i$, $t_2 = \bigwedge_{j\in J} s\rho_j$, and γ fresh

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \operatorname{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:
   $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \operatorname{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$;
3. Solve the tallying problem for
   $$\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$$
   with $t_1 = \bigwedge_{i\in I} t\rho_i$, $t_2 = \bigwedge_{j\in J} s\rho_j$, and γ fresh
   - if it fails at Step 1, then fail.
   - if it fails at Step 2, then change cardinalities (dove-tail)

The application problem

Definition (Inference application problem)

Given s and t types, find $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that:

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0}\to\mathbb{1} \quad \text{and} \quad \bigwedge_{j\in J} s\sigma_j \leq \mathrm{dom}(\bigwedge_{i\in I} t\sigma_i)$$

1. Fix the cardinalities of I and J (at the beginning both 1);
2. Split each substitution σ_k (for $k\in I\cup J$) in two: $\sigma_k = \rho_k \circ \sigma'_k$ where ρ_k is a renaming substitution mapping each variable of the domain of σ_k into a fresh variable:
   $(\bigwedge_{i\in I} t\rho_i)\sigma \leq \mathbb{0}\to\mathbb{1}$ and $(\bigwedge_{j\in J} s\rho_j)\sigma \leq \mathrm{dom}((\bigwedge_{i\in I} t\rho_i)\sigma)$;
3. Solve the tallying problem for
   $$\{(t_1, \mathbb{0}\to\mathbb{1}), (t_1, t_2\to\gamma)\}$$
   with $t_1 = \bigwedge_{i\in I} t\rho_i$, $t_2 = \bigwedge_{j\in J} s\rho_j$, and γ fresh
   - if it fails at Step 1, then fail.
   - if it fails at Step 2, then change cardinalities (dove-tail)

➡ **Every solution for γ is a solution for the application.**

Example: map even

Start with the following tallying problem:

$$\{(\alpha_1\to\beta_1)\to[\alpha_1]\to[\beta_1] \leq t\to\gamma\}$$

where $t = (\mathtt{Int}\to\mathtt{Bool})\wedge(\alpha\setminus\mathtt{Int}\to\alpha\setminus\mathtt{Int})$ is the type of `even`

Example: `map even`

Start with the following tallying problem:

$$\{(\alpha_1\to\beta_1)\to[\alpha_1]\to[\beta_1] \leq t\to\gamma\}$$

where $t = (\mathtt{Int}\to\mathtt{Bool})\wedge(\alpha\setminus\mathtt{Int}\to\alpha\setminus\mathtt{Int})$ is the type of `even`

- At step 2 the algorithm generates 9 constraint-sets: one is unsatisfiable ($t \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1{\leq}\mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1{\leq}\mathtt{Int}\ ,\ \mathtt{Bool}{\leq}\beta_1\}$,
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1{\leq}\alpha\setminus\mathtt{Int}\ ,\ \alpha\setminus\mathtt{Int}{\leq}\beta_1\}$,
  $\{\gamma \geq [\alpha_1]\to[\beta_1]\ ,\ \alpha_1{\leq}\alpha\vee\mathtt{Int}\ ,\ (\alpha\setminus\mathtt{Int})\vee\mathtt{Bool}{\leq}\beta_1\}$;

Example: `map even`

Start with the following tallying problem:

$$\{(\alpha_1\to\beta_1)\to[\alpha_1]\to[\beta_1] \le t\to\gamma\}$$

where $t = (\mathtt{Int}\to\mathtt{Bool})\wedge(\alpha\backslash\mathtt{Int}\to\alpha\backslash\mathtt{Int})$ is the type of `even`

- At step 2 the algorithm generates 9 constraint-sets: one is unsatisfiable ($t \le \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \ge [\alpha_1]\to[\beta_1]\ ,\ \alpha_1\le\mathbb{0}\}$,
  $\{\gamma \ge [\alpha_1]\to[\beta_1]\ ,\ \alpha_1\le\mathtt{Int}\ ,\ \mathtt{Bool}\le\beta_1\}$,
  $\{\gamma \ge [\alpha_1]\to[\beta_1]\ ,\ \alpha_1\le\alpha\backslash\mathtt{Int}\ ,\ \alpha\backslash\mathtt{Int}\le\beta_1\}$,
  $\{\gamma \ge [\alpha_1]\to[\beta_1]\ ,\ \alpha_1\le\alpha\vee\mathtt{Int}\ ,\ (\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}\le\beta_1\}$;
- Four solutions for γ:
  $\{\gamma=[\,]\to[\,]\}$,
  $\{\gamma = [\mathtt{Int}]\to[\mathtt{Bool}]\}$,
  $\{\gamma = [\alpha\backslash\mathtt{Int}]\to[\alpha\backslash\mathtt{Int}]\}$,
  $\{\gamma = [\alpha\vee\mathtt{Int}]\to[(\alpha\backslash\mathtt{Int})\vee\mathtt{Bool}]\}$.

Example: `map even`

Start with the following tallying problem:

$$\{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq t \to \gamma\}$$

where $t = (\mathtt{Int} \to \mathtt{Bool}) \wedge (\alpha \setminus \mathtt{Int} \to \alpha \setminus \mathtt{Int})$ is the type of `even`

- At step 2 the algorithm generates 9 constraint-sets: one is unsatisfiable ($t \leq \mathbb{0}$); four are implied by the others; remain
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathbb{0}\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \mathtt{Int}\ ,\ \mathtt{Bool} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \setminus \mathtt{Int}\ ,\ \alpha \setminus \mathtt{Int} \leq \beta_1\}$,
  $\{\gamma \geq [\alpha_1] \to [\beta_1]\ ,\ \alpha_1 \leq \alpha \vee \mathtt{Int}\ ,\ (\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool} \leq \beta_1\}$;
- Four solutions for γ:
  $\{\gamma = [] \to []\}$,
  $\{\gamma = [\mathtt{Int}] \to [\mathtt{Bool}]\}$,
  $\{\gamma = [\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]\}$,
  $\{\gamma = [\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}]\}$.
- The last two are minimal and we take their intersection:
  $\{\gamma = ([\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]) \wedge ([\alpha \vee \mathtt{Int}] \to [(\alpha \setminus \mathtt{Int}) \vee \mathtt{Bool}])\}$

On completeness and decidability

The algorithm produces a set of solutions that is sound (it finds only correct solutions) and complete (any other solution can be derived from them).

On completeness and decidability

The algorithm produces a set of solutions that is sound (it finds only correct solutions) and complete (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type-reconstruction for intersection types since we have recursive types).

On completeness and decidability

The algorithm produces a set of solutions that is sound (it finds only correct solutions) and complete (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type-reconstruction for intersection types since we have recursive types).

Completeness: For every solution of the inference problem, our algorithm finds an equivalent or more general solution. However, this solution is not necessary the first solution found.

In a dully execution of the algorithm on `map even` the good solution is the second one.

On completeness and decidability

The algorithm produces a set of solutions that is sound (it finds only correct solutions) and complete (any other solution can be derived from them).

Decidability: The algorithm is a semi-decision procedure. We conjecture decidability (N.B.: the problem is unrelated to type-reconstruction for intersection types since we have recursive types).

Completeness: For every solution of the inference problem, our algorithm finds an equivalent or more general solution. However, this solution is not necessary the first solution found.

In a dully execution of the algorithm on `map even` the good solution is the second one.

Principality: This raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist?

Type reconstruction

- Solve *sets* of contraint-sets by the tallying algorithm:

$$\frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}} \qquad \frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} e : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.e : \alpha \to \beta \rightsquigarrow \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} e_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} e_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} e_1 e_2 : \alpha \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}} \quad + \quad \text{rule for typecase}$$

Type reconstruction

- Solve *sets* of contraint-sets by the tallying algorithm:

$$\frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}} \qquad \frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} e : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.e : \alpha \to \beta \rightsquigarrow \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} e_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} e_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} e_1 e_2 : \alpha \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}} \quad + \quad \text{rule for typecase}$$

- Sound. it's a variant: fix interfaces and infer decorations

$$\lambda^{\alpha\to\beta}_{[?]} x.e$$

  Not complete: reconstruction is undecidable

Type reconstruction

- Solve *sets* of contraint-sets by the tallying algorithm:

$$\frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}} \qquad \frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} e : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.e : \alpha \to \beta \rightsquigarrow \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}$$

$$\frac{\Gamma \vdash_{\mathcal{R}} e_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} e_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} e_1 e_2 : \alpha \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}} \quad + \quad \text{rule for typecase}$$

- Sound. it's a variant: fix interfaces and infer decorations

$$\lambda^{\alpha\to\beta}_{[?]} x.e$$

  Not complete: reconstruction is undecidable

- It types more than ML

$$\lambda x.xx : \mu X.(\alpha \wedge (X \to \beta)) \to \beta \qquad (\leq \alpha \wedge (\alpha \to \beta)) \to \beta)$$

  for functions typable in ML it deduces a type at least as good:

$$map : ((\alpha \to \beta) \to [\alpha] \to [\beta]) \wedge ((\mathbb{0} \to \mathbb{1}) \to [] \to [])$$

Type Reconstruction Algorithm

$$\frac{}{\Gamma \vdash_{\mathcal{R}} c : b_c \leadsto \{\varnothing\}}(\textsc{R-const}) \qquad \frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \leadsto \{\varnothing\}}(\textsc{R-var})$$

$$\frac{\Gamma \vdash_{\mathcal{R}} m_1 : t_1 \leadsto \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \leadsto \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} m_1 m_2 : \alpha \leadsto \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \leq t_2 \to \alpha)\}\}}(\textsc{R-appl})$$

$$\frac{\Gamma, x : \alpha \vdash_{\mathcal{R}} m : t \leadsto \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \lambda x.m : \alpha \to \beta \leadsto \mathcal{S} \sqcap \{\{(t \leq \beta)\}\}}(\textsc{R-abstr})$$

$(\textsc{R-case})$

$$\frac{\begin{array}{rl} \mathcal{S} = & (\mathcal{S}_0 \sqcap \{\{(t_0 \leq \mathbb{0})\}\}) \\ \sqcup & (\mathcal{S}_0 \sqcap \mathcal{S}_1 \sqcap \{\{(t_0 \leq t), (t_1 \leq \alpha)\}\}) \\ \sqcup & (\mathcal{S}_0 \sqcap \mathcal{S}_2 \sqcap \{\{(t_0 \leq \neg t), (t_2 \leq \alpha)\}\}) \\ \sqcup & (\mathcal{S}_0 \sqcap \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1 \vee t_2 \leq \alpha)\}\}) \end{array} \\ \Gamma \vdash_{\mathcal{R}} m_0 : t_0 \leadsto \mathcal{S}_0 \qquad \Gamma \vdash_{\mathcal{R}} m_1 : t_1 \leadsto \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \leadsto \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} (m_0 {\in} t \,?\, m_1 : m_2) : \alpha \leadsto \mathcal{S}}$$

where α, α_i and β in each rule are fresh type variables.

Efficient evaluation

Monomorphic language

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e \in t\, ?\, e : e \\
v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle
\end{array}
$$

Monomorphic language

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}x.e \mid ee \mid e \in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^{t}x.e, \mathcal{E} \rangle
\end{array}
$$

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash_{\mathrm{m}} \lambda^{t}x.e \Downarrow \langle \lambda^{t}x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash_{\mathrm{m}} e_1 \Downarrow \langle \lambda^{t}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathrm{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathrm{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathrm{m}} e_1 e_2 \Downarrow v}
$$

Monomorphic language

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}x.e \mid ee \mid e \in t \mathrel{?} e : e \\
v & ::= & c \mid \langle \lambda^{t}x.e, \mathcal{E} \rangle
\end{array}
$$

save the environment

$$
(\text{Closure})\ \frac{}{\mathcal{E} \vdash_{\mathrm{m}} \lambda^{t}x.e \Downarrow \langle \lambda^{t}x.e, \mathcal{E} \rangle}
$$

$$
(\text{Apply})\ \frac{\mathcal{E} \vdash_{\mathrm{m}} e_1 \Downarrow \langle \lambda^{t}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathrm{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathrm{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathrm{m}} e_1 e_2 \Downarrow v}
$$

Monomorphic language

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle \end{array}$$

save the environment

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash_{\mathsf{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash_{\mathsf{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathsf{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathsf{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathsf{m}} e_1 e_2 \Downarrow v}$$

restore the environment

Monomorphic language

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^t x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^t x.e, \mathcal{E} \rangle \end{array}$$

$$\text{(CLOSURE)}\ \frac{}{\mathcal{E} \vdash_{\mathrm{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}$$

$$\text{(APPLY)}\ \frac{\mathcal{E} \vdash_{\mathrm{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash_{\mathrm{m}} e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash_{\mathrm{m}} e \Downarrow v}{\mathcal{E} \vdash_{\mathrm{m}} e_1 e_2 \Downarrow v}$$

$$\text{(TYPECASE TRUE)}\ \frac{\mathcal{E} \vdash_{\mathrm{m}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathrm{m}} t \qquad \mathcal{E} \vdash_{\mathrm{m}} e_2 \Downarrow v}{\mathcal{E} \vdash_{\mathrm{m}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\text{(TYPECASE FALSE)}\ \frac{\mathcal{E} \vdash_{\mathrm{m}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathrm{m}} t \qquad \mathcal{E} \vdash_{\mathrm{m}} e_3 \Downarrow v}{\mathcal{E} \vdash_{\mathrm{m}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\begin{array}{rcl} c \in_{\mathrm{m}} t & \overset{\mathrm{def}}{=} & \{c\} \leq t \\ \langle \lambda^s x.e, \mathcal{E} \rangle \in_{\mathrm{m}} t & \overset{\mathrm{def}}{=} & s \leq t \end{array}$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$e \quad ::= \quad c \mid x \mid \lambda^{t}_{\sigma_I} x.e \mid ee \mid e \in t\, ?\, e : e \mid e\sigma_I$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I} x.e \mid ee \mid e \in t\,?\,e : e \mid e\sigma_I \\ v & ::= & c \mid \langle \lambda^{t}_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I ; \mathcal{E} \vdash_{\mathsf{p}} \lambda^{t}_{\sigma_J} x.e \Downarrow \langle \lambda^{t}_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle}$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I} x.e \mid ee \mid e \in t\, ?\, e : e \mid e\sigma_I \\
v & ::= & c \mid \langle \lambda^{t}_{\sigma_J} x.e, \mathcal{E}, \sigma_I \rangle
\end{array}
$$

save the environment

$$
(\textsc{Closure})\ \frac{}{\sigma_I; \boxed{\mathcal{E}} \vdash_{\mathsf{p}} \lambda^{t}_{\sigma_J} x.e \Downarrow \langle \lambda^{t}_{\sigma_J} x.e, \boxed{\mathcal{E}}, \sigma_I \rangle}
$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\
v & ::= & c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle
\end{array}
$$

save the environment

$$
(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathsf{p}} \lambda^{t}_{\sigma_J}x.e \Downarrow \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle}
$$

save current type-substitutions

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}\qquad(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}\qquad(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

$$(\textsc{Apply})\ \frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle\quad\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_2\Downarrow v_0\quad\sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid\exists\ell\in L : v_0\in_{\mathrm{p}}s_\ell\sigma_j\}$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lll} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}\qquad(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

restore the environment

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\boxed{\mathcal{E}'},\sigma_H\rangle\qquad\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_2\Downarrow v_0\qquad\sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid\exists\ell\in L: v_0\in_{\mathrm{p}}s_\ell\sigma_j\}$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle} \qquad (\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

restore the environment

$$(\textsc{Apply})\ \frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid\exists\ell\in L: v_0\in_{\mathrm{p}}s_\ell\sigma_j\}$

restore the type substitutions

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}\qquad(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle\quad\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_2\Downarrow v_0\quad\sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid\exists\ell\in L: v_0\in_{\mathrm{p}}s_\ell\sigma_j\}$

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e : e \mid e\sigma_I \\ v & ::= & c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} \lambda^{t}_{\sigma_J}x.e \Downarrow \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle} \qquad (\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e\sigma_J\Downarrow v}$$

$$(\textsc{Apply})$$

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1\Downarrow \langle \lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e, \mathcal{E}', \sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}', x\mapsto v_0\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J \mid \exists \ell\in L : v_0\in_{\mathrm{p}} s_\ell\sigma_j\}$

Problem:

At every application compute σ_P:

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\ v & ::= & c \mid \langle \lambda^{t}_{\sigma_J}x.e, \mathcal{E}, \sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} \lambda^{t}_{\sigma_J}x.e \Downarrow \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle} \qquad (\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e\sigma_J\Downarrow v}$$

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}} e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}} e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid \exists\ell\in L: v_0\in_{\mathrm{p}} s_\ell\sigma_j\}$

Problem:

At every application compute σ_P:

1. **compose** of two sets of type-substitution

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\
v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle
\end{array}
$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}
\qquad
(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

(Apply)

$$\frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle\qquad\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_2\Downarrow v_0\qquad\sigma_P;\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid\exists\ell\in L: v_0\in_{\mathrm{p}}s_\ell\sigma_j\}$

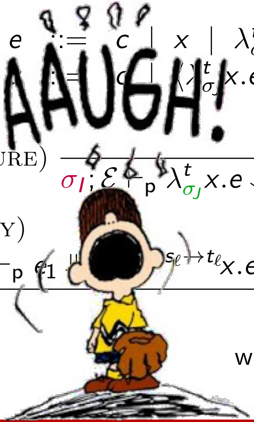
Problem:

At every application compute σ_P:

1. **compose** of two sets of type-substitution
2. **select** the substitutions compatible with the argument v_0

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)



$$e ::= c \mid x \mid \lambda^t_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I$$

[illegible] $::= c \mid \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle$

(CLOSURE) $$\frac{}{\sigma_I;\mathcal{E}\vdash_p \lambda^t_{\sigma_J}x.e \Downarrow \langle\lambda^t_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}$$

(INSTANCE) $$\frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_p e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_p e\sigma_J\Downarrow v}$$

(APPLY)

$\sigma_I;\mathcal{E}\vdash_p e_1$ [illegible] $s_\ell\to t_\ell\, x.e,\mathcal{E}',\sigma_H\rangle \quad \sigma_I;\mathcal{E}\vdash_p e_2\Downarrow v_0 \quad \sigma_P;\mathcal{E}',x\mapsto v_0\vdash_p e\Downarrow v$

$$\overline{\qquad\sigma_I;\mathcal{E}\vdash_p e_1e_2\Downarrow v\qquad}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J \mid \exists\ell\in L: v_0\in_p s_\ell\sigma_j\}$

Problem:

At every application compute σ_P:

1. **compose** of two sets of type-substitution
2. **select** the substitutions compatible with the argument v_0

Polymorphic language: naive implementation

(σ_I short for $[\sigma_i]_{i\in I}$)

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\sigma_I}x.e \mid ee \mid e\in t\,?\,e:e \mid e\sigma_I \\ v & ::= & c \mid \langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}\lambda^{t}_{\sigma_J}x.e\Downarrow\langle\lambda^{t}_{\sigma_J}x.e,\mathcal{E},\sigma_I\rangle}\qquad(\textsc{Instance})\ \frac{\sigma_I\circ\sigma_J;\mathcal{E}\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e\sigma_J\Downarrow v}$$

$$(\textsc{Apply})\ \frac{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1\Downarrow\langle\lambda^{\wedge_{\ell\in L}s_\ell\to t_\ell}_{\sigma_K}x.e,\mathcal{E}',\sigma_H\rangle\quad\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_2\Downarrow v_0\quad\boxed{\sigma_P};\mathcal{E}',x\mapsto v_0\vdash_{\mathrm{p}}e\Downarrow v}{\sigma_I;\mathcal{E}\vdash_{\mathrm{p}}e_1e_2\Downarrow v}$$

where $\sigma_J=\sigma_H\circ\sigma_K$ and $P=\{j\in J\mid\exists\ell\in L: v_0\in_{\mathrm{p}}s_\ell\sigma_j\}$

Solution:

Compute compositions and selections lazily.

Intermediate language as compilation target

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t} x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^{t} x.e, \mathcal{E} \rangle \end{array}$$

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t} x.e \Downarrow \langle \lambda^{t} x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t} x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v} \qquad (\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\begin{array}{rcl} c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\ \langle \lambda^{s} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s \leq t \end{array}$$

Intermediate language as compilation target

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^t_\Sigma x.e \mid ee \mid e \in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^t_\Sigma x.e, \mathcal{E} \rangle \\
\Sigma & ::= & \sigma_I \mid \texttt{comp}(\Sigma, \Sigma') \mid \texttt{sel}(x, t, \Sigma)
\end{array}
$$

symbolic substitutions

$$
(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathcal{E} \rangle}
$$

$$
(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^t x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}
$$

$$
\overset{(\textsc{Typecase True})}{\frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}}
\qquad
\overset{(\textsc{Typecase False})}{\frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}}
$$

$$
\begin{array}{rcl}
c \in t & \overset{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^s x.e, \mathcal{E} \rangle \in t & \overset{\text{def}}{=} & s \leq t
\end{array}
$$

Intermediate language as compilation target

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle \\ \Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma) \end{array}$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\begin{array}{rcl} c \in t & \overset{\text{def}}{=} & \{c\} \leq t \\ \langle \lambda^{s} x.e, \mathcal{E} \rangle \in t & \overset{\text{def}}{=} & s \leq t \end{array}$$

Intermediate language as compilation target

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e\in t\,?\,e : e \\
v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}\rangle \\
\Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma,\Sigma') \mid \mathtt{sel}(x,t,\Sigma)
\end{array}
$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}\rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}'\rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}
\qquad
(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$
\begin{array}{rcl}
c \in t & \overset{\text{def}}{=} & \{c\} \leq t \\
\langle \lambda^{s} x.e, \mathcal{E}\rangle \in t & \overset{\text{def}}{=} & s \leq t
\end{array}
$$

Intermediate language as compilation target

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle \\ \Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma) \end{array}$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

watch here !

$$\begin{array}{rcl} c \in t & \stackrel{\text{def}}{=} & \{c\} \leq t \\ \langle \lambda^{s} x.e, \mathcal{E} \rangle \in t & \stackrel{\text{def}}{=} & s \leq t \end{array}$$

Intermediate language as compilation target

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle \\ \Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma) \end{array}$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$\begin{array}{cc} (\textsc{Typecase True}) & (\textsc{Typecase False}) \\ \dfrac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v} & \dfrac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\,?\,e_2 : e_3 \Downarrow v} \end{array}$$

$$\begin{array}{rcl} c \in t & \overset{\text{def}}{=} & \{c\} \leq t \\ \langle \lambda^{s}_{\Sigma}x.e, \mathcal{E} \rangle \in t & \overset{\text{def}}{=} & s(\mathsf{eval}(\mathcal{E}, \Sigma)) \leq t \end{array}$$

Intermediate language as compilation target

$$\begin{array}{lcl} e & ::= & c \mid x \mid \lambda^{t}_{\Sigma}x.e \mid ee \mid e \in t\, ?\, e : e \\ v & ::= & c \mid \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle \\ \Sigma & ::= & \sigma_I \mid \texttt{comp}(\Sigma, \Sigma') \mid \texttt{sel}(x, t, \Sigma) \end{array}$$

symbolic substitutions

$$(\textsc{Closure})\ \frac{}{\mathcal{E} \vdash \lambda^{t}_{\Sigma}x.e \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E} \rangle}$$

$$(\textsc{Apply})\ \frac{\mathcal{E} \vdash e_1 \Downarrow \langle \lambda^{t}_{\Sigma}x.e, \mathcal{E}' \rangle \qquad \mathcal{E} \vdash e_2 \Downarrow v_0 \qquad \mathcal{E}', x \mapsto v_0 \vdash e \Downarrow v}{\mathcal{E} \vdash e_1 e_2 \Downarrow v}$$

$$(\textsc{Typecase True})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \in t \qquad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\, ?\, e_2 : e_3 \Downarrow v}$$

$$(\textsc{Typecase False})\ \frac{\mathcal{E} \vdash e_1 \Downarrow v_0 \qquad v_0 \notin t \qquad \mathcal{E} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash e_1 \in t\, ?\, e_2 : e_3 \Downarrow v}$$

The only difference!

$$c \in t \overset{\text{def}}{=} \{c\} \leq t$$

$$\langle \lambda^{s}_{\Sigma}x.e, \mathcal{E} \rangle \in t \overset{\text{def}}{=} s(\text{eval}(\mathcal{E}, \Sigma)) \leq t$$

Compilation

1. Compile into the intermediate language

$$
\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\mathtt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t \mathrel{?} e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \mathrel{?} [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}
$$

Compilation

1. Compile into the intermediate language

$$\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^{t}_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^{t}_{\mathtt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t \mathrel{?} e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \mathrel{?} [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}$$

2. For $\langle \lambda^{s}_{\Sigma} x.e, \mathcal{E} \rangle \in t \stackrel{\text{def}}{=} s(\text{eval}(\mathcal{E}, \Sigma)) \leq t$ we have $s(\text{eval}(\mathcal{E}, \Sigma)) \neq s$ *only if* $\lambda^{s}_{\Sigma} x.e$ results from the partial application of a polymorphic function (*ie*, in s there occur free variables bound in the context).

Compilation

1. Compile into the intermediate language

$$\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^{t}_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^{t}_{\mathtt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t \mathrel{?} e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \mathrel{?} [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}$$

2. For $\langle \lambda^{s}_{\Sigma} x.e, \mathcal{E}\rangle \in t \stackrel{\text{def}}{=} s(\text{eval}(\mathcal{E},\Sigma)) \leq t$ we have $s(\text{eval}(\mathcal{E},\Sigma)) \neq s$ *only if* $\lambda^{s}_{\Sigma} x.e$ results from the partial application of a polymorphic function (*ie*, in s there occur free variables bound in the context).

Execution is slowed *only* when testing the type of the result of a partial application of a polymorphic function.

Compilation

1. Compile into the intermediate language

$$\begin{array}{rcl}
[\![x]\!]_\Sigma & = & x \\
[\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\texttt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\texttt{sel}(x,t,\texttt{comp}(\Sigma,\sigma_I))} \\
[\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\
[\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\texttt{comp}(\Sigma,\sigma_I)} \\
[\![e_1 \in t \mathrel{?} e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \mathrel{?} [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma
\end{array}$$

2. For $\langle \lambda^s_\Sigma x.e, \mathcal{E} \rangle \in t \overset{\text{def}}{=} s(\text{eval}(\mathcal{E}, \Sigma)) \leq t$ we have $s(\text{eval}(\mathcal{E}, \Sigma)) \neq s$ *only if* $\lambda^s_\Sigma x.e$ results from the partial application of a polymorphic function (*ie*, in s there occur free variables bound in the context).

   Execution is slowed *only* when testing the type of the result of a partial application of a polymorphic function.

3. This holds also with products (used to encode lists records and XML), whose testing accounts for most of the execution time.

Conclusion

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The polymorphic extension of $\mathbb{C}$Duce is being implemented. Future applications: polymorphic extensions of XQuery and embedding some of this type machinery in ML.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The polymorphic extension of $\mathbb{C}$Duce is being implemented. Future applications: polymorphic extensions of XQuery and embedding some of this type machinery in ML.

Runtime: Relabeling cannot be avoided but it is materialized only in case of partial polymorphic applications that end up in type-cases, that is, just when it is needed.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The polymorphic extension of $\mathbb{C}\mathrm{Duce}$ is being implemented. Future applications: polymorphic extensions of XQuery and embedding some of this type machinery in ML.

Runtime: Relabeling cannot be avoided but it is materialized only in case of partial polymorphic applications that end up in type-cases, that is, just when it is needed.

Implementation: Subtyping of polymorphic types require minimal modifications to the implementation. Existing data structures (e.g., binary decision trees with lazy unions) and optimizations mostly transpose smoothly.

Theory: All the theoretical machinery necessary to design and implement a programming language is there. The practical relevance of the open theoretical issues is negligible.

Languages: The polymorphic extension of $\mathbb{C}$Duce is being implemented. Future applications: polymorphic extensions of XQuery and embedding some of this type machinery in ML.

Runtime: Relabeling cannot be avoided but it is materialized only in case of partial polymorphic applications that end up in type-cases, that is, just when it is needed.

Implementation: Subtyping of polymorphic types require minimal modifications to the implementation. Existing data structures (e.g., binary decision trees with lazy unions) and optimizations mostly transpose smoothly.

Type reconstruction: Full usage needs more research, expecially about the production of human readable types and helpful error messages, but it is mature enough to use it to type local functions.