

CDuce, an XML-Oriented Programming Language: From Theory to Practice

Giuseppe Castagna

CNRS
Université Paris 7 - Denis Diderot

Natal, 24th of May, 2007

Outline of the talk

1 XML Programming in CDuce

- XML Regular Expression Types and Patterns
- XML Programming in CDuce
- Nine good reasons to use Regular Expression types/patterns
- Tools on top of CDuce

2 Theoretical Foundations

- Semantic subtyping
- Subtyping algorithms
- CDuce functional core
- Extension to other type constructors

3 Beyond XML.

- Application of the theory to the π -calculus
- New results on Milner's encoding

Outline of the talk

1 XML Programming in CDuce

- XML Regular Expression Types and Patterns
- XML Programming in CDuce
- Nine good reasons to use Regular Expression types/patterns
- Tools on top of CDuce

2 Theoretical Foundations

- Semantic subtyping
- Subtyping algorithms
- CDuce functional core
- Extension to other type constructors

3 Beyond XML.

- Application of the theory to the π -calculus
- New results on Milner's encoding

Outline of the talk

1 XML Programming in CDuce

- XML Regular Expression Types and Patterns
- XML Programming in CDuce
- Nine good reasons to use Regular Expression types/patterns
- Tools on top of CDuce

2 Theoretical Foundations

- Semantic subtyping
- Subtyping algorithms
- CDuce functional core
- Extension to other type constructors

3 Beyond XML.

- Application of the theory to the π -calculus
- New results on Milner's encoding

Outline of the talk

1 XML Programming in CDuce

- XML Regular Expression Types and Patterns
- XML Programming in CDuce
- Nine good reasons to use Regular Expression types/patterns
- Tools on top of CDuce

2 Theoretical Foundations

- Semantic subtyping
- Subtyping algorithms
- CDuce functional core
- Extension to other type constructors

3 Beyond XML.

- Application of the theory to the π -calculus
- New results on Milner's encoding

PART 1: XML PROGRAMMING IN CDUCE

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, Xstatic
 - 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, Xstatic
 - 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, Xstatic
 - 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, Xstatic
 - 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, Xstatic
 - XQuery
 - C_ω (Microsoft)
 - ...

Presentation of CDuce

Features:

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

Inteded use:

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

Status:

- Public release available (0.4.1): documentation and tutorials.
- Integration with standards
 - Internally: Unicode, XML, Namespaces, XML Schema
 - Externally: DTD, WSDL
- Several tools: graphical queries, code embedding (*à la php*)

Presentation of CDuce

Features:

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

Inteded use:

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

Status:

- Public release available (0.4.1): documentation and tutorials.
- Integration with standards
 - Internally: Unicode, XML, Namespaces, XML Schema
 - Externally: DTD, WSDL
- Several tools: graphical queries, code embedding (*à la* php)

Presentation of CDuce

Features:

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

Inteded use:

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

Status:

- Public release available (0.4.1): documentation and tutorials.
- Integration with standards
 - Internally: Unicode, XML, Namespaces, XML Schema
 - Externally: DTD, WSDL
- Several tools: graphical queries, code embedding (*à la php*)

Presentation of CDuce

Features:

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

Inteded use:

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

Status:

- Public release available (0.4.1): documentation and tutorials.
- Integration with standards
 - Internally: Unicode, XML, Namespaces, XML Schema
 - Externally: DTD, WSDL
- Several tools: graphical queries, code embedding (*à la* php)

Presentation of CDuce

Features:

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

Inteded use:

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

Status:

- Public release available (0.4.1): documentation and tutorials.
- Integration with standards
 - Internally: Unicode, XML, Namespaces, XML Schema
 - Externally: DTD, WSDL
- Several tools: graphical queries, code embedding (*à la* php)

Used both for teaching and in production code.

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

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

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

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**

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 syntactic sugar for

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

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

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 syntactic sugar for

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

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

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 syntactic sugar for

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

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

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 syntactic sugar for

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

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

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 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

fun length (x:(List,Int)) : Int =
  match x with
  | ('nil', n) -> n
  | ((_,t), n) -> length(t,n+1)
```

So patterns are values with

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

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

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

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**,

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**,

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:

• use for types the same constructors as for values

• use values to denote concrete types

• use values to denote abstract types

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](#).

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](#).

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](#).

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 (\backslash):

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

fun length (x :(List,Int)) : Int =
  match x with
  | ('nil', n) -> n
  | ((_,t), n) -> length(t,n+1)
```

Which types should we start from?

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

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

fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

Which types should we start from?

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

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

fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

Which types should we start from?

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

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

fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

To type this function we need basic types products, singletons,...

$$t ::= \text{Int} \mid v \mid (t, t) \mid$$

Which types should we start from?

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

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

fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

To type this function we need basic types products, singletons,...

$t ::= \text{Int} \mid v \mid (t, t) \mid t|t \mid t\&t \mid t\backslash t \mid \text{Empty} \mid \text{Any}$

but also boolean type constructors.

.

Which types should we start from?

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

fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

To type this function we need basic types products, singletons,...

$t ::= \text{Int} \mid v \mid (t, t) \mid t|t \mid t\&t \mid t\backslash t \mid \text{Empty} \mid \text{Any}$

but also boolean type constructors.

Let us type the function.

Which types should we start from?

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

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

```
fun length (x :(List,Int)) : Int =  
  match x with  
  | ('nil', n) -> n  
  | ((_,t), n) -> length(t,n+1)
```

Which types should we start from?

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


 (this is a type)

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

```
fun length (x :(List,Int)) : Int =  
  match x with  
  | ('nil', n) -> n  
  | ((_,t), n) -> length(t,n+1)
```

Which types should we start from?

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

 (this is a type)

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

```
fun length (x : (List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

The first branch is executed only for values and are both in $(\text{List}, \text{Int})$ **and** in $\{('nil, n)\}$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ and $\{p\} = \{v \mid v \text{ matches pattern } p\}$
 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

The first branch is executed only for values and are both in
 (List,Int) **and** in $\{('nil,n)\} = ('nil,Any)$

Which types should we start from?

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

 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

The first branch is executed only for values and are both in
(List,Int) **and** in $\{('nil,n)\}$

(List,Int) & ('nil,Any)

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ and $\{p\} = \{v \mid v \text{ matches pattern } p\}$
 (this is a type)


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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

The first branch is executed only for values and are both in
 (List,Int) **and** in $\{('nil,n)\}$

```
(List,Int) & ('nil,Any) = ('nil,Int)
```

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ and $\{p\} = \{v \mid v \text{ matches pattern } p\}$
 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n                               Int
  | ((_,t), n) -> length(t,n+1)
```

The first branch is executed only for values and are both in
 (List,Int) **and** in $\{('nil,n)\}$

```
(List,Int) & ('nil,Any) = ('nil,Int)
```


Which types should we start from?

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

 (this is a type)

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

```
fun length (x : (List,Int)) : Int =
  match x with
  | ('nil', n) -> n                               Int
  | ((_,t), n) -> length(t,n+1)
```

The second branch is executed for values that are in
 (List,Int) **not** in $\{('nil',n)\}$ **and** in $\{((_,t),n)\}$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ and $\{p\} = \{v \mid v \text{ matches pattern } p\}$
 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil', n) -> n                               Int
  | ((_,t), n) -> length(t,n+1)
```

The second branch is executed for values that are in
 (List,Int) **not** in $\{('nil,n)\}$ **and** in $\{((_,t),n)\}$
 $((List,Int) \setminus (('nil,Any))) \& ((Any,Any), Any)$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ and $\{p\} = \{v \mid v \text{ matches pattern } p\}$
 (this is a type)

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

```
fun length (x : (List,Int)) : Int =
  match x with
  | ('nil', n) -> n                               Int
  | ((_,t), n) -> length(t,n+1)
```

The second branch is executed for values that are in

$(\text{List}, \text{Int})$ **not** in $\{('nil', n)\}$ **and** in $\{((_, t), n)\}$

$((\text{List}, \text{Int}) \setminus (('nil', \text{Any})) \& ((\text{Any}, \text{Any}), \text{Any}) = ((\text{Any}, \text{List}), \text{Int})$

Which types should we start from?

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

 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil', n) -> n                               Int
  | ((_,t), n) -> length(t,n+1)                   Int
```

The second branch is executed for values that are in
 (List,Int) **not** in $\{('nil,n)\}$ **and** in $\{((_,t),n)\}$

$((List,Int) \setminus (('nil,Any))) \& ((Any,Any), Any) = ((Any,List), Int)$

Which types should we start from?

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

 (this is a type)


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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil', n) -> n                Int
  | ((_,t), n) -> length(t,n+1)   Int
```

The match expression has type the **union** of the possible results

Which types should we start from?

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

 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil', n) -> n                Int
  | ((_,t), n) -> length(t,n+1)   Int
```

The match expression has type the **union** of the possible results

`Int | Int`

Which types should we start from?

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

 (this is a type)

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


```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil , n) -> n                Int
  | ((_,t), n) -> length(t,n+1)   Int
```

The match expression has type the union of the possible results

$\text{Int} \mid \text{Int} = \text{Int}$

Which types should we start from?

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

 (this is a type)

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

```
fun length (x :(List,Int)) : Int =
  match x with
  | ('nil', n) -> n                Int
  | ((_,t), n) -> length(t,n+1)   Int
```

The match expression has type the union of the possible results

$\text{Int} \mid \text{Int} = \text{Int}$

The function is well-typed

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- 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.

- Intersection types yield overloaded functions:

```
+ : (String,String->String) & (Int,Int->Int)
```

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- 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.

- Intersection types yield overloaded functions:

```
+ : (String,String->String) & (Int,Int->Int)
```

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- 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.

- Intersection types yield overloaded functions:

```
+ : (String,String->String) & (Int,Int->Int)
```

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- Boolean type constructors are useful for programming:

`map catalogue with`

`$x :: (\text{Car} \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used})) \rightarrow x$`

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

- Intersection types yield overloaded functions:

`+ : (String,String->String) & (Int,Int->Int)`

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- 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.

- Intersection types yield overloaded functions:

```
+ : (String,String->String) & (Int,Int->Int)
```

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

```
+ : (String, String -> String) & (Int, Int -> Int)
```

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- Intersection types yield overloaded functions:

`+ : (String, String -> String) & (Int, Int -> Int)`

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String -> String) & (Int, Int -> Int)`

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String->String) & (Int, Int->Int)`

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String -> String) & (Int, Int -> Int)`

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String->String) & (Int, Int->Int)`

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String->String) & (Int, Int->Int)`

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

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String->String) & (Int, Int->Int)`

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

Unions, intersections, differences

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

`match e with $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

- 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 $t_1 \mid t_2$.

- **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.

- **Intersection types yield overloaded functions:**

`+ : (String, String->String) & (Int, Int->Int)`

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

Unions, intersections, differences

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

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

- 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 $t_1 \mid t_2$.

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

$\text{map } catalogue \text{ with}$
 $x :: (\text{Car} \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used}))) \rightarrow x$

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

- **Intersection types yield overloaded functions:**

$+ : (\text{String}, \text{String} \rightarrow \text{String}) \& (\text{Int}, \text{Int} \rightarrow \text{Int})$

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="2007">
    <title> SBLP Proceedings </title>
    <editor>
      <last> Bigonha </last>
      <first> Roberto </first>
    </editor>
    SBC
  </book>
</bib>
```


XML syntax

```
<bib>[  
  <book year="1997">[  
    <title>['Object-Oriented Programming']  
    <author>[  
      <last>['Castagna']  
      <first>['Giuseppe']  
    ]  
    <price>['56']  
    'Bikhäuser'  
  ]  
  <book year="2007">[  
    <title>['SBLP Proceedings']  
    <editor>  
      <last>['Bigonha']  
      <first>['Roberto']  
    ]  
    'SBC'  
  ]  
</b>]
```

XML syntax

```
type Bib = <bib>[  
  <book year="1997">[  
    <title>['Object-Oriented Programming']  
    <author>[  
      <last>['Castagna']  
      <first>['Giuseppe']  
    ]  
    <price>['56']  
    'Bikhäuser'  
  ]  
  <book year="2007">[  
    <title>['SBLP Proceedings']  
    <editor>  
      <last>['Bigonha']  
      <first>['Roberto']  
    ]  
    'SBC'  
  ]  
</b>]
```

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*]
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]
```

Kleene star

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]
```

attribute types

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+)
    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]
```

unions

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

TYPES

PATTERNS

Patterns

Patterns = Types + Capture variables

TYPES

PATTERNS

```
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+ ]
```

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: 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: 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: 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 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 ]

```

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 ]
```

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 ]
```


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 ]
```

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 ]
```

Query language: CQL

Instead of just variables

```
select e from
  x1 in e1
  ⋮
  xn in en
where c
```

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

Query language: CQL

Instead of just variables use patterns

```
select e from  
  p1 in e1  
  ⋮  
  pn in en  
where c
```

```
Biblio = <bib>[Book*]
```

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

Query language: CQL

Instead of just variables use patterns

```
select e from
  p1 in e1
  ⋮
  pn in en
where c
```

```
<bib>[b::Book*]
<book year="1990">[ t::Title _+ <price>"69.99" ]
```

- (1) captures in b all the books of a bibliography
- (2) captures in t the title of a book if it is of 1990 and costs 69.99

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

Query language: CQL

Instead of just variables use patterns

```
select e from
  p1 in e1
  ⋮
  pn in en
where c
```

```
select <book>t from
  <bib>[b::Book*] in bibs,
  <book year="1990">[ t::Title _+ <price>"69.99" ] in b
```

```
Biblio = <bib>[Book*]
```

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

Query language: CQL

Instead of just variables use patterns

```
select e from
  p1 in e1
  ⋮
  pn in en
where c
```

```
select <book>t from
  <bib>[b::Book*] in bibs,
  <book year="1990">[ t::Title _+ <price>"69.99" ] in b
```

Selects from **bibs** the titles of all books of 1990 and of price 69.99

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


Query language: CQL

Instead of just variables use patterns

```
select e from
  p1 in e1
  ⋮
  pn in en
where c
```

```
fun getTitles(bibs : Biblio) : [(<book>[Title])*]
  select <book>t from
    <bib>[b::Book*] in bibs,
    <book year="1990">[ t::Title _+ <price>"69.99" ] in b
```

Selects from bibs the titles of all books of 1990 and of price 69.99 and has type `Biblio -> [(<book>[Title])*]`

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

XPath encoding

For instance in CQL (...but see Xtatic for a very different encoding):

- All children of e with tag tag (e/tag)

```
select x from <_ ..>[( x::(<tag ..>_)|_ )]* in e
```
- All attributes labelled by id ($e/@id$)

```
select x from <_ id=x ..> in e
```
- Notice that regexp patterns can define non-unary queries.

XPath encoding

For instance in CQL (...but see Xtatic for a very different encoding):

- All children of e with tag tag (e/tag)

```
select x from <_ ..>[( x::(<tag ..>_)|_ )]* in e
```

- All attributes labelled by id ($e/@id$)

```
select x from <_ id=x ..> in e
```

- Notice that regexp patterns can define non-unary queries.

XPath encoding

For instance in CQL (...but see Xtatic for a very different encoding):

- All children of e with tag tag (e/tag)

```
select x from <_ ..>[( x::(<tag ..>_)|_ )]* in e
```
- All attributes labelled by id ($e/@id$)

```
select x from <_ id=x ..> in e
```
- Notice that regexp patterns can define non-unary queries.

XPath encoding

For instance in CQL (...but see Xtatic for a very different encoding):

- All children of e with tag tag (e/tag)

```
select x from <_ ..>[( x::(<tag ..>_)|_ )]* in e
```
- All attributes labelled by id ($e/@id$)

```
select x from <_ id=x ..> in e
```
- Notice that regexp patterns can define non-unary queries.

XPath encoding

For instance in CQL (...but see Xtatic for a very different encoding):

- All children of e with tag tag (e/tag)
`select x from <_ ..>[(x::(<tag ..>_)|_)]* in e`
- All attributes labelled by id ($e/@id$)
`select x from <_ id=x ..> in e`
- Notice that regexp patterns can define non-unary queries.

Rationale

CQL, Xtatic, add syntactic sugar for XPath ...

... it is all syntactic sugar!

Types

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

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

```
Book = ('book, (Title, X ∨ Y))  
X = (Author, X ∨ (Price, 'nil) ∨ 'nil)  
Y = (Editor, Y ∨ (Price, 'nil) ∨ 'nil)
```

... it is all syntactic sugar!

Types

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

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

```
Book = ('book, (Title, X ∨ Y))  
X = (Author, X ∨ (Price, 'nil) ∨ 'nil)  
Y = (Editor, Y ∨ (Price, 'nil) ∨ 'nil)
```


... it is all syntactic sugar!

Types

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

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

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

... it is all syntactic sugar!

Types

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

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

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

Some reasons to consider regular expression types and patterns

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact**



• Classic example:

Is this a valid email?

Is this a valid URL?

Is this a valid IP?

Is this a valid date?

Is this a valid phone number?

Is this a valid password?

Is this a valid username?

Some good reasons to consider regexp patterns/types

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



- Classic usage
- Informative error messages
- Error mining
- Efficient execution
- Compact programs
- Logical optimisation of pattern-based queries
- Pattern matches as building blocks for iterators
- Type/pattern-based data pruning for memory usage optimisation
- 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

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

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

It is possible to specify constraints such as:

“The number of children of a node is at most 2”

“The number of children of a node is at least 2”

“The number of children of a node is exactly 2”

“The number of children of a node is at most 2 and at least 2”

“The number of children of a node is at most 2 and at least 2 and the children are of different types”

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

If the attribute x has value x , then e -elements that do not contain f -elements must contain two g -elements.

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.

- Types can be composed:

```
type WithPrice = <_ ..>[_* Price _*]  
type ThisYear = <_ year="2005">_
```

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.

- Types can be composed:

```
type WithPrice = <_ ..>[_* Price _*]
```

```
type ThisYear = <_ year="2005">_
```

then `<lib>[(lib|ox(ThisYear)(WithPrice))>` defines a view containing only this year's books that do not have price element.

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.

- Types can be composed:

```
type WithPrice = <_ ..>[_* Price _*]
type ThisYear = <_ year="2005">_
```

then `<bib>[((Biblio&ThisYear)\WithPrice)*]` defines a view containing only this year's books that do not have price element.

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.

- Types can be composed:

```
type WithPrice = <_ ..>[_* Price _*]  
type ThisYear = <_ year="2005">_
```

then `<bib>[((Biblio&ThisYear)\WithPrice)*]` defines a view containing only this year's books that do not have price element.

1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.

- Types can be composed:

```
type WithPrice = <_ ..>[_* Price _*]  
type ThisYear = <_ year="2005">_
```

then `<bib>[((Biblio&ThisYear)\WithPrice)*]` defines a view containing only this year's books that do not have price element.

Not very innovative but useful properties

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

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

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

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

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

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

```
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
```

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

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

```
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
```

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

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

```
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>[ ] ]
```

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

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

```
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>[ ] ]
```

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

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

```
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>[ ] ]
```

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

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

```
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>[ ] ]
```

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


3. Error mining

Detection of Empty Types

```
type Person = <person>[ Name Children ]  
type Children = <children>[Person+]  
type Name = <name>[PCDATA]
```

Warning at chars 57-76:

```
type Children = <children>[Person+]
```

This definition yields an empty type for Children

Warning at chars 14-39:

```
type Person = <person>[ Name Children ]
```

This definition yields an empty type for Person

3. Error mining

Detection of Empty Types

```
type Person = <person>[ Name Children ]  
type Children = <children>[Person+]  
type Name = <name>[PCDATA]
```

Warning at chars 57-76:

```
type Children = <children>[Person+]
```

This definition yields an empty type for Children

Warning at chars 14-39:

```
type Person = <person>[ Name Children ]
```

This definition yields an empty type for Person

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =
  select (z,y) from
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:

- no typing rule is violated

- the pattern is not useless, it can match authors

- They are not regular patterns specific

- Such errors are not always typed, they can be contextual errors

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:

- no typing rule is violated

- the type checker will not find the error (which is subtle)

- The error is not caught by the type checker

- The error is not caught by the type checker

- The error is not caught by the type checker

- The error is not caught by the type checker

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
 - no typing rule is violated
 - the pattern is not useless, it can match authors

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
 - no typing rule is violated
 - the pattern is not useless, it can match authors

• They are not regexp-patterns specific:

```
title/book/(title|price)
```

• The error is not caught by the current type checker

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
 - no typing rule is violated
 - the pattern is not useless, it can match authors
- They are not regexp-patterns specific:
 bibs/book/(title|prize)
- Such errors are not always typos: they can be conceptual errors.

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
 - no typing rule is violated
 - the pattern is not useless, it can match authors
- They are not regexp-patterns specific:
 bibs/book/(title|prize)
- Such errors are not always typos: they can be conceptual errors.

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
 - no typing rule is violated
 - the pattern is not useless, it can match authors
- They are not regexp-patterns specific:
 bibs/book/(title|prize)
- Such errors are not always typos: they can be conceptual errors.

3. Error mining (continued)

Spot subtle errors that elude current type checking technology

```
fun extract(x:[Book*]) =  
  select (z,y) from  
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
 - no typing rule is violated
 - the pattern is not useless, it can match authors
- They are not regexp-patterns specific:
 bibs/book/(title|prize)
- Such errors are not always typos: they can be conceptual errors.

Can be formally characterised and statically detected by the types/patterns presented here and integrated in current regexp type-checkers with no overhead

4. Compact programs

```
type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson|fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

4. Compact programs

```
type Person = FPerson | MPerson;;
type FPerson = <person gender = "F">[ Name Children ];;
type MPerson = <person gender = "M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

```

```

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

```

```

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson|fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;

```

4. Compact programs

```
type Person = FPerson | MPerson;;  
type FPerson = <person gender="F">[ Name Children ];;  
type MPerson = <person gender="M">[ Name Children ];;  
type Children = <children>[Person*];;  
type Name = <name>[String];;
```

```
type Man = <man>[ Name Sons Daughters ];;  
type Woman = <woman>[ Name Sons Daughters ];;  
type Sons = <sons>[ Man* ];;  
type Daughters = <daughters>[ Woman* ];;
```

```
let fun sort (MPerson -> Man ; FPerson -> Woman)  
  <person gender=gen>[ n <children>[(mc::MPerson|fc::FPerson)*] ] ->  
    let tag = match gen with "M" -> 'man | "F" -> 'woman in  
    let s = map mc with x -> sort x in  
    let d = map fc with x -> sort x in  
    <(tag)>[ n <sons>s <daughters>d ];;
```

4. Compact programs

```
type Person = FPerson | MPerson;;  
type FPerson = <person gender="F">[ Name Children ];;  
type MPerson = <person gender="M">[ Name Children ];;  
type Children = <children>[Person*];;  
type Name = <name>[String];;
```

```
type Man = <man>[ Name Sons Daughters ];;  
type Woman = <woman>[ Name Sons Daughters ];;  
type Sons = <sons>[ Man* ];;  
type Daughters = <daughters>[ Woman* ];;
```

```
let fun sort (MPerson -> Man ; FPerson -> Woman)  
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->  
    let tag = match gen with "M" -> 'man | "F" -> 'woman in  
    let s = map mc with x -> sort x in  
    let d = map fc with x -> sort x in  
    <(tag)>[ n <sons>s <daughters>d ];;
```

4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

```

```

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

```

```

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;

```


4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

```

```

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

```

```

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;

```

4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;

```

4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;

```

4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

```

```

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

```

```

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;

```

4. Compact programs

```

type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in      mc:[MPerson*] ⇒ s:[Man*]
    let d = map fc with x -> sort x in      fc:[FPerson*] ⇒ d:[Woman*]
    <(tag)>[ n <sons>s <daughters>d ];;

```

4. Compact programs

```
type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

4. Compact programs

```
type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Very hard to program it in XSLT

4. Compact programs

```
type Person = FPerson | MPerson;;
type FPerson = <person gender="F">[ Name Children ];;
type MPerson = <person gender="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;
```

```
type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;
```

```
let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[(mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Note

Although `sort:Person -> Man | Woman`, the declaration

```
fun sort (Person -> Man | Woman)
```

wouldn't type-check (fails for the recursive calls).

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*]
```

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.
- 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

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

New kind of push-down tree automata

6. Logical pattern-specific optimisation of queries

Transform the ~~from~~ clauses so as to capture in a single pattern as much information as possible

6. Logical pattern-specific optimisation of queries

Transform the ~~from~~ clauses so as to capture in a single pattern as much information as possible

- ① merge distinct patterns that work on a common sequence,
- ② transform *where* clauses into patterns,
- ③ transform paths into nested pattern-based selections, then merge.

6. Logical pattern-specific optimisation of queries

Transform the ~~from~~ clauses so as to capture in a single pattern as much information as possible

- ① merge distinct patterns that work on a common sequence,
- ② transform where clauses into patterns,
- ③ transform paths into nested pattern-based selections, then merge.

6. Logical pattern-specific optimisation of queries

Transform the ~~from~~ clauses so as to capture in a single pattern as much information as possible

- ① merge distinct patterns that work on a common sequence,
- ② transform where clauses into patterns,
- ③ transform paths into nested pattern-based selections, then merge.

6. Logical pattern-specific optimisation of queries

Transform the `from` clauses so as to capture in a single pattern as much information as possible

```
select <book year=y>[t] from  
  b in bibs/book,  
  p in b/price,  
  t in b/title,  
  y in b/@year  
where p = <price>"69.99"
```

6. Logical pattern-specific optimisation of queries

Transform the `from` clauses so as to capture in a single pattern as much information as possible

```
select <book year=y>[t] from
  b in bibs/book,
  p in b/price,
  t in b/title,
  y in b/@year
where p = <price>"69.99"
```

optimised as

```
select <book year=y> t from
  <bib>[b::Book*] in bibs,
  <book year=y>[ t::Title _+ <price>"69.99" ] in b
```


6. Logical pattern-specific optimisation of queries

Transform the `from` clauses so as to capture in a single pattern as much information as possible

```
select <book year=y>[t] from
  b in bibs/book,
  p in b/price,
  t in b/title,
  y in b/@year
where p = <price>"69.99"
```

optimised as

```
select <book year=y> t from
  <bib>[b::Book*] in bibs,
  <book year=y>[ t::Title _+ <price>"69.99" ] in b
```

These optimisations are orthogonal to the classical optimisations: they sum up and bring a further gain of performance

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

This may explain why there is less consensus on iterators than on extractors.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

This may explain why there is less consensus on iterators than on extractors.

How to define new iterators?

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

`select e from p in e'`

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

`select e from p in e' = filter[(p->e|_->[])*](e')`

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

```
select e from p in e' = filter[(p->e|_>[])*](e')  
map e with p1->e1|...|pn->en
```

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

```
select e from p in e' = filter[(p->e|_>[])*](e')  
map e with p1->e1|...|pn->en = filter[(p1->e1|...|pn->en)*(e)
```

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

$\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p \rightarrow e | _ \rightarrow [])^*](e')$

$\text{map } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[(p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n)^*](e)$

$\text{match } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n$

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

$\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p \rightarrow e | _ \rightarrow [])^*](e')$

$\text{map } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[(p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n)^*](e)$

$\text{match } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n]([e])$

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

$\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p \rightarrow e | _ \rightarrow [])^*](e')$

$\text{map } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[(p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n)^*](e)$

$\text{match } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n]([e])$

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

$\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p \rightarrow e | _ \rightarrow [])^*](e')$

$\text{map } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[(p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n)^*](e)$

$\text{match } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n]([e])$

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

7. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya's smart idea: Define regular expression over pattern-matches “ $p \rightarrow e$ ” (rather than over patterns).

$\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p \rightarrow e | _ \rightarrow [])^*](e')$

$\text{map } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[(p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n)^*](e)$

$\text{match } e \text{ with } p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n = \text{filter}[p_1 \rightarrow e_1 | \dots | p_n \rightarrow e_n]([e])$

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

Type precision obtained by specific typing, as for patterns.

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al].

We can start from the optimal compilation of regular expressions [Castagna et al. 2004] and extend it to XML queries.

Example:

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns:

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: Compile patterns in order to have as many "." wildcards as possible

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

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

compiled as

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

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

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

compiled as

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

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

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

compiled as

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

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

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

compiled as

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


8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

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

compiled as

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

Data matched by wildcards “_” not in the scope of a capture variable are not necessary to the evaluation.

8. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query q execute it on documents in which parts not necessary to evaluate q are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan *et al.*].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

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

compiled as

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

Data matched by wildcards “_” not in the scope of a capture variable are not necessary to the evaluation. Use boolean type constructors to determine the program data-need.

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- Data description is more precise:

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- Transformation description is more precise:

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a `show-element` contains `season-elements`,
then its `type-attribute` is "TV Series".*

- Transformation description is more precise:

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a **show-element** contains **season-elements**,
then its **type-attribute** is "TV Series".*

- Transformation description is more precise:

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries.

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

for `title/book/(title|author|editor)`

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

```
for      bibs/book/(title|author|editor)
infer type [(Title (Author+|Editor+))*]
rather than [(Title|Author|Editor)*]
```


9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a **show-element** contains **season-elements**,
then its **type-attribute** is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

```
for          bibs/book/(title|author|editor)
infer type   [(Title (Author+|Editor+))*]
rather than  [(Title|Author|Editor)*]
```

```
bibs : <bib>[ (<book year=String>[Title (Author+|Editor+) Price?]) * ]
```

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

```
for      bibs/book/(title|author|editor)
infer type [(Title (Author+|Editor+))*]
rather than [(Title|Author|Editor)*]
```

• DTD/Schema already used to optimise access to XML data on disk.

```
bibs : <bib>[ (<book year=String>[Title (Author+|Editor+) Price?]) * ]
```

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

```
for      bibs/book/(title|author|editor)
infer type [(Title (Author+|Editor+))*]
rather than [(Title|Author|Editor)*]
```

- DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision

```
bibs : <bib>[ (<book year=String>[Title (Author+|Editor+) Price?]) * ]
```

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

```
for          bibs/book/(title|author|editor)
infer type   [(Title (Author+|Editor+))*]
rather than  [(Title|Author|Editor)*]
```

- **DTD/Schema already used to optimise access to XML data on disk.** It should be possible to use also the precision of regexp types to optimise secondary memory queries.

9. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**

E.g. in IMDB there are constraints such as:

*If a show-element contains season-elements,
then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

- Exact type inference for pattern variables.
- Finer type inference for queries:

```
for          bibs/book/(title|author|editor)
infer type   [(Title (Author+|Editor+))*]
rather than  [(Title|Author|Editor)*]
```

- **DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.**

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Not enough

Other features

CDuce is a fully featured programming language:

- General purpose features: records, tuples, references, integers, booleans, exceptions, ...
- Strings (operators, regular expressions, lazy implementation)
- Data specific iterators
- Namespaces
- XML schema validation
- URI's, and file operations
- Unicode

Not enough
Libraries and external tools are also needed

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

- 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

- Use CDuce as an XML input/output/transformation layer
- XML Generation
- XML Transformation
- XML Validation
- XML Schema

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

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

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

Main Challenges

❶ Seamless integration:

No explicit conversion function in programs:
the compiler performs the conversions

❷ 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

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

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

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

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

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

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 \rightarrow 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;
 \Rightarrow CDuce typing would be lost.

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

\Rightarrow OCaml types are not enough to translate CDuce types.

⊖ OCaml supports type polymorphism; CDuce does not.

\Rightarrow Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

How to integrate the two type systems?

The translation can go just one way: **OCaml** \rightarrow **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;
 \Rightarrow *CDuce typing would be lost.*

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

\Rightarrow *OCaml types are not enough to translate CDuce types.*

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

\Rightarrow *Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce*

In practice

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

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

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

```
ocaml2cduce:  $t \rightarrow \mathbb{T}(t)$ 
cduce2ocaml:  $\mathbb{T}(t) \rightarrow t$ 
```

In practice

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

t (OCaml)	$\mathbb{T}(t)$ (CDuce)
<code>int</code>	<code>min_int-max_int</code>
<code>string</code>	<code>Latin1</code>
<code>$t_1 * t_2$</code>	<code>$(\mathbb{T}(t_1), \mathbb{T}(t_2))$</code>
<code>$t_1 \rightarrow t_2$</code>	<code>$\mathbb{T}(t_1) \rightarrow \mathbb{T}(t_2)$</code>
<code>t list</code>	<code>$[\mathbb{T}(t)*]$</code>
<code>t array</code>	<code>$[\mathbb{T}(t)*]$</code>
<code>t option</code>	<code>$[\mathbb{T}(t)?]$</code>
<code>t ref</code>	<code>ref $\mathbb{T}(t)$</code>
<code>A_1 of t_1 ... A_n of t_n</code>	<code>$(A_1, \mathbb{T}(t_1))$... $(A_n, \mathbb{T}(t_n))$</code>
<code>$\{l_1 = t_1; \dots; l_n = t_n\}$</code>	<code>$\{l_1 = \mathbb{T}(t_1); \dots; l_n = \mathbb{T}(t_n)\}$</code>

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

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

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

In practice

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

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

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

```
ocaml2cduce:  $t \rightarrow \mathbb{T}(t)$ 
cduce2ocaml:  $\mathbb{T}(t) \rightarrow t$ 
```

In practice

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

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

- 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`

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 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 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 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 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 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.

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)
```

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)
```

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)
```

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)
```

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)
```


Embedded CDuce

Embed CDuce in XML/XHTML à la PHP:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...>
<% include "cgi.cd" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>
      <% "Hello," @ " World !!!" %>
    </p>
  </body>
</html>
```

Compiled rather than interpreted



Embedded CDuce

Embed CDuce in XML/XHTML à la PHP:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...>
<% include "cgi.cd" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>
      <% "Hello," @ " World !!!" %>
    </p>
  </body>
</html>
```

Compiled rather than interpreted



Embedded CDuce

Embed CDuce in XML/XHTML à la PHP:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...>
<% include "cgi.cd" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>
      <% "Hello," @ " World !!!" %>
    </p>
  </body>
</html>
```

Compiled rather than interpreted

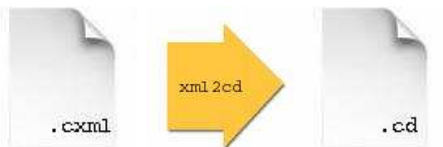


Embedded CDuce

Embed CDuce in XML/XHTML à la PHP:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...>
<% include "cgi.cd" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>
      <% "Hello," @ " World !!!" %>
    </p>
  </body>
</html>
```

Compiled rather than interpreted



Embedded CDuce

Embed CDuce in XML/XHTML à la PHP:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" ...>
<% include "cgi.cd" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>
      <% "Hello," @ " World !!!" %>
    </p>
  </body>
</html>
```

Compiled rather than interpreted



Embedded CDuce

A short demo

PART 2: THEORETICAL FOUNDATIONS

Goal

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

$\times, \rightarrow, \{\dots\}, \text{chan}(), \dots$

and add boolean combinators:

\vee, \wedge, \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

Goal

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

\times , \rightarrow , $\{\dots\}$, `chan()`, ...

and add boolean combinators:

\vee , \wedge , \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

Goal

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

\times , \rightarrow , $\{\dots\}$, `chan()`, ...

and add boolean combinators:

\vee , \wedge , \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

Goal

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

$\times, \rightarrow, \{\dots\}, \text{chan}(), \dots$

and add boolean combinators:

\vee, \wedge, \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

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

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

$$\frac{a_1 \leq a_2 \quad a_3 \leq a_4}{a_1 \rightarrow a_3 \leq a_2 \rightarrow a_4}$$

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

$$(a \vee b) \rightarrow c \leq (a \rightarrow c) \vee (b \rightarrow c)$$

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

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

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

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

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

$$(s_1 \vee s_2) \rightarrow t \not\leq (s_1 \rightarrow t) \vee (s_2 \rightarrow t)$$

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

$$t ::= B \mid t \times t \mid t \rightarrow t \mid t \forall t \mid t \wedge t \mid \neg t \mid 0 \mid 1$$

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

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

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

$$(s_1 \wedge s_2) \rightarrow t \neq (s_1 \rightarrow t) \wedge (s_2 \rightarrow t)$$

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

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

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

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

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

$$(s_1 \vee s_2) \rightarrow t \not\leq (s_1 \rightarrow t) \wedge (s_2 \rightarrow t)$$

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

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

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

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

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

$$(s_1 \vee s_2) \rightarrow t \quad \not\leq \quad (s_1 \rightarrow t) \wedge (s_2 \rightarrow t)$$

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

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

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

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

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

$$(s_1 \vee s_2) \rightarrow t \quad \begin{matrix} \geq \\ \leq \end{matrix} \quad (s_1 \rightarrow t) \wedge (s_2 \rightarrow t)$$

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

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

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

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

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

$$(s_1 \vee s_2) \rightarrow t \quad \not\geq \quad (s_1 \rightarrow t) \wedge (s_2 \rightarrow t)$$

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

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

- **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.

In details

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

- **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.

In details

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

- **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

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

$$\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$$

- 2 Define the subtyping relation as follows:

$$s \leq t \stackrel{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket$$

KEY OBSERVATION 1:

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

Hosoya and Pierce use the model of values:

$$\llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

Ok because the only values of XDuce are XML documents

Semantic subtyping

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

$$\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$$

- 2 Define the subtyping relation as follows:

$$s \leq t \stackrel{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket$$

KEY OBSERVATION 1:

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

Hosoya and Pierce use the model of values:

$$\llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

Ok because the only values of XDuce are XML documents

Semantic subtyping

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

$$\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$$

- 2 Define the subtyping relation as follows:

$$s \leq t \stackrel{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket$$

KEY OBSERVATION 1:

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

Hosoya and Pierce use the model of values:

$$\llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

Ok because the only values of XDuce are XML documents

Semantic subtyping

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

$$\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$$

- 2 Define the subtyping relation as follows:

$$s \leq t \stackrel{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket$$

KEY OBSERVATION 1:

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

Hosoya and Pierce use the model of values:

$$\llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

Ok because the only values of XDuce are XML documents

Semantic subtyping

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

$$\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$$

- 2 Define the subtyping relation as follows:

$$s \leq t \stackrel{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket$$

KEY OBSERVATION 1:

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

Hosoya and Pierce use the model of values:

$$\llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

Ok because the only values of XDuce are XML documents

Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

$$\llbracket t \rrbracket_v$$

Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

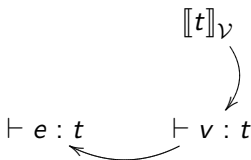
$$\begin{array}{c} \llbracket t \rrbracket_v \\ \downarrow \\ \vdash v : t \end{array}$$

Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

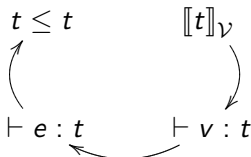


Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

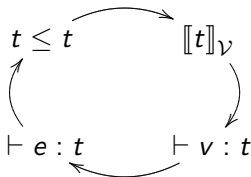


Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Circularity

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_v \subseteq \llbracket s \rrbracket_v \quad \text{where} \quad \llbracket t \rrbracket_v = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

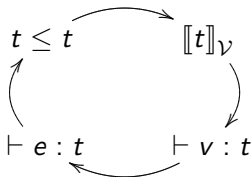


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

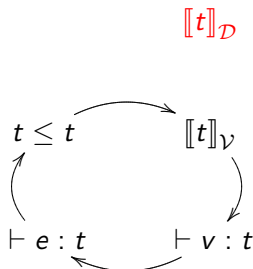


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

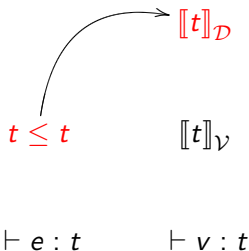


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

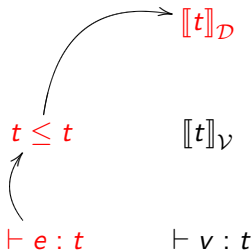


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

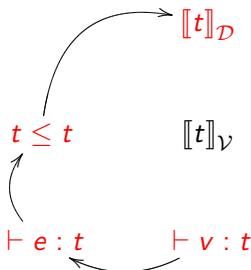


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

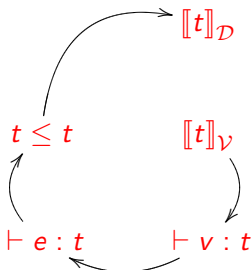


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

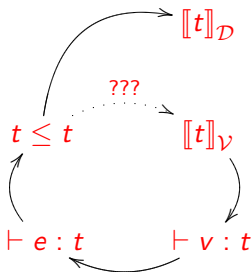


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined

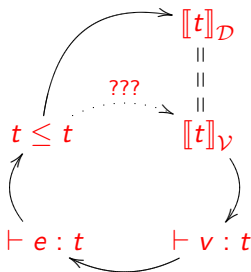


Bootstrap

Model of values

$$t \leq s \iff \llbracket t \rrbracket_{\mathcal{V}} \subseteq \llbracket s \rrbracket_{\mathcal{V}} \quad \text{where} \quad \llbracket t \rrbracket_{\mathcal{V}} = \{v \mid \vdash v : t\}$$

No longer works with arrow types: values are λ -abstractions and need (sub)typing to be defined



Step 1 : Model

Define when $\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

- Hard for constructors:

$$\llbracket t_1 \times t_2 \rrbracket =$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket =$$

Think semantically!

Step 1 : Model

Define when $\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

- Hard for constructors:

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket =$$

Think semantically!

Step 1 : Model

Define when $\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

- Hard for constructors:

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket = ???$$

Think semantically!

Step 1 : Model

Define when $\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

- Hard for constructors:

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket = ???$$

Think semantically!

Step 1 : Model

Define when $\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

- Hard for constructors:

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket = ???$$

Think semantically!

Step 1 : Model

Define when $\llbracket \cdot \rrbracket : \mathbf{Types} \longrightarrow \mathcal{P}(\mathcal{D})$ yields a *set-theoretic* model.

- Easy for the combinators:

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

- Hard for constructors:

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket = ???$$

Think semantically!

Intuition

$$\llbracket t \rightarrow s \rrbracket = ???$$

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

KEY OBSERVATION 2:

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation (*) held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \{\text{functions from } \llbracket t \rrbracket \text{ to } \llbracket s \rrbracket\}$$

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

KEY OBSERVATION 2:

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation (*) held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \{f \subseteq \mathcal{D}^2 \mid \forall (d_1, d_2) \in f. d_1 \in \llbracket t \rrbracket \Rightarrow d_2 \in \llbracket s \rrbracket\}$$

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

KEY OBSERVATION 2:

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation (*) held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \mathcal{P}(\overline{\llbracket t \rrbracket} \times \overline{\llbracket s \rrbracket}) \quad (\overline{X} \stackrel{\text{def}}{=} \text{complement of } X)$$

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

KEY OBSERVATION 2:

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation (*) held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \mathcal{P}(\overline{\llbracket t \rrbracket} \times \overline{\llbracket s \rrbracket}) \quad (*)$$

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

KEY OBSERVATION 2:

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation $(*)$ held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \mathcal{P}(\overline{\llbracket t \rrbracket} \times \overline{\llbracket s \rrbracket}) \quad (*)$$

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

KEY OBSERVATION 2:

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

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation $(*)$ held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \mathcal{P}(\overline{\llbracket t \rrbracket} \times \overline{\llbracket s \rrbracket}) \quad (*)$$

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

KEY OBSERVATION 2:

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

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation $(*)$ held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \mathcal{P}(\overline{\llbracket t \rrbracket} \times \overline{\llbracket s \rrbracket}) \quad (*)$$

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

KEY OBSERVATION 2:

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

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation $(*)$ held, namely

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

and similarly for any boolean combination of arrow types.

Intuition

$$\llbracket t \rightarrow s \rrbracket = \mathcal{P}(\overline{\llbracket t \rrbracket} \times \overline{\llbracket s \rrbracket}) \quad (*)$$

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

KEY OBSERVATION 2:

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

Accept every $\llbracket \cdot \rrbracket$ that behaves w.r.t. \subseteq **as if** equation $(*)$ held, namely

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

and similarly for any boolean combination of arrow types.

Technically ...

① **Take** $\llbracket _ \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

[combinator semantics]

② Define $\llbracket _ \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\llbracket a \times b \rrbracket \stackrel{\text{def}}{=} \llbracket a \rrbracket \times \llbracket b \rrbracket \subseteq \mathcal{D}^2$$

$$\llbracket a \rightarrow b \rrbracket \stackrel{\text{def}}{=} \mathcal{P}(\llbracket a \rrbracket \times \llbracket b \rrbracket) \subseteq \mathcal{P}(\mathcal{D}^2)$$

$$\llbracket a \vee b \rrbracket \stackrel{\text{def}}{=} \llbracket a \rrbracket \cup \llbracket b \rrbracket$$

[combinator semantics]

③ $\llbracket _ \rrbracket$ is a $\mathbf{Type} \rightarrow \mathbf{Type}$ mapping, i.e., it respects

$$\begin{aligned} \llbracket t_1 \rrbracket &\subseteq \llbracket t_2 \rrbracket && \llbracket t_1 \vee t_2 \rrbracket &\subseteq \llbracket t_2 \rrbracket \\ \llbracket t_1 \wedge t_2 \rrbracket &\subseteq \llbracket t_1 \rrbracket && \llbracket t_1 \wedge t_2 \rrbracket &\subseteq \llbracket t_2 \rrbracket \\ \llbracket t_1 \times t_2 \rrbracket &\subseteq \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket && \llbracket t_1 \times t_2 \rrbracket &\subseteq \llbracket t_2 \rrbracket \times \llbracket t_1 \rrbracket \\ \llbracket t_1 \rightarrow t_2 \rrbracket &\subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket) && \llbracket t_1 \rightarrow t_2 \rrbracket &\subseteq \mathcal{P}(\llbracket t_2 \rrbracket \times \llbracket t_1 \rrbracket) \end{aligned}$$

Technically ...

- ① **Take** $\llbracket _ \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

[combinator semantics]

- ② **Define** $\mathbb{E}[_] : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\mathbb{E}[\llbracket t_1 \times t_2 \rrbracket] \stackrel{\text{def}}{=} \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \subseteq \mathcal{D}^2$$

$$\mathbb{E}[\llbracket t_1 \rightarrow t_2 \rrbracket] \stackrel{\text{def}}{=} \mathcal{P}(\overline{\llbracket t_1 \rrbracket} \times \overline{\llbracket t_2 \rrbracket}) \subseteq \mathcal{P}(\mathcal{D}^2)$$

$$\mathbb{E}[\llbracket t_1 \vee t_2 \rrbracket] \stackrel{\text{def}}{=} \mathbb{E}[\llbracket t_1 \rrbracket] \cup \mathbb{E}[\llbracket t_2 \rrbracket]$$

⋮

[constructor semantics]

- ③ **Interpret** instead of requiring $\llbracket t \rrbracket = \emptyset$, accept $\llbracket t \rrbracket \neq \emptyset$ and

$$\llbracket t \rrbracket \neq \emptyset \Rightarrow \llbracket t \rrbracket \subseteq \mathbb{E}[\llbracket t \rrbracket]$$

Interpretation of $\llbracket t \rrbracket$ is the set of values that can be reached from t .

Technically ...

- ① **Take** $\llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

[combinator semantics]

- ② **Define** $\mathbb{E} \llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

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

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

$$\mathbb{E} \llbracket t_1 \vee t_2 \rrbracket \stackrel{\text{def}}{=} \mathbb{E} \llbracket t_1 \rrbracket \cup \mathbb{E} \llbracket t_2 \rrbracket$$

$$\vdots$$

[constructor semantics]

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

$$\llbracket x \rrbracket = \emptyset \iff \mathbb{E} \llbracket x \rrbracket = \emptyset$$

Technically ...

- ① **Take** $\llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

[combinator semantics]

- ② **Define** $\mathbb{E} \llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

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

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

$$\mathbb{E} \llbracket t_1 \vee t_2 \rrbracket \stackrel{\text{def}}{=} \mathbb{E} \llbracket t_1 \rrbracket \cup \mathbb{E} \llbracket t_2 \rrbracket$$

\vdots

[constructor semantics]

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

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

(which is equivalent to $\llbracket s \rrbracket \subseteq \llbracket t \rrbracket \iff \mathbb{E} \llbracket s \rrbracket \subseteq \mathbb{E} \llbracket t \rrbracket$)

Technically ...

- ① **Take** $\llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

[combinator semantics]

- ② **Define** $\mathbb{E}[\llbracket - \rrbracket] : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

$$\mathbb{E}[\llbracket t_1 \times t_2 \rrbracket] \stackrel{\text{def}}{=} \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \subseteq \mathcal{D}^2$$

$$\mathbb{E}[\llbracket t_1 \rightarrow t_2 \rrbracket] \stackrel{\text{def}}{=} \mathcal{P}(\overline{\llbracket t_1 \rrbracket} \times \overline{\llbracket t_2 \rrbracket}) \subseteq \mathcal{P}(\mathcal{D}^2)$$

$$\mathbb{E}[\llbracket t_1 \vee t_2 \rrbracket] \stackrel{\text{def}}{=} \mathbb{E}[\llbracket t_1 \rrbracket] \cup \mathbb{E}[\llbracket t_2 \rrbracket]$$

\vdots

[constructor semantics]

- ③ **Model:** Instead of requiring $\llbracket t \rrbracket = \mathbb{E}[\llbracket t \rrbracket]$, accept $\llbracket \rrbracket$ if

$$\llbracket t \rrbracket = \emptyset \iff \mathbb{E}[\llbracket t \rrbracket] = \emptyset$$

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

Technically ...

- ① **Take** $\llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D})$ such that

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$

$$\llbracket 0 \rrbracket = \emptyset$$

$$\llbracket 1 \rrbracket = \mathcal{D}$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

[combinator semantics]

- ② **Define** $\mathbb{E} \llbracket - \rrbracket : \mathbf{Types} \rightarrow \mathcal{P}(\mathcal{D}^2 + \mathcal{P}(\mathcal{D}^2))$ as follows

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

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

$$\mathbb{E} \llbracket t_1 \vee t_2 \rrbracket \stackrel{\text{def}}{=} \mathbb{E} \llbracket t_1 \rrbracket \cup \mathbb{E} \llbracket t_2 \rrbracket$$

\vdots

[constructor semantics]

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

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

(which is equivalent to $\llbracket s \rrbracket \subseteq \llbracket t \rrbracket \iff \mathbb{E} \llbracket s \rrbracket \subseteq \mathbb{E} \llbracket t \rrbracket$)

The main intuition

To characterize \leq all is needed is the test of emptiness

Indeed: $s \leq t \Leftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Leftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \Leftrightarrow \llbracket s \wedge \neg t \rrbracket = \emptyset$

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

$\llbracket \cdot \rrbracket$ and $\mathbb{E}[\cdot]$ must have the same zeros

We relaxed our requirement but ...

DOES A MODEL EXIST?

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

YES: an example within two slides

The main intuition

To characterize \leq all is needed is the test of emptiness

Indeed: $s \leq t \Leftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Leftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \Leftrightarrow \llbracket s \wedge \neg t \rrbracket = \emptyset$

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

$\llbracket \cdot \rrbracket$ and $\mathbb{E}[\cdot]$ must have the same zeros

We relaxed our requirement but ...

DOES A MODEL EXIST?

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

YES: an example within two slides

The main intuition

To characterize \leq all is needed is the test of emptiness

Indeed: $s \leq t \Leftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Leftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \Leftrightarrow \llbracket s \wedge \neg t \rrbracket = \emptyset$

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

$\llbracket _ \rrbracket$ and $\mathbb{E}[_]$ must have the same zeros

We relaxed our requirement but ...

DOES A MODEL EXIST?

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

YES: an example within two slides

The main intuition

To characterize \leq all is needed is the test of emptiness

Indeed: $s \leq t \Leftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Leftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \Leftrightarrow \llbracket s \wedge \neg t \rrbracket = \emptyset$

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

$\llbracket _ \rrbracket$ and $\mathbb{E}[_]$ must have the same zeros

We relaxed our requirement but ...

DOES A MODEL EXIST?

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

YES: an example within two slides

The main intuition

To characterize \leq all is needed is the test of emptiness

Indeed: $s \leq t \Leftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Leftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \Leftrightarrow \llbracket s \wedge \neg t \rrbracket = \emptyset$

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

$\llbracket _ \rrbracket$ and $\mathbb{E}[_]$ must have the same zeros

We relaxed our requirement but ...

DOES A MODEL EXIST?

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

YES: an example within two slides

The main intuition

To characterize \leq all is needed is the test of emptiness

Indeed: $s \leq t \Leftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Leftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \Leftrightarrow \llbracket s \wedge \neg t \rrbracket = \emptyset$

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

$\llbracket \cdot \rrbracket$ and $\mathbb{E}[\cdot]$ must have the same zeros

We relaxed our requirement but ...

DOES A MODEL EXIST?

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

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 $\llbracket t \rrbracket = \mathbb{E}[\llbracket t \rrbracket]$): it depends on the language the types are intended for.

Variations are possible. Our choice

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

accounts for languages that are:

- *Non-deterministic:*

Admits functions in which (d, d_1) and (d, d_2) with $d_1 \neq d_2$.

A function is allowed to be not total on $\llbracket E \rrbracket$.

A function is allowed to be not total on $\llbracket E \rrbracket$.

A function is allowed to be not total on $\llbracket E \rrbracket$.

The role of $\mathbb{E}[\]$

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

Variations are possible. Our choice

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

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 $\llbracket t \rightarrow s \rrbracket$ may be not total on $\llbracket t \rrbracket$.

The role of $\mathbb{E}[\]$

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

Variations are possible. Our choice

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

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 $\llbracket t \rightarrow s \rrbracket$ may be not total on $\llbracket t \rrbracket$. E.g.

$\llbracket t \rightarrow 0 \rrbracket =$ functions diverging on t

The role of $\mathbb{E}[\]$

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

Variations are possible. Our choice

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

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 $\llbracket t \rightarrow s \rrbracket$ may be not total on $\llbracket t \rrbracket$. E.g.

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

③ *Overloaded*:

$$\llbracket (t_1 \vee t_2) \rightarrow (s_1 \wedge s_2) \rrbracket \subseteq \llbracket (t_1 \rightarrow s_1) \wedge (t_2 \rightarrow s_2) \rrbracket$$

The role of $\mathbb{E}[\]$

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

Variations are possible. Our choice

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

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 $\llbracket t \rightarrow s \rrbracket$ may be not total on $\llbracket t \rrbracket$. E.g.

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

③ *Overloaded*:

$$\llbracket (t_1 \vee t_2) \rightarrow (s_1 \wedge s_2) \rrbracket \subsetneq \llbracket (t_1 \rightarrow s_1) \wedge (t_2 \rightarrow s_2) \rrbracket$$

The role of $\mathbb{E}[\]$

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

Variations are possible. Our choice

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

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 $\llbracket t \rightarrow s \rrbracket$ may be not total on $\llbracket t \rrbracket$. E.g.

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

③ *Overloaded*:

$$\llbracket (t_1 \vee t_2) \rightarrow (s_1 \wedge s_2) \rrbracket \subsetneq \llbracket (t_1 \rightarrow s_1) \wedge (t_2 \rightarrow s_2) \rrbracket$$

Closing the circle

- 1 Take any model $(\mathcal{B}, \llbracket \cdot \rrbracket_{\mathcal{B}})$ to bootstrap the definition.

- 2 Define

$$s \leq_{\mathcal{B}} t \iff \llbracket s \rrbracket_{\mathcal{B}} \subseteq \llbracket t \rrbracket_{\mathcal{B}}$$

- 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 $\llbracket t \rrbracket_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and

$$s \leq_{\mathcal{V}} t \iff \llbracket s \rrbracket_{\mathcal{V}} \subseteq \llbracket t \rrbracket_{\mathcal{V}}$$

- 5 If \mathcal{L} is “appropriate” ($\vdash_{\mathcal{B}} v : t \iff \not\vdash_{\mathcal{B}} v : \neg t$) then $\llbracket \cdot \rrbracket_{\mathcal{V}}$ is a model and

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

The circle is closed

Closing the circle

① Take any model $(\mathcal{B}, \llbracket \cdot \rrbracket_{\mathcal{B}})$ to bootstrap the definition.

② Define

$$s \leq_{\mathcal{B}} t \iff \llbracket s \rrbracket_{\mathcal{B}} \subseteq \llbracket t \rrbracket_{\mathcal{B}}$$

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 $\llbracket t \rrbracket_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and

$$s \leq_{\mathcal{V}} t \iff \llbracket s \rrbracket_{\mathcal{V}} \subseteq \llbracket t \rrbracket_{\mathcal{V}}$$

⑤ If \mathcal{L} is “appropriate” ($\vdash_{\mathcal{B}} v : t \iff \not\vdash_{\mathcal{B}} v : \neg t$) then $\llbracket \cdot \rrbracket_{\mathcal{V}}$ is a model and

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

The circle is closed

Closing the circle

① Take any model $(\mathcal{B}, \llbracket \cdot \rrbracket_{\mathcal{B}})$ to bootstrap the definition.

② Define

$$s \leq_{\mathcal{B}} t \quad \Longleftrightarrow \quad \llbracket s \rrbracket_{\mathcal{B}} \subseteq \llbracket t \rrbracket_{\mathcal{B}}$$

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 $\llbracket t \rrbracket_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and

$$s \leq_{\mathcal{V}} t \quad \Longleftrightarrow \quad \llbracket s \rrbracket_{\mathcal{V}} \subseteq \llbracket t \rrbracket_{\mathcal{V}}$$

⑤ If \mathcal{L} is “appropriate” ($\vdash_{\mathcal{B}} v : t \Longleftrightarrow \not\vdash_{\mathcal{B}} v : \neg t$) then $\llbracket \cdot \rrbracket_{\mathcal{V}}$ is a model and

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

The circle is closed

Closing the circle

① Take any model $(\mathcal{B}, \llbracket \cdot \rrbracket_{\mathcal{B}})$ to bootstrap the definition.

② Define

$$s \leq_{\mathcal{B}} t \iff \llbracket s \rrbracket_{\mathcal{B}} \subseteq \llbracket t \rrbracket_{\mathcal{B}}$$

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 $\llbracket t \rrbracket_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and

$$s \leq_{\mathcal{V}} t \iff \llbracket s \rrbracket_{\mathcal{V}} \subseteq \llbracket t \rrbracket_{\mathcal{V}}$$

⑤ If \mathcal{L} is “appropriate” ($\vdash_{\mathcal{B}} v : t \iff \not\vdash_{\mathcal{B}} v : \neg t$) then $\llbracket \cdot \rrbracket_{\mathcal{V}}$ is a model and

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

The circle is closed

Closing the circle

① Take any model $(\mathcal{B}, \llbracket \cdot \rrbracket_{\mathcal{B}})$ to bootstrap the definition.

② Define

$$s \leq_{\mathcal{B}} t \quad \Longleftrightarrow \quad \llbracket s \rrbracket_{\mathcal{B}} \subseteq \llbracket t \rrbracket_{\mathcal{B}}$$

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 $\llbracket t \rrbracket_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and

$$s \leq_{\mathcal{V}} t \quad \Longleftrightarrow \quad \llbracket s \rrbracket_{\mathcal{V}} \subseteq \llbracket t \rrbracket_{\mathcal{V}}$$

⑤ If \mathcal{L} is “appropriate” ($\vdash_{\mathcal{B}} v : t \Longleftrightarrow \not\vdash_{\mathcal{B}} v : \neg t$) then $\llbracket \cdot \rrbracket_{\mathcal{V}}$ is a model and

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

The circle is closed

Closing the circle

① Take any model $(\mathcal{B}, \llbracket \cdot \rrbracket_{\mathcal{B}})$ to bootstrap the definition.

② Define

$$s \leq_{\mathcal{B}} t \iff \llbracket s \rrbracket_{\mathcal{B}} \subseteq \llbracket t \rrbracket_{\mathcal{B}}$$

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 $\llbracket t \rrbracket_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : t\}$ and

$$s \leq_{\mathcal{V}} t \iff \llbracket s \rrbracket_{\mathcal{V}} \subseteq \llbracket t \rrbracket_{\mathcal{V}}$$

⑤ If \mathcal{L} is “appropriate” ($\vdash_{\mathcal{B}} v : t \iff \not\vdash_{\mathcal{B}} v : \neg t$) then $\llbracket \cdot \rrbracket_{\mathcal{V}}$ is a model and

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

The circle is closed

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

- $\llbracket \cdot \rrbracket_{\mathcal{U}}$ is defined as:

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \overline{\mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

• $\llbracket \cdot \rrbracket_{\mathcal{U}}$ is defined as:

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \overline{\mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket x \rrbracket_{\mathcal{U}} &= \{x\} & \llbracket \lambda x. t \rrbracket_{\mathcal{U}} &= \{\lambda x. \llbracket t \rrbracket_{\mathcal{U}}\} & \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket_{\mathcal{U}} &= \llbracket t_2 \rrbracket_{\mathcal{U}} \\ \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket_{\mathcal{U}} &= \llbracket t_2 \rrbracket_{\mathcal{U}} & \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket_{\mathcal{U}} &= \llbracket t_2 \rrbracket_{\mathcal{U}} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \overline{\mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

$$t_1 \leq_{\mathcal{D}} t_2 \implies t_1 \leq_{\mathcal{U}} t_2$$

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket x \rrbracket_{\mathcal{U}} &= \{x\} & \llbracket \lambda x. t \rrbracket_{\mathcal{U}} &= \{\lambda x. \llbracket t \rrbracket_{\mathcal{U}}\} & \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket_{\mathcal{U}} &= \llbracket t_2 \rrbracket_{\mathcal{U}} \\ \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket_{\mathcal{U}} &= \llbracket t_2 \rrbracket_{\mathcal{U}} & \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket_{\mathcal{U}} &= \llbracket t_2 \rrbracket_{\mathcal{U}} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \overline{\mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathcal{U}} &= \emptyset & \llbracket 1 \rrbracket_{\mathcal{U}} &= \mathcal{U} & \llbracket \neg t \rrbracket_{\mathcal{U}} &= \mathcal{U} \setminus \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \vee t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cup \llbracket t \rrbracket_{\mathcal{U}} & \llbracket s \wedge t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cap \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \times t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \times \llbracket t \rrbracket_{\mathcal{U}} & \llbracket t \rightarrow s \rrbracket_{\mathcal{U}} &= \overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}) = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathcal{U}} &= \emptyset & \llbracket 1 \rrbracket_{\mathcal{U}} &= \mathcal{U} & \llbracket \neg t \rrbracket_{\mathcal{U}} &= \mathcal{U} \setminus \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \vee t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cup \llbracket t \rrbracket_{\mathcal{U}} & \llbracket s \wedge t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cap \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \times t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \times \llbracket t \rrbracket_{\mathcal{U}} & \llbracket t \rightarrow s \rrbracket_{\mathcal{U}} &= \overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}) = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathcal{U}} &= \emptyset & \llbracket 1 \rrbracket_{\mathcal{U}} &= \mathcal{U} & \llbracket \neg t \rrbracket_{\mathcal{U}} &= \mathcal{U} \setminus \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \vee t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cup \llbracket t \rrbracket_{\mathcal{U}} & \llbracket s \wedge t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cap \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \times t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \times \llbracket t \rrbracket_{\mathcal{U}} & \llbracket t \rightarrow s \rrbracket_{\mathcal{U}} &= \mathcal{P}_f(\overline{\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}}) \end{aligned}$$

It is a model: $\mathcal{P}_f(\overline{\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}}) = \emptyset \iff \mathcal{P}(\overline{\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}}) = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathcal{U}} &= \emptyset & \llbracket 1 \rrbracket_{\mathcal{U}} &= \mathcal{U} & \llbracket \neg t \rrbracket_{\mathcal{U}} &= \mathcal{U} \setminus \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \vee t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cup \llbracket t \rrbracket_{\mathcal{U}} & \llbracket s \wedge t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cap \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \times t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \times \llbracket t \rrbracket_{\mathcal{U}} & \llbracket t \rightarrow s \rrbracket_{\mathcal{U}} &= \overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}) = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathcal{U}} &= \emptyset & \llbracket 1 \rrbracket_{\mathcal{U}} &= \mathcal{U} & \llbracket \neg t \rrbracket_{\mathcal{U}} &= \mathcal{U} \setminus \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \vee t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cup \llbracket t \rrbracket_{\mathcal{U}} & \llbracket s \wedge t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cap \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \times t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \times \llbracket t \rrbracket_{\mathcal{U}} & \llbracket t \rightarrow s \rrbracket_{\mathcal{U}} &= \overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}) = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Exhibit a model

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

YES: take $(\mathcal{U}, \llbracket \cdot \rrbracket_{\mathcal{U}})$ where

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

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

$$\begin{aligned} \llbracket 0 \rrbracket_{\mathcal{U}} &= \emptyset & \llbracket 1 \rrbracket_{\mathcal{U}} &= \mathcal{U} & \llbracket \neg t \rrbracket_{\mathcal{U}} &= \mathcal{U} \setminus \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \vee t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cup \llbracket t \rrbracket_{\mathcal{U}} & \llbracket s \wedge t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \cap \llbracket t \rrbracket_{\mathcal{U}} \\ \llbracket s \times t \rrbracket_{\mathcal{U}} &= \llbracket s \rrbracket_{\mathcal{U}} \times \llbracket t \rrbracket_{\mathcal{U}} & \llbracket t \rightarrow s \rrbracket_{\mathcal{U}} &= \overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} \end{aligned}$$

It is a model: $\overline{\mathcal{P}_f(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}})} = \emptyset \iff \mathcal{P}(\llbracket t \rrbracket_{\mathcal{U}} \times \llbracket s \rrbracket_{\mathcal{U}}) = \emptyset$

It is the **best** model: for any other model $\llbracket \cdot \rrbracket_{\mathcal{D}}$

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

Subtyping Algorithms.

Canonical forms

Every (recursive) type

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

is equivalent (semantically, that is w.r.t. \leq) to a type of the form:

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

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 Transformations have only homogeneous intersections, e.g.

$$((a_1 \wedge a_2) \vee (a_3 \wedge a_4)) \vee ((\neg a_5 \wedge \neg a_6) \wedge (a_7 \wedge a_8)) \vee (a_9 \wedge a_{10})$$

③ Group negative and positive atoms in the intersections

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

Canonical forms

Every (recursive) type

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

is equivalent (semantically, that is w.r.t. \leq) to a type of the form:

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

- 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.

$$((s_1 \times t_1) \wedge \neg(s_2 \times t_2)) \vee (\neg(s_3 \rightarrow t_3) \wedge \neg(s_4 \rightarrow t_4)) \vee (s_5 \times t_5)$$

- 3 Group negative and positive atoms in the intersections:

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

Canonical forms

Every (recursive) type

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

is equivalent (semantically, that is w.r.t. \leq) to a type of the form:

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

- 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.

$$((s_1 \times t_1) \wedge \neg(s_2 \times t_2)) \vee (\neg(s_3 \rightarrow t_3) \wedge \neg(s_4 \rightarrow t_4)) \vee (s_5 \times t_5)$$

- 3 Group negative and positive atoms in the intersections:

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

Canonical forms

Every (recursive) type

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

is equivalent (semantically, that is w.r.t. \leq) to a type of the form:

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

- 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.

$$((s_1 \times t_1) \wedge \neg(s_2 \times t_2)) \vee (\neg(s_3 \rightarrow t_3) \wedge \neg(s_4 \rightarrow t_4)) \vee (s_5 \times t_5)$$

- 3 Group negative and positive atoms in the intersections:

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

Canonical forms

Every (recursive) type

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

is equivalent (semantically, that is w.r.t. \leq) to a type of the form:

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

- 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.

$$((s_1 \times t_1) \wedge \neg(s_2 \times t_2)) \vee (\neg(s_3 \rightarrow t_3) \wedge \neg(s_4 \rightarrow t_4)) \vee (s_5 \times t_5)$$

- 3 Group negative and positive atoms in the intersections:

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

Subtyping decomposition

Some ugly formulas:

$$\bigwedge_{i \in I} t_i \times s_i \leq \bigvee_{i \in J} t_i \times s_i$$

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

$$\bigwedge_{i \in I} t_i \rightarrow s_i \leq \bigvee_{i \in J} t_i \rightarrow s_i$$

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

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \iff \llbracket s \wedge \neg t \rrbracket = \emptyset \iff s \wedge \neg t = 0$$

1. Consider $s \wedge \neg t$

2. Put it in canonical form

$$\bigvee_{(P_1, \dots, P_n) \in \text{can}(s)} (P_1 \wedge \dots \wedge P_n) \wedge \neg \bigvee_{(Q_1, \dots, Q_m) \in \text{can}(t)} (Q_1 \wedge \dots \wedge Q_m) = \bigvee_{(P_1, \dots, P_n) \in \text{can}(s)} (P_1 \wedge \dots \wedge P_n) \wedge \bigwedge_{(Q_1, \dots, Q_m) \in \text{can}(t)} \neg(Q_1 \wedge \dots \wedge Q_m)$$

3. Decide (collectively) whether the two summands are both empty by walking through formulae of the canonical form.

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \iff \llbracket s \wedge \neg t \rrbracket = \emptyset \iff s \wedge \neg t = 0$$

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

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

- 3 Decide (coinductively) whether the two summands are both empty by applying the ugly formulas of the previous slide.

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \iff \llbracket s \wedge \neg t \rrbracket = \emptyset \iff s \wedge \neg t = \emptyset$$

① Consider $s \wedge \neg t$

② Put it in canonical form

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

③ Decide (coinductively) whether the two summands are both empty by applying the ugly formulas of the previous slide.

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \iff \llbracket s \wedge \neg t \rrbracket = \emptyset \iff s \wedge \neg t = \emptyset$$

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

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

- ③ Decide (coinductively) whether the two summands are both empty by applying the ugly formulas of the previous slide.

Decision procedure

$$s \leq t?$$

Recall that:

$$s \leq t \iff \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \emptyset \iff \llbracket s \wedge \neg t \rrbracket = \emptyset \iff s \wedge \neg t = \emptyset$$

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

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

- ③ Decide (coinductively) whether the two summands are both empty by applying the ugly formulas of the previous slide.

Application to a language.

Language

$e ::=$	x	variable
	$\mu f(s_1 \rightarrow t_1; \dots; s_n \rightarrow t_n)(x).e$	abstraction, $n \geq 1$
	$e_1 e_2$	application
	(e_1, e_2)	pair
	$\pi_i(e)$	projection, $i = 1, 2$
	$(x = e \in t)?e_1 : e_2$	binding type case

Typing

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

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

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

$$\frac{\Gamma \vdash e : s \quad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \text{ (typecase)}$$

Consider:

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

Typing

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

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

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

$$\frac{\Gamma \vdash e : s \quad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \text{ (typecase)}$$

Consider:

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

Typing

$$\frac{\Gamma \vdash e : s \leq_B t}{\Gamma \vdash e : t} \text{ (subsumption)}$$

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

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

$$\frac{\Gamma \vdash e : s \quad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \text{ (typecase)}$$

Consider:

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

Typing

$$\frac{\Gamma \vdash e : s \leq_B t}{\Gamma \vdash e : t} \text{ (subsumption)}$$

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

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

$$\frac{\Gamma \vdash e : s \quad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \text{ (typecase)}$$

Consider:

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

Typing

$$\frac{\Gamma \vdash e : s \leq_B t}{\Gamma \vdash e : t} \text{ (subsumption)}$$

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

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

$$\frac{\Gamma \vdash e : s \quad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \text{ (typecase)}$$

Consider:

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

Typing

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

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

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

$$\frac{\Gamma \vdash e : s \quad \Gamma, (x : s_1) \vdash e_1 : t_1 \quad \Gamma, (x : s_2) \vdash e_2 : t_2}{\Gamma \vdash (x = e \in t)?e_1 : e_2 : \bigvee_{\{i | s_i \neq 0\}} t_i} \text{ (typecase)}$$

Consider:

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

Reduction

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

where

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

And we have

$$s \leq_B t \iff s \leq_V t$$

The circle is closed

Reduction

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

where

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

And we have

$$s \leq_B t \iff s \leq_V t$$

The circle is closed

Reduction

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

where

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

And we have

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

The circle is closed

Reduction

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

where

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

And we have

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

The circle is closed

Why does it work?

$$s \leq_B t \iff s \leq_V t \quad (1)$$

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 B ,

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

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 $\mu f^{(s_1 \rightarrow t_1; \dots; s_k \rightarrow t_k)}(x).e$

Why does it work?

$$s \leq_{\mathcal{B}} t \iff s \leq_{\nu} t \quad (1)$$

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} ,

$$s \not\leq_{\mathcal{B}} t \implies \text{there exists } \nu \text{ such that } \vdash \nu : s \text{ and } \not\vdash \nu : t$$

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 $\mu f^{(s_1 \rightarrow t_1; \dots; s_k \rightarrow t_k)}(x).e$

Why does it work?

$$s \leq_{\mathcal{B}} t \iff s \leq_v t \quad (1)$$

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} ,

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

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 $\mu f^{(s_1 \rightarrow t_1; \dots; s_k \rightarrow t_k)}(x).e$

Advantages for the programmer

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

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

Furthermore the property

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

is fundamental for meaningful error messages:

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

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

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

is fundamental for meaningful error messages:

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

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

$s \not\leq t \implies$ there exists v such that $\vdash v : s$ and $\not\vdash v : t$

is fundamental for meaningful error messages:

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

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

$s \not\leq t \implies$ there exists v such that $\vdash v : s$ and $\not\vdash v : t$

is fundamental for meaningful error messages:

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

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

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

is fundamental for meaningful error messages:

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

Extensions.

ref types

$$\llbracket \text{ref } t \rrbracket = \{ \text{ref } v \mid v \in \llbracket t \rrbracket \}$$

In practice equivalent to

$$\llbracket \text{ref } t \rrbracket = \begin{cases} \{ \llbracket t \rrbracket \} & \text{if } \llbracket t \rrbracket \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

Deduce the subtyping relation

$$\left(\bigwedge_{\text{ref } s \in P} \text{ref } s \right) \leq \left(\bigvee_{\text{ref } t \in N} \text{ref } t \right) \iff \begin{aligned} & \exists \text{ref } s \in P, s \simeq 0, \text{ or} \\ & \exists \text{ref } s_1 \in P, \exists \text{ref } s_2 \in P, s_1 \not\preceq s_2, \text{ or} \\ & \exists \text{ref } s \in P, \exists \text{ref } t \in N, s \simeq t \end{aligned}$$

ref types

$$\llbracket \text{ref } t \rrbracket = \{ \text{ref } v \mid v \in \llbracket t \rrbracket \}$$

In practice equivalent to

$$\llbracket \text{ref } t \rrbracket = \begin{cases} \{ \llbracket t \rrbracket \} & \text{if } \llbracket t \rrbracket \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

Deduce the subtyping relation

$$\left(\bigwedge_{\text{ref } s \in P} \text{ref } s \right) \leq \left(\bigvee_{\text{ref } t \in N} \text{ref } t \right) \iff \begin{aligned} & \exists \text{ref } s \in P, s \simeq 0, \text{ or} \\ & \exists \text{ref } s_1 \in P, \exists \text{ref } s_2 \in P, s_1 \not\preceq s_2, \text{ or} \\ & \exists \text{ref } s \in P, \exists \text{ref } t \in N, s \simeq t \end{aligned}$$

ref types

$$\llbracket \text{ref } t \rrbracket = \{ \text{ref } v \mid v \in \llbracket t \rrbracket \}$$

In practice equivalent to

$$\mathbb{E} \llbracket \text{ref } t \rrbracket = \begin{cases} \{ \llbracket t \rrbracket \} & \text{if } \llbracket t \rrbracket \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

Deduce the subtyping relation

$$\left(\bigwedge_{\text{ref } s \in P} \text{ref } s \right) \leq \left(\bigvee_{\text{ref } t \in N} \text{ref } t \right) \iff \begin{aligned} & \exists \text{ref } s \in P, s \simeq 0, \text{ or} \\ & \exists \text{ref } s_1 \in P, \exists \text{ref } s_2 \in P, s_1 \not\approx s_2, \text{ or} \\ & \exists \text{ref } s \in P, \exists \text{ref } t \in N, s \simeq t \end{aligned}$$

ref types

$$\llbracket \text{ref } t \rrbracket = \{ \text{ref } v \mid v \in \llbracket t \rrbracket \}$$

In practice equivalent to

$$\mathbb{E} \llbracket \text{ref } t \rrbracket = \begin{cases} \{ \llbracket t \rrbracket \} & \text{if } \llbracket t \rrbracket \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

Deduce the subtyping relation

$$\left(\bigwedge_{\text{ref } s \in P} \text{ref } s \right) \leq \left(\bigvee_{\text{ref } t \in N} \text{ref } t \right) \iff \begin{aligned} & \exists \text{ref } s \in P, s \simeq 0, \text{ or} \\ & \exists \text{ref } s_1 \in P, \exists \text{ref } s_2 \in P, s_1 \not\leq s_2, \text{ or} \\ & \exists \text{ref } s \in P, \exists \text{ref } t \in N, s \simeq t \end{aligned}$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq 0$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy } (f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq 0$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy } (f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq \emptyset$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy}(f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq \emptyset$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy } (f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq 0$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy } (f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq 0$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy } (f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\mathbb{E} \llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

lazy types

If we define $t = \text{Int} \times t$ then $t \simeq 0$.

Use $s = \text{Int} \times \text{lazy } s$.

$$(\mu f^{(\text{Int} \rightarrow s)}(x).(x, \text{lazy } (f(x + 1))))0$$

$$\llbracket \text{lazy } t \rrbracket = \{ \text{lazy } e \mid e \text{ is closed and } e : t \}$$

But each $\text{lazy } e$ is identified by all the possible results it can return, namely $\{v \mid e \rightarrow^* v\}$, from which we deduce:

$$\mathbb{E} \llbracket \text{lazy } t \rrbracket = \mathcal{P}(\llbracket t \rrbracket)$$

Deduce the subtyping relation

$$(\bigwedge_{\text{lazy } s \in P} \text{lazy } s) \leq (\bigvee_{\text{lazy } t \in N} \text{lazy } t) \iff \exists \text{lazy } t \in N : \forall P' \subseteq P (\bigwedge_{\text{lazy } s \in P'} s) \leq t$$

Summarizing

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

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket;$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket;$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket;$$

$$\llbracket t \rrbracket = \emptyset \iff E[\llbracket t \rrbracket] = \emptyset$$

where the extensional interpretation associated to $\llbracket \cdot \rrbracket$ is defined as:

$$E[\llbracket t \rightarrow s \rrbracket] = \overline{\mathcal{P}(\llbracket t \rrbracket \times \llbracket s \rrbracket)}$$

$$E[\llbracket t \times s \rrbracket] = \llbracket t \rrbracket \times \llbracket s \rrbracket$$

$$E[\llbracket \text{lazy } t \rrbracket] = \mathcal{P}(\llbracket t \rrbracket)$$

$$E[\llbracket \text{ref } t \rrbracket] = \begin{cases} \{\llbracket t \rrbracket\} & \text{if } \llbracket t \rrbracket \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

...

Summarizing

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

$$\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket;$$

$$\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket;$$

$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket;$$

$$\llbracket t \rrbracket = \emptyset \iff \mathbb{E}[\llbracket t \rrbracket] = \emptyset$$

where the extensional interpretation associated to $\llbracket \cdot \rrbracket$ is defined as:

$$\mathbb{E}[\llbracket t \rightarrow s \rrbracket] = \overline{\mathcal{P}(\llbracket t \rrbracket \times \llbracket s \rrbracket)}$$

$$\mathbb{E}[\llbracket t \times s \rrbracket] = \llbracket t \rrbracket \times \llbracket s \rrbracket$$

$$\mathbb{E}[\llbracket \text{lazy } t \rrbracket] = \mathcal{P}(\llbracket t \rrbracket)$$

$$\mathbb{E}[\llbracket \text{ref } t \rrbracket] = \begin{cases} \{\llbracket t \rrbracket\} & \text{if } \llbracket t \rrbracket \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

...

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:

- 1. Define $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 $\llbracket \cdot \rrbracket$ 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 $\llbracket \cdot \rrbracket$) to decompose the emptiness test for your type constructors, and hence derive a subtyping algorithm.
- 5 Enjoy.

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 emptiness test for your type constructors, and hence derive a subtyping algorithm.
- 5 Enjoy.

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 $\llbracket \cdot \rrbracket$ 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 $\llbracket \cdot \rrbracket$) to decompose the emptiness test for your type constructors, and hence derive a subtyping algorithm.
- 5 Enjoy.

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 emptiness test for your type constructors, and hence derive a subtyping algorithm.
- 5 Enjoy.

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 emptiness test for your type constructors, and hence **derive a subtyping algorithm**. [may be not easy/possible]
- 5 **Enjoy**.

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 emptiness test for your type constructors, and hence **derive a subtyping algorithm**.
- 5 **Enjoy**.

PART 3: BEYOND XML

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

Types	t	$::=$	b	basic types
			$t \times t$	products
			$ch(t)$	I/O channel type
			$\neg t$	
			$t \vee t$	
			$t \wedge t$	
			0	
			1	

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

Types	t	$::=$	b	basic types
			$t \times t$	products
			$ch(t)$	I/O channel type
			$\neg t$	
			$t \vee t$	
			$t \wedge t$	
			0	
			1	

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

Types	t	$::=$	b	basic types
			$t \times t$	products
			$ch(t)$	I/O channel type
			$\neg t$	
			$t \vee t$	
			$t \wedge t$	
			0	
			1	

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

Types	t	$::=$	b	basic types
			$t \times t$	products
			$ch(t)$	I/O channel type
			$ch^+(t)$	input channel type
			$ch^-(t)$	output channel type
			$\neg t$	
			$t \vee t$	
			$t \wedge t$	
			0	
			1	

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

The subtyping relation is not very rich:

- compare the level of nesting of the same constructor
- type constructors do not mix

Types	t	$::=$	b	basic types
			$t \times t$	products
			$ch(t)$	I/O channel type
			$ch^+(t)$	input channel type
			$ch^-(t)$	output channel type
			$\neg t$	
			$t \vee t$	
			$t \wedge t$	
			0	
			1	

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

Enrich subtyping by adding type combinators

Types	t	$::=$	b	basic types
			$t \times t$	products
			$ch(t)$	I/O channel type
			$ch^+(t)$	input channel type
			$ch^-(t)$	output channel type
			$\neg t$	
			$t \vee t$	
			$t \wedge t$	
			0	
			1	

Types for π

In the beginning was the channel type (well, the sorts) ...

Then came subtyping with its polarities ...

Enrich subtyping by adding type combinators

Types	$t ::=$	b	basic types
		$t \times t$	products
		$ch(t)$	I/O channel type
		$ch^+(t)$	input channel type
		$ch^-(t)$	output channel type
		$\neg t$	} boolean combinators
		$t \vee t$	
		$t \wedge t$	
		0	
		1	

Set-theoretic model: **intuition**

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape



$ch^-(t)$ types all channels on which I can write a t -message



Set-theoretic model: **intuition**

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape



$ch^-(t)$ types all channels on which I can write a t -message



Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{ \text{channels whose shape fits objects of type } t \}$

$ch^-(t)$ types all channels on which I can write a t -message

•

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{ \text{channels whose shape fits objects of type } t \}$

From the viewpoint of subtyping, channels can be distinguished only on the objects they transport

$ch^-(t)$ types all channels on which I can write a t -message



Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\text{set of objects of type } t\}$

From the viewpoint of subtyping, channels can be distinguished only on the objects they transport

$ch^-(t)$ types all channels on which I can write a t -message

•

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$

From the viewpoint of subtyping, channels can be distinguished only on the objects they transport

$ch^-(t)$ types all channels on which I can write a t -message

•

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$

invariance of channel types

From the viewpoint of subtyping, channels can be distinguished only on the objects they transport

$ch^-(t)$ types all channels on which I can write a t -message

•

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message



$ch^-(t)$ types all channels on which I can write a t -message



Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message

- $\llbracket ch^+(t) \rrbracket = \{\llbracket s \rrbracket \mid s \leq t\}$

$ch^-(t)$ types all channels on which I can write a t -message

•

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message

- $\llbracket ch^+(t) \rrbracket = \{\llbracket s \rrbracket \mid s \leq t\}$

$ch^-(t)$ types all channels on which I can write a t -message



Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message

- $\llbracket ch^+(t) \rrbracket = \{\llbracket s \rrbracket \mid s \leq t\}$

$ch^-(t)$ types all channels on which I can write a t -message

- $\llbracket ch^-(t) \rrbracket = \{\llbracket s \rrbracket \mid s \geq t\}$

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message

- $\llbracket ch^+(t) \rrbracket = \{\llbracket s \rrbracket \mid s \leq t\}$ covariance of input channels

$ch^-(t)$ types all channels on which I can write a t -message

- $\llbracket ch^-(t) \rrbracket = \{\llbracket s \rrbracket \mid s \geq t\}$ contravariance of output chans

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message

- $\llbracket ch^+(t) \rrbracket = \{\llbracket s \rrbracket \mid s \leq t\}$

$ch^-(t)$ types all channels on which I can write a t -message

- $\llbracket ch^-(t) \rrbracket = \{\llbracket s \rrbracket \mid s \geq t\}$

Set-theoretic model: intuition

A type t denotes the set of objects of type t .

A channel is like a box with a particular shape

The box can contain only objects that fit that shape

- $\llbracket ch(t) \rrbracket = \{\llbracket t \rrbracket\}$ invariance of channel types

$ch^+(t)$ types all channels on which I expect to read a t -message

- $\llbracket ch^+(t) \rrbracket = \{\llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket\}$

$ch^-(t)$ types all channels on which I can write a t -message

- $\llbracket ch^-(t) \rrbracket = \{\llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket\}$

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{[s] \mid [s] \subseteq [t]\}$
- $\llbracket ch^-(t) \rrbracket = \{[s] \mid [s] \supseteq [t]\}$

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

We need $\mathcal{D} = \mathbb{B} + \llbracket \mathcal{D} \rrbracket$ (not straightforward)

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

Then define

$$t \leq s \stackrel{\text{def}}{\iff} \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$$

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

Then define

$$t \leq s \stackrel{\text{def}}{\iff} \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$$

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

Then define

$$t \leq s \stackrel{\text{def}}{\iff} \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$$

Some induced equations:

$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

Set-theoretic model: **formal definition**

Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:

- $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$, $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$,
 $\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
- $\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$
- $\llbracket ch^+(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \}$
- $\llbracket ch^-(t) \rrbracket = \{ \llbracket s \rrbracket \mid \llbracket s \rrbracket \supseteq \llbracket t \rrbracket \}$

Then define

$$t \leq s \stackrel{\text{def}}{\iff} \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$$

Some induced equations:

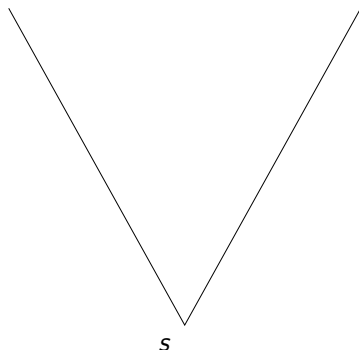
$$ch^-(t) \wedge ch^+(t) = ch(t)$$

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

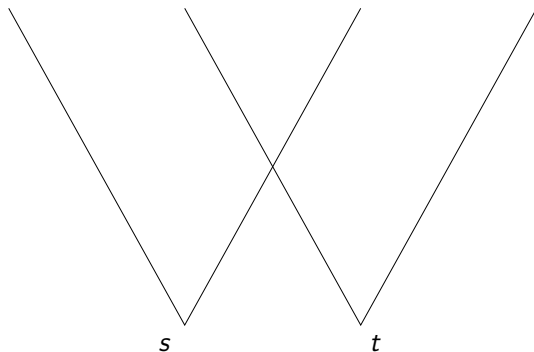
$$ch^-(s) \vee ch^-(t) \leq ch^-(s \wedge t)$$

Can be checked graphically

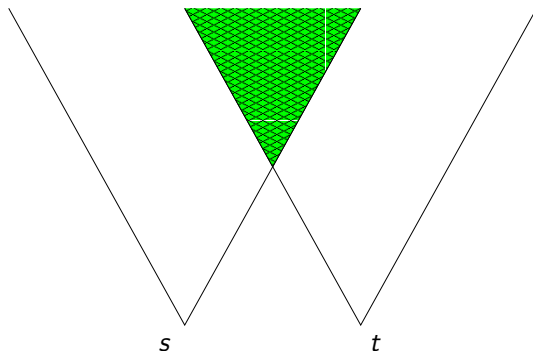
$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$



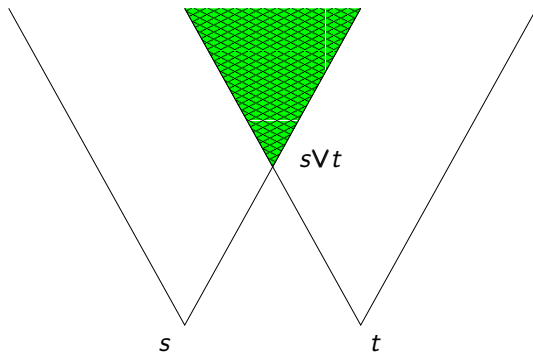
$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$



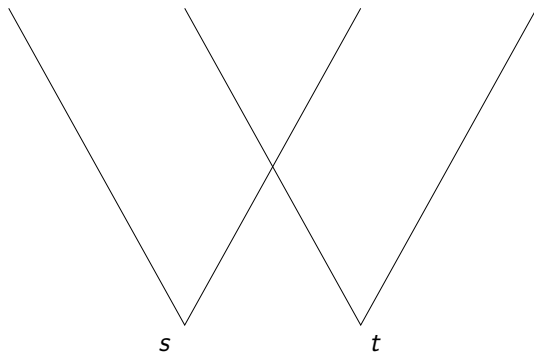
$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$



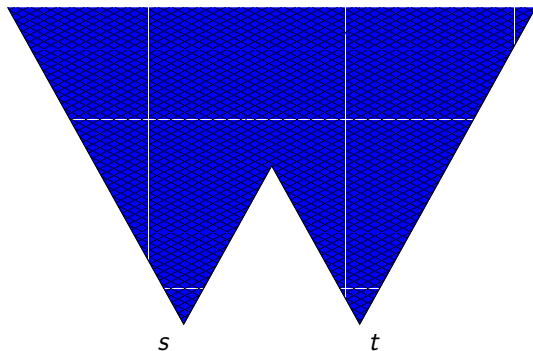
$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$



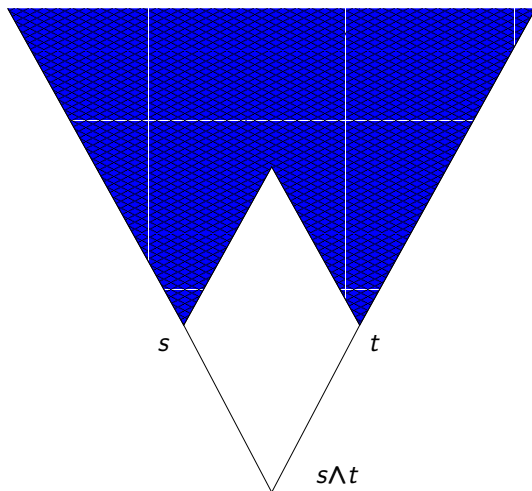
$$ch^-(s) \vee ch^-(t) \not\leq ch^-(s \wedge t)$$



$$ch^-(s) \vee ch^-(t) \not\leq ch^-(s \wedge t)$$



$$ch^-(s) \vee ch^-(t) \not\leq ch^-(s \wedge t)$$



Atoms

Checking subtyping is not always so straightforward:

Remember?

$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$

In some cases the condition to check involves **atoms** (i.e., types with a singleton interpretation)

Consider:

- two types $t_1 \neq t_2$
- $t_2 \leq t_1$
- let us try to check the relation above

Atoms

Checking subtyping is not always so straightforward:

Remember?

$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$

In some cases the condition to check involves **atoms** (i.e., types with a singleton interpretation)

Consider:

- two types $t_1 \neq t_2$
- $t_2 \leq t_1$
- let us try to check the relation above

Atoms

Checking subtyping is not always so straightforward:

Remember?

$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$

In some cases the condition to check involves **atoms** (i.e., types with a singleton interpretation)

Consider:

- two types $t_1 \neq t_2$
- $t_2 \leq t_1$
- let us try to check the relation above

Atoms

Checking subtyping is not always so straightforward:

Remember?

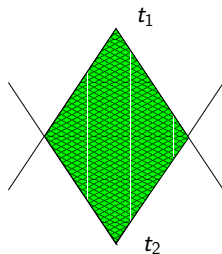
$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$

In some cases the condition to check involves **atoms** (i.e., types with a singleton interpretation)

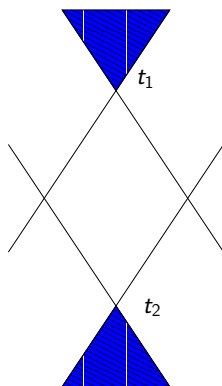
Consider:

- two types $t_1 \neq t_2$
- $t_2 \leq t_1$
- let us try to check the relation above

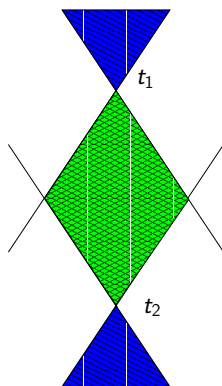
$$ch^+(t_1) \wedge ch^-(t_2) \quad ch^+(t_2) \vee ch^-(t_1)$$



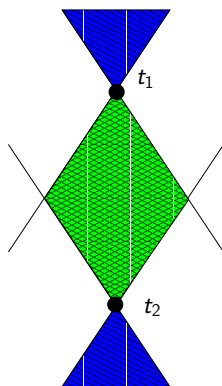
$$ch^+(t_1) \wedge ch^-(t_2) \quad ch^+(t_2) \vee ch^-(t_1)$$



$$ch^+(t_1) \wedge ch^-(t_2) \quad ch^+(t_2) \vee ch^-(t_1)$$

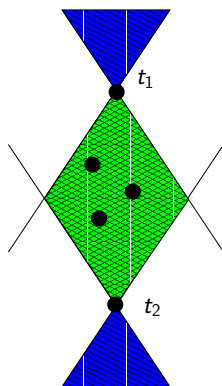


$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$



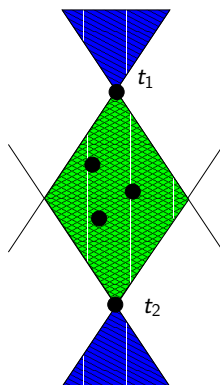
The two sets have these two points in common, namely $ch(t_1)$ and $ch(t_2)$.

$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$



The disequation holds if there are no other points in the left hand side

$$ch^+(t_1) \wedge ch^-(t_2) \leq ch^+(t_2) \vee ch^-(t_1)$$



It depends on whether $t_1 \wedge \neg t_2$ is atomic: that is whether there is nothing between t_1 and t_2

The $\pi^{\wedge, \vee}$ -calculus

Which kind of π -calculus fit these types?

Consider again

$$ch^+(t_1) \vee ch^+(t_2) \not\leq ch^+(t_1 \vee t_2)$$

- Containment is strict, so we want programs that distinguish these two types
- We must be able to dynamically check the type of messages arriving a channel
- Use **type-case** in read actions.

The $\pi^{\wedge, \vee}$ -calculus

Which kind of π -calculus fit these types?

Consider again

$$ch^+(t_1) \vee ch^+(t_2) \not\preceq ch^+(t_1 \vee t_2)$$

- Containment is strict, so we want programs that distinguish these two types
- We must be able to dynamically check the type of messages arriving a channel
- Use **type-case** in read actions.

The $\pi^{\wedge, \vee}$ -calculus

Which kind of π -calculus fit these types?

Consider again

$$ch^+(t_1) \vee ch^+(t_2) \not\leq ch^+(t_1 \vee t_2)$$

- Containment is strict, so we want programs that distinguish these two types
- We must be able to dynamically check the type of messages arriving a channel
- Use **type-case** in read actions.

The $\pi^{\wedge, \vee}$ -calculus

Which kind of π -calculus fit these types?

Consider again

$$ch^+(t_1) \vee ch^+(t_2) \not\leq ch^+(t_1 \vee t_2)$$

- Containment is strict, so we want programs that distinguish these two types
- We must be able to dynamically check the type of messages arriving a channel
- Use **type-case** in read actions.

The $\pi^{\wedge, \vee}$ -calculus

Which kind of π -calculus fit these types?

Consider again

$$ch^+(t_1) \vee ch^+(t_2) \not\preceq ch^+(t_1 \vee t_2)$$

- Containment is strict, so we want programs that distinguish these two types
- We must be able to dynamically check the type of messages arriving a channel
- Use **type-case** in read actions.

The $\pi^{\wedge, \vee}$ -calculus

<i>Channels</i>	α	$::=$	x	variables
			c^t	channel constants
<i>Processes</i>	P	$::=$	$\bar{\alpha}(\alpha)$	output
			$\sum_{i \in I} \alpha(x:t_i).P_i$	guarded input
			$P_1 \parallel P_2$	parallel
			$(\nu c^t)P$	restriction
			$!P$	replication

Reduction

$$\bar{c}_1^s(c_2^t) \parallel \sum_{i \in I} c_1^s(x:t_i)P_i \rightarrow P_j[c_2^t/x] \quad \text{if } ch(t) \leq t_j$$

The $\pi^{\wedge, \vee}$ -calculus

Channels $\alpha ::= x$ variables
 $| c^t$ channel constants

Processes $P ::= \bar{\alpha}(\alpha)$ output
 $| \sum_{i \in I} \alpha(x:t_i).P_i$ guarded input
 $| P_1 \parallel P_2$ parallel
 $| (\nu c^t)P$ restriction
 $| !P$ replication

Reduction

$$\bar{c}_1^s(c_2^t) \parallel \sum_{i \in I} c_1^s(x:t_i)P_i \rightarrow P_j[c_2^t/x] \quad \text{if } ch(t) \leq t_j$$

The $\pi^{\wedge, \vee}$ -calculus

Channels $\alpha ::= x$ variables
 $| c^t$ channel constants

Processes $P ::= \bar{\alpha}(\alpha)$ output
 $| \sum_{i \in I} \alpha(x:t_i).P_i$ guarded input
 $| P_1 \parallel P_2$ parallel
 $| (\nu c^t)P$ restriction
 $| !P$ replication

Reduction

$$\bar{c}_1^s(c_2^t) \parallel \sum_{i \in I} c_1^s(x:t_i)P_i \rightarrow P_j[c_2^t/x] \quad \text{if } ch(t) \leq t_j$$

Type case

The $\pi^{\wedge, \vee}$ -calculus

Channels $\alpha ::= x$ variables
 $| c^t$ channel constants

Processes $P ::= \bar{\alpha}(\alpha)$ output
 $| \sum_{i \in I} \alpha(x:t_i).P_i$ guarded input
 $| P_1 \parallel P_2$ parallel
 $| (\nu c^t)P$ restriction
 $| !P$ replication

Reduction

$$\bar{c}_1^s(c_2^t) \parallel \sum_{i \in I} c_1^s(x:t_i)P_i \rightarrow P_j[c_2^t/x] \quad \text{if } ch(t) \leq t_j$$

Type case

The $\pi^{\wedge, \vee}$ -calculus

Channels $\alpha ::= x$ variables
 $| c^t$ channel constants

Processes $P ::= \bar{\alpha}(\alpha)$ output
 $| \sum_{i \in I} \alpha(x:t_i).P_i$ guarded input
 $| P_1 \parallel P_2$ parallel
 $| (\nu c^t)P$ restriction
 $| !P$ replication

Reduction

$$\bar{c}_1^s(c_2^t) \parallel \sum_{i \in I} c_1^s(x:t_i)P_i \rightarrow P_j[c_2^t/x] \quad \text{if } ch(t) \leq t_j$$

Type case call-by-value

The $\pi^{\wedge, \vee}$ -calculus

<i>Channels</i>	α	$::=$	x	variables
			c^t	channel constants
<i>Processes</i>	P	$::=$	$\bar{\alpha}(\alpha)$	output
			$\sum_{i \in I} \alpha(x:t_i).P_i$	guarded input
			$P_1 \parallel P_2$	parallel
			$(\nu c^t)P$	restriction
			$!P$	replication

Reduction

$$\bar{c}_1^s(c_2^t) \parallel \sum_{i \in I} c_1^s(x:t_i)P_i \rightarrow P_j[c_2^t/x] \quad \text{if } ch(t) \leq t_j$$

Type case call-by-value

Encoding $\lambda^{\wedge, \vee}$ into $\pi^{\wedge, \vee}$.

$\lambda^{\wedge, \vee}$ -calculus types

Types	$\tau ::=$	b	basic types
		$\tau \times \tau$	product types
		$ch^+(\tau) \mid ch^-(\tau)$	channel types
		$\neg \tau$	} boolean combinators
		$\tau \wedge \tau$	
		$\tau \vee \tau$	
		0	
		1	

They may be recursive

$\lambda^{\wedge, \vee}$ -calculus types

Trade channels for arrows

Types	$\tau ::=$	b	basic types
		$\tau \times \tau$	product types
		$ch^+(\tau) \mid ch^-(\tau)$	channel types
		$\neg \tau$	} boolean combinators
		$\tau \wedge \tau$	
		$\tau \vee \tau$	
		0	
		1	

They may be recursive

$\lambda^{\wedge, \vee}$ -calculus types

Trade channels for arrows

Types	τ	$::=$	b	
			$\tau \times \tau$	basic types
			$\tau \rightarrow \tau$	product types
			$\neg \tau$	arrow types
			$\tau \wedge \tau$	} boolean combinators
			$\tau \vee \tau$	
			0	
			1	

They may be recursive

$\lambda^{\wedge, \vee}$ -calculus types

Trade channels for arrows

Types	τ	$::=$	b	
			$\tau \times \tau$	basic types
			$\tau \rightarrow \tau$	product types
			$\neg \tau$	arrow types
			$\tau \wedge \tau$	} boolean combinators
			$\tau \vee \tau$	
			0	
			1	

They may be recursive

$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$



$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$



$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$



$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$



$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_{\mathbf{f}}(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$



$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$

- As before

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$$

- Two interesting equations:

$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$

- As before

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$$

- Two interesting equations:

$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$

- As before

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$$

- Two interesting equations:

$$(\sigma \rightarrow \tau) \wedge (\sigma \rightarrow \tau') = (\sigma \rightarrow \tau \wedge \tau')$$

$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$

- As before

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$$

- Two interesting equations:

$$(\sigma \rightarrow \tau) \wedge (\sigma \rightarrow \tau') = (\sigma \rightarrow \tau \wedge \tau')$$

$$(\sigma_1 \vee \sigma_2) \rightarrow (\tau_1 \wedge \tau_2) \not\leq (\sigma_1 \rightarrow \tau_1) \wedge (\sigma_2 \rightarrow \tau_2)$$

$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$
- As before

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$$

- Two interesting equations:

$$(\sigma \rightarrow \tau) \wedge (\sigma \rightarrow \tau') = (\sigma \rightarrow \tau \wedge \tau')$$

$$(\sigma_1 \vee \sigma_2) \rightarrow (\tau_1 \wedge \tau_2) \not\leq (\sigma_1 \rightarrow \tau_1) \wedge (\sigma_2 \rightarrow \tau_2)$$

Overloading

$\lambda^{\wedge, \vee}$ set-theoretic model

- Define $\llbracket - \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$:
 - $\llbracket \tau_1 \vee \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cup \llbracket \tau_2 \rrbracket$, $\llbracket \tau_1 \wedge \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$,
 $\llbracket \neg \tau \rrbracket = \mathcal{D} \setminus \llbracket \tau \rrbracket$, $\llbracket 0 \rrbracket = \emptyset$, $\llbracket 1 \rrbracket = \mathcal{D}$.
 - $\llbracket \tau_1 \times \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$
 - $\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \overline{\mathcal{P}_f(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket)}$
- As before

$$\tau_1 \leq \tau_2 \stackrel{\text{def}}{\iff} \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$$

- Two interesting equations:

$$\begin{aligned}
 &(\sigma \rightarrow \tau) \wedge (\sigma \rightarrow \tau') = (\sigma \rightarrow \tau \wedge \tau') \\
 &(\sigma_1 \vee \sigma_2) \rightarrow (\tau_1 \wedge \tau_2) \not\leq (\sigma_1 \rightarrow \tau_1) \wedge (\sigma_2 \rightarrow \tau_2)
 \end{aligned}$$

Degenerated form of overloading

$\lambda^{\wedge, \vee}$ -terms

<i>Terms</i>	$e ::=$	x	variable
		(e, e)	pair
		$\pi_i(e)$	projections
		ee	application
		$\lambda^{\wedge_i \sigma_i \mapsto \tau_i} x. e$	abstraction
		$(e \in \tau)?e:e$	type case

$\lambda^{\wedge, \vee}$ -terms

<i>Terms</i>	$e ::=$	x	variable
		(e, e)	pair
		$\pi_i(e)$	projections
		ee	application
		$\lambda^{\wedge_i \sigma_i \rightarrow \tau_i} x. e$	abstraction
		$(e \in \tau)?e:e$	type case

$\lambda^{\wedge, \vee}$ -terms

<i>Terms</i>	$e ::=$	x	variable
		(e, e)	pair
		$\pi_i(e)$	projections
		ee	application
		$\lambda^{\wedge_i \sigma_i \rightarrow \tau_i} x. e$	abstraction
		$(e \in \tau)?e:e$	type case

Overloaded function **opposite** : $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

$\lambda^{(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})} x. (x \in \text{int})(-x):(-x)$

$\lambda^{\wedge, \vee}$ -terms

<i>Terms</i>	$e ::=$	x	variable
		(e, e)	pair
		$\pi_i(e)$	projections
		ee	application
		$\lambda^{\wedge_i \sigma_i \rightarrow \tau_i} x. e$	abstraction
		$(e \in \tau)?e:e$	type case

Overloaded function **opposite** : $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

$$\lambda^{(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})} x. (x \in \text{int}) (\neg x) : (\neg x)$$

Note that

$$\mathbf{opposite} \notin (\text{int} \vee \text{bool}) \rightarrow \underbrace{(\text{int} \wedge \text{bool})}_0$$

The broken but instructive encoding

Functions as processes

Milner's encoding:

$$\{\{x\}\}_\alpha = \bar{\alpha}(x)$$

$$\{\{\lambda x.e\}\}_\alpha = (\nu f)(\bar{\alpha}(f) \parallel !(f(x, b).\{\{e\}\}_b))$$

$$\{\{e_1 e_2\}\}_\alpha = (\nu a)(\nu b)(\{\{e_1\}\}_a \parallel a(w).(\{\{e_2\}\}_b \parallel b(h).\bar{w}(h, \alpha)))$$

Intuition:

- A function is called by providing an input and a channel on which the output should be returned
- Akin to CPS transform

The encoding of types

All types can be encoded homomorphically

$$\{\{\sigma \times \tau\}\} = \{\{\sigma\}\} \times \{\{\tau\}\}$$

$$\{\{\sigma \wedge \tau\}\} = \{\{\sigma\}\} \wedge \{\{\tau\}\}$$

$$\vdots$$

Except arrow types:

$$\{\{\sigma \rightarrow \tau\}\} = ch^{-}(\{\{\sigma\}\} \times ch^{-}(\{\{\tau\}\}))$$

A function is represented by a channel on which one can send the input value and the return channel

Problem

The encoding of types does not respect the equations

We have

$$\sigma \rightarrow (\tau_1 \wedge \tau_2) = (\sigma \rightarrow \tau_1) \wedge (\sigma \rightarrow \tau_2)$$

but

$$\{\{\sigma \rightarrow (\tau_1 \wedge \tau_2)\}\} \not\leq \{\{(\sigma \rightarrow \tau_1) \wedge (\sigma \rightarrow \tau_2)\}\}$$

Problem

The encoding of types does not respect the equations

Technically the problem is in the inequation

$$ch^-(s \wedge t) \not\leq ch^-(s) \vee ch^-(t)$$

that implies

$$\{\{\sigma \rightarrow (\tau_1 \wedge \tau_2)\}\} \not\leq \{\{(\sigma \rightarrow \tau_1) \wedge (\sigma \rightarrow \tau_2)\}\}$$

Problem

The reason is “output-based” overloading

A $\lambda^{\wedge, \vee}$ -function does type-case matching only on input:

$$\sigma \rightarrow (\tau_1 \wedge \tau_2) = (\sigma \rightarrow \tau_1) \wedge (\sigma \rightarrow \tau_2)$$

Encoding can perform type-case on the expected output too

$$\{\{\sigma \rightarrow (\tau_1 \wedge \tau_2)\}\} \preceq \{\{(\sigma \rightarrow \tau_1) \wedge (\sigma \rightarrow \tau_2)\}\}$$

The working encoding

Toward a solution

We want to stay as close as possible to Milner's encoding:

$$\{\sigma \rightarrow \tau\} = ch^{-}(\{\sigma\} \times ch^{-}(\{\tau\}))$$

where:

1. $ch^{-}(t) \leq ch^{\lambda}(t)$
2. $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$
3. $s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$

Toward a solution

We want to stay as close as possible to Milner's encoding:

$$\{\sigma \rightarrow \tau\} = ch^{-}(\{\sigma\} \times ch^{\lambda}(\{\tau\}))$$

where:

1. $ch^{-}(t) \leq ch^{\lambda}(t)$
2. $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$
3. $s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$

Toward a solution

We want to stay as close as possible to Milner's encoding:

$$\{\sigma \rightarrow \tau\} = ch^{-}(\{\sigma\} \times ch^{\lambda}(\{\tau\}))$$

where:

1. $ch^{-}(t) \leq ch^{\lambda}(t)$
2. $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$
3. $s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$

Toward a solution

We want to stay as close as possible to Milner's encoding:

$$\{\sigma \rightarrow \tau\} = ch^{-}(\{\sigma\} \times ch^{\lambda}(\{\tau\}))$$

where:

$$1. \quad ch^{-}(t) \leq ch^{\lambda}(t)$$

it can transport return channels

$$2. \quad ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$$

$$3. \quad s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$$

Toward a solution

We want to stay as close as possible to Milner's encoding:

$$\{\sigma \rightarrow \tau\} = ch^{-}(\{\sigma\} \times ch^{\lambda}(\{\tau\}))$$

where:

1. $ch^{-}(t) \leq ch^{\lambda}(t)$ it can transport return channels
 2. $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$
 3. $s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$
- } it preserves type equations

Toward a solution

We want to stay as close as possible to Milner's encoding:

$$\{\sigma \rightarrow \tau\} = ch^{-}(\{\sigma\} \times ch^{\lambda}(\{\tau\}))$$

where:

1. $ch^{-}(t) \leq ch^{\lambda}(t)$ it can transport return channels
 2. $ch^{\lambda}(s \wedge t) = ch^{\lambda}(s) \vee ch^{\lambda}(t)$
 3. $s \leq t \iff ch^{\lambda}(t) \leq ch^{\lambda}(s)$
- } it preserves type equations

Does it exist?

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(-ch^-(-t) \vee ch(\mathbb{1}))$$

$$\bullet \quad ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t);$$

$$\bullet \quad s \leq t \implies ch^\lambda(s) \leq ch^\lambda(t);$$

$$\bullet \quad ch^-(\mathbb{1}) \leq ch^\lambda(t).$$

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(-ch^-(t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$; must distribute on intersections
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$; must distribute on intersections
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

must be contravariant

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

must be contravariant

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

must contain the channel type

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

must contain the channel type

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Key point

$ch^\lambda(t)$ adds to $ch^-(t)$ all channels that can send values both inside and outside t . That is, all the channels for which the result of checking if their messages are of type t cannot be predicted.

Definition of $ch^\lambda(-)$

Define $ch^\lambda(t)$ as

$$(\neg ch^-(\neg t) \vee ch(\mathbb{1}))$$

- ① $ch^\lambda(s \wedge t) = ch^\lambda(s) \vee ch^\lambda(t)$;
- ② $s \leq t \iff ch^\lambda(t) \leq ch^\lambda(s)$;
- ③ $ch^-(t) \leq ch^\lambda(t)$;

Observe:

$$\llbracket ch^\lambda(t) \setminus ch^-(t) \rrbracket = \{c^s \mid t \not\leq s \ \& \ \neg t \not\leq s\} \cup \{c^{\mathbb{1}}\}$$

Type obfuscation

λ -channels introduce some latent noise that makes it impossible to determine which (output) type they encode

Definition of $ch^\lambda(-)$

This is why it works.

- It rules out “output-based” overloading functions
- It is impossible to test whether a channel of type $ch^\lambda(s) \vee ch^\lambda(t)$ was meant to be of type $ch^\lambda(s)$ or $ch^\lambda(t)$;
- The caller can in principle confuse such functions (by using a channel that is in $ch^\lambda(t) \setminus ch^-(t)$)
- Callers don't do it, but types must reflect this

Type obfuscation

λ -channels introduce some latent noise that makes it impossible to determine which (output) type they encode

Definition of $ch^\lambda(-)$

This is why it works.

- It rules out “output-based” overloading functions
- It is impossible to test whether a channel of type $ch^\lambda(s) \vee ch^\lambda(t)$ was meant to be of type $ch^\lambda(s)$ or $ch^\lambda(t)$;
- The caller can in principle confuse such functions (by using a channel that is in $ch^\lambda(t) \setminus ch^-(t)$)
- Callers don't do it, but types must reflect this

Like the Police in Utopia

Everybody behaves well in Utopia, but the Police is the visible representation of the fact the everybody behaves well

Encoding of types

A map $\{\!\{-\}\!\} : \mathcal{T}_\lambda \rightarrow \mathcal{T}_\pi$

- $\{\!\{0\}\!\} = 0, \quad \{\!\{1\}\!\} = 1$
- $\{\!\{\neg\tau\}\!\} = \neg\{\!\{\tau\}\!\}$
- $\{\!\{\sigma \vee \tau\}\!\} = \{\!\{\sigma\}\!\} \vee \{\!\{\tau\}\!\}, \quad \{\!\{\sigma \wedge \tau\}\!\} = \{\!\{\sigma\}\!\} \wedge \{\!\{\tau\}\!\}$
- $\{\!\{\sigma \times \tau\}\!\} = \{\!\{\sigma\}\!\} \times \{\!\{\tau\}\!\}$
- $\{\!\{\sigma \rightarrow \tau\}\!\} = ch^-(\{\!\{\sigma\}\!\} \times ch^\lambda(\{\!\{\tau\}\!\}))$

Theorem

Let σ and τ be $\lambda^{\wedge, \vee}$ -types, then:

$$\sigma \leq \tau \iff \{\!\{\sigma\}\!\} \leq \{\!\{\tau\}\!\}.$$

Encoding of terms

A glance at the encoding

- $\{(e \in t)?e_1:e_2\}_\alpha =$
 $(\nu a)((a(x:\{t\}).\{e_1\}_\alpha + a(x:\neg\{t\}).\{e_2\}_\alpha) \parallel \{e\}_a)$
- $\{\mathbf{opposite}\}_\alpha =$
 $(\nu f)(\overline{\alpha}(f) \parallel f(x:\text{int}, b:\text{ch}^-(\text{int})).\{-x\}_b$
 $+ f(x:\text{bool}, b:\text{ch}^-(\text{bool})).\{\neg x\}_b$
 $+ f(x:\text{int} \vee \text{bool}, b:\text{"noise"}).0)$

Recall the type of **opposite** is $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

$$\text{"noise"} = (\text{ch}^\lambda(\text{int}) \setminus \text{ch}^-(\text{int})) \vee (\text{ch}^\lambda(\text{bool}) \setminus \text{ch}^-(\text{bool}))$$

Correctness

Theorem

The encoding preserves typability

Theorem

The encoding preserves behaviour:

If $e \longrightarrow^ v$, then $\{e\}_c \longrightarrow^* \cong \{v\}_c \not\longrightarrow^*$*

If e diverges, then so $\{e\}_c$ does.

So what?

Contribution: yet another encoding!

However

- it shows the relevance of semantic subtyping
- it is a study of overloading and CPS
- the encoded function has more power
(emphasized by the presence of type case)
- introduction of "type obfuscation" technique

So what?

Contribution: yet another encoding!

However

- it shows the relevance of semantic subtyping
- it is a study of overloading and CPS
- the encoded function has more power
(emphasized by the presence of type case)
- introduction of “type obfuscation” technique

So what?

Contribution: yet another encoding!

However

- it shows the relevance of semantic subtyping
- it is a study of overloading and CPS
- the encoded function has more power
(emphasized by the presence of type case)
- introduction of “type obfuscation” technique

So what?

Contribution: yet another encoding!

However

- it shows the relevance of semantic subtyping
- it is a study of overloading and CPS
- the encoded function has more power
(emphasized by the presence of type case)
- introduction of “type obfuscation” technique

PART 4: CONCLUSION

Conclusion

- **Regex patterns start from two simple ideas:**

- Use the same constructors for types and value
- Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**

- Patterns are composed of small, reusable building blocks, and differences behave sub-structurally
- Concepts are easier for the programmer (e.g. subtyping)
- Informative error messages
- Flexible and powerful specification language

- **Several benefits:**

- Types yield the significant reduction in code size (e.g. 1000 lines of code to 100 lines) and the reduction of boilerplate code
- Discover some useful, previously unknown properties such as the Church-Turing thesis
- High expressivity in writing queries, rewriting, compiler transformation, etc.
- High generality without the need of introducing new specific languages (e.g. writing data parsing, logical specifications, etc.)
- High generality in applications

Conclusion

- **Regex patterns start from two simple ideas:**

- Use the same constructors for types and value
- Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**

- Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
- Concepts are simpler for the programmer (e.g. subtyping)
- Informative error messages
- Flexible and powerful specification language

- **Several benefits:**

- Types and values share the same constructors and operators
- Use the same constructors and operators for types and values
- Close connection with the regular expressions, pattern matching, and boolean combinators
- Tightly connected with the queries, pattern, groups, transformations
- Simple and efficient framework for specifying non-specific operations
- Operations using value binding, local and sections

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**

Types are easy to read and understand, and can be used to specify complex patterns in a concise way. The patterns are easy to write and understand, and can be used to specify complex patterns in a concise way. The patterns are easy to write and understand, and can be used to specify complex patterns in a concise way.

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**
 - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [20] (see also XQuery Jc07-Cat5 benchmarks).

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**
 - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
 - High precision in typing queries, iterators, complex transformations.
 - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications,...)

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**
 - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
 - High precision in typing queries, iterators, complex transformations.
 - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, ...)

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**
 - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
 - High precision in typing queries, iterators, complex transformations.
 - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, ...)

Conclusion

- **Regex patterns start from two simple ideas:**
 - Use the same constructors for types and value
 - Define patterns as types with capture variables
- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
 - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
 - Concepts are easier for the programmer (e.g. subtyping)
 - Informative error messages.
 - Precise and powerful specification language
- **Several benefits:**
 - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [[XMark and XQuery Use Cases benchmarks](#)].
 - High precision in typing queries, iterators, complex transformations.
 - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, ...)

PART 5: REFERENCES

References

On the web

- www.cduce.org. Distribution, documentations, tutorials, mailing lists, tips and tricks,...
- Um [curso](#) de programação CDuce, **em Português**, está disponível na Universidade Nova de Lisboa

Language

- V. Benzaken, G. Castagna, and A. Frisch:
[CDuce: an XML-Centric General-Purpose Language](#). In ICFP '03, ACM Press, 2003.
- H. Hosoya, A. Frisch, and G. Castagna:
[Parametric Polymorphism for XML](#). In POPL '05, ACM Press, 2005.
- G. Castagna, D. Colazzo, and A. Frisch:
[Error Mining for Regular Expression Patterns](#). In ICTCS 2005, LNCS n.3701, Springer, 2005.

References

On the web

- www.cduce.org. Distribution, documentations, tutorials, mailing lists, tips and tricks,...
- Um [curso](#) de programação CDuce, *em Português*, está disponível na Universidade Nova de Lisboa

Language

- V. Benzaken, G. Castagna, and A. Frisch:
[CDuce: an XML-Centric General-Purpose Language](#). In ICFP '03, ACM Press, 2003.
- H. Hosoya, A. Frisch, and G. Castagna:
[Parametric Polymorphism for XML](#). In POPL '05, ACM Press, 2005.
- G. Castagna, D. Colazzo, and A. Frisch:
[Error Mining for Regular Expression Patterns](#). In ICTCS 2005, LNCS n.3701, Springer, 2005.

References (continued)

Theory

- G. Castagna and A. Frisch: [A gentle introduction to semantic subtyping](#). In PPDP '05, ACM Press (full version) and ICALP '05, LNCS n. 3580, Springer (summary), 2005. Joint ICALP-PPDP keynote talk.
- A. Frisch, G. Castagna, and V. Benzaken: [Semantic Subtyping](#). In LICS '02, IEEE Computer Society Press, 2002.
- G. Castagna: [Semantic subtyping: challenges, perspectives, and open problems](#). In ICTCS 2005, LNCS 3701, Springer, 2005.

Queries

- V. Benzaken, G. Castagna, and C. Miachon: [A Full Pattern-based Paradigm for XML Query Processing](#). In PADL '05, LNCS n.3350, Springer, January, 2005.
- G. Castagna: [Patterns and types for querying XML](#). In Proceedings of DBPL 2005, LNCS n.3774 (full version) and XSym 2005, LNCS n.3671 (summary), Springer, 2005. Joint invited talk.
- V. Benzaken G. Castagna, D. Colazzo, and K. Nguyen: [Type-Based XML Projection](#). In VLDB 2006, pag.271-282, 2006.

References (continued)

Theory

- G. Castagna and A. Frisch: [A gentle introduction to semantic subtyping](#). In PPDP '05, ACM Press (full version) and ICALP '05, LNCS n. 3580, Springer (summary), 2005. Joint ICALP-PPDP keynote talk.
- A. Frisch, G. Castagna, and V. Benzaken: [Semantic Subtyping](#). In LICS '02, IEEE Computer Society Press, 2002.
- G. Castagna: [Semantic subtyping: challenges, perspectives, and open problems](#). In ICTCS 2005, LNCS 3701, Springer, 2005.

Queries

- V. Benzaken, G. Castagna, and C. Miachon: [A Full Pattern-based Paradigm for XML Query Processing](#). In PADL '05, LNCS n.3350, Springer, January, 2005.
- G. Castagna: [Patterns and types for querying XML](#). In Proceedings of DBPL 2005, LNCS n.3774 (full version) and XSym 2005, LNCS n.3671 (summary), Springer, 2005. Joint invited talk.
- V. Benzaken G. Castagna, D. Colazzo, and K. Nguyen: [Type-Based XML Projection](#). In VLDB 2006, pag.271-282, 2006.

References (end)

Implementation

- A. Frisch [Regular tree language recognition with static information](#) In TCS 2004 LNCS Springer.
- A. Frisch, and L. Cardelli [Greedy regular expression matching](#). In ICALP 2004 LNCS Springer.

Concurrency

- G. Castagna, R. De Nicola, and D. Varacca:
[Semantic subtyping for the Pi-calculus](#). In LICS '05, IEEE Computer Society Press, 2005.
- G. Castagna, M. Dezani-Ciancaglini, and D. Varacca:
[Encoding CDuce into the Pi-calculus](#). In CONCUR 200, LNCS n.4137, LNCS, Springer, 2006.

Security

- V. Benzaken, M. Burelle, and G. Castagna:
[Information flow security for XML transformations](#). In ASIAN '03, LNCS n.2896, LNCS, Springer, December, 2003.

References (end)

Implementation

- A. Frisch [Regular tree language recognition with static information](#) In TCS 2004 LNCS Springer.
- A. Frisch, and L. Cardelli [Greedy regular expression matching](#). In ICALP 2004 LNCS Springer.

Concurrency

- G. Castagna, R. De Nicola, and D. Varacca:
[Semantic subtyping for the Pi-calculus](#). In LICS '05, IEEE Computer Society Press, 2005.
- G. Castagna, M. Dezani-Ciancaglini, and D. Varacca:
[Encoding CDuce into the Pi-calculus](#). In CONCUR 200, LNCS n.4137, LNCS, Springer, 2006.

Security

- V. Benzaken, M. Burelle, and G. Castagna:
[Information flow security for XML transformations](#). In ASIAN '03, LNCS n.2896, LNCS, Springer, December, 2003.

References (end)

Implementation

- A. Frisch [Regular tree language recognition with static information](#) In TCS 2004 LNCS Springer.
- A. Frisch, and L. Cardelli [Greedy regular expression matching](#). In ICALP 2004 LNCS Springer.

Concurrency

- G. Castagna, R. De Nicola, and D. Varacca:
[Semantic subtyping for the Pi-calculus](#). In LICS '05, IEEE Computer Society Press, 2005.
- G. Castagna, M. Dezani-Ciancaglini, and D. Varacca:
[Encoding CDuce into the Pi-calculus](#). In CONCUR 200, LNCS n.4137, LNCS, Springer, 2006.

Security

- V. Benzaken, M. Burelle, and G. Castagna:
[Information flow security for XML transformations](#). In ASIAN '03, LNCS n.2896, LNCS, Springer, December, 2003.

PART 5: ADDITIONAL SLIDES

Addendum 1: a model may not exist

$$t = \text{int} \vee (\text{ref}(\text{int}) \wedge \text{ref}(t))$$

Is t equal to int ?

$$t = \text{int} \iff (\text{ref}(\text{int}) \wedge \text{ref}(t)) = \emptyset \iff t \neq \text{int}$$

but also

$$t \neq \text{int} \iff (\text{ref}(\text{int}) \wedge \text{ref}(t)) \neq \emptyset \iff t = \text{int}$$

Addendum 1: a model may not exist

$$t = \text{int} \vee (\text{ref}(\text{int}) \wedge \text{ref}(t))$$

Is t equal to int ?

$$t = \text{int} \iff (\text{ref}(\text{int}) \wedge \text{ref}(t)) = \emptyset \iff t \neq \text{int}$$

but also

$$t \neq \text{int} \iff (\text{ref}(\text{int}) \wedge \text{ref}(t)) \neq \emptyset \iff t = \text{int}$$

Addendum 1: a model may not exist

$$t = \text{int} \vee (\text{ref}(\text{int}) \wedge \text{ref}(t))$$

Is t equal to int ?

$$t = \text{int} \iff (\text{ref}(\text{int}) \wedge \text{ref}(t)) = \emptyset \iff t \neq \text{int}$$

but also

$$t \neq \text{int} \iff (\text{ref}(\text{int}) \wedge \text{ref}(t)) \neq \emptyset \iff t = \text{int}$$

Addendum 2: the real *abstr* typing rule

$$t \equiv (\bigwedge_{i=1..n} s_i \rightarrow t_i) \setminus (\bigvee_{j=1..m} s'_j \rightarrow t'_j) \not\leq \mathbb{0}$$

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

Addendum 3: A different definition for $\mathbb{E}[\]$

Note that according to the previous $\mathbb{E}[\]$:

$$s \rightarrow t \leq \mathbb{1} \rightarrow \mathbb{1} \quad (3)$$

Every application is well typed. Add a distinguished Ω to denote a runtime type error, modify

$$\mathbb{E}[t \rightarrow s] = \{f \subseteq \mathcal{D} \times (\mathcal{D} \cup \{\Omega\}) \mid \forall (d_1, d_2) \in f. d_1 \in \llbracket t \rrbracket \Rightarrow d_2 \in \llbracket s \rrbracket\}$$

(3) no longer holds since the constant map $\neg s \mapsto \Omega$, is in the left hand type but not in the right one.

$$\begin{aligned} \bigwedge_{i \in I} (t_i \rightarrow s_i) \leq \bigvee_{j \in J} (t'_j \rightarrow s'_j) &\iff \\ \exists j \in J. \left\{ \begin{array}{l} t'_j \leq \bigvee_{i \in I} t_i \wedge \\ \forall I' \subseteq I. (t'_j \leq \bigvee_{i \in I'} t_i) \vee (\bigwedge_{i \in I \setminus I'} s_i \leq s'_j) \end{array} \right. \end{aligned}$$

Addendum 3: A different definition for $\mathbb{E}[\]$

Note that according to the previous $\mathbb{E}[\]$:

$$s \rightarrow t \leq \mathbb{1} \rightarrow \mathbb{1} \quad (3)$$

Every application is well typed. Add a distinguished Ω to denote a runtime type error, modify

$$\mathbb{E}[t \rightarrow s] = \{f \subseteq \mathcal{D} \times (\mathcal{D} \cup \{\Omega\}) \mid \forall (d_1, d_2) \in f. d_1 \in \llbracket t \rrbracket \Rightarrow d_2 \in \llbracket s \rrbracket\}$$

(3) no longer holds since the constant map $\neg s \mapsto \Omega$, is in the left hand type but not in the right one.

$$\begin{aligned} \bigwedge_{i \in I} (t_i \rightarrow s_i) \leq \bigvee_{j \in J} (t'_j \rightarrow s'_j) &\iff \\ \exists j \in J. \left\{ \begin{array}{l} t'_j \leq \bigvee_{i \in I} t_i \wedge \\ \forall I' \subseteq I. (t'_j \leq \bigvee_{i \in I'} t_i) \vee (\bigwedge_{i \in I \setminus I'} s_i \leq s'_j) \end{array} \right. \end{aligned}$$

Addendum 3: A different definition for $\mathbb{E}[\]$

Note that according to the previous $\mathbb{E}[\]$:

$$s \rightarrow t \leq \mathbb{1} \rightarrow \mathbb{1} \quad (3)$$

Every application is well typed. Add a distinguished Ω to denote a runtime type error, modify

$$\mathbb{E}[t \rightarrow s] = \{f \subseteq \mathcal{D} \times (\mathcal{D} \cup \{\Omega\}) \mid \forall (d_1, d_2) \in f. d_1 \in \llbracket t \rrbracket \Rightarrow d_2 \in \llbracket s \rrbracket\}$$

(3) no longer holds since the constant map $\neg s \mapsto \Omega$, is in the left hand type but not in the right one.

$$\begin{aligned} \bigwedge_{i \in I} (t_i \rightarrow s_i) \leq \bigvee_{j \in J} (t'_j \rightarrow s'_j) &\iff \\ \exists j \in J. \left\{ \begin{array}{l} t'_j \leq \bigvee_{i \in I} t_i \wedge \\ \forall I' \subseteq I. (t'_j \leq \bigvee_{i \in I'} t_i) \vee (\bigwedge_{i \in I \setminus I'} s_i \leq s'_j) \end{array} \right. \end{aligned}$$

Addendum 3: A different definition for $\mathbb{E}[\]$

Note that according to the previous $\mathbb{E}[\]$:

$$s \rightarrow t \leq \mathbb{1} \rightarrow \mathbb{1} \quad (3)$$

Every application is well typed. Add a distinguished Ω to denote a runtime type error, modify

$$\mathbb{E}[t \rightarrow s] = \{f \subseteq \mathcal{D} \times (\mathcal{D} \cup \{\Omega\}) \mid \forall (d_1, d_2) \in f. d_1 \in \llbracket t \rrbracket \Rightarrow d_2 \in \llbracket s \rrbracket\}$$

(3) no longer holds since the constant map $\neg s \mapsto \Omega$, is in the left hand type but not in the right one.

$$\begin{aligned} \bigwedge_{i \in I} (t_i \rightarrow s_i) \leq \bigvee_{j \in J} (t'_j \rightarrow s'_j) &\iff \\ \exists j \in J. \left\{ \begin{array}{l} t'_j \leq \bigvee_{i \in I} t_i \wedge \\ \forall I' \subseteq I. (t'_j \leq \bigvee_{i \in I'} t_i) \vee (\bigwedge_{i \in I \setminus I'} s_i \leq s'_j) \end{array} \right. \end{aligned}$$