Polymorphic Type Inference for Dynamic Languages

Reconstructing Types for Systems combining
Parametric, Ad-Hoc, and Subtyping Polymorphism

Mickaël Laurent, supervised by Giuseppe Castagna and Kim Nguyen

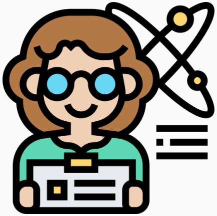
June 21, 2024

IRIF (Université Paris Cité, France), LMF (Université Paris-Saclay, France)

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

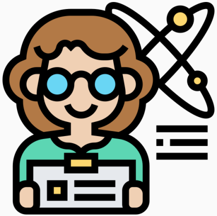
Laboratoire
Méthodes
Formelles

Introduction



country	city	pop	density	⋯
USA	Chicago	2665039	4398	⋯
USA	Boston	675647	2911	⋯
France	Gif-sur-Yvette	22352	1900	⋯
France	Pontamafrey	307	26	⋯
⋮	⋮	⋮	⋮	⋱

Introduction



country	city	pop	density	⋯
USA	Chicago	2665039	4398	⋯
USA	Boston	675647	2911	⋯
France	Gif-sur-Yvette	22352	1900	⋯
France	Pontamafrey	307	26	⋯
⋮	⋮	⋮	⋮	⋱

How to retrieve the population of a city?

get_population in Rust

```rust
fn get_population(data: &str, city: &str) -> Option<u32> {
    let mut rdr = csv::Reader::from_reader(data.as_bytes());
    let city_index = rdr.headers().unwrap().iter()
      .position(|h| h == "city").unwrap();
    let pop_index = rdr.headers().unwrap().iter()
      .position(|h| h == "pop").unwrap();
    for result in rdr.records() {
        let record = result.unwrap();
        if record.get(city_index).unwrap() == v {
            return Some(record.get(pop_index).parse().unwrap());
        }
    }
    None
}
```

get_population in Rust

```rust
fn get_population(data: &str, city: &str) -> Option<u32> {
    let mut rdr = csv::Reader::from_reader(data.as_bytes());
    let city_index = rdr.headers().unwrap().iter()
      .position(|h| h == "city").unwrap();
    let pop_index = rdr.headers().unwrap().iter()
      .position(|h| h == "pop").unwrap();
    for result in rdr.records() {
        let record = result.unwrap();
        if record.get(city_index).unwrap() == v {
            return Some(record.get(pop_index).parse().unwrap());
        }
    }
    None
}
```

get_population in Rust

```rust
fn get_population(data: &str, city: &str) -> Option<u32> {
    let mut rdr = csv::Reader::from_reader(data.as_bytes());
    let city_index = rdr.headers().unwrap().iter()
      .position(|h| h == "city").unwrap();
    let pop_index = rdr.headers().unwrap().iter()
      .position(|h| h == "pop").unwrap();
    for result in rdr.records() {
        let record = result.unwrap();
        if record.get(city_index).unwrap() == v {
            return Some(record.get(pop_index).parse().unwrap());
        }
    }
    None
}
```

get_population in Rust

```rust
fn get_population(data: &str, city: &str) -> Option<u32> {
    let mut rdr = csv::Reader::from_reader(data.as_bytes());
    let city_index = rdr.headers().unwrap().iter()
      .position(|h| h == "city").unwrap();
    let pop_index = rdr.headers().unwrap().iter()
      .position(|h| h == "pop").unwrap();
    for result in rdr.records() {
        let record = result.unwrap();
        if record.get(city_index).unwrap() == v {
            return Some(record.get(pop_index).parse().unwrap());
        }
    }
    None
}
```

get_population in Python

```python
def get_population(data, city):
    d = csv.DictReader(StringIO(data))
    for row in d:
        if row['city'] == city:
            return int(row['pop'])
    return None
```

get_population in Python

```python
def get_population(data, city):
    d = csv.DictReader(StringIO(data))
    for row in d:
        if row['city'] == city:
            return int(row['pop'])
    return None
```

No need to unwrap or pattern-match the result like in Rust.

In Rust:

```
get_population(data, "Gif-sur-Yvette").unwrap() / 1000          // 22
```

In Python:

```
get_population(data, "Gif-sur-Yvette") // 1000                  # 22
```

get_population in Python

```python
def get_population(data, city):
    d = csv.DictReader(StringIO(data))
    for row in d:
        if row['city'] == city:
            return int(row['pop'])
    return None

def in_thousands(n):
    if type(n) is int:
        return n // 1000
    else:
        return None
```

get_population in Python

```python
def get_population(data, city):
    d = csv.DictReader(StringIO(data))
    for row in d:
        if row['city'] == city:
            return int(row['pop'])
    return None

def in_thousands(n):
    if type(n) is int:
        return n // 1000
    else:
        return None

in_thousands(get_population(data, "Gif-sur-Yvette"))  # 22
in_thousands(get_population(data, "Jpeg-sur-Yvette")) # None
```

Dynamic Languages

✓ Programmer does not need to write type annotations

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types

  ⇒ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)

Dynamic Languages

✓ Programmer does not need to write type annotations

✓ Functions can accept and return data of different types

$\Rightarrow$ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)

✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types

  ⇒ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

⇒ Flexible, concise, good for experimenting and prototyping

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types
  ⇒ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

⇒ Flexible, concise, good for experimenting and prototyping

- × It is not clear where program can fail (no explicit `unwrap`, etc.)

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types
  $\Rightarrow$ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

$\Rightarrow$ Flexible, concise, good for experimenting and prototyping

- × It is not clear where program can fail (no explicit `unwrap`, etc.)
- × No type safety guarantees (`TypeError` exceptions can be raised at runtime)

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types
  ⇒ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

⇒ Flexible, concise, good for experimenting and prototyping

- × It is not clear where program can fail (no explicit `unwrap`, etc.)
- × No type safety guarantees (`TypeError` exceptions can be raised at runtime)
- × No static type ⇒ provide little information to the programmer (documentation) ...

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types
  ⇒ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

⇒ Flexible, concise, good for experimenting and prototyping

- × It is not clear where program can fail (no explicit `unwrap`, etc.)
- × No type safety guarantees (`TypeError` exceptions can be raised at runtime)
- × No static type ⇒ provide little information to the programmer (documentation) ...
- × ... and to the toolchain (optimizer, linter, auto-complete, etc.)

Dynamic Languages

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types
  ⇒ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch: which implementation to execute is determined at runtime

⇒ Flexible, concise, good for experimenting and prototyping

- × It is not clear where program can fail (no explicit `unwrap`, etc.)
- × No type safety guarantees (`TypeError` exceptions can be raised at runtime)
- × No static type ⇒ provide little information to the programmer (documentation) ...
- × ... and to the toolchain (optimizer, linter, auto-complete, etc.)

⇒ Unsafe, bad for production code and maintenance of large projects

Motivations

Goal: statically typing dynamic languages without hindering their flexibility.

Motivations

Goal: statically typing dynamic languages without hindering their flexibility.

- ✓ Programmer does not need to write type annotations
- ✓ Functions can accept and return data of different types
- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch

Motivations

Goal: statically typing dynamic languages without hindering their flexibility.

- ✓ Programmer does not need to write type annotations
  $\Rightarrow$ Static types should be inferred (as much as possible)

- ✓ Functions can accept and return data of different types

- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch

Motivations

Goal: statically typing dynamic languages without hindering their flexibility.

- ✓ Programmer does not need to write type annotations
  $\Rightarrow$ Static types should be inferred (as much as possible)

- ✓ Functions can accept and return data of different types
  $\Rightarrow$ Our type system should feature union types and subtyping

- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch

Motivations

Goal: statically typing dynamic languages without hindering their flexibility.

- ✓ Programmer does not need to write type annotations
  $\Rightarrow$ Static types should be inferred (as much as possible)

- ✓ Functions can accept and return data of different types
  $\Rightarrow$ Our type system should feature union types and subtyping

- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch
  $\Rightarrow$ Our type system should be able to type type-cases
  and capture overloaded behaviors using intersection types

Motivations

Goal: statically typing dynamic languages without hindering their flexibility.

- ✓ Programmer does not need to write type annotations
  $\Rightarrow$ Static types should be inferred (as much as possible)

- ✓ Functions can accept and return data of different types
  $\Rightarrow$ Our type system should feature union types and subtyping

- ✓ Overloaded functions, via explicit type-cases or via dynamic dispatch
  $\Rightarrow$ Our type system should be able to type type-cases
  and capture overloaded behaviors using intersection types

My contribution: conception of a type system for a language with type-cases, featuring many forms of polymorphism (parametric, ad-hoc, subtyping) and a type inference.

Summary

Types & Core Language

Declarative Type System

Algorithmic Type System

Reconstruction of the Annotation Tree

Conclusion and Perspective

Types & Core Language

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with `ToBoolean`:

- For `false`, `null`, `0`, `±0.0`, `""`, etc. ⇒ returns `false`
- For other values ⇒ returns `true`

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\mathtt{ToBoolean}:$

- For $\underbrace{\mathtt{false}, \mathtt{null}, 0, \pm 0.0, \mathtt{""}, \text{ etc.}}_{\mathtt{Falsy}}$ $\Rightarrow$ returns $\mathtt{false}$
- For $\underbrace{\text{other values}}_{\mathtt{Truthy}}$ $\Rightarrow$ returns $\mathtt{true}$

```
type Falsy = false | null | 0 | 0.0 | ""
type Truthy = ~Falsy
```

$$\mathtt{ToBoolean}: \ (\mathtt{Falsy} \to \mathtt{false}) \wedge (\mathtt{Truthy} \to \mathtt{true})$$

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: (\texttt{Falsy} \to \texttt{false}) \wedge (\texttt{Truthy} \to \texttt{true})$

$$\texttt{LogicalOr}: \quad \forall \alpha, \beta \,. \quad ((\alpha \wedge \texttt{Truthy}, \texttt{Any}) \to \alpha \wedge \texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$$

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: (\texttt{Falsy} \to \texttt{false}) \wedge (\texttt{Truthy} \to \texttt{true})$

and $\texttt{LogicalOr}: \forall \alpha, \beta.\ ((\alpha \wedge \texttt{Truthy}, \texttt{Any}) \to \alpha \wedge \texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$

Challenges:

- Type narrowing: type the first branch under the hypothesis that `x` is `Truthy`

  $\Rightarrow$ union types

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean} : (\texttt{Falsy} \to \texttt{false}) \wedge (\texttt{Truthy} \to \texttt{true})$

and $\texttt{LogicalOr} : \forall\alpha,\beta.\ ((\alpha\wedge\texttt{Truthy}, \texttt{Any}) \to \alpha\wedge\texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$

Challenges:

- Type narrowing: type the first branch under the hypothesis that `x` is `Truthy`

  $\Rightarrow$ union types

- Capture overloaded behaviors: `LogicalOr` has different behaviors depending on `x`

  $\Rightarrow$ intersection types

Example: Typing JavaScript's "||" (Logical Or)

```
function LogicalOr (x, y) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: (\texttt{Falsy} \to \texttt{false}) \wedge (\texttt{Truthy} \to \texttt{true})$

and $\texttt{LogicalOr}: \forall\alpha,\beta.\ ((\alpha\wedge\texttt{Truthy}, \texttt{Any}) \to \alpha\wedge\texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$

Challenges:

- Type narrowing: type the first branch under the hypothesis that `x` is `Truthy`

  $\Rightarrow$ union types
- Capture overloaded behaviors: `LogicalOr` has different behaviors depending on `x`

  $\Rightarrow$ intersection types
- Capture genericity: `LogicalOr` returns its first or second parameter, unchanged

  $\Rightarrow$ parametric polymorphism

Set-Theoretic Types [Frisch, 2004]

Constants	c	$::=$	$\texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$
Basic Types	b	$::=$	$\texttt{Bool} \mid \texttt{Int} \mid \ldots$
Set-Theoretic Types	t	$::=$	$c \mid b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Set-Theoretic Types [Frisch, 2004]

$$\begin{array}{lrcl}
\textbf{Constants} & c & ::= & \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots \\
\textbf{Basic Types} & b & ::= & \texttt{Bool} \mid \texttt{Int} \mid \ldots \\
\textbf{Set-Theoretic Types} & t & ::= & c \mid b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}
\end{array}$$

Types are interpreted as sets of values:

$$\begin{array}{ll}
[\![\texttt{false}]\!] = \{\texttt{false}\} & [\![\texttt{Any}]\!] = \mathcal{V} \quad \text{(with } \mathcal{V} \text{ the set of all values)} \\
[\![\texttt{Int}]\!] = \{0, 1, \ldots\} & [\![\texttt{Empty}]\!] = \varnothing
\end{array}$$

Set-Theoretic Types [Frisch, 2004]

$$
\begin{array}{llll}
\textbf{Constants} & c & ::= & \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots \\
\textbf{Basic Types} & b & ::= & \texttt{Bool} \mid \texttt{Int} \mid \ldots \\
\textbf{Set-Theoretic Types} & t & ::= & c \mid b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}
\end{array}
$$

Types are interpreted as sets of values:

$[\![\texttt{false}]\!] = \{\texttt{false}\}$ $\qquad$ $[\![\texttt{Any}]\!] = \mathcal{V}$ $\qquad$ (with $\mathcal{V}$ the set of all values)

$[\![\texttt{Int}]\!] = \{0, 1, \ldots\}$ $\qquad$ $[\![\texttt{Empty}]\!] = \varnothing$

$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$ $\qquad$ $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$ $\qquad$ $[\![\neg t]\!] = \mathcal{V} \setminus [\![t]\!]$

$[\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”}$ $\qquad$ $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$

Set-Theoretic Types [Frisch, 2004]

Constants	c	$::=$	$\texttt{false} \mid \texttt{true} \mid \texttt{0} \mid \texttt{1} \mid \ldots$
Basic Types	b	$::=$	$\texttt{Bool} \mid \texttt{Int} \mid \ldots$
Set-Theoretic Types	t	$::=$	$c \mid b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Types are interpreted as sets of values:

$$[\![\texttt{false}]\!] = \{\texttt{false}\} \qquad [\![\texttt{Any}]\!] = \mathcal{V} \quad \text{(with } \mathcal{V} \text{ the set of all values)}$$

$$[\![\texttt{Int}]\!] = \{0, 1, \ldots\} \qquad [\![\texttt{Empty}]\!] = \varnothing$$

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![\neg t]\!] = \mathcal{V} \setminus [\![t]\!]$$

$$[\![t_1 \to t_2]\!] = \text{``}[\![t_2]\!]^{[\![t_1]\!]}\text{''} \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: $\quad t_1 \leq t_2 \overset{def}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$

Set-Theoretic Types [Frisch, 2004] [Castagna and Xu, 2011]

Constants	c	$::=$	$\texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$
Basic Types	b	$::=$	$\texttt{Bool} \mid \texttt{Int} \mid \ldots$
Set-Theoretic Types	t	$::=$	$c \mid b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any} \mid \alpha$

Types are interpreted as sets of values:

$[\![\texttt{false}]\!] = \{\texttt{false}\}$ $\qquad$ $[\![\texttt{Any}]\!] = \mathcal{V}$ $\qquad$ (with $\mathcal{V}$ the set of all values)

$[\![\texttt{Int}]\!] = \{0, 1, \ldots\}$ $\qquad$ $[\![\texttt{Empty}]\!] = \varnothing$

$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$ $\qquad$ $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$ $\qquad$ $[\![\neg t]\!] = \mathcal{V} \setminus [\![t]\!]$

$[\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”}$ $\qquad$ $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$

Semantic subtyping: $\quad t_1 \leq t_2 \overset{def}{\iff} \forall\sigma.\ t_1\sigma \leq t_2\sigma \overset{def}{\iff} \forall\sigma.\ [\![t_1\sigma]\!] \subseteq [\![t_2\sigma]\!]$

Syntax and Semantics

$$\begin{array}{llcl} \textbf{Expressions} & e & ::= & c \,|\, x \,|\, \lambda x.e \,|\, e\,e \,|\, (e,e) \,|\, \pi_i e \,|\, (e \in t)\,?\,e : e \\ \textbf{Values} & v & ::= & c \,|\, \lambda x.e \,|\, (v,v) \end{array}$$

with the usual call-by-value semantics (w/ leftmost outermost strategy):

$$\begin{array}{rcl} (\lambda x.e)v & \rightsquigarrow & e\{v/x\} \\ \pi_1(v_1,v_2) & \rightsquigarrow & v_1 \\ \pi_2(v_1,v_2) & \rightsquigarrow & v_2 \end{array}$$

$$\boxed{\begin{array}{rcll} (v \in t)\,?\,e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } v \text{ has type } t \\ (v \in t)\,?\,e_1 : e_2 & \rightsquigarrow & e_2 & \text{otherwise} \end{array}}$$

Declarative Type System

Usual Rules

$$[\text{Const}]\ \frac{}{\Gamma \vdash c : c} \qquad\qquad [\text{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \text{dom}(\Gamma)$$

Usual Rules

$$[\text{Const}]\ \frac{}{\Gamma \vdash c : c} \qquad [\text{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \text{dom}(\Gamma)$$

$$[\times\text{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\text{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\times\text{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

Usual Rules

$$[\text{Const}]\ \frac{}{\Gamma \vdash c : c} \qquad [\text{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \text{dom}(\Gamma)$$

$$[\times\text{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\text{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\times\text{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

$$[\to\text{I}]\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

Usual Rules

$$[\text{Const}]\ \frac{}{\Gamma \vdash c : c} \qquad [\text{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \text{dom}(\Gamma)$$

$$[\times\text{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\text{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\times\text{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_2 e : t_2}$$

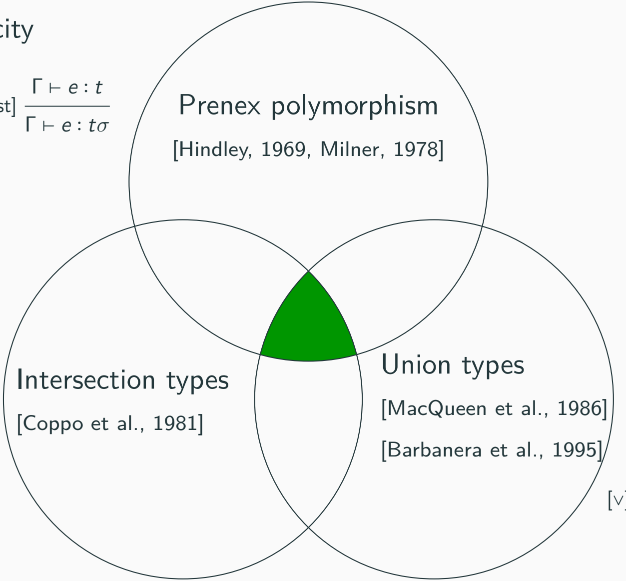
$$[\to\text{I}]\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

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

Mixing Union, Intersection, and HM Polymorphism

For genericity

$$[\text{Gen}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\{\gamma \leadsto \alpha\}} \qquad [\text{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma}$$



For overloading

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

For type narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s_1 \vee s_2 \qquad \Gamma, x : s_1 \vdash e : t \quad \Gamma, x : s_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

Instantiation and Generalization (Hindley Milner)

Some type variables are polymorphic: $\alpha, \beta \in \textbf{Vars}_P$

Some type variables are monomorphic: $\boldsymbol{\gamma}, \boldsymbol{\delta} \in \textbf{Vars}_M$

$$\textbf{Vars} = \textbf{Vars}_P \cup \textbf{Vars}_M$$

Instantiation and Generalization (Hindley Milner)

Some type variables are polymorphic: $\alpha, \beta \in \mathbf{Vars}_P$
Some type variables are monomorphic: $\gamma, \delta \in \mathbf{Vars}_M$

$$\mathbf{Vars} = \mathbf{Vars}_P \cup \mathbf{Vars}_M$$

We can instantiate polymorphic type variables:

$$[\text{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma}\ \text{dom}(\sigma) \subseteq \mathbf{Vars}_P$$

Instantiation and Generalization (Hindley Milner)

Some type variables are polymorphic: $\alpha, \beta \in \mathbf{Vars}_P$
Some type variables are monomorphic: $\boldsymbol{\gamma}, \boldsymbol{\delta} \in \mathbf{Vars}_M$

$$\mathbf{Vars} = \mathbf{Vars}_P \cup \mathbf{Vars}_M$$

We can instantiate polymorphic type variables:

$$[\text{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma}\ \text{dom}(\sigma) \subseteq \mathbf{Vars}_P$$

We can generalize a monomorphic type variable γ into a polymorphic type variable α (only if γ is not bound to the environment):

$$[\text{Gen}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\{\gamma \rightsquigarrow \alpha\}}\ \gamma \notin \text{vars}(\Gamma)$$

$$\begin{aligned}
&\texttt{let id} &&= \lambda x.x \\
&\texttt{let test} &&= (\texttt{id}\ 42, \texttt{id}\ \texttt{true})
\end{aligned}$$

$$
\begin{array}{llll}
\texttt{let} & \texttt{id} & \texttt{=} & \lambda x.x \\
\texttt{let} & \texttt{test} & \texttt{=} & (\texttt{id}\ 42, \texttt{id}\ \texttt{true})
\end{array}
$$

We first type `id` under the empty environment $\varnothing$:

$$
[\rightarrow\mathrm{I}]\ \dfrac{[\mathrm{Var}]\ \dfrac{}{x : \gamma \vdash x : \gamma}}{\varnothing \vdash \lambda x.x : \gamma \rightarrow \gamma}\ \text{with } \gamma \text{ monomorphic}
$$

$$
\begin{aligned}
&\texttt{let}\ \ \texttt{id} && \texttt{=}\ \ \lambda x.x \\
&\texttt{let}\ \ \texttt{test} && \texttt{=}\ \ (\texttt{id}\ 42, \texttt{id}\ \texttt{true})
\end{aligned}
$$

We first type `id` under the empty environment $\varnothing$:

$$
\dfrac{\dfrac{\dfrac{}{x : \gamma \vdash x : \gamma}\,[\text{Var}]}{\varnothing \vdash \lambda x.x : \gamma \to \gamma}\,[\to\text{I}] \text{ with } \gamma \text{ monomorphic}}{\varnothing \vdash \lambda x.x : \alpha \to \alpha}\,[\text{Gen}] \text{ with } \alpha \text{ polymorphic}
$$

$$\begin{array}{llcl}\texttt{let} & \texttt{id} & \texttt{=} & \lambda x.x \\ \texttt{let} & \texttt{test} & \texttt{=} & (\texttt{id}\ 42, \texttt{id}\ \texttt{true})\end{array}$$

We first type `id` under the empty environment $\varnothing$:

$$\dfrac{\dfrac{\dfrac{}{x : \gamma \vdash x : \gamma}\,[\mathrm{Var}]}{\varnothing \vdash \lambda x.x : \gamma \to \gamma}\,[\to\mathrm{I}] \text{ with } \gamma \text{ monomorphic}}{\varnothing \vdash \lambda x.x : \alpha \to \alpha}\,[\mathrm{Gen}] \text{ with } \alpha \text{ polymorphic}$$

We then type `test` under the environment $\Gamma = (\texttt{id} : \alpha \to \alpha)$:

$$\dfrac{\dfrac{\dfrac{[\mathrm{Var}]}{\Gamma \vdash \texttt{id} : \alpha \to \alpha}}{\Gamma \vdash \texttt{id} : 42 \to 42}\,[\mathrm{Inst}] \quad \dfrac{[\mathrm{Const}]}{\Gamma \vdash 42 : 42}}{\Gamma \vdash \texttt{id}\ 42 : 42}\,[\to\mathrm{E}]$$

$$\begin{array}{llcl} \texttt{let} & \texttt{id} & \texttt{=} & \lambda x.x \\ \texttt{let} & \texttt{test} & \texttt{=} & (\texttt{id}\ 42, \texttt{id}\ \texttt{true}) \end{array}$$

We first type `id` under the empty environment $\varnothing$:

$$\dfrac{\dfrac{\dfrac{}{x : \gamma \vdash x : \gamma}\,[\mathrm{Var}]}{\varnothing \vdash \lambda x.x : \gamma \to \gamma}\,[\to\mathrm{I}] \text{ with } \gamma \text{ monomorphic}}{\varnothing \vdash \lambda x.x : \alpha \to \alpha}\,[\mathrm{Gen}] \text{ with } \alpha \text{ polymorphic}$$

We then type `test` under the environment $\Gamma = (\texttt{id} : \alpha \to \alpha)$:

$$\dfrac{\dfrac{\dfrac{[\mathrm{Var}]}{\Gamma \vdash \texttt{id} : \alpha \to \alpha}}{\Gamma \vdash \texttt{id} : 42 \to 42}\,[\mathrm{Inst}] \quad \dfrac{[\mathrm{Const}]}{\Gamma \vdash 42 : 42}}{\Gamma \vdash \texttt{id}\ 42 : 42}\,[\to\mathrm{E}] \qquad \dfrac{\dfrac{\dfrac{[\mathrm{Var}]}{\Gamma \vdash \texttt{id} : \alpha \to \alpha}}{\Gamma \vdash \texttt{id} : \texttt{true} \to \texttt{true}}\,[\mathrm{Inst}] \quad \dfrac{[\mathrm{Const}]}{\Gamma \vdash \texttt{true} : \texttt{true}}}{\Gamma \vdash \texttt{id}\ \texttt{true} : \texttt{true}}\,[\to\mathrm{E}]$$

$$\begin{array}{llcl} \texttt{let} & \texttt{id} & \texttt{=} & \lambda x.x \\ \texttt{let} & \texttt{test} & \texttt{=} & (\texttt{id}\ 42, \texttt{id}\ \texttt{true}) \end{array}$$

We first type `id` under the empty environment $\varnothing$:

$$\dfrac{\dfrac{\dfrac{}{x : \gamma \vdash x : \gamma}[\text{Var}]}{\varnothing \vdash \lambda x.x : \gamma \to \gamma}[\to\text{I}] \text{ with } \gamma \text{ monomorphic}}{\varnothing \vdash \lambda x.x : \alpha \to \alpha}[\text{Gen}] \text{ with } \alpha \text{ polymorphic}$$

We then type `test` under the environment $\Gamma = (\texttt{id} : \alpha \to \alpha)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\text{Var}]}{\Gamma \vdash \texttt{id} : \alpha \to \alpha}}{\Gamma \vdash \texttt{id} : 42 \to 42}[\text{Inst}] \quad \dfrac{[\text{Const}]}{\Gamma \vdash 42 : 42}}{\Gamma \vdash \texttt{id}\ 42 : 42}[\to\text{E}] \quad \dfrac{\dfrac{\dfrac{[\text{Var}]}{\Gamma \vdash \texttt{id} : \alpha \to \alpha}}{\Gamma \vdash \texttt{id} : \texttt{true} \to \texttt{true}}[\text{Inst}] \quad \dfrac{[\text{Const}]}{\Gamma \vdash \texttt{true} : \texttt{true}}}{\Gamma \vdash \texttt{id}\ \texttt{true} : \texttt{true}}[\to\text{E}]}{\Gamma \vdash (\texttt{id}\ 42, \texttt{id}\ \texttt{true}) : 42 \times \texttt{true}}[\times\text{I}]$$

Intersection

Intersection introduction:

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

Intersection

Intersection introduction:

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

Intersection elimination can be derived from subsumption:

$$[\leq]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \qquad \longrightarrow \qquad [\leq]\ \frac{\Gamma \vdash e : t_1 \wedge t_2}{\Gamma \vdash e : t_1}\ t_1 \wedge t_2 \leq t_1$$

For instance, we can type $\lambda x.x$:

For instance, we can type $\lambda x.x$:

- A first time for the domain `Bool`, yielding `Bool` $\to$ `Bool`,

$$
\dfrac{[\text{Var}]\ \dfrac{}{x : \mathtt{Bool} \vdash x : \mathtt{Bool}}}{\varnothing \vdash \lambda x.x : \mathtt{Bool} \to \mathtt{Bool}}\ [\to\text{I}]
$$

For instance, we can type $\lambda x.x$:

- A first time for the domain `Bool`, yielding `Bool` $\to$ `Bool`,
- A second time for the domain `Int`, yielding `Int` $\to$ `Int`,

$$\dfrac{[\mathsf{Var}]\,\dfrac{}{x : \texttt{Bool} \vdash x : \texttt{Bool}}}{\varnothing \vdash \lambda x.x : \texttt{Bool} \to \texttt{Bool}}\,[{\to}\mathsf{I}] \qquad \dfrac{[\mathsf{Var}]\,\dfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}}{\varnothing \vdash \lambda x.x : \texttt{Int} \to \texttt{Int}}\,[{\to}\mathsf{I}]$$

For instance, we can type $\lambda x.x$:

- A first time for the domain $\texttt{Bool}$, yielding $\texttt{Bool} \to \texttt{Bool}$,
- A second time for the domain $\texttt{Int}$, yielding $\texttt{Int} \to \texttt{Int}$,
- Then, we can use the intersection introduction rule to derive the type $(\texttt{Bool} \to \texttt{Bool}) \wedge (\texttt{Int} \to \texttt{Int})$

$$
\dfrac{
\dfrac{
\dfrac{}{x : \texttt{Bool} \vdash x : \texttt{Bool}}\,[\mathrm{Var}]
}{\varnothing \vdash \lambda x.x : \texttt{Bool} \to \texttt{Bool}}\,[\to\mathrm{I}]
\qquad
\dfrac{
\dfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}\,[\mathrm{Var}]
}{\varnothing \vdash \lambda x.x : \texttt{Int} \to \texttt{Int}}\,[\to\mathrm{I}]
}{\varnothing \vdash \lambda x.x : (\texttt{Bool} \to \texttt{Bool}) \wedge (\texttt{Int} \to \texttt{Int})}\,[\wedge]
$$

Union

Union introduction can be derived from subsumption:

$$[\leq]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \quad \longrightarrow \quad [\leq]\ \frac{\Gamma \vdash e : t_1}{\Gamma \vdash e : t_1 \vee t_2}\ t_1 \leq t_1 \vee t_2$$

Union

Union introduction can be derived from subsumption:

$$[\leq]\ \frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \quad \longrightarrow \quad [\leq]\ \frac{\Gamma \vdash e : t_1}{\Gamma \vdash e : t_1 \vee t_2}\ t_1 \leq t_1 \vee t_2$$

Union elimination:

$$[\vee]\ \frac{\Gamma \vdash e' : s_1 \vee s_2 \qquad \Gamma, x : s_1 \vdash e : t \quad \Gamma, x : s_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$(\text{ f } 42 \text{ , } \text{ f } 42 \text{ }) \qquad \text{with } f : \texttt{Int} \to \texttt{Bool}$$

$$(\underbrace{\mathsf{f}\ 42}_{x}, \underbrace{\mathsf{f}\ 42}_{x}) \qquad \text{with } f : \mathtt{Int} \to \mathtt{Bool}$$

$$\text{with } x : \mathtt{Bool} \simeq \mathtt{true} \vee \mathtt{false}$$

$$(\underbrace{\mathrm{f}\ 42}_{x}, \underbrace{\mathrm{f}\ 42}_{x}) \qquad \text{with } f : \mathtt{Int} \to \mathtt{Bool}$$

$$\text{with } x : \mathtt{Bool} \simeq \mathtt{true} \vee \mathtt{false}$$

We type (x, x):

- First, by assuming that $x : \mathtt{true} \quad \Rightarrow \quad \mathtt{true} \times \mathtt{true}$,

$$(\underbrace{\text{f } 42}_{x}, \underbrace{\text{f } 42}_{x}) \qquad \text{with } f : \mathtt{Int} \to \mathtt{Bool}$$

$$\text{with } x : \mathtt{Bool} \simeq \mathtt{true} \vee \mathtt{false}$$

We type (x, x):

- First, by assuming that $x : \mathtt{true} \quad \Rightarrow \quad \mathtt{true} \times \mathtt{true}$,
- Then, by assuming that $x : \mathtt{false} \quad \Rightarrow \quad \mathtt{false} \times \mathtt{false}$

$$(\underbrace{\text{f } 42}_{x}, \underbrace{\text{f } 42}_{x}) \qquad \text{with } f : \texttt{Int} \to \texttt{Bool}$$

$$\text{with } x : \texttt{Bool} \simeq \texttt{true} \vee \texttt{false}$$

We type (x, x):

- First, by assuming that $x : \texttt{true} \quad \Rightarrow \quad \texttt{true} \times \texttt{true}$,
- Then, by assuming that $x : \texttt{false} \quad \Rightarrow \quad \texttt{false} \times \texttt{false}$

$$[\vee]\ \frac{\Gamma \vdash f\ 42 : \texttt{true} \vee \texttt{false} \qquad \Gamma, x : \texttt{true} \vdash (x, x) : \texttt{true} \times \texttt{true} \qquad \Gamma, x : \texttt{false} \vdash (x, x) : \texttt{false} \times \texttt{false}}{\Gamma \vdash (x, x)\{(f\ 42)/x\} : (\texttt{true} \times \texttt{true}) \vee (\texttt{false} \times \texttt{false})}$$

Unsound in the presence of polymorphic type variables:

$$(\ \mathsf{f}\ 42\ ,\ \mathsf{f}\ 42\ ) \qquad \text{with } f : \mathtt{Int} \to \mathtt{Bool}$$

Unsound in the presence of polymorphic type variables:

$$(\underbrace{\text{f } 42}_{x}, \underbrace{\text{f } 42}_{x}) \qquad \text{with } f : \texttt{Int} \to \texttt{Bool}$$

$$\text{with } x : \texttt{Bool} \simeq (\texttt{Bool} \wedge \alpha) \vee (\texttt{Bool} \wedge \neg\alpha) \text{ (with } \alpha \text{ polymorphic)}$$

Unsound in the presence of polymorphic type variables:

$$(\underbrace{\mathsf{f}\ 42}_{x}, \underbrace{\mathsf{f}\ 42}_{x}) \qquad \text{with } f : \mathtt{Int} \to \mathtt{Bool}$$

$$\text{with } x : \mathtt{Bool} \simeq (\mathtt{Bool} \wedge \alpha) \vee (\mathtt{Bool} \wedge \neg\alpha) \text{ (with } \alpha \text{ polymorphic)}$$

We type (x, x):

- First, by assuming that $x : \mathtt{Bool} \wedge \alpha \quad \Rightarrow \quad \mathtt{Empty}$ (by substituting α by $\mathtt{Empty}$),

Unsound in the presence of polymorphic type variables:

$$(\underbrace{\text{f } 42}_{x}, \underbrace{\text{f } 42}_{x}) \qquad \text{with } f : \texttt{Int} \to \texttt{Bool}$$

$$\text{with } x : \texttt{Bool} \simeq (\texttt{Bool} \wedge \alpha) \vee (\texttt{Bool} \wedge \neg\alpha) \text{ (with } \alpha \text{ polymorphic)}$$

We type (x, x):

- First, by assuming that $x : \texttt{Bool} \wedge \alpha \quad \Rightarrow \quad \texttt{Empty}$ (by substituting α by $\texttt{Empty}$),
- Then, by assuming that $x : \texttt{Bool} \wedge \neg\alpha \quad \Rightarrow \quad \texttt{Empty}$ (by substituting α by $\texttt{Any}$)

Unsound in the presence of polymorphic type variables:

$$(\underbrace{\mathsf{f}\ 42}_{x}, \underbrace{\mathsf{f}\ 42}_{x}) \qquad \text{with } f : \mathtt{Int} \to \mathtt{Bool}$$

$$\text{with } x : \mathtt{Bool} \simeq (\mathtt{Bool} \wedge \alpha) \vee (\mathtt{Bool} \wedge \neg\alpha) \text{ (with } \alpha \text{ polymorphic)}$$

We type (x, x):

- First, by assuming that $x : \mathtt{Bool} \wedge \alpha \quad \Rightarrow \quad \mathtt{Empty}$ (by substituting α by $\mathtt{Empty}$),
- Then, by assuming that $x : \mathtt{Bool} \wedge \neg\alpha \quad \Rightarrow \quad \mathtt{Empty}$ (by substituting α by $\mathtt{Any}$)

We must prevent the type decomposition from containing polymorphic type variables:

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

where u does not contain any polymorphic type variable: $\mathrm{vars}(u) \cap \mathbf{Vars}_P = \varnothing$

Typing Type-Cases

Two cases:

$$\boxed{\begin{array}{llll} (v{\in}t)\ ?\ e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } v \text{ has type } t \\ (v{\in}t)\ ?\ e_1 : e_2 & \rightsquigarrow & e_2 & \text{otherwise} \end{array}}$$

Typing Type-Cases

Two cases:

$$\boxed{\begin{array}{lcll} (v{\in}t)\ ?\ e_1 : e_2 & \rightsquigarrow & e_1 & \text{if } v \text{ has type } t \\ (v{\in}t)\ ?\ e_1 : e_2 & \rightsquigarrow & e_2 & \text{otherwise} \end{array}}$$

Two rules:

$$[\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\ ?\ e_1 : e_2 : t_2}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\begin{array}{c}\Gamma \vdash e' : s \\ \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t\end{array}}{\Gamma \vdash e\{e'/x\} : t} \qquad [\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_2}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t} \qquad [\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e{\in}t)\,?\,e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e{\in}t)\,?\,e_1 : e_2 : t_2}$$

$$\lambda x.\ (\ \mathtt{x} \in \mathtt{Int})\,?\ \ \mathtt{x}\ + 1 : \mathtt{false}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t} \qquad [\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma = \{\ x : \texttt{Any}\ \} \qquad \lambda x.\ (\ \texttt{x} \in \texttt{Int})\ ?\ \ \texttt{x} + 1 : \texttt{false}$$

$$[\to\text{I}]\ \frac{x : \texttt{Any} \vdash (x \in \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Int} \vee \texttt{false}}{\varnothing \vdash \lambda x.\ (x \in \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Any} \to (\texttt{Int} \vee \texttt{false})}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$[\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma = \{\ x : \underbrace{\texttt{Any}}_{\texttt{Int} \vee \neg\texttt{Int}}\ \} \qquad \lambda x.\ (\texttt{ x } \in \texttt{Int})\ ?\ \texttt{ x } + 1 : \texttt{false}$$

$$[\to\mathrm{I}]\ \dfrac{[\vee]\ \dfrac{x : \texttt{Int} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Int} \qquad x : \neg\texttt{Int} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{false}}{x : \texttt{Any} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Int} \vee \texttt{false}}}{\varnothing \vdash \lambda x.\ (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Any} \to (\texttt{Int} \vee \texttt{false})}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \qquad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$[\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma = \{\ x : \underbrace{\texttt{Any}}_{\texttt{Int} \vee \neg\texttt{Int}}\ \} \qquad \lambda x.\ (\underbrace{x}_{\texttt{Int}} \in \texttt{Int})\ ?\ \underbrace{x}_{\texttt{Int}} + 1 : \texttt{false}$$

$$[\to\mathrm{I}]\ \dfrac{[\vee]\ \dfrac{[\epsilon_1]\ \dfrac{x : \texttt{Int} \vdash x + 1 : \texttt{Int}}{x : \texttt{Int} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Int}} \qquad x : \neg\texttt{Int} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{false}}{x : \texttt{Any} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Int} \vee \texttt{false}}}{\varnothing \vdash \lambda x.\ (x \epsilon \texttt{Int})\ ?\ x + 1 : \texttt{false} : \texttt{Any} \to (\texttt{Int} \vee \texttt{false})}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \qquad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$[\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_1}$$

$$[\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma \;=\; \{\ x : \underbrace{\texttt{Any}}_{\texttt{Int} \vee \neg\texttt{Int}}\ \}$$

$$\lambda x.\ (\underbrace{x}_{\neg\texttt{Int}} \in \texttt{Int})\ ?\ x+1 : \texttt{false}$$

$$[\to\text{I}]\ \dfrac{[\vee]\ \dfrac{[\epsilon_1]\ \dfrac{x : \texttt{Int} \vdash x+1 : \texttt{Int}}{x : \texttt{Int} \vdash (x\epsilon\texttt{Int})\ ?\ x+1 : \texttt{false} : \texttt{Int}} \qquad [\epsilon_2]\ \dfrac{x : \neg\texttt{Int} \vdash \texttt{false} : \texttt{false}}{x : \neg\texttt{Int} \vdash (x\epsilon\texttt{Int})\ ?\ x+1 : \texttt{false} : \texttt{false}}}{x : \texttt{Any} \vdash (x\epsilon\texttt{Int})\ ?\ x+1 : \texttt{false} : \texttt{Int} \vee \texttt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\ ?\ x+1 : \texttt{false} : \texttt{Any} \to (\texttt{Int} \vee \texttt{false})}$$

Typing Type-Cases: Union Elimination and Type Narrowing

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$[\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e\epsilon t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e\epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\lambda x.\ (\,\mathtt{x} \in \mathtt{Int})\ ?\ \ \mathtt{x}\ + 1\ :\ \mathtt{false}$$

$$[\to\mathrm{I}]\ \dfrac{[\vee]\ \dfrac{[\epsilon_1]\ \dfrac{x : \mathtt{Int} \vdash x + 1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int}} \qquad [\epsilon_2]\ \dfrac{x : \neg\mathtt{Int} \vdash \mathtt{false} : \mathtt{false}}{x : \neg\mathtt{Int} \vdash (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{false}}}{x : \mathtt{Any} \vdash (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int} \vee \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Any} \to (\mathtt{Int} \vee \mathtt{false})}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_2}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 \mathbin{:} e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 \mathbin{:} e_2 : t_2}$$

$$\lambda x.\ (x \epsilon \mathtt{Int})\ ?\ x + 1 : \mathtt{false}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 \mathbin{:} e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 \mathbin{:} e_2 : t_2}$$

$$\Gamma \ = \ \{\ x : \texttt{Int}\ \} \qquad\qquad \lambda x.\ (x \epsilon \texttt{Int})\ ?\ x + 1 \ \mathbf{:}\ \texttt{false}$$

$$[\to\mathrm{I}]\ \frac{x : \texttt{Int} \vdash (x \epsilon \texttt{Int})\ ?\ x + 1 \mathbin{:} \texttt{false} : \texttt{Int}}{\varnothing \vdash \lambda x.\ (x \epsilon \texttt{Int})\ ?\ x + 1 \mathbin{:} \texttt{false} : \texttt{Int} \to \texttt{Int}}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\epsilon_1]\ \frac{\Gamma \vdash e : t \quad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \in t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma = \{\ x : \mathtt{Int}\ \} \qquad \lambda x.\ (x \in \mathtt{Int})\ ?\ x + 1 : \mathtt{false}$$

$$[\to\mathrm{I}]\ \dfrac{[\epsilon_1]\ \dfrac{x : \mathtt{Int} \vdash x + 1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x \in \mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int}}}{\varnothing \vdash \lambda x.\ (x \in \mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int} \to \mathtt{Int}}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma \quad = \quad \{\ x : \neg\mathtt{Int}\ \} \qquad\qquad \lambda x.\ (x \epsilon \mathtt{Int})\ ?\ x + 1\ :\ \mathtt{false}$$

$$[\to\mathrm{I}]\ \frac{[\epsilon_1]\ \dfrac{x : \mathtt{Int} \vdash x + 1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x \epsilon \mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int}}}{\varnothing \vdash \lambda x.\ (x \epsilon \mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int} \to \mathtt{Int}} \qquad [\to\mathrm{I}]\ \frac{x : \neg\mathtt{Int} \vdash (x \epsilon \mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{false}}{\varnothing \vdash \lambda x.\ (x \epsilon \mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \neg\mathtt{Int} \to \mathtt{false}}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e\epsilon t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e\epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\Gamma \ = \ \{\ x : \neg\mathtt{Int}\ \} \qquad\qquad \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x+1\ :\ \mathtt{false}$$

$$[\to\mathrm{I}]\ \dfrac{[\epsilon_1]\ \dfrac{x : \mathtt{Int} \vdash x+1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x\epsilon\mathtt{Int})\ ?\ x+1 : \mathtt{false} : \mathtt{Int}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x+1 : \mathtt{false} : \mathtt{Int} \to \mathtt{Int}} \qquad [\to\mathrm{I}]\ \dfrac{[\epsilon_2]\ \dfrac{x : \neg\mathtt{Int} \vdash \mathtt{false} : \mathtt{false}}{x : \neg\mathtt{Int} \vdash (x\epsilon\mathtt{Int})\ ?\ x+1 : \mathtt{false} : \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x+1 : \mathtt{false} : \neg\mathtt{Int} \to \mathtt{false}}$$

Capturing Overloaded Behaviors: Intersection Introduction

$$[\wedge]\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2} \qquad [\epsilon_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e\epsilon t)\ ?\ e_1 : e_2 : t_1} \qquad [\epsilon_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e\epsilon t)\ ?\ e_1 : e_2 : t_2}$$

$$\lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false}$$

$$[\wedge]\ \dfrac{[\to\mathrm{I}]\ \dfrac{[\epsilon_1]\ \dfrac{x : \mathtt{Int} \vdash x + 1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{Int} \to \mathtt{Int}} \qquad [\to\mathrm{I}]\ \dfrac{[\epsilon_2]\ \dfrac{x : \neg\mathtt{Int} \vdash \mathtt{false} : \mathtt{false}}{x : \neg\mathtt{Int} \vdash (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : \neg\mathtt{Int} \to \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\ ?\ x + 1 : \mathtt{false} : (\mathtt{Int} \to \mathtt{Int}) \wedge (\neg\mathtt{Int} \to \mathtt{false})}$$

$$(\mathtt{Int} \to \mathtt{Int}) \wedge (\neg\mathtt{Int} \to \mathtt{false}) \quad \leq \quad \mathtt{Any} \to (\mathtt{Int} \vee \mathtt{false})$$

Contribution 1

Type safety of the declarative type system

For every expression e, if $\varnothing \vdash e : t$, then:

– either e reduces to a value v of type t,

– or e diverges.

Contribution 1

Type safety of the declarative type system

For every expression e, if $\varnothing \vdash e : t$, then:

– either e reduces to a value v of type t,

– or e diverges.

However, this type system is not algorithmic.

How to turn it into an algorithm?

Algorithmic Type System

Declarative = Non-algorithmic

$$[\wedge]\dfrac{[\to\mathrm{I}]\dfrac{[\epsilon_1]\dfrac{x : \mathtt{Int} \vdash x+1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{Int}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{Int} \to \mathtt{Int}} \qquad [\to\mathrm{I}]\dfrac{[\epsilon_2]\dfrac{x : \neg\mathtt{Int} \vdash \mathtt{false} : \mathtt{false}}{x : \neg\mathtt{Int} \vdash (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \neg\mathtt{Int} \to \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : (\mathtt{Int} \to \mathtt{Int}) \wedge (\neg\mathtt{Int} \to \mathtt{false})}$$

$$[\to\mathrm{I}]\dfrac{[\vee]\dfrac{[\epsilon_1]\dfrac{x : \mathtt{Int} \vdash x+1 : \mathtt{Int}}{x : \mathtt{Int} \vdash (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{Int}} \qquad [\epsilon_2]\dfrac{x : \neg\mathtt{Int} \vdash \mathtt{false} : \mathtt{false}}{x : \neg\mathtt{Int} \vdash (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{false}}}{x : \mathtt{Any} \vdash (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{Int} \vee \mathtt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\mathtt{Int})\,?\,x+1 : \mathtt{false} : \mathtt{Any} \to (\mathtt{Int} \vee \mathtt{false})}$$

Many possible derivations:

Declarative = Non-algorithmic

$$\dfrac{\dfrac{\dfrac{x : \texttt{Int} \vdash x+1 : \texttt{Int}}{x : \texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int}}[\epsilon_1]}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int} \to \texttt{Int}}[\to\mathrm{I}] \quad \dfrac{\dfrac{x : \neg\texttt{Int} \vdash \texttt{false} : \texttt{false}}{x : \neg\texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{false}}[\epsilon_2]}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \neg\texttt{Int} \to \texttt{false}}[\to\mathrm{I}]}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : (\texttt{Int} \to \texttt{Int}) \wedge (\neg\texttt{Int} \to \texttt{false})}[\wedge]$$

$$\dfrac{\dfrac{\dfrac{x : \texttt{Int} \vdash x+1 : \texttt{Int}}{x : \texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int}}[\epsilon_1] \quad \dfrac{x : \neg\texttt{Int} \vdash \texttt{false} : \texttt{false}}{x : \neg\texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{false}}[\epsilon_2]}{x : \texttt{Any} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int} \vee \texttt{false}}[\vee]}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Any} \to (\texttt{Int} \vee \texttt{false})}[\to\mathrm{I}]$$

Many possible derivations:

- Some rules can be applied on every expression (the system is not syntax-directed):
  - Union elimination [$\vee$]
  - Intersection introduction [$\wedge$]
  - Instantiation [Inst]
  - Subsumption [$\leq$]

Declarative = Non-algorithmic

$$[\wedge]\ \dfrac{[\to\mathrm{I}]\ \dfrac{[\epsilon_1]\ \dfrac{x : \texttt{Int} \vdash x+1 : \texttt{Int}}{x : \texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int}}}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int} \to \texttt{Int}} \qquad [\to\mathrm{I}]\ \dfrac{[\epsilon_2]\ \dfrac{x : \neg\texttt{Int} \vdash \texttt{false} : \texttt{false}}{x : \neg\texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \neg\texttt{Int} \to \texttt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : (\texttt{Int} \to \texttt{Int}) \wedge (\neg\texttt{Int} \to \texttt{false})}$$

$$[\to\mathrm{I}]\ \dfrac{[\vee]\ \dfrac{[\epsilon_1]\ \dfrac{x : \texttt{Int} \vdash x+1 : \texttt{Int}}{x : \texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int}} \qquad [\epsilon_2]\ \dfrac{x : \neg\texttt{Int} \vdash \texttt{false} : \texttt{false}}{x : \neg\texttt{Int} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{false}}}{x : \texttt{Any} \vdash (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Int} \vee \texttt{false}}}{\varnothing \vdash \lambda x.\ (x\epsilon\texttt{Int})\,?\,x+1 : \texttt{false} : \texttt{Any} \to (\texttt{Int} \vee \texttt{false})}$$

Many possible derivations:

- Some rules can be applied on every expression (the system is not syntax-directed):
  - Union elimination [∨]
  - Intersection introduction [∧]
  - Instantiation [Inst]
  - Subsumption [≤]
- Some premises cannot be guessed from the conclusion (rules are not analytic):
  - The types forming the union in [∨]
  - The type of the parameter in [→I]

Declarative = Non-algorithmic

$$\dfrac{\dfrac{\dfrac{x:\mathtt{Int}\vdash x+1:\mathtt{Int}}{x:\mathtt{Int}\vdash (x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{Int}}[\in_1]}{\varnothing\vdash\lambda x.\,(x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{Int}\to\mathtt{Int}}[\to\mathrm{I}] \quad \dfrac{\dfrac{x:\neg\mathtt{Int}\vdash\mathtt{false}:\mathtt{false}}{x:\neg\mathtt{Int}\vdash (x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{false}}[\in_2]}{\varnothing\vdash\lambda x.\,(x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\neg\mathtt{Int}\to\mathtt{false}}[\to\mathrm{I}]}{\varnothing\vdash\lambda x.\,(x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:(\mathtt{Int}\to\mathtt{Int})\wedge(\neg\mathtt{Int}\to\mathtt{false})}[\wedge]$$

$$\dfrac{\dfrac{\dfrac{x:\mathtt{Int}\vdash x+1:\mathtt{Int}}{x:\mathtt{Int}\vdash (x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{Int}}[\in_1] \quad \dfrac{x:\neg\mathtt{Int}\vdash\mathtt{false}:\mathtt{false}}{x:\neg\mathtt{Int}\vdash (x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{false}}[\in_2]}{x:\mathtt{Any}\vdash (x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{Int}\vee\mathtt{false}}[\vee]}{\varnothing\vdash\lambda x.\,(x\in\mathtt{Int})\,?\,x+1:\mathtt{false}:\mathtt{Any}\to(\mathtt{Int}\vee\mathtt{false})}[\to\mathrm{I}]$$

Many possible derivations:

- Some rules can be applied on every expression (the system is not syntax-directed):
  - Union elimination [∨]
  - Intersection introduction [∧]
  - Instantiation [Inst]
  - Subsumption [≤]
- Some premises cannot be guessed from the conclusion (rules are not analytic):
  - The types forming the union in [∨]
  - The type of the parameter in [→I]

How to make the type system algorithmic?

Making the Type System Syntax-Directed

Solution to make the type system syntax directed without loosing generality:

- Subsumption [$\leq$] and instantiation [Inst] are embedded in destructor rules:

Making the Type System Syntax-Directed

Solution to make the type system syntax directed without loosing generality:

- Subsumption [≤] and instantiation [Inst] are embedded in destructor rules:

$$[\to\text{E}]\ \frac{\Gamma \vdash e_1 : s \to t \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : t} \quad + \quad [\text{Inst}]\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t\sigma} \quad + \quad [\leq]\ \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

$$\Rightarrow \quad [\text{App}]\ \frac{\Gamma \vdash e_1 : t_1 \text{ with } t_1\sigma_1 \leq s \to t \qquad \Gamma \vdash e_2 : t_2 \text{ with } t_2\sigma_2 \leq s}{\Gamma \vdash e_1 e_2 : t}$$

Making the Type System Syntax-Directed

- The union elimination [$\vee$] should be applied once on every distinct subexpression

Making the Type System Syntax-Directed

- The union elimination $[\vee]$ should be applied once on every distinct subexpression

  $\Rightarrow$ We transform the expression in Maximal Sharing Canonical (MSC) form, which gives a unique name to each distinct subexpression:

$$(f\ x, f\ x) \quad \rightsquigarrow \quad \begin{array}{l} \texttt{bind}\ \mathbf{u}\ \texttt{=}\ f\ x\ \texttt{in} \\ \texttt{bind}\ \mathbf{v}\ \texttt{=}\ (\mathbf{u}, \mathbf{u})\ \texttt{in}\ \mathbf{v} \end{array}$$

Making the Type System Syntax-Directed

- The union elimination $[\vee]$ should be applied once on every distinct subexpression

  $\Rightarrow$ We transform the expression in Maximal Sharing Canonical (MSC) form, which gives a unique name to each distinct subexpression:

$$(f\ x, f\ x) \quad \rightsquigarrow \quad \begin{array}{l} \texttt{bind}\ \mathbf{u}\ \texttt{=}\ f\ x\ \texttt{in} \\ \texttt{bind}\ \mathbf{v}\ \texttt{=}\ (\mathbf{u},\mathbf{u})\ \texttt{in}\ \mathbf{v} \end{array}$$

$$[\vee]\ \frac{\Gamma \vdash e' : s \qquad \Gamma, x : s \wedge u \vdash e : t \quad \Gamma, x : s \wedge \neg u \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

$$\Rightarrow \quad [\text{Bind}]\ \frac{\Gamma \vdash a : s \qquad (\forall i \in I)\ \Gamma, \mathbf{u} : s \wedge u_i \vdash \kappa : t_i}{\Gamma \vdash \ \texttt{bind}\ \mathbf{u}\ \texttt{=}\ a\ \texttt{in}\ \kappa\ : \bigvee_{i \in I} t_i}\ \{u_i\}_{i \in I}\ \text{a partition of}\ \texttt{Any}$$

Making the Rules Analytic

Solution to make the rules analytic:

Making the Rules Analytic

Solution to make the rules analytic:

- In addition of a MSC form, our algorithmic type system takes as input an annotation tree that specifies:
  $\rightarrow$ the type decompositions $s_1 \vee \cdots \vee s_n$ to use in $[\vee]$ rules
  $\rightarrow$ the types of the parameters of λ-abstractions

Making the Rules Analytic

Solution to make the rules analytic:

- In addition of a MSC form, our algorithmic type system takes as input an annotation tree that specifies:
  $\rightarrow$ the type decompositions $s_1 \vee \cdots \vee s_n$ to use in $[\vee]$ rules
  $\rightarrow$ the types of the parameters of λ-abstractions
- The pair $[\text{MSC} \mid \text{annotation tree}]$ uniquely encodes a derivation:

Making the Rules Analytic

Solution to make the rules analytic:

- In addition of a MSC form, our algorithmic type system takes as input an annotation tree that specifies:
  → the type decompositions $s_1 \vee \cdots \vee s_n$ to use in $[\vee]$ rules
  → the types of the parameters of λ-abstractions

- The pair [MSC | annotation tree] uniquely encodes a derivation:

$$[\ \underbrace{\texttt{bind}\ \mathbf{u} = a\ \texttt{in}\ \kappa}_{\text{MSC}}\ \ |\ \ \underbrace{[\vee]\ (\ldots,\{(\texttt{Int},\ldots),(\neg\texttt{Int},\ldots)\})}_{\text{annotation tree}}\ ]$$

Making the Rules Analytic

Solution to make the rules analytic:

- In addition of a MSC form, our algorithmic type system takes as input an annotation tree that specifies:
  → the type decompositions $s_1 \vee \cdots \vee s_n$ to use in $[\vee]$ rules
  → the types of the parameters of λ-abstractions
- The pair [MSC | annotation tree] uniquely encodes a derivation:

$$\Big[\ \underbrace{\texttt{bind}\ \mathbf{u}\ \texttt{=}\ a\ \texttt{in}\ \kappa}_{\text{MSC}}\ \Big|\ \underbrace{[\vee]\ (\ldots,\{(\texttt{Int},\ldots),(\neg\texttt{Int},\ldots)\})}_{\text{annotation tree}}\ \Big]$$

$$\downarrow$$

$$[\text{Bind-Alg}]\ \dfrac{\dfrac{\cdots}{\Gamma \vdash a : s} \quad \dfrac{\cdots}{\Gamma, \mathbf{u} : s \wedge \texttt{Int} \vdash e : t_1} \quad \dfrac{\cdots}{\Gamma, \mathbf{u} : s \wedge \neg\texttt{Int} \vdash e : t_2}}{\Gamma \vdash [\texttt{bind}\ \mathbf{u}\ \texttt{=}\ a\ \texttt{in}\ \kappa \mid [\vee]\ (\ldots,\{(\texttt{Int},\ldots),(\neg\texttt{Int},\ldots)\})] : t_1 \vee t_2}$$

Contribution 2

Equivalence between declarative and algorithmic type system

e is typeable with the declarative type system

if and only if

there exists an annotation such that $\mathsf{MSC}(e)$ is typeable with the algorithmic system.

Contribution 2

Equivalence between declarative and algorithmic type system

e is typeable with the declarative type system

if and only if

there exists an annotation such that $\mathrm{MSC}(e)$ is typeable with the algorithmic system.

But how to infer annotation trees?

Reconstruction of the Annotation Tree

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering `(z∈Int)?x:y`,
  we backtrack to the `bind` definition of `z` and split its type into `Int` ; `¬Int`

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering `(z∈Int)?x:y`,
  we backtrack to the `bind` definition of `z` and split its type into `Int` ; `¬Int`
- Then, we backpropagate this split on the variables used in the definition of `z`.

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering $(\mathbf{z}{\in}\mathtt{Int})\,?\,\mathbf{x}:\mathbf{y}$,
  we backtrack to the `bind` definition of $\mathbf{z}$ and split its type into $\mathtt{Int}$; $\neg\mathtt{Int}$

- Then, we backpropagate this split on the variables used in the definition of $\mathbf{z}$.

$$(\mathtt{id}\ x{\in}\mathtt{Int})\,?\,x:\mathtt{false} \quad \text{with} \quad \mathtt{id}:\alpha\to\alpha \ \ \text{and}\ \ x:\mathtt{Any}$$

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering `(`$\mathbf{z}$`∈Int)?`$\mathbf{x}$`:`$\mathbf{y}$,
  we backtrack to the `bind` definition of $\mathbf{z}$ and split its type into `Int` ; ¬`Int`
- Then, we backpropagate this split on the variables used in the definition of $\mathbf{z}$.

$$(\texttt{id}\ x{\in}\texttt{Int})\,?\,x\texttt{:false} \quad \text{with} \quad \texttt{id} : \alpha \to \alpha \quad \text{and} \quad x : \texttt{Any}$$

```
bind x = x in
bind y = false in
bind z = id x in
bind u = (z∈Int) ? x : y in
u
```

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering $(\mathbf{z}\in\texttt{Int})\,?\,\mathbf{x}:\mathbf{y}$,
  we backtrack to the `bind` definition of $\mathbf{z}$ and split its type into $\texttt{Int}\ ;\ \neg\texttt{Int}$

- Then, we backpropagate this split on the variables used in the definition of $\mathbf{z}$.

$$(\texttt{id}\ x\in\texttt{Int})\,?\,x:\texttt{false} \quad \text{with} \quad \texttt{id}:\alpha\to\alpha \quad \text{and} \quad x:\texttt{Any}$$

$\texttt{bind}\,\mathbf{x} : \texttt{Any} = x\ \texttt{in}$

$\texttt{bind}\,\mathbf{y} : \texttt{false} = \texttt{false}\ \texttt{in}$

$\texttt{bind}\,\mathbf{z} : \texttt{Any} = \texttt{id}\ \mathbf{x}\ \texttt{in}$

$\texttt{bind}\,\mathbf{u} = (\mathbf{z}\in\texttt{Int})\,?\,\mathbf{x}:\mathbf{y}\ \texttt{in}$

$\mathbf{u}$

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types: when encountering `(z∈Int) ? x : y`, we backtrack to the `bind` definition of **z** and split its type into `Int` ; `¬Int`
- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\mathtt{id}\ x{\in}\mathtt{Int})\ ?\ x : \mathtt{false} \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \quad \text{and} \quad x : \mathtt{Any}$$

```
bind x : Any = x in
bind y : false = false in
bind z : Any = id x in
bind u = (z∈Int) ? x : y in
u
```

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types: when encountering $(\mathbf{z}{\in}\texttt{Int})\,?\,\mathbf{x}:\mathbf{y}$, we backtrack to the `bind` definition of **z** and split its type into $\texttt{Int} \,;\, \neg\texttt{Int}$

- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\texttt{id}\ x{\in}\texttt{Int})\,?\,x:\texttt{false} \quad \text{with} \quad \texttt{id} : \alpha \to \alpha \quad \text{and} \quad x : \texttt{Any}$$

$\texttt{bind}\,\mathbf{x} : \texttt{Any} = x\ \texttt{in}$

$\texttt{bind}\,\mathbf{y} : \texttt{false} = \texttt{false}\ \texttt{in}$

$\texttt{bind}\,\mathbf{z} : \texttt{Int} \,;\, \neg\texttt{Int} = \texttt{id}\ \mathbf{x}\ \texttt{in}$

$\texttt{bind}\,\mathbf{u} = (\mathbf{z}{\in}\texttt{Int})\,?\,\mathbf{x}:\mathbf{y}\ \texttt{in}$

$\mathbf{u}$

Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering `(z∈Int) ? x : y`,
  we backtrack to the `bind` definition of **z** and split its type into `Int` ; `¬Int`

- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\mathtt{id}\ x{\in}\mathtt{Int})\,?\,x\!:\!\mathtt{false} \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \quad \text{and} \quad x : \mathtt{Any}$$

```
bind x : Int ; ¬Int = x in
bind y : false = false in
bind z : Int ; ¬Int = id x in
bind u = (z∈Int) ? x : y in
u
```

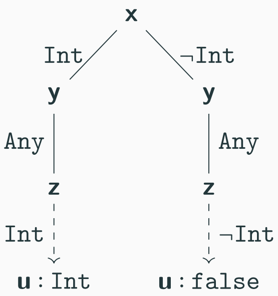
Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering (**z**∈Int) ? **x** : **y**,
  we backtrack to the `bind` definition of **z** and split its type into Int ; ¬Int

- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\mathtt{id}\ x \in \mathtt{Int})\ ?\ x : \mathtt{false} \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \ \text{ and } \ x : \mathtt{Any}$$

```
bind x : Int ; ¬Int = x in
bind y : false = false in
bind z : Int ; ¬Int = id x in
bind u = (z∈Int) ? x : y in
u
```



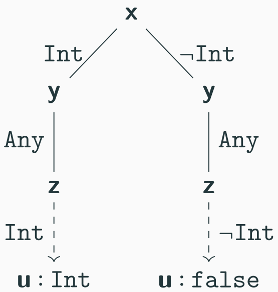
Reconstruction of Type Decompositions

- We use type-cases to deduce how to decompose union types:
  when encountering `(z∈Int) ? x : y`,
  we backtrack to the `bind` definition of **z** and split its type into `Int ; ¬Int`

- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\texttt{id}\ x{\in}\texttt{Int})\,?\,x:\texttt{false} \quad \text{with} \quad \texttt{id} : \alpha \to \alpha \quad \text{and} \quad x : \texttt{Any}$$

```
bind x : Int ; ¬Int = x in
bind y : false = false in
bind z : Int ; ¬Int = id x in
bind u = (z∈Int) ? x : y in
u
```



$\mathbf{u} : \texttt{Int} \vee \texttt{false} \ \Leftarrow$

Reconstruction of the Type of Parameters

- We use tallying to find type substitutions and to infer the type of parameters (just like Algorithm $\mathcal{W}$ uses unification).

Reconstruction of the Type of Parameters

- We use tallying to find type substitutions and to infer the type of parameters (just like Algorithm $\mathcal{W}$ uses unification).

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\mathtt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

For our subtyping relation, tallying is decidable.

Reconstruction of the Type of Parameters

- We use tallying to find type substitutions and to infer the type of parameters (just like Algorithm $\mathcal{W}$ uses unification).

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

For our subtyping relation, tallying is decidable.

- Solutions are characterized by a principal finite set of substitutions (compared to at most one principal substitution for unification).

Reconstruction of the Type of Parameters

- We use tallying to find type substitutions and to infer the type of parameters (just like Algorithm $\mathcal{W}$ uses unification).

Tallying [Castagna et al., 2015] ("unification, but with subtyping constraints"):

$$\texttt{tally}(t_1, t_2) = \{\sigma \mid t_1\sigma \leq t_2\sigma\}$$

For our subtyping relation, tallying is decidable.

- Solutions are characterized by a principal finite set of substitutions (compared to at most one principal substitution for unification).
- Each solution is considered in a separate branch.

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x:γ, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\mathtt{ToBoolean}: (\mathtt{Truthy} \rightarrow \mathtt{true}) \wedge (\mathtt{Falsy} \rightarrow \mathtt{false})$

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x:γ, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x:γ, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: \ (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

find σ, such that

$$\underbrace{((\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false}))}_{\texttt{ToBoolean}} \ \sigma \ \leq \ \underbrace{(\gamma \to \alpha)}_{x \to \text{result}} \ \sigma$$

for some fresh type variable α representing the result of the application

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x:γ, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: \ (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

find σ, such that

$$\underbrace{((\texttt{Truthy} \to \texttt{true}) \ \wedge \ (\texttt{Falsy} \to \texttt{false}))}_{\texttt{ToBoolean}} \ \sigma \ \leq \ \underbrace{(\gamma \to \alpha)}_{x \,\to\, \text{result}} \ \sigma$$

for some fresh type variable α representing the result of the application $\quad \Rightarrow$

$\{\gamma \rightsquigarrow \gamma' \wedge \texttt{Truthy} \ ; \ \alpha \rightsquigarrow \alpha' \vee \texttt{true}\} \ ; \ \{\gamma \rightsquigarrow \gamma'' \wedge \texttt{Falsy} \ ; \ \alpha \rightsquigarrow \alpha'' \vee \texttt{false}\}$

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x: {γ′ ∧ Truthy ; γ″ ∧ Falsy}, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: \ (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

find σ, such that

$$\underbrace{((\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false}))}_{\texttt{ToBoolean}} \sigma \leq \underbrace{(\gamma \to \alpha)}_{x \to \text{result}} \sigma$$

for some fresh type variable α representing the result of the application $\Rightarrow$

$\{\gamma \rightsquigarrow \gamma' \wedge \texttt{Truthy} \ ; \ \alpha \rightsquigarrow \alpha' \vee \texttt{true}\} \ ; \ \{\gamma \rightsquigarrow \gamma'' \wedge \texttt{Falsy} \ ; \ \alpha \rightsquigarrow \alpha'' \vee \texttt{false}\}$

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x: {γ′ ∧ Truthy ; γ″ ∧ Falsy}, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: \; (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

Found two substitutions $\Rightarrow$ we type the body twice (once for each hypothesis)

Reconstruction of the Type of Parameters (example)

$$\texttt{\textbf{function} LogicalOr (x: \{}\gamma' \wedge \texttt{Truthy ; }\gamma'' \wedge \texttt{Falsy\}, y:}\delta\texttt{) \{}$$

$$\quad\texttt{\textbf{if} (ToBoolean(x)) \{ \textbf{return} x; \} \textbf{else} \{ \textbf{return} y; \}}$$

$$\texttt{\}}$$

with $\texttt{ToBoolean}: \ (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

Found two substitutions $\Rightarrow$ we type the body twice (once for each hypothesis)

$$(\gamma' \wedge \texttt{Truthy}, \delta)$$

$$\Downarrow$$

$$\gamma' \wedge \texttt{Truthy}$$

Reconstruction of the Type of Parameters (example)

```
function LogicalOr (x: {γ′ ∧ Truthy ; γ″ ∧ Falsy}, y:δ) {
  if (ToBoolean(x)) { return x; } else { return y; }
}
```

with $\texttt{ToBoolean}: \; (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

Found two substitutions ⇒ we type the body twice (once for each hypothesis)

$(\gamma' \wedge \texttt{Truthy}, \delta)$	$(\gamma'' \wedge \texttt{Falsy}, \delta)$
$\Downarrow$	$\Downarrow$
$\gamma' \wedge \texttt{Truthy}$	δ

Reconstruction of the Type of Parameters (example)

$$\textbf{function}\ \texttt{LogicalOr (x: \{}\gamma' \wedge \texttt{Truthy ; }\gamma'' \wedge \texttt{Falsy\}, y:}\delta\texttt{) \{}$$
$$\quad \textbf{if}\ \texttt{(ToBoolean(x)) \{ }\textbf{return}\texttt{ x; \} }\textbf{else}\texttt{ \{ }\textbf{return}\texttt{ y; \}}$$
$$\texttt{\}}$$

with $\texttt{ToBoolean}: \ (\texttt{Truthy} \to \texttt{true}) \wedge (\texttt{Falsy} \to \texttt{false})$

Found two substitutions $\Rightarrow$ we type the body twice (once for each hypothesis)

$(\gamma' \wedge \texttt{Truthy}, \delta)$	$(\gamma'' \wedge \texttt{Falsy}, \delta)$
$\Downarrow$	$\Downarrow$
$\gamma' \wedge \texttt{Truthy}$	δ

$$((\gamma' \wedge \texttt{Truthy}, \delta) \to \gamma' \wedge \texttt{Truthy}) \ \wedge \ ((\gamma'' \wedge \texttt{Falsy}, \delta) \to \delta)$$

Contribution 3

- Conception of a sound and terminating (but incomplete) algorithm to reconstruct annotation trees, using tallying and backtracking

Contribution 3

- Conception of a sound and terminating (but incomplete) algorithm to reconstruct annotation trees, using tallying and backtracking
- Fully implemented (OCaml, ~ 4600 loc): `https://www.cduce.org/dynlang/`

Contribution 3

- Conception of a sound and terminating (but incomplete) algorithm to reconstruct annotation trees, using tallying and backtracking
- Fully implemented (OCaml, ~ 4600 loc): `https://www.cduce.org/dynlang/`
- Several extensions: pattern matching, records, regular expression types (lists)

Contribution 3

- Conception of a sound and terminating (but incomplete) algorithm to reconstruct annotation trees, using tallying and backtracking
- Fully implemented (OCaml, ~ 4600 loc): `https://www.cduce.org/dynlang/`
- Several extensions: pattern matching, records, regular expression types (lists)
- Several optimizations: tree pruning, memoization, type simplification

Demo (https://www.cduce.org/dynlang/)

```
type Falsy = False | "" | 0 | Null
type Truthy = ~Falsy

let to_boolean x =
  if x is Truthy then true else false
```
type> $(\mathtt{Truthy} \to \mathtt{true}) \;\; \wedge \;\; (\mathtt{Falsy} \to \mathtt{false})$

```
let logical_or (x,y) = if to_boolean x then x else y
```
type> $((\alpha \wedge \mathtt{Truthy}, \mathtt{Any}) \to \alpha \wedge \mathtt{Truthy}) \;\; \wedge \;\; ((\mathtt{Falsy}, \beta) \to \beta)$

```
let id x = logical_or (x,x)
```
type> $\alpha \to \alpha$

Demo (https://www.cduce.org/dynlang/)

```
let fixpoint = fun f ->
  let delta = fun x -> f ( fun  v -> x x v ) in
  delta delta
```

type> $((\alpha \to \beta) \to (\alpha \to \beta) \wedge \gamma) \to (\alpha \to \beta) \wedge \gamma$

```
let map_stub map f lst =
  match lst with
  | [] -> []
  | (e,lst) -> (f e, map f lst)
```

```
let map = fixpoint map_stub
```

type> $(\mathtt{Any} \to [\,] \to [\,]) \;\wedge\; ((\alpha \to \beta) \to [\alpha+] \to [\beta+])$

Demo (https://www.cduce.org/dynlang/)

```
let rec filter (f: (α → Any) ∧ (β → Falsy)) (l: [(α ∨ β)*]) =
  match l with
  | [] -> []
  | (e,l) -> if f e is Truthy then (e, filter f l) else filter f l
  end
type> (α → Any) ∧ (β → Falsy) → [(α ∨ β)*] → [(α ∖ β)*]

let filtered_list = filter to_boolean [42;37;null;42;"";4]
type> [(4 ∨ 37 ∨ 42)*]

let test = map ((+)1) filtered_list
type> [Int*]
```

Conclusion and Perspective

Conclusion

Goal: statically type dynamic languages without hindering their flexibility

Conclusion

Goal: statically type dynamic languages without hindering their flexibility

My contributions:

- Declarative type system mixing union types, intersection types, and polymorphism
- Algorithmic type system, sound and complete, but that requires annotations
- Inference of these annotations using tallying and backtracking
- Fully implemented (OCaml, $\sim$ 4600 loc): `https://www.cduce.org/dynlang/`

Conclusion

Goal: statically type dynamic languages without hindering their flexibility

My contributions:

- Declarative type system mixing union types, intersection types, and polymorphism
- Algorithmic type system, sound and complete, but that requires annotations
- Inference of these annotations using tallying and backtracking
- Fully implemented (OCaml, ~ 4600 loc): `https://www.cduce.org/dynlang/`

Publications:

- Science of Computer Programming: "Revisiting occurrence typing"
- POPL'22: "On Type-Cases, Union Elimination, and Occurrence Typing"
- POPL'24: "Polymorphic Type Inference for Dynamic Languages"

Future Work

Which features do we support?

- Overloaded functions, dynamic dispatch (type-cases)
- Generics (parametric polymorphism)
- Structural subtyping (pairs, records)

Future Work

Which features do we support?

- Overloaded functions, dynamic dispatch (type-cases)
- Generics (parametric polymorphism)
- Structural subtyping (pairs, records)

Which features are missing?

Future Work

Which features do we support?

- Overloaded functions, dynamic dispatch (type-cases)
- Generics (parametric polymorphism)
- Structural subtyping (pairs, records)

Which features are missing?

- Nominal subtyping (abstract data types)

Future Work

Which features do we support?

- Overloaded functions, dynamic dispatch (type-cases)
- Generics (parametric polymorphism)
- Structural subtyping (pairs, records)

Which features are missing?

- Nominal subtyping (abstract data types)
- Mutability of the state (references)

Future Work

Which features do we support?

- Overloaded functions, dynamic dispatch (type-cases)
- Generics (parametric polymorphism)
- Structural subtyping (pairs, records)

Which features are missing?

- Nominal subtyping (abstract data types)
- Mutability of the state (references)
- Gradual typing, for a seamless integration and even more flexibility

Future Work

Which features do we support?

- Overloaded functions, dynamic dispatch (type-cases)
- Generics (parametric polymorphism)
- Structural subtyping (pairs, records)

Which features are missing?

- Nominal subtyping (abstract data types)
- Mutability of the state (references)
- Gradual typing, for a seamless integration and even more flexibility
- Language-specific features
  (example: pattern guards in `Elixir` [Castagna et al., 2023])