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

May 7, 2024

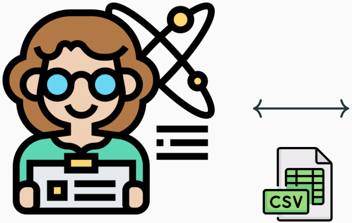
IRIF (Université Paris Cité, France), LMF (Université Paris-Saclay, France)

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

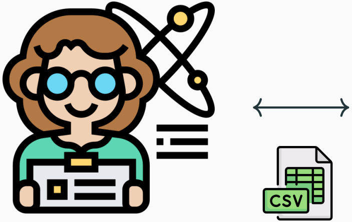
LMF Laboratoire Méthodes Formelles

Background and Motivations

Introduction



`country`	`city`	`pop`	`density`	⋯
USA	Chicago	2665039	4398	⋯
USA	Boston	675647	2911	⋯
France	Pontamafrey	307	26	⋯
⋮	⋮	⋮	⋮	⋱



country	city	pop	density	···
USA	Chicago	2665039	4398	···
USA	Boston	675647	2911	···
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


def get_population_2(data, city):
    df = pandas.read_csv(StringIO(data))
    return df[df['city'] == city].loc[0,'pop']
```

Dynamic Languages

✓ No need to write static types (types only “exist” at runtime)

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

⇒ Flexible, concise, good for prototyping

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

⇒ Flexible, concise, good for prototyping

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

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

⇒ Flexible, concise, good for 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

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

⇒ Flexible, concise, good for prototyping

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

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

⇒ Flexible, concise, good for prototyping

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

Dynamic Languages

- ✓ No need to write static types (types only "exist" at runtime)
- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
- ✓ Overloaded functions with dynamic dispatch:
  which function to call is determined at runtime

⇒ Flexible, concise, good for prototyping

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

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

Motivations

Our goal: statically typing dynamic languages without hindering their flexibility.

Motivations

Our goal: statically typing dynamic languages without hindering their flexibility.

✓ No need to specify types (types only “exist” at runtime)

⇒ Now static types exist, but should be inferred (as much as possible)

Motivations

Our goal: statically typing dynamic languages without hindering their flexibility.

✓ No need to specify types (types only "exist" at runtime)
⇒ Now static types exist, but should be inferred (as much as possible)

✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
⇒ Our type system should feature union types and subtyping

Motivations

Our goal: statically typing dynamic languages without hindering their flexibility.

- ✓ No need to specify types (types only "exist" at runtime)
  ⇒ Now static types exist, but should be inferred (as much as possible)

- ✓ No need to explicitly wrap/unwrap data (`Some` and `None`, `unwrap`, etc.)
  ⇒ Our type system should feature union types and subtyping

- ✓ Overloaded functions with dynamic dispatch
  ⇒ Our type system should be able to type type-cases
  (i.e., conditionals that test the dynamic type of an expression)

Example: Logical Or in JavaScript

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

Example: Logical Or in JavaScript

```
function lOr (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: Logical Or in JavaScript

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

with ToBoolean:

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

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

$$\texttt{ToBoolean}: \ (\texttt{Falsy} \rightarrow \texttt{false}) \land (\texttt{Truthy} \rightarrow \texttt{true})$$

Example: Logical Or in JavaScript

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

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

$$\begin{aligned} \texttt{lOr}: \quad & \forall \alpha, \beta \,. \\ & \quad ((\alpha \wedge \texttt{Truthy}, \texttt{Any}) \to \alpha \wedge \texttt{Truthy}) \\ & \wedge \; ((\texttt{Falsy}, \beta) \to \beta) \end{aligned}$$

Example: Logical Or in JavaScript

```
function lOr (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 $\texttt{lOr} : \forall \alpha, \beta.\ ((\alpha \wedge \texttt{Truthy}, \texttt{Any}) \to \alpha \wedge \texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$

Challenges:

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

  $\Rightarrow$ union types

Example: Logical Or in JavaScript

```
function lOr (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 $\texttt{lOr} : \forall \alpha, \beta.\ ((\alpha \wedge \texttt{Truthy}, \texttt{Any}) \to \alpha \wedge \texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$

Challenges:

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

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

  $\Rightarrow$ intersection types

Example: Logical Or in JavaScript

```
function lOr (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 $\texttt{lOr}: \forall\alpha,\beta.\ ((\alpha{\wedge}\texttt{Truthy}, \texttt{Any}) \to \alpha{\wedge}\texttt{Truthy}) \wedge ((\texttt{Falsy}, \beta) \to \beta)$

Challenges:

- Type narrowing: type the first branch under the hypothesis that `x` is `Truthy`
  $\Rightarrow$ union types
- Capture overloaded behaviors: `lOr` has different behaviors depending on `x`
  $\Rightarrow$ intersection types
- Capture genericity: `lOr` returns its first or second parameter, unchanged
  $\Rightarrow$ parametric polymorphism

Set-Theoretic Types

Set-Theoretic Types $t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Set-Theoretic Types

$$\textbf{Set-Theoretic Types} \quad t \quad ::= \quad b \,|\, t \to t \,|\, t \times t \,|\, t \vee t \,|\, t \wedge t \,|\, \neg t \,|\, \texttt{Empty} \,|\, \texttt{Any}$$

Types are interpreted as subsets of an interpretation domain $\mathcal{D}$ ($\approx$ the set of values):

$$[\![\texttt{false}]\!] = \{\texttt{false}\} \qquad [\![\texttt{Any}]\!] = \mathcal{D}$$

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

Set-Theoretic Types

Set-Theoretic Types $t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Types are interpreted as subsets of an interpretation domain $\mathcal{D}$ ($\approx$ the set of values):

$$\begin{aligned} [\![\texttt{false}]\!] &= \{\texttt{false}\} \\ [\![\texttt{Int}]\!] &= \{0, 1, \ldots\} \end{aligned} \qquad \begin{aligned} [\![\texttt{Any}]\!] &= \mathcal{D} \\ [\![\texttt{Empty}]\!] &= \varnothing \end{aligned}$$

$$\begin{aligned} [\![t_1 \times t_2]\!] &= [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] &= \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} \end{aligned} \qquad \begin{aligned} [\![t_1 \vee t_2]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \wedge t_2]\!] &= [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\neg t]\!] &= \mathcal{D} \setminus [\![t]\!] \end{aligned}$$

Set-Theoretic Types

Set-Theoretic Types $t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Types are interpreted as subsets of an interpretation domain $\mathcal{D}$ ($\approx$ the set of values):

$$\llbracket \texttt{false} \rrbracket = \{\texttt{false}\} \qquad \llbracket \texttt{Any} \rrbracket = \mathcal{D}$$
$$\llbracket \texttt{Int} \rrbracket = \{0, 1, \dots\} \qquad \llbracket \texttt{Empty} \rrbracket = \varnothing$$

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \qquad \llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$$
$$\llbracket t_1 \to t_2 \rrbracket = \text{“}\llbracket t_2 \rrbracket^{\llbracket t_1 \rrbracket}\text{”} \qquad \llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$$
$$\llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket$$

Semantic subtyping:

$$t_1 \leq t_2 \quad \overset{def}{\Longleftrightarrow} \quad \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$$

Set-Theoretic Types (examples)

$$\textbf{Set-Theoretic Types} \quad t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

Semantic subtyping:

$$t_1 \leq t_2 \quad \stackrel{def}{\Longleftrightarrow} \quad [\![t_1]\!] \subseteq [\![t_2]\!]$$

Examples:

- $\texttt{Empty} \leq \texttt{Bool} \leq \texttt{Any}$

Set-Theoretic Types (examples)

$$\textbf{Set-Theoretic Types} \quad t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

Semantic subtyping:

$$t_1 \leq t_2 \quad \stackrel{def}{\Longleftrightarrow} \quad [\![t_1]\!] \subseteq [\![t_2]\!]$$

Examples:

- $\texttt{Empty} \leq \texttt{Bool} \leq \texttt{Any}$
- $\texttt{Bool} \times \texttt{Any} \leq \texttt{Any} \times \texttt{Any}$ (covariant)
- $\texttt{Any} \times \texttt{Bool} \leq \texttt{Any} \times \texttt{Any}$ (covariant)

Set-Theoretic Types (examples)

Set-Theoretic Types $t \ ::= \ b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Semantic subtyping:

$$t_1 \leq t_2 \ \stackrel{def}{\Leftrightarrow} \ [\![t_1]\!] \subseteq [\![t_2]\!]$$

Examples:

- $\texttt{Empty} \leq \texttt{Bool} \leq \texttt{Any}$
- $\texttt{Bool} \times \texttt{Any} \leq \texttt{Any} \times \texttt{Any}$ (covariant)
- $\texttt{Any} \times \texttt{Bool} \leq \texttt{Any} \times \texttt{Any}$ (covariant)
- $\texttt{Any} \to \texttt{Bool} \leq \texttt{Any} \to \texttt{Any}$ (covariant)
- $\texttt{Bool} \to \texttt{Any} \geq \texttt{Any} \to \texttt{Any}$ (contravariant)

Set-Theoretic Types (examples)

Set-Theoretic Types $t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$

Semantic subtyping:

$$t_1 \leq t_2 \quad \overset{def}{\Longleftrightarrow} \quad [\![t_1]\!] \subseteq [\![t_2]\!]$$

Examples:

- $\texttt{Empty} \leq \texttt{Bool} \leq \texttt{Any}$
- $\texttt{Bool} \times \texttt{Any} \leq \texttt{Any} \times \texttt{Any}$ (covariant)
- $\texttt{Any} \times \texttt{Bool} \leq \texttt{Any} \times \texttt{Any}$ (covariant)
- $\texttt{Any} \to \texttt{Bool} \leq \texttt{Any} \to \texttt{Any}$ (covariant)
- $\texttt{Bool} \to \texttt{Any} \geq \texttt{Any} \to \texttt{Any}$ (contravariant)
- $\neg\texttt{Bool} \geq \neg\texttt{Any} (\simeq \texttt{Empty})$ (contravariant)

Adding Type Variables

$$\textbf{Set-Theoretic Types} \quad t \quad ::= \quad b \,|\, t \to t \,|\, t \times t \,|\, t \vee t \,|\, t \wedge t \,|\, \neg t \,|\, \texttt{Empty} \,|\, \texttt{Any} \,|\, \alpha$$

Type substitutions σ : **Vars** → **Types**.

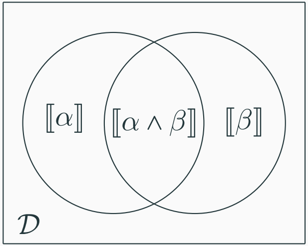
Adding Type Variables

$$\textbf{Set-Theoretic Types} \quad t \; ::= \; b \,|\, t \to t \,|\, t \times t \,|\, t \vee t \,|\, t \wedge t \,|\, \neg t \,|\, \texttt{Empty} \,|\, \texttt{Any} \,|\, \alpha$$

Type substitutions σ : **Vars** → **Types**.

We modify the interpretation domain $\mathcal{D}$ and interpretation $[\![.]\!]$ such that:

$$[\![t_1]\!] \subseteq [\![t_2]\!] \quad \Rightarrow \quad \forall \sigma.\ [\![t_1\sigma]\!] \subseteq [\![t_2\sigma]\!]$$



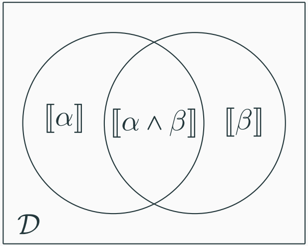
Adding Type Variables

Set-Theoretic Types $t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any} \mid \alpha$

Type substitutions σ : **Vars** → **Types**.

We modify the interpretation domain $\mathcal{D}$ and interpretation $[\![.]\!]$ such that:

$$t_1 \leq t_2 \quad \Rightarrow \quad \forall \sigma.\ t_1\sigma \leq t_2\sigma$$



Syntax and Semantics

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

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

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

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

Declarative Type System

Usual Rules

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \mathsf{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathrm{dom}(\Gamma)$$

Usual Rules

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \mathsf{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

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

Usual Rules

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \mathsf{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

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

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

Usual Rules

$$[\textsc{Const}]\ \frac{}{\Gamma \vdash c : \mathsf{b}_c} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x : \Gamma(x)}\ x \in \mathsf{dom}(\Gamma)$$

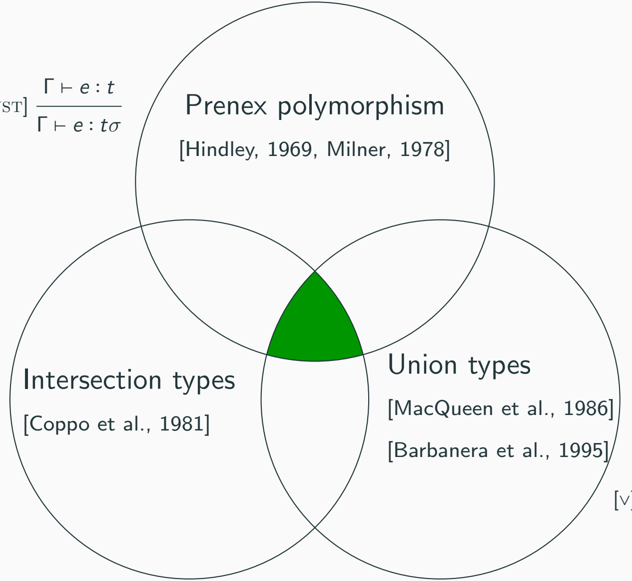
$$[\times\mathrm{I}]\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad [\times\mathrm{E}_1]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_1 e : t_1} \qquad [\times\mathrm{E}_2]\ \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma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

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

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

Mixing Union, Intersection, and HM Polymorphism

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



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

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

Instantiation and Generalization (Hindley Milner)

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

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

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

Instantiation and Generalization (Hindley Milner)

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

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

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

We can instantiate polymorphic type variables:

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

Instantiation and Generalization (Hindley Milner)

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

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

We can instantiate polymorphic type variables:

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

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

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

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

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

We first type `id` using rules [→I] and [Var],

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

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

We first type `id` using rules [→I] and [Var],

Then, we generalize the resulting type using [Gen]: $\gamma \to \gamma$ becomes $\alpha \to \alpha$

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

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

We first type `id` using rules [→I] and [Var],

Then, we generalize the resulting type using [Gen]: $\gamma \rightarrow \gamma$ becomes $\alpha \rightarrow \alpha$

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

Now, when typing `test`, we can instantiate the type of `id`:

- When typing `id` 42, we substitute α by 42 using [Inst]
- When typing `id true`, we substitute α by `true` using [Inst]

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:

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

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` → `Bool`,

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

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

- A first time for the domain `Bool`, yielding `Bool` → `Bool`,
- A second time for the domain ¬`Bool`, yielding ¬`Bool` → ¬`Bool`,

$$[\rightarrow\mathrm{I}]\ \dfrac{[\textsc{Var}]\ \dfrac{}{x : \texttt{Bool} \vdash x : \texttt{Bool}}}{\varnothing \vdash \lambda x.x : \texttt{Bool} \rightarrow \texttt{Bool}} \qquad [\rightarrow\mathrm{I}]\ \dfrac{[\textsc{Var}]\ \dfrac{}{x : \neg\texttt{Bool} \vdash x : \neg\texttt{Bool}}}{\varnothing \vdash \lambda x.x : \neg\texttt{Bool} \rightarrow \neg\texttt{Bool}}$$

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

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

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

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:

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

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

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

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

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

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

We type (x, x):

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

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

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

We type (x, x):

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

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

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

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:

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

Unsound in the presence of polymorphic type variables:

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

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

Unsound in the presence of polymorphic type variables:

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

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

We type (x, x):

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

Unsound in the presence of polymorphic type variables:

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

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

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: $\text{vars}(u) \cap \mathbf{Vars}_P = \varnothing$

Typing Type-Cases

Two cases:

$$(v\in t)\,?\,e_1 : e_2 \quad \rightsquigarrow \quad e_1 \qquad \text{if } v \text{ has type } t$$
$$(v\in t)\,?\,e_1 : e_2 \quad \rightsquigarrow \quad e_2 \qquad \text{otherwise}$$

Typing Type-Cases

Two cases:

$$(v \in t)\ ?\ e_1 : e_2 \quad \rightsquigarrow \quad e_1 \qquad \text{if } v \text{ has type } t$$
$$(v \in t)\ ?\ e_1 : e_2 \quad \rightsquigarrow \quad e_2 \qquad \text{otherwise}$$

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

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

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

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

Typing Type-Cases: Union Elimination and Type Narrowing

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

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

Typing Type-Cases: Union Elimination and Type Narrowing

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

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

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

Typing Type-Cases: Union Elimination and Type Narrowing

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

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

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

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

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

Typing Type-Cases: Union Elimination and Type Narrowing

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

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

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

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

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

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

Typing Type-Cases: Union Elimination and Type Narrowing

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

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

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

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

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

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

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

Capturing Overloaded Behaviors: Intersection Introduction

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

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

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

Capturing Overloaded Behaviors: Intersection Introduction

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

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

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

Capturing Overloaded Behaviors: Intersection Introduction

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

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

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

Capturing Overloaded Behaviors: Intersection Introduction

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

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

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

Capturing Overloaded Behaviors: Intersection Introduction

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

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

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

Algorithmic Type System

Sources of Non-Determinism

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

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

Many possible derivations:

Sources of Non-Determinism

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

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

Many possible derivations:

- Some rules can be applied on every expression (the system is not syntax-directed):
  - Union elimination [∨]
  - Intersection introduction [∧]
  - Instantiation [Inst]
  - Subsumption [≤]

Sources of Non-Determinism

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

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

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 [$\leq$]
- Some premises cannot be guessed from the conclusion (rules are not analytic):
  - The types forming the union in [$\vee$]
  - The type of the parameter in [$\to$I]
  - The substitution in [Inst]

Sources of Non-Determinism

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

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

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]
  - The substitution in [Inst]

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:

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

$$\Rightarrow \quad [\text{App}]\ \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : s' \circ t'} \begin{array}{l} t' = \bigwedge_{i \in I} t\sigma_i \text{ for some } \{\sigma_i\}_{i \in I} \\ s' = \bigwedge_{j \in J} s\sigma'_j \text{ for some } \{\sigma'_j\}_{j \in J} \end{array}$$

$$\text{where} \quad t \circ s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\} \quad (\text{for } s \leq \text{dom}(t))$$

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:

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

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:

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

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

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

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:
  → the type decompositions $s_1 \vee \cdots \vee s_n$ to use in $[\vee]$ rules
  → 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 [MSC | 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:

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

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:

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

$$\downarrow$$

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

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:

$$[\ \underbrace{\texttt{id}\ 42}_{\text{MSC}}\ \mid\ \underbrace{\texttt{@}(\{\alpha \to 42\}, \{\varnothing\})}_{\text{annotation tree}}\ ] \qquad (\text{with } \Gamma(\texttt{id}) = \alpha \to \alpha)$$

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:

$$[\ \underbrace{\texttt{id}\ 42}_{\text{MSC}}\ \mid\ \underbrace{\texttt{@}(\{\alpha \to 42\}, \{\varnothing\})}_{\text{annotation tree}}\ ] \qquad (\text{with } \Gamma(\texttt{id}) = \alpha \to \alpha)$$

$$\downarrow$$

$$[\text{App-Alg}]\ \frac{\Gamma \vdash \texttt{id} : \alpha \to \alpha \qquad \Gamma \vdash 42 : 42}{\Gamma \vdash\ [\texttt{id}\ 42 \mid \texttt{@}(\{\alpha \to 42\}, \{\varnothing\})]\ :\ t \circ s \simeq 42} \quad \begin{array}{l} t = (\alpha \to \alpha)\{\alpha \to 42\} = 42 \to 42 \\ s = 42 \end{array}$$

Properties

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.

Properties

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.

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 e is typeable with the algorithmic type system.

Properties

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.

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 e is typeable with the algorithmic type 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∈true)?x:y`,
  we backtrack to the `bind` definition of **z** and split its type into `true` ; `¬true`

Reconstruction of Type Decompositions

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

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

$$(\texttt{id}\ x \in \texttt{Int})\ ?\ x : 0 \quad \text{with} \quad \texttt{id} : \alpha \to \alpha \ \text{ and } \ x : \texttt{Any}$$

Reconstruction of Type Decompositions

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

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

$$(\mathtt{id}\ x \in \mathtt{Int})\,?\,x : 0 \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \ \text{ and } \ x : \mathtt{Any}$$

$$\mathtt{bind}\,\mathbf{x} \ = \ x \ \mathtt{in}$$
$$\mathtt{bind}\,\mathbf{y} \ = \ 0 \ \mathtt{in}$$
$$\mathtt{bind}\,\mathbf{z} \ = \ \mathtt{id}\ \mathbf{x} \ \mathtt{in}$$
$$\mathtt{bind}\,\mathbf{u} \ = \ (\mathbf{z} \in \mathtt{Int})\,?\,\mathbf{x} : \mathbf{y} \ \mathtt{in}$$
$$\mathbf{u}$$

Reconstruction of Type Decompositions

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

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

$$(\mathtt{id}\ x{\in}\mathtt{Int})\ ?\ x : 0 \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \quad \text{and} \quad x : \mathtt{Any}$$

```
bind x : Any = x in
bind y : 0 = 0 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 `(z∈true)?x:y`, we backtrack to the `bind` definition of **z** and split its type into `true` ; ¬`true`

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

$$(\texttt{id}\ x{\in}\texttt{Int})\ ?\ x : 0 \quad \text{with} \quad \texttt{id} : \alpha \to \alpha \quad \text{and} \quad x : \texttt{Any}$$

```
bind x : Any = x in
bind y : 0 = 0 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 $(\mathbf{z} \in \mathtt{true})\,?\,\mathbf{x}:\mathbf{y}$, we backtrack to the bind definition of **z** and split its type into $\mathtt{true} \,;\, \neg\mathtt{true}$
- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\mathtt{id}\ x \in \mathtt{Int})\,?\,x:0 \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \quad \text{and} \quad x : \mathtt{Any}$$

```
bind x : Any = x in
bind y : 0 = 0 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∈true) ? x : y`,
  we backtrack to the `bind` definition of **z** and split its type into `true` ; `¬true`
- Then, we backpropagate this split on the variables used in the definition of **z**.

$$(\texttt{id}\ x \in \texttt{Int})\ ?\ x : 0 \quad \text{with} \quad \texttt{id} : \alpha \to \alpha \quad \text{and} \quad x : \texttt{Any}$$

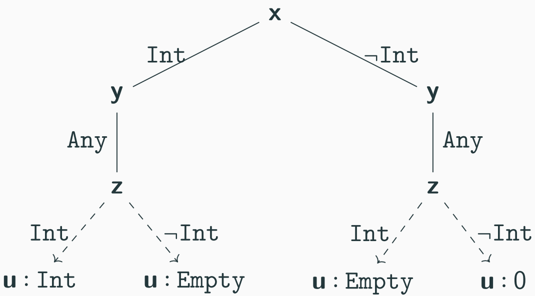
```
bind x : Int ; ¬Int = x in
bind y : 0 = 0 in
bind z : Int ; ¬Int = id x in
bind u  = (z∈Int) ? x : y in
u
```

Reconstruction of Type Decompositions

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

$$(\mathtt{id}\ x\in\mathtt{Int})\,?\,x:0 \quad \text{with} \quad \mathtt{id} : \alpha \to \alpha \quad \text{and} \quad x : \mathtt{Any}$$

```
bind x : Int ; ¬Int =  x in
bind y : 0 =  0 in
bind z : Int ; ¬Int =  id x in
bind u  =  (z∈Int) ? x : y in
u
```



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 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"):

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

For our subtyping relation, tallying is decidable.

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

Reconstruction of the Type of Parameters (example)

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

with ToBoolean : (Truthy → true) ∧ (Falsy → false)

Reconstruction of the Type of Parameters (example)

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

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

find σ, such that

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

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

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

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

Reconstruction of the Type of Parameters (example)

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

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

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

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

Reconstruction of the Type of Parameters (example)

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

with 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)

```
function lOr (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

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

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

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

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

```
let toBoolean x =
  if x is Truthy then true else false
```

$((\alpha \wedge \mathtt{Truthy}, \mathtt{Any}) \to \alpha \wedge \mathtt{Truthy}) \quad \wedge \quad ((\mathtt{Falsy}, \beta) \to \beta)$

```
let lOr (x,y) = if toBoolean x then x else y
```

$\alpha \to \alpha$

```
let id x = lOr (x,x)
```

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

$((\alpha \to \beta) \to (\alpha \to \beta) \land \gamma) \to (\alpha \to \beta) \land \gamma$

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

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

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

```
let map = fixpoint map_stub
```

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

$(\alpha \to \texttt{Any}) \wedge (\beta \to \texttt{Falsy}) \to [(\alpha \vee \beta)*] \to [(\alpha \setminus \beta)*]$

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

$[(4 \vee 37 \vee 42)*]$

```
let filtered_list = filter toBoolean [42;37;null;42;"";4]
```

$[\texttt{Int}*]$

```
let test = map ((+)1) filtered_list
```

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

$[\alpha*] \to [\beta*] \to [\alpha*; \beta*]$

```
let rec concat (x: [α*]) (y: [β*]) = match x with
  | :[] -> y
  | (h, t) -> (h, concat t y)
  end
```

$(\texttt{Tree}\ \alpha \to [(\alpha \setminus \texttt{List})*]) \wedge (\beta \setminus \texttt{List} \to [\beta \setminus \texttt{List}])$

where $\texttt{Tree}\ \alpha = (\alpha \setminus \texttt{List}) \vee [(\texttt{Tree}\ \alpha)*]$

```
let rec deep_flatten x = match x with
  | :[] -> []
  | (h, t) & :List -> concat (deep_flatten h) (deep_flatten t)
  | _ -> [x]
  end
```

Conclusion and Perspective

Conclusion

Objective: type inference of polymorphic and overloaded functions

Our solution:

- 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 backpropagation, tallying, and backtracking
- Fully implemented (OCaml, ~ 4600 loc): `https://www.cduce.org/dynlang/`

Perspective

Which features do we support?

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

Perspective

Which features do we support?

- ✓ Overloaded functions with 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 for more flexibility
- × Language-specific features
  (example: testing the arity of a function in `Elixir` [Castagna et al., 2023])