# Type inference for functional and imperative dynamic languages

using set-theoretic types



Mickaël Laurent, Jan Vitek

October, 2026

Sorbonne Université, Paris

Charles University, Prague

SORBONNE UNIVERSITÉ

CHARLES UNIVERSITY

# Summary

Context and motivations

Pure core calculus with set-theoretic types

How to encode imperative features (opaque data types)

# Context and motivations

Context and motivations

Pure core calculus with set-theoretic types

How to encode imperative features (opaque data types)



# Example: filter function

```
def filter(f, l):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
```

# Example: filter function

```
def filter(f: α → bool, l: list(α)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ∀α. (α → bool, list(α)) → list(α)
```

# Example: filter function

```
def filter(f: α → bool, l: list(α)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
```

type> $\forall\alpha.\ (\alpha \to \texttt{bool},\ \texttt{list}(\alpha)) \to \texttt{list}(\alpha)$

```
filter(lambda x: type(x) is int, [42, None, 13])
```

type> ???

# Example: filter function

```
def filter(f: α → bool, l: list(α)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
```

type> $\forall\alpha.\ (\alpha \to \texttt{bool},\ \texttt{list}(\alpha)) \to \texttt{list}(\alpha)$

```
filter(lambda x: type(x) is int, [42, None, 13])
```

type> $\texttt{list}(\texttt{int} \vee \texttt{None})$

Can we do better?

# Example: filter function

```
def filter(f: (α → bool) ∧ (β → false), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → bool) ∧ (β → false), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
```

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
```

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α \ β)

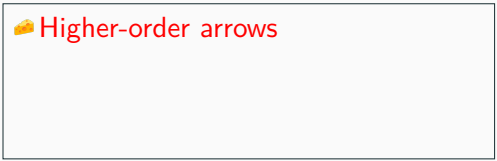
filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```



# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

- Higher-order arrows
- Mutability

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
```

- Higher-order arrows
- Mutability
- Unions

```
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
```

- Higher-order arrows
- Mutability
- Unions
- Intersections (overloads)

```
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α \ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α \ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

- Higher-order arrows
- Mutability
- Unions
- Intersections (overloads)
- Parametric polymorphism (generics)

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

- Higher-order arrows
- Mutability
- Unions
- Intersections (overloads)
- Parametric polymorphism (generics)
- Type narrowing

# Example: filter function

```
type falsy = False ∨ "" ∨ 0 ∨ None ∨ [] ∨ () ∨ {}
type truthy = ¬falsy

def filter(f: (α → any) ∧ (β → falsy), l: list(α ∨ β)):
    res = []
    for e in l:
        if f(e):
            res.append(e)
    return res
type> ((α → any) ∧ (β → falsy), list(α ∨ β)) → list(α ∖ β)

filter(lambda x: type(x) is int, [42, None, 13])
type> list(int)
```

- Higher-order arrows
- Mutability
- Unions
- Intersections (overloads)
- Parametric polymorphism (generics)
- Type narrowing
- Type inference

# Pure core calculus with set-theoretic types

Context and motivations

Pure core calculus with set-theoretic types

How to encode imperative features (opaque data types)



# Set-Theoretic Types [Frisch, 2004]

| | | | |
|---|---|---|---|
| **Basic Types** | $b$ | ::= | $\texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$ |
| **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 [Frisch, 2004]

| | | | |
|---|---|---|---|
| **Basic Types** | $b$ | ::= | $\texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$ |
| **Set-Theoretic Types** | $t$ | ::= | $b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any}$ |

Types are interpreted as sets of values:

$$[\![\texttt{false}]\!] = \{\texttt{false}\} \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\} \qquad [\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} * \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

* Not the actual interpretation (for cardinality reasons).

# Set-Theoretic Types [Frisch, 2004]

$$\begin{array}{lrcl} \textbf{Basic Types} & b & ::= & \texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots \\ \textbf{Set-Theoretic Types} & t & ::= & b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \end{array}$$

Types are interpreted as sets of values:

$$\llbracket \texttt{false} \rrbracket = \{\texttt{false}\} \qquad \llbracket \texttt{bool} \rrbracket = \{\texttt{false}, \texttt{true}\}$$

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \qquad \llbracket t_1 \to t_2 \rrbracket = \text{“}\llbracket t_2 \rrbracket^{\llbracket t_1 \rrbracket}\text{”} \ *$$

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

Semantic subtyping: $t_1 \leq t_2 \overset{def}{\Leftrightarrow} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$

* Not the actual interpretation (for cardinality reasons).

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

| | | | |
|---|---|---|---|
| **Basic Types** | $b$ | ::= | $\texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$ |
| **Set-Theoretic Types** | $t$ | ::= | $b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \mid \alpha$ |

Types are interpreted as sets of values:

$$[\![\texttt{false}]\!] = \{\texttt{false}\} \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\} \qquad [\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} * \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: $t_1 \leq t_2 \iff \forall \sigma.\ t_1\sigma \leq t_2\sigma \iff \forall \sigma.\ [\![t_1\sigma]\!] \subseteq [\![t_2\sigma]\!]$

* Not the actual interpretation (for cardinality reasons).

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

| | | | |
|---|---|---|---|
| **Basic Types** | $b$ | ::= | $\texttt{bool} \mid \texttt{int} \mid \ldots \mid \texttt{false} \mid \texttt{true} \mid 0 \mid 1 \mid \ldots$ |
| **Set-Theoretic Types** | $t$ | ::= | $b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \mid \alpha$ |

Types are interpreted as sets of values:

$$[\![\texttt{false}]\!] = \{\texttt{false}\} \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\texttt{bool}]\!] = \{\texttt{false}, \texttt{true}\} \qquad [\![t_1 \to t_2]\!] = \text{“}[\![t_2]\!]^{[\![t_1]\!]}\text{”} * \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: $t_1 \leq t_2 \iff \forall \sigma.\ t_1\sigma \leq t_2\sigma \iff \forall \sigma.\ [\![t_1\sigma]\!] \subseteq [\![t_2\sigma]\!]$

- The semantic subtyping relation $\leq$ is decidable.
- All set properties are inherited: commutativity, idempotence, distributivity, etc.
- Intersections can be used to type overloaded functions:
  $\texttt{f} : (\alpha \to \texttt{any}) \wedge (\beta \to \texttt{falsy})$

* Not the actual interpretation (for cardinality reasons).

# Core type system [Castagna et al., 2024]

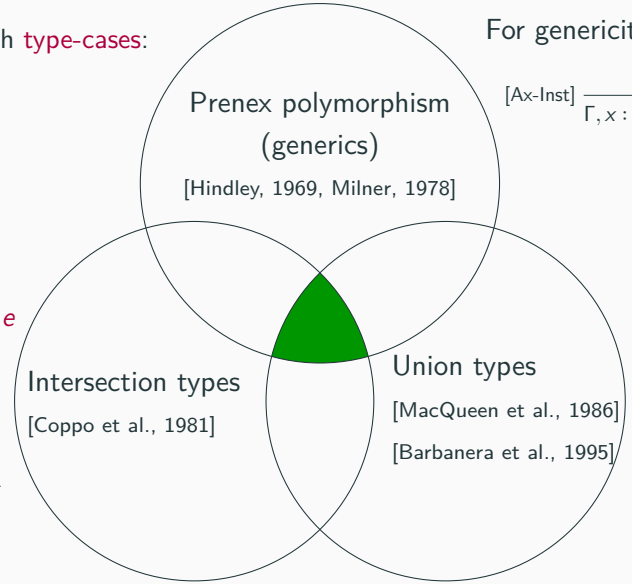
Pure $\lambda$-calculus with type-cases:

$$
\begin{array}{rcl}
e & ::= & c \\
& \mid & x \\
& \mid & \lambda x.e \\
& \mid & e\,e \\
& \mid & (e, e) \\
& \mid & \pi_i e \\
& \mid & (e \in t)\,?\,e : e
\end{array}
$$

# Core type system [Castagna et al., 2024]

Pure $\lambda$-calculus with type-cases:

$$
\begin{array}{rcl}
e & ::= & c \\
& \mid & x \\
& \mid & \lambda x.e \\
& \mid & e\,e \\
& \mid & (e, e) \\
& \mid & \pi_i e \\
& \mid & (e \in t)\,?\,e : e
\end{array}
$$




For genericity

$$[\text{Ax-Inst}]\ \frac{}{\Gamma, x : \forall \vec{\alpha}.t \vdash x : t\sigma}\ \mathrm{dom}(\sigma) \subseteq \vec{\alpha}$$

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

$$[\text{Let}]\ \frac{\Gamma \vdash e_1 : \bigvee_{i \in I} s_i \qquad (\forall i \in I)\ \Gamma, x : s_i \vdash e_2 : t}{\Gamma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : t}$$

# Type inference and tallying [Castagna et al., 2015]

Type inference based on tallying ($\simeq$ unification with subtyping constraints), similar to algorithm $\mathcal{W}$ of Hindley-Milner systems.

# Type inference and tallying [Castagna et al., 2015]

Type inference based on tallying ($\simeq$ unification with subtyping constraints), similar to algorithm $\mathcal{W}$ of Hindley-Milner systems.

$\lambda x.\ x + x$ where $+ : (\texttt{int} \times \texttt{int} \to \texttt{int}) \wedge (\texttt{string} \times \texttt{string} \to \texttt{string})$

# Type inference and tallying [Castagna et al., 2015]

Type inference based on tallying ($\simeq$ unification with subtyping constraints), similar to algorithm $\mathcal{W}$ of Hindley-Milner systems.

$\lambda x : \alpha.\ x + x$ where $+ : (\texttt{int} \times \texttt{int} \to \texttt{int}) \wedge (\texttt{string} \times \texttt{string} \to \texttt{string})$

# Type inference and tallying [Castagna et al., 2015]

Type inference based on tallying ($\simeq$ unification with subtyping constraints), similar to algorithm $\mathcal{W}$ of Hindley-Milner systems.

$\lambda x : \alpha.\ x + x$ where $+ : (\texttt{int} \times \texttt{int} \to \texttt{int}) \wedge (\texttt{string} \times \texttt{string} \to \texttt{string})$

We need to find every $\sigma$ such that:

$$\underbrace{((\texttt{int} \times \texttt{int} \to \texttt{int}) \wedge (\texttt{string} \times \texttt{string} \to \texttt{string}))}_{\texttt{type of +}}\,\sigma \;\leq\; (\ \underbrace{\alpha \times \alpha}_{\text{type of arg}} \to \underbrace{\beta}_{\text{result}}\ )\,\sigma$$

# Type inference and tallying [Castagna et al., 2015]

Type inference based on tallying ($\simeq$ unification with subtyping constraints), similar to algorithm $\mathcal{W}$ of Hindley-Milner systems.

$\lambda x : \alpha.\ x + x$ where $+ : (\texttt{int} \times \texttt{int} \to \texttt{int}) \wedge (\texttt{string} \times \texttt{string} \to \texttt{string})$

We need to find every $\sigma$ such that:

$$\underbrace{((\texttt{int} \times \texttt{int} \to \texttt{int}) \wedge (\texttt{string} \times \texttt{string} \to \texttt{string}))}_{\texttt{type of +}}\,\sigma \;\leq\; (\ \underbrace{\alpha \times \alpha}_{\text{type of arg}} \to \underbrace{\beta}_{\text{result}}\ )\,\sigma$$

Two solutions: $\sigma_1 = \{\alpha \rightsquigarrow \texttt{int},\ \beta \rightsquigarrow \texttt{int}\}$ and $\sigma_2 = \{\alpha \rightsquigarrow \texttt{string},\ \beta \rightsquigarrow \texttt{string}\}$

# Type inference and tallying [Castagna et al., 2015]

Type inference based on tallying ($\simeq$ unification with subtyping constraints), similar to algorithm $\mathcal{W}$ of Hindley-Milner systems.

$\lambda x : \alpha.\ x + x$ where $+ : (\mathtt{int} \times \mathtt{int} \to \mathtt{int}) \wedge (\mathtt{string} \times \mathtt{string} \to \mathtt{string})$

We need to find every $\sigma$ such that:

$$\underbrace{((\mathtt{int} \times \mathtt{int} \to \mathtt{int}) \wedge (\mathtt{string} \times \mathtt{string} \to \mathtt{string}))}_{\text{type of } +}\ \sigma \ \leq\ (\ \underbrace{\alpha \times \alpha}_{\text{type of arg}} \ \to \ \underbrace{\beta}_{\text{result}}\ )\ \sigma$$

Two solutions: $\sigma_1 = \{\alpha \rightsquigarrow \mathtt{int},\ \beta \rightsquigarrow \mathtt{int}\}$ and $\sigma_2 = \{\alpha \rightsquigarrow \mathtt{string},\ \beta \rightsquigarrow \mathtt{string}\}$

$$\begin{array}{ccc} \lambda x : \mathtt{int}.\ x + x & & \lambda x : \mathtt{string}.\ x + x \\ \Downarrow & & \Downarrow \\ (\mathtt{int} \to \mathtt{int}) & \wedge & (\mathtt{string} \to \mathtt{string}) \end{array}$$

# How to encode imperative features (opaque data types)

Context and motivations

Pure core calculus with set-theoretic types

How to encode imperative features (opaque data types)



# Towards a real-world imperative language

Our pure $\lambda$-calculus is lacking features:

# Towards a real-world imperative language

Our pure $\lambda$-calculus is lacking features:

- 🧀 Imperativity
  - 🍷 Mutable variables
  - 🍷 Mutable data structures (arrays, dictionaries, etc.)
  - 🍷 Imperative control flow (loops, break, return, exception, etc.)

# Towards a real-world imperative language

Our pure $\lambda$-calculus is lacking features:

- Imperativity
  - Mutable variables
  - Mutable data structures (arrays, dictionaries, etc.)
  - Imperative control flow (loops, break, return, exception, etc.)
- Objects, modules, traits, etc.

# Towards a real-world imperative language

Our pure $\lambda$-calculus is lacking features:

- Imperativity
  - Mutable variables
  - Mutable data structures (arrays, dictionaries, etc.)
  - Imperative control flow (loops, break, return, exception, etc.)
- Objects, modules, traits, etc.
- Other language-specific features (e.g. laziness, dynamic dispatch mechanisms)

# Opaque data types

To help us encode these features,

we introduce a general notion of opaque data types in the set-theoretic type algebra.

# Opaque data types

To help us encode these features,
we introduce a general notion of opaque data types in the set-theoretic type algebra.

We have seen that each type constructor has an interpretation:

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

# Opaque data types

To help us encode these features,
we introduce a general notion of opaque data types in the set-theoretic type algebra.

We have seen that each type constructor has an interpretation:

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

A parametric opaque data type $\mathsf{o}(.)$ has an unknown interpretation:

$$\llbracket \mathsf{o}(t) \rrbracket \stackrel{\text{def}}{=} f(\llbracket t \rrbracket) \quad \text{for some unknown } f : \mathcal{P}(\mathcal{V}) \to \mathcal{P}(\mathcal{V})$$

# Opaque data types

To help us encode these features,
we introduce a general notion of opaque data types in the set-theoretic type algebra.

We have seen that each type constructor has an interpretation:

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

A parametric opaque data type $\mathsf{o}(.)$ has an unknown interpretation:

$$\llbracket \mathsf{o}(t) \rrbracket \stackrel{\text{def}}{=} f(\llbracket t \rrbracket) \quad \text{for some unknown } f : \mathcal{P}(\mathcal{V}) \to \mathcal{P}(\mathcal{V})$$

Subtyping must be sound w.r.t every possible interpretation.

# Opaque data types

To help us encode these features,
we introduce a general notion of opaque data types in the set-theoretic type algebra.

We have seen that each type constructor has an interpretation:

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

A parametric opaque data type $\mathsf{o}(.)$ has an unknown interpretation:

$$\llbracket \mathsf{o}(t) \rrbracket \stackrel{\text{def}}{=} f(\llbracket t \rrbracket) \quad \text{for some unknown } f : \mathcal{P}(\mathcal{V}) \to \mathcal{P}(\mathcal{V})$$

Subtyping must be sound w.r.t every possible interpretation.

If $\mathtt{f}$ satisfies some properties $\Rightarrow$ more precise subtyping relation.

# Opaque data types

| **Property of $f$** | **Description of $\mathsf{o}$** | **Subtyping relation** $\bigwedge_{p\in P} \mathsf{o}(t_p) \leq \bigvee_{n\in N} \mathsf{o}(s_n)$ holds when... |
|---|---|---|
| No property | Invariant parameter | $\exists p \in P.\ \exists n \in N.\ t_p \simeq s_n$ |
| Monotonic | Covariant parameter | $\exists p \in P.\ \exists n \in N.\ t_p \leq s_n$ |
| Isomorphic to identity | Just tagging/boxing | $\bigwedge_{p\in P} t_p \leq \bigvee_{n\in N} s_n$ |

# Opaque data types

| **Property of $f$** | **Description of $\circ$** | **Subtyping relation** $\bigwedge_{p\in P} \circ(t_p) \leq \bigvee_{n\in N} \circ(s_n)$ holds when... |
|---|---|---|
| No property | Invariant parameter | $\exists p \in P.\ \exists n \in N.\ t_p \simeq s_n$ |
| Monotonic | Covariant parameter | $\exists p \in P.\ \exists n \in N.\ t_p \leq s_n$ |
| Isomorphic to identity | Just tagging/boxing | $\bigwedge_{p\in P} t_p \leq \bigvee_{n\in N} s_n$ |

Can be refined with additional properties:
preservation of $\wedge$, preservation of $\vee$, preservation of `any`, preservation of `empty`

# Encodings

| Feature | Encoded with... |
|---|---|
| Mutable variables | Invariant opaque data type `ref(.)` + value restriction<br>`assign: ref('a) -> 'a -> unit`<br>`read: ref('a) -> 'a` |
| Mutable arrays | Invariant opaque data type `array(.)` + value restriction |
| Laziness | Covariant opaque data type `thunk(.)` around arguments<br>`force: thunk('a) -> 'a` |
| Objects | Record type tagged with an opaque data type,<br>e.g. `object(`{`attr1` : $t_1$, ..., `attrn` : $t_n$}`)` |

# Program transformations (just for typing)

```
let filter f arr =
  let res = array () in
  let mut i = 0 in

  while i < len arr do

    let e = arr[i] in

    if f e do push res e end ;

    i := i + 1
  end ;
  return res
```

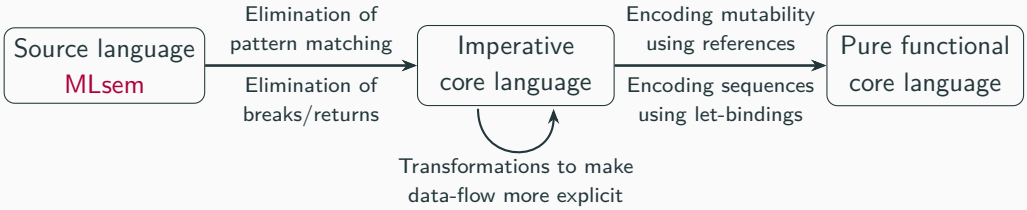
⇒

```
fun f -> fun arr ->
  let res = array () in
  let i = ref 0 in
  let _ =
    let i1 = read i in
    if i1 < len arr then
      let e = arr[i1] in
      let _ =
        if f e then push res e else ()
      in
      assign (i, i1 + 1)
    else () in
  res
```

# MLsem (https://github.com/E-Sh4rk/MLsem)

MLsem is a type-checking library for dynamic languages, featuring:

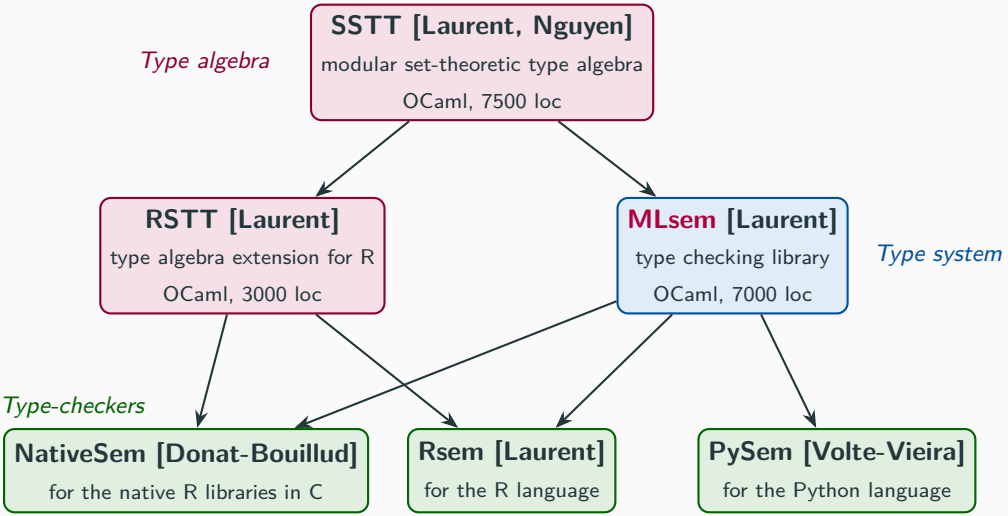
- polymorphism (subtyping, ad-hoc, parametric),
- imperative constructs,
- type narrowing and type inference [Castagna et al., 2024],
- gradual typing [Castagna et al., 2019]



# MLsem (https://github.com/E-Sh4rk/MLsem)

```
14  abstract type array('a)
15  val array : () -> array('a)
16  val to_array : ['a*] -> array('a)
17  val ([]<-) : (array('b), int, 'b) -> ()
18  val ([]) : (array('b), int) -> 'b
19  val push : array('a) -> 'a -> ()
20  val len : array('a) -> int
21
    ∀'a,'b,'c. ('b -> false) & ('a -> bool) -> array('b | 'a) -> array('a \ 'b | 'c)
22  let filter (f:('a -> bool) & ('b -> false)) (arr:array('a|'b)) =
23    let res = array () in
24    let mut i = 0 in
25    while i < len arr do
26      let e = arr[i] in
27      if f e do push res e end ;
28      i := i + 1
29    end ;
30    return res
31
    array(int)
32  let filter_test = filter (fun x -> (x is int)) (to_array [42 ; None ; 13])
```

# A modular typing framework



# Conclusion

- Type system for a pure $\lambda$-calculus:
  - Ad-hoc polymorphism (overloading), parametric polymorphism (genericity)
  - Type narrowing, gradual typing
  - Type inference based on tallying ($\simeq$ HM unification)
- Applications to « real » dynamic languages using encodings:
  - Opaque data types can be used to type many data structures
  - Program transformations to eliminate imperative control flow, etc.
- Related work:
  - Algebraic subtyping [Dolan and Mycroft, 2017, Parreaux and Chau, 2022]
    $\Rightarrow$ no ad hoc polymorphism, but a principal (polar) type inference
  - Other type systems based on set-theoretic types:
    for Elixir [Castagna et al., 2023], for Erlang [Schimpf et al., 2023]

# References i


Barbanera, F., Dezani-Ciancaglini, M., and de'Liguoro, U. (1995).
**Intersection and union types.**
*Inf. Comput.*, 119(2):202–230.

Castagna, G., Duboc, G., and Valim, J. (2023).
**The design principles of the elixir type system.**
*The Art, Science, and Engineering of Programming*, 8(2).

Castagna, G., Lanvin, V., Petrucciani, T., and Siek, J. G. (2019).
**Gradual typing: a new perspective.**
*Proc. ACM Program. Lang.*, 3(POPL).

# References ii



Castagna, G., Laurent, M., and Nguyễn, K. (2024).
**Polymorphic type inference for dynamic languages.**
*Proc. ACM Program. Lang.*, 8(POPL).


Castagna, G., Nguyen, K., Xu, Z., and Abate, P. (2015).
**Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction.**
In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '15, pages 289–302.

# References iii



Castagna, G. and Xu, Z. (2011).
**Set-theoretic foundation of parametric polymorphism and subtyping.**
In *ICFP '11: 16th ACM-SIGPLAN International Conference on Functional Programming*, pages 94–106.


Coppo, M., Dezani-Ciancaglini, M., and Venneri, B. (1981).
**Functional characters of solvable terms.**
*Mathematical Logic Quarterly*, 27(2-6):45–58.


Dolan, S. and Mycroft, A. (2017).
**Polymorphism, subtyping, and type inference in mlsub.**
In Castagna, G. and Gordon, A. D., editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 60–72, New York, NY, USA. Association for Computing Machinery.


Frisch, A. (2004).
***Théorie, conception et réalisation d'un langage de programmation adapté à XML.***
PhD thesis, Université Paris Diderot.


Hindley, J. R. (1969).
**The principal type-scheme of an object in combinatory logic.**
*Transactions of the American Mathematical Society*, 146:29–60.


MacQueen, D., Plotkin, G., and Sethi, R. (1986).
**An ideal model for recursive polymorphic types.**
*Information and Control*, 71(1):95–130.


Milner, R. (1978).
**A theory of type polymorphism in programming.**
*Journal of Computer and System Sciences*, 17(3):348–375.

Parreaux, L. and Chau, C. Y. (2022).
**MlStruct: Principal type inference in a boolean algebra of structural types.**
*Proc. ACM Program. Lang.*, 6(OOPSLA2).

Schimpf, A., Wehr, S., and Bieniusa, A. (2023).
**Set-theoretic types for erlang.**
In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*, IFL '22, New York, NY, USA. Association for Computing Machinery.