Implementing Set-Theoretic Types

Mickaël Laurent, Kim Nguyễn

OOPSLA, Oakland, CA, October 2026







Primer on set-theoretic types

- **Primer on set-theoretic types**

  Implementation

  Evaluation

Typing dynamic languages

```
def f(x):
  if type(x) is int:
          return x+1
  return x
```

Typing dynamic languages

```
def f(x):
  if type(x) is int:
          return x+1
  return x
```

- any → any *correct, but imprecise*

Typing dynamic languages

```
def f(x):
  if type(x) is int:
          return x+1
  return x
```

- $\texttt{any} \to \texttt{any}$ — *correct, but imprecise*
- $(\texttt{int} \to \texttt{int}) \wedge (\neg\texttt{int} \to \neg\texttt{int})$ — *overloading, type narrowing*

Typing dynamic languages

```
def f(x):
  if type(x) is int:
          return x+1
  return x
```

- $\mathtt{any} \to \mathtt{any}$ — *correct, but imprecise*
- $(\mathtt{int} \to \mathtt{int}) \wedge (\neg\mathtt{int} \to \neg\mathtt{int})$ — *overloading, type narrowing*
- $(\mathtt{int} \to \mathtt{int}) \wedge (\alpha \setminus \mathtt{int} \to \alpha \setminus \mathtt{int})$ — *+ genericity*

Typing dynamic languages

```
def f(x):
  if type(x) is int:
          return x+1
  return x
```

- $\mathtt{any} \to \mathtt{any}$ — *correct, but imprecise*
- $(\mathtt{int} \to \mathtt{int}) \wedge (\neg\mathtt{int} \to \neg\mathtt{int})$ — *overloading, type narrowing*
- $(\mathtt{int} \to \mathtt{int}) \wedge (\alpha \setminus \mathtt{int} \to \alpha \setminus \mathtt{int})$ — *+ genericity*

Set-theoretic types: unions, intersections, negations, type variables

A natural type algebra for dynamic languages:

Elixir [Castagna et al., 2023], Erlang [Schimpf et al., 2023], MLsem [Laurent and Vitek, 2026]

Typing dynamic languages

```
def f(x):
  if type(x) is int:
          return x+1
  return x
```

- $\texttt{any} \to \texttt{any}$ — *correct, but imprecise*
- $(\texttt{int} \to \texttt{int}) \wedge (\neg\texttt{int} \to \neg\texttt{int})$ — *overloading, type narrowing*
- $(\texttt{int} \to \texttt{int}) \wedge (\alpha \setminus \texttt{int} \to \alpha \setminus \texttt{int})$ — *+ genericity*

Set-theoretic types: unions, intersections, negations, type variables

A natural type algebra for dynamic languages:

Elixir [Castagna et al., 2023], Erlang [Schimpf et al., 2023], MLsem [Laurent and Vitek, 2026]

- *Type inference for functional and imperative dynamic languages* (this morning)
- *Revisiting Row Polymorphism for Set-Theoretic Types* (next session)

Set-theoretic types [Frisch et al., 2008]

Set-Theoretic Types	t	$::=$	$b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \mid \alpha$
Basic Types	b	$::=$	$\texttt{int} \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid \texttt{42} \mid \ldots$

*co*inductively defined: regular + contractive ⇒ equirecursive types, no explicit binder:

Set-theoretic types [Frisch et al., 2008]

Set-Theoretic Types	t	$::=$	$b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \mid \alpha$
Basic Types	b	$::=$	$\texttt{int} \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid \texttt{42} \mid \ldots$

*co*inductively defined: regular + contractive ⇒ equirecursive types, no explicit binder:

```
int list
```

Set-theoretic types [Frisch et al., 2008]

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

Basic Types $b ::= \texttt{int} \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid \texttt{42} \mid \ldots$

*co*inductively defined: regular + contractive ⇒ equirecursive types, no explicit binder:

$\texttt{int list} \quad \Rightarrow \quad X = \texttt{nil} \vee (\texttt{int} \times X)$

Set-theoretic types [Frisch et al., 2008]

$$\textbf{Set-Theoretic Types} \quad t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{empty} \mid \texttt{any} \mid \alpha$$
$$\textbf{Basic Types} \quad b ::= \texttt{int} \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid 42 \mid \ldots$$

*co*inductively defined: regular + contractive ⇒ equirecursive types, no explicit binder:

$$\texttt{int list} \;\Rightarrow\; X = \texttt{nil} \vee (\texttt{int} \times X) \;\Rightarrow\; \texttt{nil} \vee (\texttt{int} \times \texttt{nil}) \vee (\texttt{int} \times (\texttt{int}, \texttt{nil})) \vee \ldots$$

Set-theoretic types [Frisch et al., 2008]

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

Basic Types $b ::= \texttt{int} \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid 42 \mid \ldots$

*co*inductively defined: regular + contractive ⇒ equirecursive types, no explicit binder:

$$\texttt{int list} \quad \Rightarrow \quad X = \texttt{nil} \vee (\texttt{int} \times X) \quad \Rightarrow \quad \texttt{nil} \vee (\texttt{int} \times \texttt{nil}) \vee (\texttt{int} \times (\texttt{int}, \texttt{nil})) \vee \ldots$$

Types are interpreted as sets of values:

$$[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\} \qquad [\![\texttt{any}]\!] = \mathcal{V} \text{ (all the values)} \qquad [\![\texttt{empty}]\!] = \varnothing$$

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad \ldots$$

Set-theoretic types [Frisch et al., 2008]

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

Basic Types $b ::= \texttt{int} \mid \texttt{bool} \mid \texttt{true} \mid \texttt{false} \mid 42 \mid \ldots$

*co*inductively defined: regular + contractive ⇒ equirecursive types, no explicit binder:

$$\texttt{int list} \;\Rightarrow\; X = \texttt{nil} \vee (\texttt{int} \times X) \;\Rightarrow\; \texttt{nil} \vee (\texttt{int} \times \texttt{nil}) \vee (\texttt{int} \times (\texttt{int}, \texttt{nil})) \vee \ldots$$

Types are interpreted as sets of values:

$$[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\} \qquad [\![\texttt{any}]\!] = \mathcal{V} \text{ (all the values)} \qquad [\![\texttt{empty}]\!] = \varnothing$$

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad \ldots$$

Subtyping is set containment (or testing for emptiness):

$$t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1 \setminus t_2]\!] = \varnothing \qquad t_1 \simeq t_2 \overset{\text{def}}{\iff} t_1 \leq t_2 \text{ and } t_2 \leq t_1$$

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Tallying: "unification with subtyping constraints": $\exists\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \exists\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

It is complete: a finite set of substitutions characterizes all solutions.

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Tallying: "unification with subtyping constraints": $\exists\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \exists\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

It is complete: a finite set of substitutions characterizes all solutions.

Example: typing the partial application `map even`, with

$\texttt{map} : (\alpha \to \beta) \to [\alpha*] \to [\beta*] \qquad \texttt{even} : (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int})$

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Tallying: "unification with subtyping constraints": $\exists\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \exists\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

It is complete: a finite set of substitutions characterizes all solutions.

Example: typing the partial application `map even`, with

$$\texttt{map} : (\alpha \to \beta) \to [\alpha*] \to [\beta*] \qquad \texttt{even} : (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int})$$

$$\exists\sigma. \quad (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int}) \ \leq\ \alpha \to \beta$$

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Tallying: "unification with subtyping constraints": $\exists\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \exists\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

It is complete: a finite set of substitutions characterizes all solutions.

Example: typing the partial application `map even`, with

$$\texttt{map} : (\alpha \to \beta) \to [\alpha*] \to [\beta*] \qquad \texttt{even} : (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int})$$

$$\exists\sigma.\quad (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int}) \ \leq\ \alpha \to \beta$$

$\sigma_1 = \{\alpha : \texttt{empty}\}$ $\quad \sigma_2 = \{\alpha : \texttt{int}, \beta : \texttt{bool}\}$ $\quad \sigma_3 = \{\alpha : \gamma \setminus \texttt{int}, \beta : \gamma \setminus \texttt{int}\}$

$\sigma_4 = \{\alpha : \texttt{int} \vee (\gamma \setminus \texttt{int}), \beta : \texttt{bool} \vee (\gamma \setminus \texttt{int})\}$

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Tallying: "unification with subtyping constraints": $\exists\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \exists\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

It is complete: a finite set of substitutions characterizes all solutions.

Example: typing the partial application `map even`, with

$$\texttt{map} : (\alpha \to \beta) \to [\alpha*] \to [\beta*] \qquad \texttt{even} : (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int})$$

$$\exists\sigma.\quad (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int}) \ \leq\ \alpha \to \beta$$

$\sigma_1 = \{\alpha : \texttt{empty}\}$ $\quad \sigma_2 = \{\alpha : \texttt{int}, \beta : \texttt{bool}\}$ $\quad \sigma_3 = \{\alpha : \gamma \setminus \texttt{int}, \beta : \gamma \setminus \texttt{int}\}$

$\sigma_4 = \{\alpha : \texttt{int} \vee (\gamma \setminus \texttt{int}), \beta : \texttt{bool} \vee (\gamma \setminus \texttt{int})\}$

- All properties inherited from set theory (commutativity, De Morgan's laws, ...)
- Supports extensible records, n-arry tuples, regular expression types, ...

Subtyping [Castagna and Xu, 2011], Tallying [Castagna et al., 2015]

Subtyping must hold for all substitutions: $\forall\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

Tallying: "unification with subtyping constraints": $\exists\sigma.\ s\sigma \leq t\sigma \ \Leftrightarrow\ \exists\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

It is complete: a finite set of substitutions characterizes all solutions.

Example: typing the partial application `map even`, with

$$\texttt{map} : (\alpha \to \beta) \to [\alpha*] \to [\beta*] \qquad \texttt{even} : (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int})$$

$$\exists\sigma. \quad (\texttt{int} \to \texttt{bool}) \wedge (\gamma \setminus \texttt{int} \to \gamma \setminus \texttt{int}) \ \leq \ \alpha \to \beta$$

$$\sigma_1 = \{\alpha : \texttt{empty}\} \qquad \sigma_2 = \{\alpha : \texttt{int}, \beta : \texttt{bool}\} \qquad \sigma_3 = \{\alpha : \gamma \setminus \texttt{int}, \beta : \gamma \setminus \texttt{int}\}$$

$$\sigma_4 = \{\alpha : \texttt{int} \vee (\gamma \setminus \texttt{int}), \beta : \texttt{bool} \vee (\gamma \setminus \texttt{int})\}$$

- All properties inherited from set theory (commutativity, De Morgan's laws, ...)
- Supports extensible records, n-arry tuples, regular expression types, ...
- Subtyping is $2^{O(n)}$

Why (yet) another implementation?

The theory is well established. The implementations are not:

- CDuce [Frisch, 2004]: the historical one, but designed around XML;
  its polymorphic part [Castagna et al., 2015] was never described or evaluated
- Elixir [Castagna et al., 2023] (no type variables nor recursive types yet)
- Etylizer (Erlang) [Schimpf et al., 2023] (interesting but still slow)

Why (yet) another implementation?

The theory is well established. The implementations are not:

- CDuce [Frisch, 2004]: the historical one, but designed around XML;
  its polymorphic part [Castagna et al., 2015] was never described or evaluated
- Elixir [Castagna et al., 2023] (no type variables nor recursive types yet)
- Etylizer (Erlang) [Schimpf et al., 2023] (interesting but still slow)

⇒ Sstt, the *simple set-theoretic type library*: OCaml, 6 900 loc, open and modular

Why (yet) another implementation?

The theory is well established. The implementations are not:

- CDuce [Frisch, 2004]: the historical one, but designed around XML;
  its polymorphic part [Castagna et al., 2015] was never described or evaluated
- Elixir [Castagna et al., 2023] (no type variables nor recursive types yet)
- Etylizer (Erlang) [Schimpf et al., 2023] (interesting but still slow)

⇒ Sstt, the *simple set-theoretic type library*: OCaml, 6 900 loc, open and modular

Contributions:

- A novel semantic simplification operation for Binary Decision Trees (BDTs)
- A layered representation: constructors / connectives / variables / recursion
- Revisited subtyping and tallying algorithms, evaluated against CDuce

Implementation

Primer on set-theoretic types

- **Implementation**

Evaluation

Disjunctive normal forms

Deciding $t_1 \leq t_2$ = deciding $t_1 \setminus t_2 \simeq \texttt{empty}$, and emptiness is decided on the DNF:

$$\mathrm{DNF}(t) \;=\; \bigvee \Big(\underbrace{\textstyle\bigwedge_i \alpha_i \wedge \bigwedge_j \neg\beta_j}_{\text{type variables}} \wedge \underbrace{\textstyle\bigwedge_k (t_k \to s_k) \wedge \bigwedge_l \neg(t'_l \to s'_l)}_{\text{a single kind of type constructor}} \Big)$$

Disjunctive normal forms

Deciding $t_1 \leq t_2$ = deciding $t_1 \setminus t_2 \simeq \mathtt{empty}$, and emptiness is decided on the DNF:

$$\mathrm{DNF}(t) \;=\; \bigvee \Big(\underbrace{\textstyle\bigwedge_i \alpha_i \wedge \bigwedge_j \neg\beta_j}_{\text{type variables}} \wedge \underbrace{\textstyle\bigwedge_k (t_k \to s_k) \wedge \bigwedge_l \neg(t'_l \to s'_l)}_{\text{a single kind of type constructor}} \Big)$$

Example, for arrows:

$$\begin{aligned} & \big((\mathtt{int} \to \mathtt{int}) \vee (\mathtt{bool} \to \mathtt{bool})\big) \setminus (\alpha \to \alpha) \\ \overset{\text{dnf}}{=}\; & \big((\mathtt{int} \to \mathtt{int}) \wedge \neg(\alpha \to \alpha)\big) \;\vee\; \big((\mathtt{bool} \to \mathtt{bool}) \wedge \neg(\alpha \to \alpha)\big) \end{aligned}$$

Disjunctive normal forms

Deciding $t_1 \leq t_2$ = deciding $t_1 \setminus t_2 \simeq \texttt{empty}$, and emptiness is decided on the DNF:

$$\mathrm{DNF}(t) \;=\; \bigvee \Big(\underbrace{\textstyle\bigwedge_i \alpha_i \wedge \bigwedge_j \neg\beta_j}_{\text{type variables}} \wedge \underbrace{\textstyle\bigwedge_k (t_k \to s_k) \wedge \bigwedge_l \neg(t'_l \to s'_l)}_{\text{a single kind of type constructor}} \Big)$$

Example, for arrows:

$$\begin{aligned} & \big((\texttt{int} \to \texttt{int}) \vee (\texttt{bool} \to \texttt{bool})\big) \setminus (\alpha \to \alpha) \\ \overset{\text{dnf}}{=}\; & \big((\texttt{int} \to \texttt{int}) \wedge \neg(\alpha \to \alpha)\big) \;\vee\; \big((\texttt{bool} \to \texttt{bool}) \wedge \neg(\alpha \to \alpha)\big) \end{aligned}$$

Question: how do we store such a normal form?

A list of clauses duplicates atoms and explodes after a few unions and intersections.

Binary Decision Trees

(Ordered) BDTs $B ::= \mathtt{L}(l) \mid \mathtt{N}(a?B^+ : B^-)$

leaves: l, elements of some lattice

atoms: a, any object with total order $\preceq$, strictly increasing along a path

$[\![_]\!]$**:** from atom/leaves to types, $[\![\mathtt{N}(a?B^+ : B^-)]\!] = ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$

Binary Decision Trees

(Ordered) BDTs $B ::= \mathtt{L}(l) \mid \mathtt{N}(a?B^+ : B^-)$

leaves: l, elements of some lattice

atoms: a, any object with total order $\preceq$, strictly increasing along a path

$[\![_]\!]$**:** from atom/leaves to types, $[\![\mathtt{N}(a?B^+ : B^-)]\!] = ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$

Example: the arrow DNF of the previous slide:

leaves: Booleans, with $[\![\bot]\!] = \mathtt{empty}$ and $[\![\top]\!] = \mathtt{any}$

atoms: $a_1 = \mathtt{int} \to \mathtt{int} \preceq a_2 = \mathtt{bool} \to \mathtt{bool} \preceq a_3 = \alpha \to \alpha$

Binary Decision Trees

(Ordered) BDTs $B ::= \mathtt{L}(l) \mid \mathtt{N}(a?B^+ : B^-)$

leaves: l, elements of some lattice

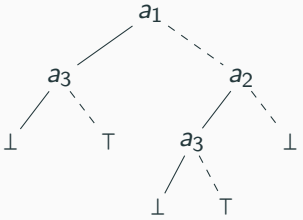
atoms: a, any object with total order $\preceq$, strictly increasing along a path

$[\![_]\!]$**:** from atom/leaves to types, $[\![\mathtt{N}(a?B^+ : B^-)]\!] = ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$

Example: the arrow DNF of the previous slide:

leaves: Booleans, with $[\![\bot]\!] = \mathtt{empty}$ and $[\![\top]\!] = \mathtt{any}$

atoms: $a_1 = \mathtt{int} \to \mathtt{int} \preceq a_2 = \mathtt{bool} \to \mathtt{bool} \preceq a_3 = \alpha \to \alpha$



Binary Decision Trees

(Ordered) BDTs $B \;::=\; \mathtt{L}(l) \mid \mathtt{N}(a?B^+ : B^-)$

leaves: l, elements of some lattice

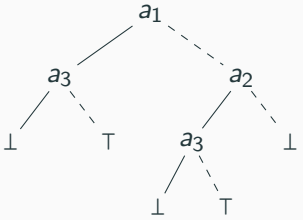
atoms: a, any object with total order $\preceq$, strictly increasing along a path

$[\![_]\!]$**:** from atom/leaves to types, $[\![\mathtt{N}(a?B^+ : B^-)]\!] = ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$

Example: the arrow DNF of the previous slide:

leaves: Booleans, with $[\![\bot]\!] = \mathtt{empty}$ and $[\![\top]\!] = \mathtt{any}$

atoms: $a_1 = \mathtt{int} \to \mathtt{int} \preceq a_2 = \mathtt{bool} \to \mathtt{bool} \preceq a_3 = \alpha \to \alpha$



As a type: $a_1 \wedge \neg a_3 \;\vee\; \neg a_1 \wedge a_2 \wedge \neg a_3$

Binary Decision Trees

(Ordered) BDTs $B ::= \mathtt{L}(l) \mid \mathtt{N}(a?B^+ : B^-)$

leaves: l, elements of some lattice

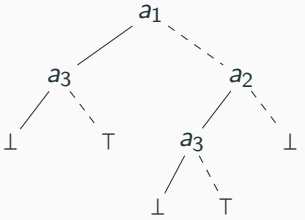
atoms: a, any object with total order $\preceq$, strictly increasing along a path

$[\![_]\!]$**:** from atom/leaves to types, $[\![\mathtt{N}(a?B^+ : B^-)]\!] = ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$

Example: the arrow DNF of the previous slide:

leaves: Booleans, with $[\![\bot]\!] = \mathtt{empty}$ and $[\![\top]\!] = \mathtt{any}$

atoms: $a_1 = \mathtt{int} \to \mathtt{int} \preceq a_2 = \mathtt{bool} \to \mathtt{bool} \preceq a_3 = \alpha \to \alpha$



As a type: $a_1 \wedge \neg a_3 \ \vee \ \neg a_1 \wedge a_2 \wedge \neg a_3$

$\neg$ negates the leaves *in place*

$\wedge, \vee, \setminus$ merge two BDTs following $\preceq$

Binary Decision Trees

(Ordered) BDTs $B ::= \mathtt{L}(l) \mid \mathtt{N}(a?B^+ : B^-)$

leaves: l, elements of some lattice

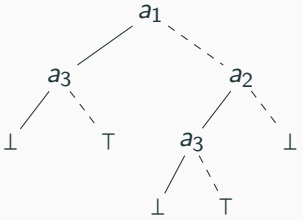
atoms: a, any object with total order $\preceq$, strictly increasing along a path

$[\![_]\!]$**:** from atom/leaves to types, $[\![\mathtt{N}(a?B^+ : B^-)]\!] = ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$

Example: the arrow DNF of the previous slide:

leaves: Booleans, with $[\![\bot]\!] = \mathtt{empty}$ and $[\![\top]\!] = \mathtt{any}$

atoms: $a_1 = \mathtt{int} \to \mathtt{int} \preceq a_2 = \mathtt{bool} \to \mathtt{bool} \preceq a_3 = \alpha \to \alpha$



As a type: $a_1 \wedge \neg a_3 \ \vee \ \neg a_1 \wedge a_2 \wedge \neg a_3$

$\neg$ negates the leaves *in place*

$\wedge, \vee, \setminus$ merge two BDTs following $\preceq$

Like ROBDDs [Bryant, 1986]: leaves range over a lattice, no maximal sharing

A layered representation

$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

A layered representation

$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

$\texttt{nil}$ $\quad (\texttt{int} \times X)$

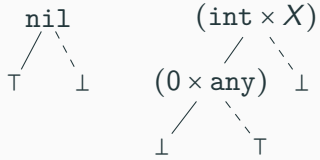
$(0 \times \texttt{any})$

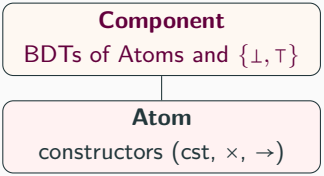
Atom

constructors (cst, $\times$, $\rightarrow$)

A layered representation

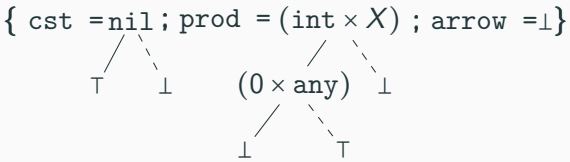
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

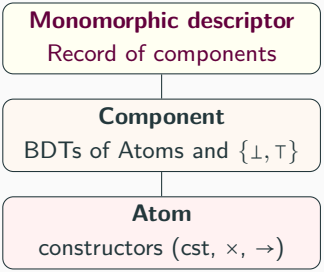




A layered representation

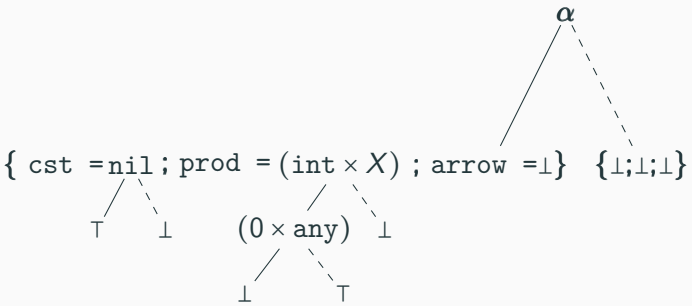
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

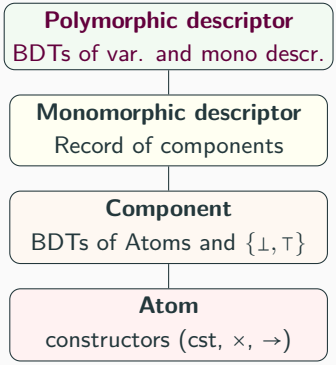




A layered representation

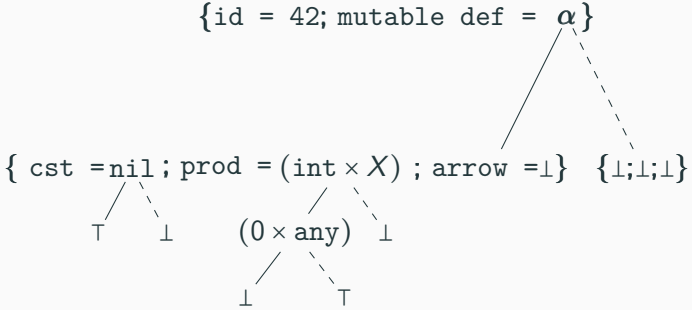
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

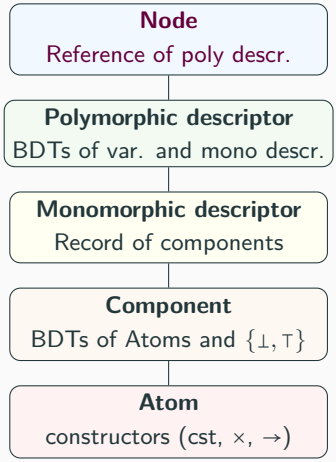




A layered representation

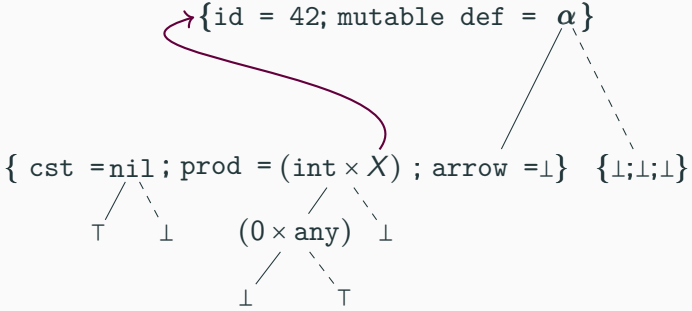
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

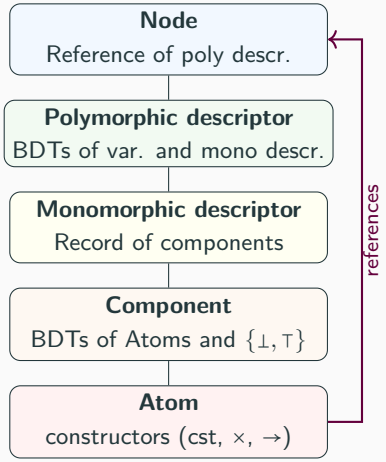




A layered representation

$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$





Subtyping and Tallying

Subtyping (emptiness): returns a Boolean. Base is $\bot$ or $\top$, combined with $\wedge$ and $\vee$.

Tallying: returns a set of sets of constraints (each inner set is a possible solution).

Base is $\{\{s \leq \alpha \leq t\}\}$, combined with $\sqcap$ and $\sqcup$

Subtyping and Tallying

Subtyping (emptiness): returns a Boolean. Base is $\bot$ or $\top$, combined with $\wedge$ and $\vee$.
Tallying: returns a set of sets of constraints (each inner set is a possible solution).

Base is $\{\{s \leq \alpha \leq t\}\}$, combined with $\sqcap$ and $\sqcup$

Layer	Subtyping	Tallying
Node	handles recursion/caching	
Poly. Descr.	ignored	single out each variable $\{\{s \leq \alpha \leq t\}\}$
Mono. Descr.	$\wedge$	$\sqcap$
Component	$\wedge$, $\vee$	$\sqcap$, $\sqcup$
Atom	$\bot$, $\top$ or recurse through Node	`Sat`, `Unsat` or recurse through Node

Recursion: each Node is kept in a cache and visited at most once.
The algorithms explore finitely (possibly exponentially) many nodes.

Optimization: Semantic simplification

Atoms may be semantically redundant without being syntactically redundant:

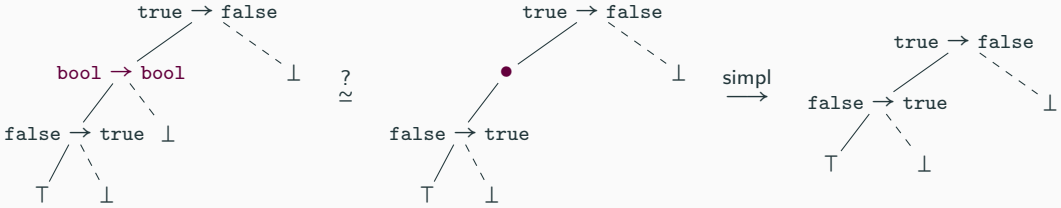
$$(\texttt{bool} \to \texttt{bool}) \wedge (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true}) \quad \simeq \quad (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true})$$

Optimization: Semantic simplification

Atoms may be semantically redundant without being syntactically redundant:

$$(\texttt{bool} \to \texttt{bool}) \wedge (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true}) \quad \simeq \quad (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true})$$

We define a semantic simplification $\mathrm{simpl}(B)$, traversing the BDT:

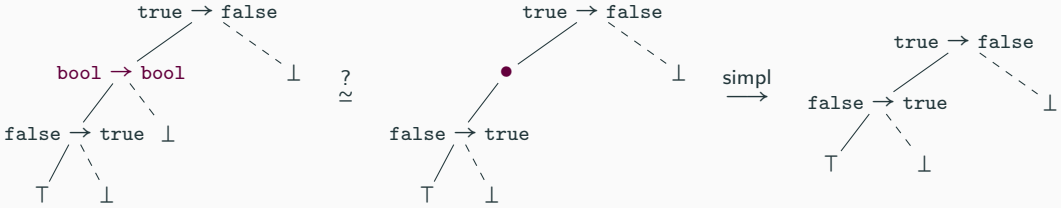


Optimization: Semantic simplification

Atoms may be semantically redundant without being syntactically redundant:

$$(\texttt{bool} \to \texttt{bool}) \wedge (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true}) \quad \simeq \quad (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true})$$

We define a semantic simplification $\mathrm{simpl}(B)$, traversing the BDT:



$\neg\,\mathrm{simpl}(B) = \mathrm{simpl}(\neg B)$

Negations stay simplified

$[\![B]\!] \simeq \texttt{empty} \iff \mathrm{simpl}(B) = \bot$

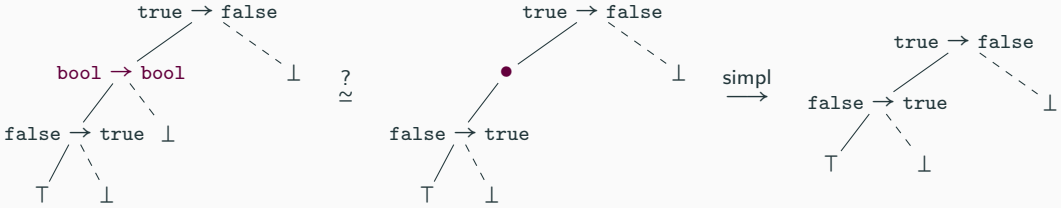
Emptiness check is $O(1)$

Optimization: Semantic simplification

Atoms may be semantically redundant without being syntactically redundant:

$$(\texttt{bool} \to \texttt{bool}) \wedge (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true}) \quad \simeq \quad (\texttt{true} \to \texttt{false}) \wedge (\texttt{false} \to \texttt{true})$$

We define a semantic simplification $\mathrm{simpl}(B)$, traversing the BDT:



$\neg\,\mathrm{simpl}(B) = \mathrm{simpl}(\neg B)$

Negations stay simplified

$[\![B]\!] \simeq \texttt{empty} \iff \mathrm{simpl}(B) = \bot$

Emptiness check is $O(1)$

$\Rightarrow$ Sstt simplifies systematically: after every $\wedge$, $\vee$, substitution and `build`.

Optimization: Subtyping and Tallying

Subtyping:

- Nodes use a semi-persistent cache to detect recursion and reuse computations
- Components use set-theoretic properties (e.g. pushing $\wedge$ below $\times$)

Optimization: Subtyping and Tallying

Subtyping:

- Nodes use a semi-persistent cache to detect recursion and reuse computations
- Components use set-theoretic properties (e.g. pushing $\wedge$ below $\times$)

Tallying: contrary to [Castagna et al., 2015] propagate constraints during generation:

- Eliminate unsatisfiable constraint sets as early as possible
- Avoid redundant solutions by only keeping "most general" sets

Optimization: Subtyping and Tallying

Subtyping:

- Nodes use a semi-persistent cache to detect recursion and reuse computations
- Components use set-theoretic properties (e.g. pushing $\wedge$ below $\times$)

Tallying: contrary to [Castagna et al., 2015] propagate constraints during generation:

- Eliminate unsatisfiable constraint sets as early as possible
- Avoid redundant solutions by only keeping "most general" sets

Everything adds up:

⇒ Subtyping cache makes simpl(_) faster

⇒ simpl(_) keeps types small and make subsequent subtyping free

⇒ Cheap subtyping makes tallying much faster
(cheaper to detect redundant and unsat. sets ⇒ generate fewer solutions)

Evaluation

Primer on set-theoretic types

Implementation

- **Evaluation**

How did we evaluate our library?

We used three corpora of MLsem code [Laurent and Vitek, 2026]:

Phase 1: Typecheck with MLsem, record every tallying problem in a JSON file

Phase 2: Build types from JSON, solve the tallying instances with various optimization disabled, and with CDuce

A: ML style	**B: Type narrowing**	**C: Mixed**
Iterators on trees and lists, pattern-matching	Type cases, overloaded functions	Functional, imperative, overloaded, ...
852 instances	5520 instances	2335 instances

Key takeaways

- Without semantic simplification, phase 1 does not terminate
  $\Rightarrow$ types grow too big, full of redundant atoms (esp. after substitutions)

Key takeaways

- Without semantic simplification, phase 1 does not terminate
  $\Rightarrow$ types grow too big, full of redundant atoms (esp. after substitutions)
- Hash-consing (making BDTs into BDDs with max. sharing) is not worth it
  $\Rightarrow$ maintaining the hash tables is too costly for almost no gain

Key takeaways

- Without semantic simplification, phase 1 does not terminate
  $\Rightarrow$ types grow too big, full of redundant atoms (esp. after substitutions)
- Hash-consing (making BDTs into BDDs with max. sharing) is not worth it
  $\Rightarrow$ maintaining the hash tables is too costly for almost no gain
- Without the smart cache, subtyping is between 10% and 50% slower

Key takeaways

- Without semantic simplification, phase 1 does not terminate
  $\Rightarrow$ types grow too big, full of redundant atoms (esp. after substitutions)
- Hash-consing (making BDTs into BDDs with max. sharing) is not worth it
  $\Rightarrow$ maintaining the hash tables is too costly for almost no gain
- Without the smart cache, subtyping is between 10% and 50% slower
- Without tallying optimizations, up to 68× slower and 10× number of (redundant) solutions (some problems don't terminate)

Key takeaways

- Without semantic simplification, phase 1 does not terminate
  $\Rightarrow$ types grow too big, full of redundant atoms (esp. after substitutions)
- Hash-consing (making BDTs into BDDs with max. sharing) is not worth it
  $\Rightarrow$ maintaining the hash tables is too costly for almost no gain
- Without the smart cache, subtyping is between 10% and 50% slower
- Without tallying optimizations, up to 68× slower and 10× number of (redundant) solutions (some problems don't terminate)
- Always faster than CDuce (between 20% and 50% faster, depending on corpus)

Conclusion

Sstt: an open, modular OCaml library for set-theoretic types.

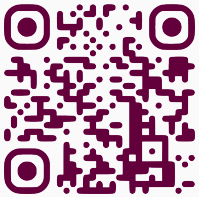
- BDTs + systematic semantic simplification: the key design decision
- A layered representation: constructors, connectives, type variables, recursion
- Revisited subtyping (smart caching) and tallying (constraint simplification)
- Consistently faster than CDuce, the historical implementation
- Much more in the paper, much, much more in the code!

`https://github.com/E-Sh4rk/sstt`









References i


Bryant, R. E. (1986).
Graph-based algorithms for Boolean function manipulation.
IEEE Transactions on Computers, C-35(8):677–691.


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).

References ii


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.


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.

References iii


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


Frisch, A., Castagna, G., and Benzaken, V. (2008).
Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types.
Journal of the ACM, 55(4):19:1–19:64.


Laurent, M. and Vitek, J. (2026).
Type inference for functional and imperative dynamic languages .
working paper or preprint.


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.