Designing a Set-Theoretic Type Library in OCaml

Mickaël Laurent, Kim Nguyễn

November, 2026

Sorbonne Université

Université Paris-Saclay





Summary

Primer on set-theoretic types

Type representation: a recursive layered structure

Type manipulations: implementation of caching

Primer on set-theoretic types

Primer on set-theoretic types

Type representation: a recursive layered structure

Type manipulations: implementation of caching

Typing dynamic languages

```
def even(x):
  if type(x) is int:
          return x % 2 == 0
  return x
```

Typing dynamic languages

```
def even(x):
  if type(x) is int:
          return x % 2 == 0
  return x
```

- any → any *correct, but imprecise*

Typing dynamic languages

```
def even(x):
  if type(x) is int:
          return x % 2 == 0
  return x
```

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

Typing dynamic languages

```
def even(x):
  if type(x) is int:
          return x % 2 == 0
  return x
```

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

Typing dynamic languages

```
def even(x):
  if type(x) is int:
          return x % 2 == 0
  return x
```

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

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

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

*co*inductively defined: regular + contractive $\Rightarrow$ 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]

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:

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

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:

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

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

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

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

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: $\forall\sigma.\ s\sigma \leq t\sigma \iff \forall\sigma.(s \setminus t)\sigma \leq \texttt{empty}$

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

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

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

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

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

- 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

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

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

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

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

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

Sstt: a library for set-theoretic types

Sstt: *Simple Set-Theoretic Types*, an OCaml library (~7k loc)

- Set connectives ($\wedge$, $\vee$, $\neg$), subtyping, substitutions, tallying, pretty-printing, ...
- Not tied to a specific language: used by MLsem, PySem, TypR
- Extensible through encodings (lists, strings, objects, ...)

Sstt: a library for set-theoretic types

Sstt: *Simple Set-Theoretic Types*, an OCaml library (~7k loc)

- Set connectives ($\wedge$, $\vee$, $\neg$), subtyping, substitutions, tallying, pretty-printing, ...
- Not tied to a specific language: used by MLsem, PySem, TypR
- Extensible through encodings (lists, strings, objects, ...)

```
open Sstt
let arrow a b = Descr.mk_arrow (a, b) |> Ty.mk_descr          (* a -> b *)
let int = Descr.mk_intervals Intervals.any |> Ty.mk_descr       (* int *)
let a = Ty.mk_var (Var.mk "a")                                   (* 'a *)
let f = Ty.cap (arrow int int) (arrow a a) (* (int -> int)&('a -> 'a) *)
let () = assert (Ty.leq f (arrow int Ty.any))     (* f <= (int -> any) *)
```

This talk: two design problems we faced, and how OCaml helped... or not.

Type representation: a recursive layered structure

Primer on set-theoretic types

Type representation: a recursive layered structure

Type manipulations: implementation of caching

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:

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

Disjunctive normal forms

Deciding $t_1 \leq t_2$ = deciding $t_1 \setminus t_2 \simeq \texttt{empty}$, 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:

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

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:

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

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

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

leaves: l, elements of some lattice implementing $\wedge$, $\vee$, $\neg$

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

Binary Decision Trees

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

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

leaves: l, elements of some lattice implementing $\wedge$, $\vee$, $\neg$

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

Example: $((\mathtt{int} \to \mathtt{int}) \wedge \neg(\alpha \to \alpha)) \vee (\neg(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \wedge \neg(\alpha \to \alpha))$

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

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

Binary Decision Trees

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

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

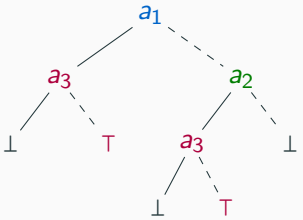
leaves: l, elements of some lattice implementing $\wedge$, $\vee$, $\neg$

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

Example: $((\mathtt{int} \to \mathtt{int}) \wedge \neg(\alpha \to \alpha)) \vee (\neg(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \wedge \neg(\alpha \to \alpha))$

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

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



Binary Decision Trees

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

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

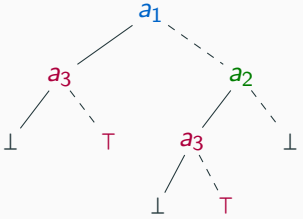
leaves: l, elements of some lattice implementing $\wedge$, $\vee$, $\neg$

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

Example: $((\mathtt{int} \to \mathtt{int}) \wedge \neg(\alpha \to \alpha)) \vee (\neg(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \wedge \neg(\alpha \to \alpha))$

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

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



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

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

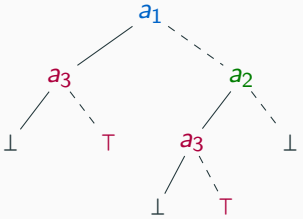
leaves: l, elements of some lattice implementing $\wedge$, $\vee$, $\neg$

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

Example: $((\mathtt{int} \to \mathtt{int}) \wedge \neg(\alpha \to \alpha)) \vee (\neg(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \wedge \neg(\alpha \to \alpha))$

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

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



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

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

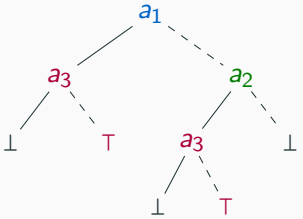
leaves: l, elements of some lattice implementing $\wedge$, $\vee$, $\neg$

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

Example: $((\mathtt{int} \to \mathtt{int}) \wedge \neg(\alpha \to \alpha)) \vee (\neg(\mathtt{int} \to \mathtt{int}) \wedge (\mathtt{bool} \to \mathtt{bool}) \wedge \neg(\alpha \to \alpha))$

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

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



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

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

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

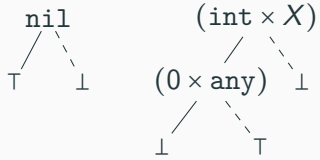
$(0 \times \texttt{any})$

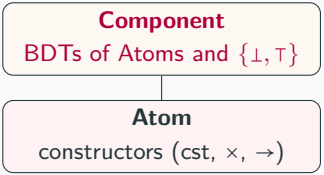
Atom

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

A layered representation

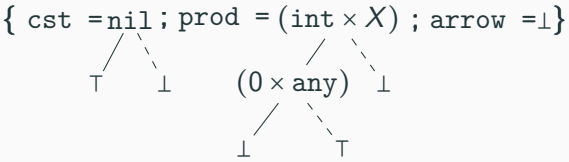
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

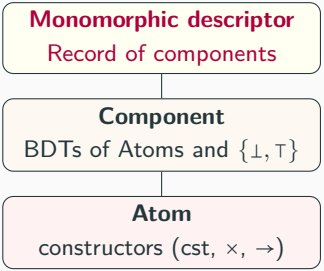




A layered representation

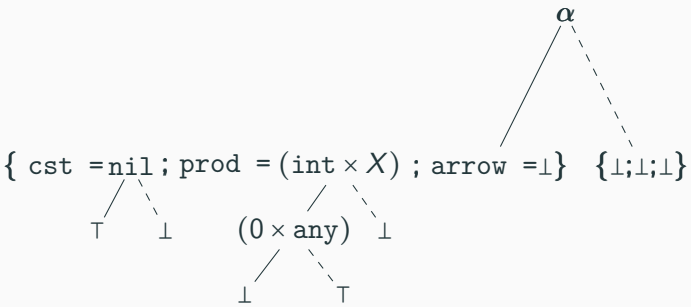
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

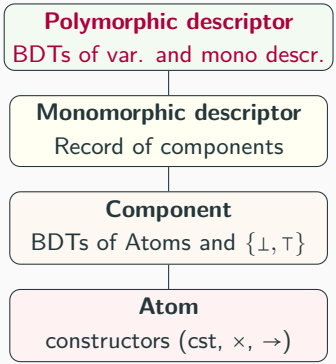




A layered representation

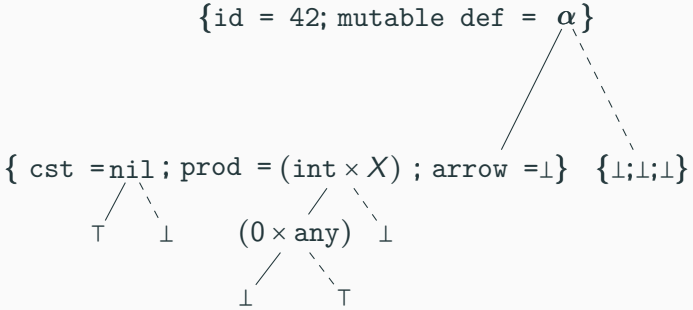
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

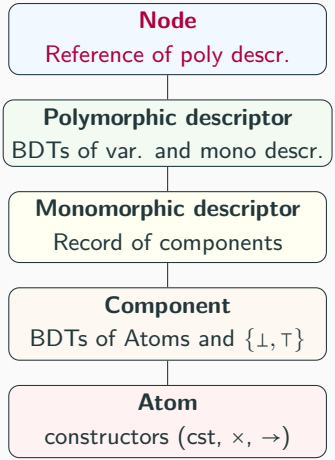




A layered representation

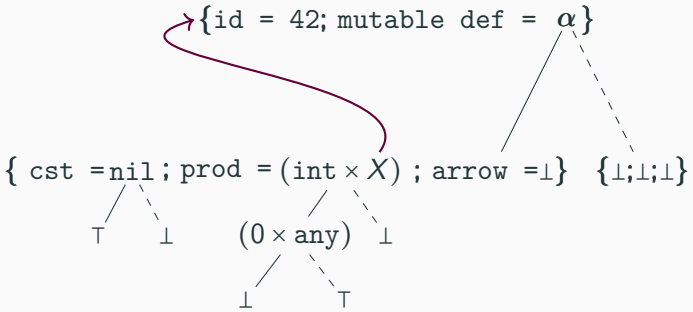
$$X = \alpha \wedge (\texttt{nil} \vee (\texttt{int} \times X) \setminus (0 \times \texttt{any}))$$

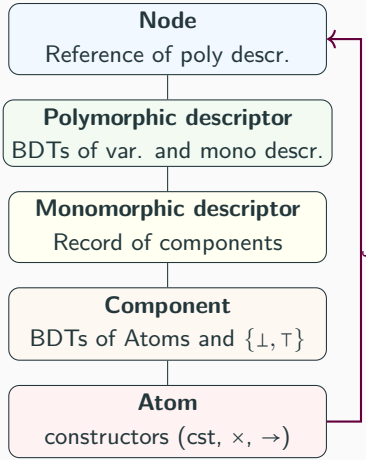




A layered representation

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





Every layer is a functor

Every layer is parameterized by the Node module it will eventually point to:

```
(* components/arrows.ml *)
module Make( N:Node ) = struct
  module Atom = struct
    type t = N.t * N.t
    ...
  end
  module Bdd = Bdd.Make(Atom)(BoolLeaf)
  type t = Bdd.t
  let rec psi t1 t2 ps =
    N.is_empty t1 || N.is_empty t2 || ...
  let is_empty t = ...
end
```

```
(* descr.ml *)
module Make( N:Node ) = struct
  module Intervals = Intervals
  module Arrows = Arrows.Make( N )
  module Tuples = Tuples.Make( N )
  ...
  type t = { intervals : Intervals.t ;
             arrows : Arrows.t ;
             tuples : Tuples.t ; ... }
end

(* vdescr.ml *)
module Make( N:Node ) = struct
  module Descr = Descr.Make( N )
  include Polymorphic.Make(Var)(Descr)
end
```

Recursion closed at the `Node` module

The `Node` module instantiates the inner layers with itself:

```
(* node.ml *)
module rec Node : Node = struct
  type t = { id : int ; mutable def : VDescr.t }
  let mk () = { id = fresh_id () ; def = VDescr.empty }
  let define n d = n.def <- d
  let is_empty n = VDescr.is_empty n.def
  ...
end
and VDescr : VDescr with type node = Node.t = Vdescr.Make( Node )
```

Recursion closed at the `Node` module

The `Node` module instantiates the inner layers with itself:

```
(* node.ml *)
module rec Node : Node = struct
  type t = { id : int ; mutable def : VDescr.t }
  let mk () = { id = fresh_id () ; def = VDescr.empty }
  let define n d = n.def <- d
  let is_empty n = VDescr.is_empty n.def
  ...
end
and VDescr : VDescr with type node = Node.t = Vdescr.Make( Node )
```

- `Vdescr.Make(Node)` $\rightsquigarrow$ `Descr.Make(Node)` $\rightsquigarrow$ `Arrows.Make(Node)`, ...
- The knot is tied once, in a single place

Recursion closed at the `Node` module

The `Node` module instantiates the inner layers with itself:

```
(* node.ml *)
module rec Node : Node = struct
  type t = { id : int ; mutable def : VDescr.t }
  let mk () = { id = fresh_id () ; def = VDescr.empty }
  let define n d = n.def <- d
  let is_empty n = VDescr.is_empty n.def
  ...
end
and VDescr : VDescr with type node = Node.t = Vdescr.Make( Node )
```

- `Vdescr.Make(Node)` $\rightsquigarrow$ `Descr.Make(Node)` $\rightsquigarrow$ `Arrows.Make(Node)`, ...
- The knot is tied once, in a single place

One file per component, independent and testable. New component = one functor + one field.

In comparison: CDuce

CDuce defines everything as mutually recursive `modules`, in `types.ml` (~3 300 loc):

```
module rec Descr : sig
  include Custom.T with type t = (VarAtoms.t, VarIntervals.t, VarChars.t,
                                  VarTimes.t, VarXml.t, VarArrow.t, ...) descr_
end = struct ... end
and Node : sig
  type t = { id : int; cu: Compunit.t; mutable descr : Descr.t }
end = struct ... end
and VarAtoms : VarTypeSig with type Atom.t = Atoms.t ... = struct ... end
and VarIntervals : VarTypeSig with type Atom.t = Intervals.t ... = struct ... end
and Pair : Bool.S with type elem = (Node.t * Node.t) = Bool.Make(...)
and VarTimes : VarTypeSig with module Atom = Pair ... = struct ... end
and VarArrow : VarTypeSig with module Atom = Pair ... = struct ... end
and Rec : Bool.S with type elem = bool * Node.t Ident.label_map = ...
...
```

In comparison: CDuce

CDuce defines everything as mutually recursive `modules`, in `types.ml` (~3 300 loc):

```
module rec Descr : sig
  include Custom.T with type t = (VarAtoms.t, VarIntervals.t, VarChars.t,
                                  VarTimes.t, VarXml.t, VarArrow.t, ...) descr_
end = struct ... end
and Node : sig
  type t = { id : int; cu: Compunit.t; mutable descr : Descr.t }
end = struct ... end
and VarAtoms : VarTypeSig with type Atom.t = Atoms.t ... = struct ... end
and VarIntervals : VarTypeSig with type Atom.t = Intervals.t ... = struct ... end
and Pair : Bool.S with type elem = (Node.t * Node.t) = Bool.Make(...)
and VarTimes : VarTypeSig with module Atom = Pair ... = struct ... end
and VarArrow : VarTypeSig with module Atom = Pair ... = struct ... end
and Rec : Bool.S with type elem = bool * Node.t Ident.label_map = ...
...
```

- Subtyping, products, records, arrows... are then all implemented in `types.ml`
- Hard to extend, hard to reuse

✗ Closing the recursion

Naive attempt:

```
module rec Node : Node = struct
  type t = { id : int ; mutable def : VDescr.t }
  let any = { id = fresh_id () ; def = VDescr.any }
  let is_empty n = VDescr.is_empty n.def
  ...
end
and VDescr : VDescr with type node = Node.t = Vdescr.Make(Node)
```

✗ Closing the recursion

Naive attempt:

```
module rec Node : Node = struct
  type t = { id : int ; mutable def : VDescr.t }
  let any = { id = fresh_id () ; def = VDescr.any }
  let is_empty n = VDescr.is_empty n.def
  ...
end
and VDescr : VDescr with type node = Node.t = Vdescr.Make(Node)
```

```
Error: Cannot safely evaluate the definition of the following cycle
       of recursively-defined modules: Node -> VDescr -> Node.
       There are no safe modules in this cycle (see manual section 12.2).
  Module "Node" defines an unsafe value, "any" .
  Module "VDescr" defines an unsafe value, "any" .
```

✗ Closing the recursion

Naive attempt:

```
module rec Node : Node = struct
  type t = { id : int ; mutable def : VDescr.t }
  let any = { id = fresh_id () ; def = VDescr.any }
  let is_empty n = VDescr.is_empty n.def
  ...
end
and VDescr : VDescr with type node = Node.t = Vdescr.Make(Node)
```

```
Error: Cannot safely evaluate the definition of the following cycle
       of recursively-defined modules: Node -> VDescr -> Node.
       There are no safe modules in this cycle (see manual section 12.2).
  Module "Node" defines an unsafe value, "any" .
  Module "VDescr" defines an unsafe value, "any" .
```

- A recursive module is safe if it only contains functions
- Every layer has constants `any` and `empty`… and we want them

The PreNode trick

```
module rec TyRef : sig
  type t = {id:int ; mutable def:VDescr.t option ...}
end = TyRef (* type-only: its own definition *)

and AnyEmpty : sig
  type t = TyRef.t
  val any : t   val empty : t
  val init : VDescr.t -> VDescr.t -> unit
end = struct ... end (* any/empty: undefined *)

and Node : Node with type t = AnyEmpty.t ... = struct
  include PreNode
  let any = AnyEmpty.any  let empty = AnyEmpty.empty
end

and VDescr : VDescr' with ... = Vdescr.Make(Node)
and PreNode : PreNode with ... = struct
  ... (* only functions: our safe module *)
end
let () = AnyEmpty.init VDescr.empty VDescr.any
```

1. PreNode has no constant: it is safe, initialized with stubs

The PreNode trick

```
module rec TyRef : sig
  type t = {id:int ; mutable def:VDescr.t option ...}
end = TyRef (* type-only: its own definition *)

and AnyEmpty : sig
  type t = TyRef.t
  val any : t   val empty : t
  val init : VDescr.t -> VDescr.t -> unit
end = struct ... end (* any/empty: undefined *)

and Node : Node with type t = AnyEmpty.t ... = struct
  include PreNode
  let any = AnyEmpty.any  let empty = AnyEmpty.empty
end

and VDescr : VDescr' with ... = Vdescr.Make(Node)
and PreNode : PreNode with ... = struct
  ... (* only functions: our safe module *)
end
let () = AnyEmpty.init VDescr.empty VDescr.any
```

1. PreNode has no constant: it is safe, initialized with stubs
2. any/empty are allocated without their definition

The PreNode trick

```
module rec TyRef : sig
  type t = {id:int ; mutable def:VDescr.t option ...}
end = TyRef (* type-only: its own definition *)

and AnyEmpty : sig
  type t = TyRef.t
  val any : t   val empty : t
  val init : VDescr.t -> VDescr.t -> unit
end = struct ... end (* any/empty: undefined *)

and Node : Node with type t = AnyEmpty.t ... = struct
  include PreNode
  let any = AnyEmpty.any  let empty = AnyEmpty.empty
end

and VDescr : VDescr' with ... = Vdescr.Make(Node)
and PreNode : PreNode with ... = struct
  ... (* only functions: our safe module *)
end
let () = AnyEmpty.init VDescr.empty VDescr.any
```

1. PreNode has no constant: it is safe, initialized with stubs
2. any/empty are allocated without their definition
3. Node = PreNode + these constants ⇒ it is safe

The PreNode trick

```
module rec TyRef : sig
  type t = {id:int ; mutable def:VDescr.t option ...}
end = TyRef (* type-only: its own definition *)

and AnyEmpty : sig
  type t = TyRef.t
  val any : t   val empty : t
  val init : VDescr.t -> VDescr.t -> unit
end = struct ... end (* any/empty: undefined *)

and Node : Node with type t = AnyEmpty.t ... = struct
  include PreNode
  let any = AnyEmpty.any  let empty = AnyEmpty.empty
end

and VDescr : VDescr' with ... = Vdescr.Make(Node)
and PreNode : PreNode with ... = struct
  ... (* only functions: our safe module *)
end
let () = AnyEmpty.init VDescr.empty VDescr.any
```

1. `PreNode` has no constant: it is safe, initialized with stubs
2. `any`/`empty` are allocated without their definition
3. `Node` = `PreNode` + these constants ⇒ it is safe
4. `Vdescr.Make(Node)` is evaluated

The PreNode trick

```
module rec TyRef : sig
  type t = {id:int ; mutable def:VDescr.t option ...}
end = TyRef (* type-only: its own definition *)

and AnyEmpty : sig
  type t = TyRef.t
  val any : t   val empty : t
  val init : VDescr.t -> VDescr.t -> unit
end = struct ... end (* any/empty: undefined *)

and Node : Node with type t = AnyEmpty.t ... = struct
  include PreNode
  let any = AnyEmpty.any  let empty = AnyEmpty.empty
end

and VDescr : VDescr' with ... = Vdescr.Make(Node)
and PreNode : PreNode with ... = struct
  ... (* only functions: our safe module *)
end
let () = AnyEmpty.init VDescr.empty VDescr.any
```

1. PreNode has no constant: it is safe, initialized with stubs
2. any/empty are allocated without their definition
3. Node = PreNode + these constants ⇒ it is safe
4. Vdescr.Make(Node) is evaluated
5. The definitions of any/empty are patched at the end

The PreNode trick

```
module rec TyRef : sig
  type t = {id:int ; mutable def:VDescr.t option ...}
end = TyRef (* type-only: its own definition *)

and AnyEmpty : sig
  type t = TyRef.t
  val any : t   val empty : t
  val init : VDescr.t -> VDescr.t -> unit
end = struct ... end (* any/empty: undefined *)

and Node : Node with type t = AnyEmpty.t ... = struct
  include PreNode
  let any = AnyEmpty.any  let empty = AnyEmpty.empty
end

and VDescr : VDescr' with ... = Vdescr.Make(Node)
and PreNode : PreNode with ... = struct
  ... (* only functions: our safe module *)
end
let () = AnyEmpty.init VDescr.empty VDescr.any
```

1. PreNode has no constant: it is safe, initialized with stubs
2. any/empty are allocated without their definition
3. Node = PreNode + these constants ⇒ it is safe
4. Vdescr.Make(Node) is evaluated
5. The definitions of any/empty are patched at the end

Invariant: no code in the cycle may dereference Node.any at toplevel.

✗ Internal vs. exported signatures

To close the recursion, each module of the cycle needs a strong internal signature...
... but not the one we want to export:

- internally: `simplify`, `direct_nodes`, `map_nodes`, `define`, caches...
- externally: a clean API (`Ty`, `VDescr`, `Descr`, ...)

✗ Internal vs. exported signatures

To close the recursion, each module of the cycle needs a strong internal signature…
… but not the one we want to export:

- internally: `simplify`, `direct_nodes`, `map_nodes`, `define`, caches…
- externally: a clean API (`Ty`, `VDescr`, `Descr`, …)

Trade-off:

- specify the full exported signatures in a file `sigs.ml` (1 300 lines), with many `with type ... := ...` due to their recursive nature
- re-specify only a few internal signatures (`PreNode`, `Node`, etc.) needed for the recursive definition, let the rest be inferred from the functors

✓ Local imperativity

Building a recursive type = creating a node first, defining it later:

```
type node = { id : int ; mutable def : VDescr.t option }

(* List(a) = nil | (a, List(a)) *)
let x = { id = fresh_id () ; def = None } in  (* fresh node, undefined *)
let d = VDescr.cup nil (VDescr.mk_tuple [a ; x]) in   (* refers to x *)
x.def <- Some d                                          (* tie the knot *)
```

✓ Local imperativity

Building a recursive type = creating a node first, defining it later:

```
type node = { id : int ; mutable def : VDescr.t option }

(* List(a) = nil | (a, List(a)) *)
let x = { id = fresh_id () ; def = None } in  (* fresh node, undefined *)
let d = VDescr.cup nil (VDescr.mk_tuple [a ; x]) in   (* refers to x *)
x.def <- Some d                                          (* tie the knot *)
```

- Mutation is local: once built, a node is (morally) immutable

✓ Local imperativity

Building a recursive type = creating a node first, defining it later:

```
type node = { id : int ; mutable def : VDescr.t option }

(* List(a) = nil | (a, List(a)) *)
let x = { id = fresh_id () ; def = None } in  (* fresh node, undefined *)
let d = VDescr.cup nil (VDescr.mk_tuple [a ; x]) in   (* refers to x *)
x.def <- Some d                                          (* tie the knot *)
```

- Mutation is local: once built, a node is (morally) immutable
- Hidden to the users: we only expose a builder from a system of equations

```
Ty.of_eqs : (Var.t * Ty.t) list -> Ty.t list
```

✓ Local imperativity

Building a recursive type = creating a node first, defining it later:

```
type node = { id : int ; mutable def : VDescr.t option }

(* List(a) = nil | (a, List(a)) *)
let x = { id = fresh_id () ; def = None } in  (* fresh node, undefined *)
let d = VDescr.cup nil (VDescr.mk_tuple [a ; x]) in   (* refers to x *)
x.def <- Some d                                          (* tie the knot *)
```

- Mutation is local: once built, a node is (morally) immutable
- Hidden to the users: we only expose a builder from a system of equations

  `Ty.of_eqs : (Var.t * Ty.t) list -> Ty.t list`
- Nodes also cache derived information in mutable fields: `simplified`, `dependencies`…

✓ Local imperativity

Building a recursive type = creating a node first, defining it later:

```
type node = { id : int ; mutable def : VDescr.t option }

(* List(a) = nil | (a, List(a)) *)
let x = { id = fresh_id () ; def = None } in  (* fresh node, undefined *)
let d = VDescr.cup nil (VDescr.mk_tuple [a ; x]) in   (* refers to x *)
x.def <- Some d                                          (* tie the knot *)
```

- Mutation is local: once built, a node is (morally) immutable
- Hidden to the users: we only expose a builder from a system of equations

  `Ty.of_eqs : (Var.t * Ty.t) list -> Ty.t list`
- Nodes also cache derived information in mutable fields: `simplified`, `dependencies`…

In a pure language, *e.g.* Etylizer (Erlang) [Schimpf et al., 2023]: recursive types go through named references into a global store.

✓ Local imperativity

Building a recursive type = creating a node first, defining it later:

```
type node = { id : int ; mutable def : VDescr.t option }

(* List(a) = nil | (a, List(a)) *)
let x = { id = fresh_id () ; def = None } in  (* fresh node, undefined *)
let d = VDescr.cup nil (VDescr.mk_tuple [a ; x]) in   (* refers to x *)
x.def <- Some d                                          (* tie the knot *)
```

- Mutation is local: once built, a node is (morally) immutable
- Hidden to the users: we only expose a builder from a system of equations

  `Ty.of_eqs : (Var.t * Ty.t) list -> Ty.t list`
- Nodes also cache derived information in mutable fields: `simplified`, `dependencies`…

In a pure language, *e.g.* Etylizer (Erlang) [Schimpf et al., 2023]: recursive types go through named references into a global store.

✗ The price: Sstt is not thread-safe for now (single thread, single domain).

Insight: whose problem is it?

The cycle $N \to$ `VDescr` $\to$ `Descr` $\to$ `Arrows` $\to N$ is only here because OCaml compiles modules one at a time, in dependency order.

Insight: whose problem is it?

The cycle $N \to$ `VDescr` $\to$ `Descr` $\to$ `Arrows` $\to N$ is only here because OCaml compiles modules one at a time, in dependency order.

- With late binding (Python, JavaScript, Erlang, …): `arrows` just calls `node.is_empty`, resolved at runtime

  no functor, no `module rec`, no safety check… and no static guarantee

Insight: whose problem is it?

The cycle $N \to$ `VDescr` $\to$ `Descr` $\to$ `Arrows` $\to N$ is only here because OCaml compiles modules one at a time, in dependency order.

- With late binding (Python, JavaScript, Erlang, …): `arrows` just calls `node.is_empty`, resolved at runtime
  no functor, no `module rec`, no safety check… and no static guarantee
- In Rust: a crate is compiled as a whole,
  mutually recursive modules across files are fine

Insight: whose problem is it?

The cycle $N \to$ `VDescr` $\to$ `Descr` $\to$ `Arrows` $\to N$ is only here because OCaml compiles modules one at a time, in dependency order.

- With late binding (Python, JavaScript, Erlang, …): `arrows` just calls `node.is_empty`, resolved at runtime
  no functor, no `module rec`, no safety check… and no static guarantee
- In Rust: a crate is compiled as a whole,
  mutually recursive modules across files are fine
- In OCaml: functors give modularity, but closing the knot is on us

Insight: whose problem is it?

The cycle $N \to$ `VDescr` $\to$ `Descr` $\to$ `Arrows` $\to N$ is only here because OCaml compiles modules one at a time, in dependency order.

- With late binding (Python, JavaScript, Erlang, …): `arrows` just calls `node.is_empty`, resolved at runtime
  no functor, no `module rec`, no safety check… and no static guarantee
- In Rust: a crate is compiled as a whole,
  mutually recursive modules across files are fine
- In OCaml: functors give modularity, but closing the knot is on us

Recursive modules: powerful, but still a sharp tool in OCaml.

Type manipulations: implementation of caching

Primer on set-theoretic types

Type representation: a recursive layered structure

Type manipulations: implementation of caching

Subtyping and Tallying

$s \leq t \iff s \setminus t \leq \texttt{empty}$: emptiness is decided on the DNF, component by component, by an inductive relation:

- Base cases: constants, `empty`, `any` — *decided directly*
- Inductive cases: products and arrows — *recurse on their sub-terms*

$$\bigwedge_{i \in P}(s_i \times t_i) \wedge \bigwedge_{j \in N} \neg(s'_j \times t'_j) \leq \texttt{empty} \iff \forall N' \subseteq N.\ \bigwedge_P s_i \leq \bigvee_{N'} s'_j \text{ or } \bigwedge_P t_i \leq \bigvee_{N \setminus N'} t'_j$$

Subtyping and Tallying

$s \leq t \iff s \setminus t \leq \texttt{empty}$: emptiness is decided on the DNF, component by component, by an inductive relation:

- Base cases: constants, `empty`, `any` — *decided directly*
- Inductive cases: products and arrows — *recurse on their sub-terms*

$\bigwedge_{i \in P}(s_i \times t_i) \wedge \bigwedge_{j \in N} \neg(s'_j \times t'_j) \leq \texttt{empty} \iff \forall N' \subseteq N.\ \bigwedge_P s_i \leq \bigvee_{N'} s'_j$ or $\bigwedge_P t_i \leq \bigvee_{N \setminus N'} t'_j$

Tallying follows the same structure, but returns a set of sets of constraints:

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

Is $\texttt{List(int)} \leq \texttt{List(any)}$, with $X = \texttt{nil} \vee (\texttt{int} \times X)$ and $Y = \texttt{nil} \vee (\texttt{any} \times Y)$?

Recursion

Is $\texttt{List(int)} \leq \texttt{List(any)}$, with $X = \texttt{nil} \vee (\texttt{int} \times X)$ and $Y = \texttt{nil} \vee (\texttt{any} \times Y)$?

$$X \setminus Y \leq \texttt{empty} \iff \texttt{nil} \setminus \texttt{nil} \leq \texttt{empty} \checkmark \text{ and } (\texttt{int} \times X) \leq (\texttt{any} \times Y)$$

Recursion

Is $\texttt{List(int)} \leq \texttt{List(any)}$, with $X = \texttt{nil} \vee (\texttt{int} \times X)$ and $Y = \texttt{nil} \vee (\texttt{any} \times Y)$?

$$X \setminus Y \leq \texttt{empty} \iff \texttt{nil} \setminus \texttt{nil} \leq \texttt{empty} \ \checkmark \ \text{and} \ (\texttt{int} \times X) \leq (\texttt{any} \times Y)$$

$$(\texttt{int} \times X) \setminus (\texttt{any} \times Y) \leq \texttt{empty} \iff (\texttt{int} \leq \texttt{empty} \text{ or } X \leq Y) \ \text{and} \ (\texttt{int} \leq \texttt{any} \text{ or } X \leq \texttt{empty})$$

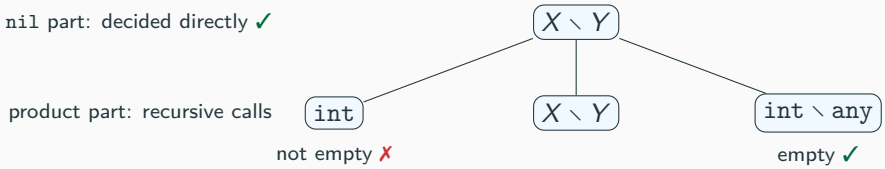
Recursion

Is $\texttt{List(int)} \leq \texttt{List(any)}$, with $X = \texttt{nil} \vee (\texttt{int} \times X)$ and $Y = \texttt{nil} \vee (\texttt{any} \times Y)$?

$$X \setminus Y \leq \texttt{empty} \iff \texttt{nil} \setminus \texttt{nil} \leq \texttt{empty} \checkmark \text{ and } (\texttt{int} \times X) \leq (\texttt{any} \times Y)$$

$$(\texttt{int} \times X) \setminus (\texttt{any} \times Y) \leq \texttt{empty} \iff (\texttt{int} \leq \texttt{empty} \text{ or } X \leq Y) \text{ and } (\texttt{int} \leq \texttt{any} \text{ or } X \leq \texttt{empty})$$

Nodes explored for emptiness:



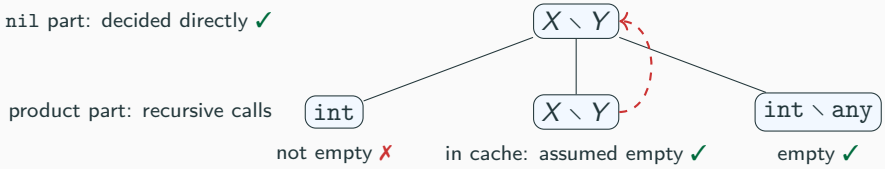
Recursion

Is $\texttt{List(int)} \leq \texttt{List(any)}$, with $X = \texttt{nil} \vee (\texttt{int} \times X)$ and $Y = \texttt{nil} \vee (\texttt{any} \times Y)$?

$$X \setminus Y \leq \texttt{empty} \iff \texttt{nil} \setminus \texttt{nil} \leq \texttt{empty} \checkmark \text{ and } (\texttt{int} \times X) \leq (\texttt{any} \times Y)$$

$$(\texttt{int} \times X) \setminus (\texttt{any} \times Y) \leq \texttt{empty} \iff (\texttt{int} \leq \texttt{empty} \text{ or } X \leq Y) \text{ and } (\texttt{int} \leq \texttt{any} \text{ or } X \leq \texttt{empty})$$

Nodes explored for emptiness:



Recursion: each Node is kept in a cache and visited at most once.

The algorithms explore finitely (possibly exponentially) many nodes.

Why a cache?

Operations on recursive types must remember what they have already seen:

- Subtyping, tallying: a node met during its own traversal is assumed empty
- Simplification, substitution, …: must not create infinitely many nodes
  $\Rightarrow$ memoize node construction ($\texttt{VDescr.t} \mapsto N$) and binary operations ($\wedge$, $\vee$, $\setminus$)
- Optimisation: reuse previous computations when possible to improve performance

Why a cache?

Operations on recursive types must remember what they have already seen:

- Subtyping, tallying: a node met during its own traversal is assumed empty
- Simplification, substitution, …: must not create infinitely many nodes
  $\Rightarrow$ memoize node construction ($\texttt{VDescr.t} \mapsto N$) and binary operations ($\wedge$, $\vee$, $\setminus$)
- Optimisation: reuse previous computations when possible to improve performance

Question: where does the cache live, and for how long?

Where does the cache live?

A global cache? This is what CDuce does:

```
let memo = DescrHash.create 8191
```

✗ During type inference, many (and big) types are generated: memory keeps growing.

Where does the cache live?

A global cache? This is what CDuce does:

```
let memo = DescrHash.create 8191
```

✗ During type inference, many (and big) types are generated: memory keeps growing.

A global weak hash table?

✗ Entries may disappear in the middle of an operation: termination is lost.

Where does the cache live?

A global cache? This is what CDuce does:

```
let memo = DescrHash.create 8191
```

✗ During type inference, many (and big) types are generated: memory keeps growing.

A global weak hash table?

✗ Entries may disappear in the middle of an operation: termination is lost.

A cache per operation?

✓ Safe, bounded lifetime… but the right scope depends on the user:

- one subtyping test? one tallying instance? a whole top-level definition?

Where does the cache live?

A global cache? This is what CDuce does:

```
let memo = DescrHash.create 8191
```

✗ During type inference, many (and big) types are generated: memory keeps growing.

A global weak hash table?

✗ Entries may disappear in the middle of an operation: termination is lost.

A cache per operation?

✓ Safe, bounded lifetime… but the right scope depends on the user:

- one subtyping test? one tallying instance? a whole top-level definition?

$\Rightarrow$ The user should decide the scope of the cache.

✗ Solution 1: explicit parameter

The recursion goes through every layer:

N.`is_empty` → `VDescr` → `Descr` → `Arrows.psi` → N.`is_empty` → ...

✗ Solution 1: explicit parameter

The recursion goes through every layer:

N.`is_empty` → `VDescr` → `Descr` → `Arrows.psi` → *N*.`is_empty` → ...

Solution 1: an explicit `cache` parameter, everywhere:

```
module type Node = sig
  type t
  val is_empty : cache -> t -> bool
  val leq      : cache -> t -> t -> bool
  val cap      : cache -> t -> t -> t
  ...
end
(* and in every component: *)
let rec psi cache t1 t2 ps =
  N.is_empty cache t1 || N.is_empty cache t2 || ...
```

✗ Solution 1: explicit parameter

The recursion goes through every layer:

N.`is_empty` → `VDescr` → `Descr` → `Arrows.psi` → N.`is_empty` → ...

Solution 1: an explicit `cache` parameter, everywhere:

```
module type Node = sig
  type t
  val is_empty : cache -> t -> bool
  val leq      : cache -> t -> t -> bool
  val cap      : cache -> t -> t -> t
  ...
end
(* and in every component: *)
let rec psi cache t1 t2 ps =
  N.is_empty cache t1 || N.is_empty cache t2 || ...
```

- Pollutes every signature (including the public API)
- Components that do not care about it (`Intervals`, ...) must still pass it along

✓ Solution 2: algebraic effects

```
type _ Effect.t += GetCache : cache t

let get_cache () = perform GetCache
(* Re-entrant: the topmost handler wins *)
let with_shared_cache f t =
  let cache = ref None in
  try f t with
  | effect GetCache, k ->
    match !cache with
    | Some c -> continue k c
    | None ->
      let c = try get_cache () (* outer handler? *)
        with Unhandled _ -> create_cache () in
      cache := Some c ; continue k c
```

- We define an effect `GetCache`

✓ Solution 2: algebraic effects

```
type _ Effect.t += GetCache : cache t

let get_cache () = perform GetCache
(* Re-entrant: the topmost handler wins *)
let with_shared_cache f t =
  let cache = ref None in
  try f t with
  | effect GetCache, k ->
    match !cache with
    | Some c -> continue k c
    | None ->
      let c = try get_cache () (* outer handler? *)
        with Unhandled _ -> create_cache () in
      cache := Some c ; continue k c
```

- We define an effect `GetCache`
- The cache is fetched where needed, not passed around

✓ Solution 2: algebraic effects

```
type _ Effect.t += GetCache : cache t

let get_cache () = perform GetCache
(* Re-entrant: the topmost handler wins *)
let with_shared_cache f t =
  let cache = ref None in
  try f t with
  | effect GetCache, k ->
    match !cache with
    | Some c -> continue k c
    | None ->
      let c = try get_cache () (* outer handler? *)
        with Unhandled _ -> create_cache () in
      cache := Some c ; continue k c
```

- We define an effect `GetCache`
- The cache is fetched where needed, not passed around
- `with_shared_cache f x` installs a handler, providing a shared cache for `f x`

✓ Solution 2: algebraic effects

```
type _ Effect.t += GetCache : cache t

let get_cache () = perform GetCache
(* Re-entrant: the topmost handler wins *)
let with_shared_cache f t =
  let cache = ref None in
  try f t with
  | effect GetCache, k ->
    match !cache with
    | Some c -> continue k c
    | None ->
      let c = try get_cache () (* outer handler? *)
        with Unhandled _ -> create_cache () in
      cache := Some c ; continue k c
```

- We define an effect `GetCache`
- The cache is fetched where needed, not passed around
- `with_shared_cache f x` installs a handler, providing a shared cache for `f x`
- If a handler already exists, `with_shared_cache` reuses its cache ⇒ nested calls share the outermost cache

✓ Solution 2: algebraic effects

```
type _ Effect.t += GetCache : cache t

let get_cache () = perform GetCache
(* Re-entrant: the topmost handler wins *)
let with_shared_cache f t =
  let cache = ref None in
  try f t with
  | effect GetCache, k ->
    match !cache with
    | Some c -> continue k c
    | None ->
      let c = try get_cache () (* outer handler? *)
        with Unhandled _ -> create_cache () in
      cache := Some c ; continue k c
```

✓ The cache crosses all the functor layers without appearing in any signature.

- We define an effect `GetCache`
- The cache is fetched where needed, not passed around
- `with_shared_cache f x` installs a handler, providing a shared cache for `f x`
- If a handler already exists, `with_shared_cache` reuses its cache ⇒ nested calls share the outermost cache

✓ Solution 2: algebraic effects

```
type _ Effect.t += GetCache : cache t

let get_cache () = perform GetCache
(* Re-entrant: the topmost handler wins *)
let with_shared_cache f t =
  let cache = ref None in
  try f t with
  | effect GetCache, k ->
    match !cache with
    | Some c -> continue k c
    | None ->
      let c = try get_cache () (* outer handler? *)
        with Unhandled _ -> create_cache () in
      cache := Some c ; continue k c
```

- We define an effect `GetCache`
- The cache is fetched where needed, not passed around
- `with_shared_cache f x` installs a handler, providing a shared cache for `f x`
- If a handler already exists, `with_shared_cache` reuses its cache ⇒ nested calls share the outermost cache

✓ The cache crosses all the functor layers without appearing in any signature.

✓ The cache scope can be controlled dynamically.

Using the cache

```
(* Subtyping always needs a cache *)
let is_empty_rec t =
  let cache = get_is_empty_cache () in
  match Table.find cache t with
  | Some b -> b
  | None ->
    let b = VDescr.is_empty (def t) in
    Table.update cache t b ; b
let is_empty_rec t =
  with_shared_cache is_empty_rec t

(* Node construction: memoized
   only if someone asked for it *)
let cons d =
  try
    let cache = get_cons_cache () in
    ... (* lookup / add *)
  with Unhandled GetCache ->
    let t = mk () in define t d ; t
```

- Some operations require a cache (e.g. subtyping)
- Some operations use a cache only if provided (e.g. type constructors)

Using the cache

```
(* Subtyping always needs a cache *)
let is_empty_rec t =
  let cache = get_is_empty_cache () in
  match Table.find cache t with
  | Some b -> b
  | None ->
    let b = VDescr.is_empty (def t) in
    Table.update cache t b ; b
let is_empty_rec t =
  with_shared_cache is_empty_rec t

(* Node construction: memoized
   only if someone asked for it *)
let cons d =
  try
    let cache = get_cons_cache () in
    ... (* lookup / add *)
  with Unhandled GetCache ->
    let t = mk () in define t d ; t
```

- Some operations require a cache (e.g. subtyping)
- Some operations use a cache only if provided (e.g. type constructors)

The user chooses the scope:

```
(* in SSTT: one tallying instance *)
let tally cs =
  Ty.with_shared_cache tally cs

(* in MLsem: e.g. a whole definition *)
Ty.with_shared_cache infer_def d
```

- No handler: one cache per subtyping test
- Outermost handler: cache freed when it returns

Conclusion

- A recursive architecture composed of several layers and components
  - Every component/layer is a functor, recursion closed in a single module
  - Struggling with the type-checker (safe recursive modules, signatures)
  - Local imperativity makes recursive types easy to build

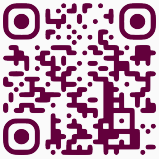
- Operations on recursive types require caching at different levels
  - ✗ *Global cache*: prevent GC (+ weak hash tables do not ensure termination)
  - ✗ Solution 1: *cache* parameter through all the different components and layers
  - ✓ Solution 2: algebraic effects, the user chooses the scope

`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., 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.