

# Compositional Static Value Analysis for Higher-Order Programs

---

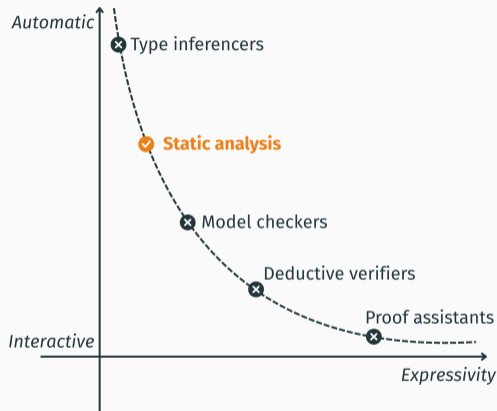
Milla Valnet <sup>1</sup>, Raphaël Monat <sup>2</sup>, Antoine Miné <sup>1</sup>  
Cambium Seminar

<sup>1</sup>Sorbonne Université, <sup>2</sup>Inria Lille

# Introduction

---

# Abstract Interpretation



from X. Leroy, Collège de France, "Le logiciel, entre esprit et matière"

**Figure 1:** Verification approaches

# Functional programming

New features to handle!



# Functional programming

New features to handle!



- Algebraic Data Types
- Pattern-matching
- Recursivity
- Higher Order
- Polymorphism
- Combined with side effects

# Functional programming

New features to handle!



- Algebraic Data Types
- Pattern-matching
- Recursivity
- Higher Order
- Polymorphism
- Combined with side effects

# Functional programming

New features to handle!



- Algebraic Data Types
- Pattern-matching
- Recursivity
- Higher Order
- Polymorphism
- Combined with side effects

Our analysis:

- Static Value Analysis
- Relational
- Compositional
- Domain-parametric

Compositional analysis [CC02]:

# Compositionality and Relationality

## Compositional analysis [CC02]:

- Automatically generates contracts

# Compositionality and Relationality

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability
- At definition site: analyze the function once and for all

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability
- At definition site: analyze the function once and for all
- At call site: re-use the generated summary

# Compositionality and Relationality

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability
- At definition site: analyze the function once and for all
- At call site: re-use the generated summary

```
let max x y = if x > y then x else y
```

# Compositionality and Relationality

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability
- At definition site: analyze the function once and for all
- At call site: re-use the generated summary

```
let max x y = if x > y then x else y
```

Relationality needed!

# Compositionality and Relationality

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability
- At definition site: analyze the function once and for all
- At call site: re-use the generated summary

```
let max x y = if x > y then x else y
```

## Relationality needed!

✗ Interval domain:  $\max : x = T \wedge y = T \wedge r = T$

# Compositionality and Relationality

## Compositional analysis [CC02]:

- Automatically generates contracts
- Often improves scalability
- At definition site: analyze the function once and for all
- At call site: re-use the generated summary

```
let max x y = if x > y then x else y
```

## Relationality needed!

- ✗ Interval domain:  $\max : x = T \wedge y = T \wedge r = T$
- ✓ Polyhedra domain:  $\max : x \leq r \wedge y \leq r$

## Motivating example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

## Motivating example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

- This program is well-typed, but it does not prove the assertion.

## Motivating example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

- This program is well-typed, but it does not prove the assertion.
- Deductive methods would require annotations.

## Motivating example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

- This program is well-typed, but it does not prove the assertion.
- Deductive methods would require annotations.

What about static analysis by abstract interpretation?

- Type systems [Vaz+14] and deductive methods [PR21]: SAT/SMT solvers, annotations

- Type systems [Vaz+14] and deductive methods [PR21]: SAT/SMT solvers, annotations
- Control Flow Analysis [MJ21], termination or strictness analysis [CC94], etc.

- Type systems [Vaz+14] and deductive methods [PR21]: SAT/SMT solvers, annotations
- Control Flow Analysis [MJ21], termination or strictness analysis [CC94], etc.
- [BJM24]: precise domain for non-recursive algebraic values

- Type systems [Vaz+14] and deductive methods [PR21]: SAT/SMT solvers, annotations
- Control Flow Analysis [MJ21], termination or strictness analysis [CC94], etc.
- [BJM24]: precise domain for non-recursive algebraic values
- [LM24]: Salto, an analyzer for a large subset of OCaml, but neither compositional nor relational

# Recursive algebraic data types

---

## User-defined list type

```
type list = Cons of int * list | Nil
```

## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```

## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```

## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```



## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```



## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```

```
let y = Nil
```

```
» ((y.Cons.0: ⊥, y.Cons.1: ⊥), {Nil})
```

## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```

```
let y = Nil
```

```
» ((y.Cons.0: ⊥, y.Cons.1: ⊥), {Nil})
```

```
let z = Cons(4, x)
```

```
» ((z.Cons.0:[1, 4], z.Cons.1:{Nil, Cons}), {Cons})
```

## User-defined list type

```
type list = Cons of int * list | Nil
```

```
let x = Cons( 1, Cons(2, Cons(3, Nil)))
```

```
» ((x.Cons.0:[1, 3], x.Cons.1:{Nil, Cons}), {Cons})
```

```
let y = Nil
```

```
» ((y.Cons.0: ⊥, y.Cons.1: ⊥), {Nil})
```

```
let z = Cons(4, x)
```

```
» ((z.Cons.0:[1, 4], z.Cons.1:{Nil, Cons}), {Cons})
```

```
⇒ Abstracts non-empty lists of elements between 1 and 4
```

## User-defined Algebraic Data Types

```
type t =  
  | C1 of t1,1 * ... * t1,n_1  
  | ...  
  | Cm of tm,1 * ... * tn,n_m
```

The resulting abstract domain:

- ✓ Supports any user-defined type

## User-defined Algebraic Data Types

```
type t =  
  | C1 of t1,1 * ... * t1,n_1  
  | ...  
  | Cm of tm,1 * ... * tn,n_m
```

The resulting abstract domain:

- ✓ Supports any user-defined type
- ✓ Is domain-parametric (independent of the chosen domain for  $t_{i,j}$ )

## User-defined Algebraic Data Types

```
type t =  
  | C1 of t1,1 * ... * t1,n_1  
  | ...  
  | Cm of tm,1 * ... * tn,n_m
```

The resulting abstract domain:

- ✓ Supports any user-defined type
- ✓ Is domain-parametric (independent of the chosen domain for  $t_{i,j}$ )
- ✓ Supports relationality

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$\sigma^\checkmark \leftarrow \text{match}(e, p1)$

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$\sigma^\checkmark \leftarrow \text{match}(e, p1)$

$E^\# [ e1 ] \sigma^\checkmark$

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$\sigma^{\checkmark} \leftarrow \text{match}(e, p1)$

$\sigma^{\times} \leftarrow \text{no\_match}(e, p1)$

$E^{\#}[ e1 ]\sigma^{\checkmark}$

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$\sigma^{\checkmark} \leftarrow \text{match}(e, p1)$

$\sigma^{\times} \leftarrow \text{no\_match}(e, p1)$

$\mathbb{E}^{\#}[\text{e1}] \sigma^{\checkmark}$

$\mathbb{E}^{\#}[\text{match } e \text{ with } | p2 -> e2 | \dots | pn -> en] \sigma^{\times}$

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$\sigma^{\checkmark} \leftarrow \text{match}(e, p1)$

$\sigma^{\times} \leftarrow \text{no\_match}(e, p1)$

$\mathbb{E}^{\#}[\text{e1}] \sigma^{\checkmark}$

$\cup^{\#}$

$\mathbb{E}^{\#}[\text{match } e \text{ with } | p2 -> e2 | \dots | pn -> en] \sigma^{\times}$

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$\sigma^{\checkmark} \leftarrow \text{match}(e, p1)$

$\sigma^{\times} \leftarrow \text{no\_match}(e, p1)$

$\mathbb{E}^{\#}[\text{e1}] \sigma^{\checkmark} \quad \cup^{\#} \quad \mathbb{E}^{\#}[\text{match } e \text{ with } | p2 \text{ -> } e2 \text{ | ... | } pn \text{ -> } en] \sigma^{\times}$

- Flow sensitive
- Handle when clauses

# Pattern-matching

```
match e with | p1 -> e1 | ... | pn -> en
```

$$\sigma^{\checkmark} \leftarrow \text{match}(e, p1)$$
$$\sigma^{\times} \leftarrow \text{no\_match}(e, p1)$$
$$\mathbb{E}^{\#}[\![ e1 ]\!] \sigma^{\checkmark} \quad \cup^{\#} \quad \mathbb{E}^{\#}[\![ \text{match } e \text{ with } | p2 -> e2 | \dots | pn -> en ]\!] \sigma^{\times}$$

- Flow sensitive
- Handle when clauses

$\Rightarrow$  This can check pattern exhaustivity!

# Compositional Analysis for First Order Functions

---

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

# Functions as relations

## Function analysis:

1. Initialize inputs to T.
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

Here,  $f^\# : r = x.0 + x.1$ .

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

Here,  $f^\# : r = x.0 + x.1$ .

## Application:

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

Here,  $f^\# : r = x.0 + x.1$ .

## Application:

```
f (42, 12)
```

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

Here,  $f^\# : r = x.0 + x.1$ .

## Application:

```
f (42, 12)
```

Add constraints:

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

Here,  $f^\# : r = x.0 + x.1$ .

## Application:

```
f (42, 12)
```

Add constraints: here,  $r = x.0 + x.1 \wedge x.0 = 42 \wedge x.1 = 12$

# Functions as relations

## Function analysis:

1. Initialize inputs to  $T$ .
2. Analyze the body.
3. Deduce the relation between inputs and output.

```
let f = fun x -> match x with (a,b) -> a + b
```

Here,  $f^\# : r = x.0 + x.1$ .

## Application:

```
f (42, 12)
```

Add constraints: here,  $r = x.0 + x.1 \wedge x.0 = 42 \wedge x.1 = 12 \implies r = 54$ .

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
  2. Analyze body.
  3. Get a new summary for  $f$
  4. Go back to 2.
- ⇒ Iterate until fixpoint (Kleene iterations)

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

$\Rightarrow$  Iterate until fixpoint (Kleene iterations)

Abstract semantics of recursive functions...

```
let rec f = fun x1 ... xn -> body in
```

Concrete semantics of recursive functions... Fixpoints!

1. Assume  $f \rightarrow \perp$ .
2. Analyze body.
3. Get a new summary for  $f$
4. Go back to 2.

$\Rightarrow$  Iterate until fixpoint (Kleene iterations)

Abstract semantics of recursive functions... Fixpoint approximation with widening!

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c : T$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c = \{Cons\}, h > inf$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$r = \perp$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c = \{Cons\}, h \leq inf$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$r = \perp$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c = \{Nil\}$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$r = \{nil\}$

## On the example

```
let rec filter_lt = fun inf l -> match l with  
  | Cons(h, q) when h > inf -> filter_lt inf q  
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))  
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$$\Rightarrow r = \perp \cup^{\#} \perp \cup^{\#} \{nil\}$$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c : T$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c = \{Cons\}, h > inf$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$r = \{Nil\}$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c = \{Cons\}, h \leq inf$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$$
$$r = ((r.Cons.0 \leq inf, \{Nil\}), \{Cons\})$$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$inf : T, l.Cons.0 : T, l.Cons.1 : T, l_c = \{Nil\}$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$r = \{Nil\}$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$\Rightarrow r = \{nil\} \cup^\# ((r.Cons.0 \leq inf, \{Nil\}), \{Cons\}) \cup^\# \{nil\}$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$\Rightarrow r = ((r.Cons.0 \leq inf, \{Nil\}), \{Cons, Nil\})$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 \leq inf, \{Nil\}), \{Cons, Nil\})$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 \leq inf, \{Nil\}), \{Cons, Nil\})$

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 \leq inf, \{Cons, Nil\}), \{Cons, Nil\})$

## On the example

```
let rec filter_lt = fun inf l -> match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil
```

We initialize  $filter\_lt : ((l.Cons.0 : T, l.Cons.1 : T), l_c : T) \rightarrow \perp$ .

We iteratively analyze the body.

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 : \perp, \emptyset), \{Nil\})$

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 \leq inf, \{Nil\}), \{Cons, Nil\})$

$filter\_lt : l \rightarrow inf \rightarrow ((r.Cons.0 \leq inf, \{Cons, Nil\}), \{Cons, Nil\})$

The next iteration gives same result: this is a fixpoint.

## On the example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

- $\text{filter\_lt} : l \rightarrow \text{inf} \rightarrow ((r.\text{Cons}.0 \leq \text{inf}, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\})$

## On the example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

- $\text{filter\_lt} : l \rightarrow \text{inf} \rightarrow ((r.\text{Cons.}0 \leq \text{inf}, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\})$
- ✓ The assertion succeeds...

## On the example

```
type list = Cons of int * list | Nil

let rec filter_lt inf l = match l with
  | Cons(h, q) when h > inf -> filter_lt inf q
  | Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
  | Nil -> Nil

let y = filter_lt 4 (Cons(0, Cons(5, Cons(11, Nil))))
assert(match y with Cons(h, q) -> h <= 4 | Nil -> true )
```

- $\text{filter\_lt} : l \rightarrow \text{inf} \rightarrow ((r.\text{Cons.}0 \leq \text{inf}, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\})$
- ✓ The assertion succeeds... and the pattern is proved exhaustive!(\*)

Key ingredients:

- Smashing ([Gop+04])
- Heterogeneous environments ([Jou19])
- Relationality
- Domain-parametricity

## Higher Order

---

## Motivation

```
type cst_or_fun = Cst of int | Fun of int -> int
let to_fun a =
  match a with
  | Cst n -> fun x -> n
  | Fun f -> f

let f1 = to_fun (Cst 5)
let f2 = to_fun (Fun(fun x -> x + 4))
let r1 = f1 4
let r2 = f2 5
```

- **Non-compositional analysis:** Analysis of `to_fun` when parameters are known:
  - ⇒  $r_1 = 5, r_2 = 9$ , at the cost of an analysis per call
  - ⇒ This is what Salto does!

- **Non-compositional analysis:** Analysis of `to_fun` when parameters are known:
  - ⇒  $r_1 = 5, r_2 = 9$ , at the cost of an analysis per call
  - ⇒ This is what Salto does!
- **Standard compositional analysis:** One input-output relation for `to_fun`:
  - ⇒  $r_1 = \top, r_2 = \top$

State of the art:

- [Bou92]: dynamic partitioning, non-relational
- [BH19]: 1st order relational partitioning

State of the art:

- [Bou92]: dynamic partitioning, non-relational
- [BH19]: 1st order relational partitioning

Our method at higher-order:

- Domain abstracting functions (for higher-order)

State of the art:

- [Bou92]: dynamic partitioning, non-relational
- [BH19]: 1st order relational partitioning

Our method at higher-order:

- Domain abstracting functions (for higher-order)
- Parametric in the domain for inputs and output.

# Partitioning at higher-order

State of the art:

- [Bou92]: dynamic partitioning, non-relational
- [BH19]: 1st order relational partitioning

Our method at higher-order:

- Domain abstracting functions (for higher-order)
- Parametric in the domain for inputs and output.
- Lifts Boutonnet's disjunctive relational summaries to higher-order

- ✓ ADT domain parametric in leafs' domains
- ⇒ ADTs can contain functions!

✓ ADT domain parametric in leafs' domains

⇒ ADTs can contain functions!

✓ Function domain parametric in inputs and output's domains

⇒ Functions can manipulate ADTs!

## On the example

```
type cst_or_fun = Cst of int | Fun of int -> int
let to_fun a = match a with
  | Cst n -> fun x -> n
  | Fun f -> f

let f1, f2 = to_fun (Cst 5), to_fun (Fun(fun x -> x + 4))
let r1, r2 = f1 4, f2 5
```

- $to\_fun: a \rightarrow \{a\_c = Cst \wedge r = (x \rightarrow a.Cst.0)\} \vee \{a\_c = Fun \wedge r = a.Fun.0\}$

## On the example

```
type cst_or_fun = Cst of int | Fun of int -> int
let to_fun a = match a with
  | Cst n -> fun x -> n
  | Fun f -> f

let f1, f2 = to_fun (Cst 5), to_fun (Fun(fun x -> x + 4))
let r1, r2 = f1 4, f2 5
```

- $to\_fun : a \rightarrow \{a\_c = Cst \wedge r = (x \rightarrow a.Cst.0)\} \vee \{a\_c = Fun \wedge r = a.Fun.0\}$
- $f_1 : x \rightarrow r = 5, f_2 : x \rightarrow r = x + 4$

## On the example

```
type cst_or_fun = Cst of int | Fun of int -> int
let to_fun a = match a with
  | Cst n -> fun x -> n
  | Fun f -> f

let f1, f2 = to_fun (Cst 5), to_fun (Fun(fun x -> x + 4))
let r1, r2 = f1 4, f2 5
```

- $to\_fun : a \rightarrow \{a\_c = Cst \wedge r = (x \rightarrow a.Cst.0)\} \vee \{a\_c = Fun \wedge r = a.Fun.0\}$
- $f_1 : x \rightarrow r = 5, f_2 : x \rightarrow r = x + 4$
- $r_1 : 5, r_2 : 9$

With our compositional analysis, partial application comes for free!

With our compositional analysis, partial application comes for free!

```
let max x y = if x > y then x else y
let partial_max = max 5
```

- $\text{max} : x, y \rightarrow \{x > y \wedge r = x\} \vee \{x \leq y \wedge r = y\}$

## Partial application

With our compositional analysis, partial application comes for free!

```
let max x y = if x > y then x else y
let partial_max = max 5
```

- $\text{max} : x, y \rightarrow \{x > y \wedge r = x\} \vee \{x \leq y \wedge r = y\}$
- $\text{partial\_max} : y \rightarrow \{x > y \wedge r = x\} \vee \{x \leq y \wedge r = y\} [\mathbf{x = 5}]$

## Partial application

With our compositional analysis, partial application comes for free!

```
let max x y = if x > y then x else y
let partial_max = max 5
```

- $\text{max} : x, y \rightarrow \{x > y \wedge r = x\} \vee \{x \leq y \wedge r = y\}$
- $\text{partial\_max} : y \rightarrow \{5 > y \wedge r = 5\} \vee \{5 \leq y \wedge r = y\}$

Improve precision: a new domain  
for ADTs

---

## Need for a new abstract domain

```
let rec filter_lt inf l = match l with
| Cons(h, q) when h > inf -> filter_lt inf q
| Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
| Nil -> Nil
```

- Current summary:  $\text{filter\_lt} : l \rightarrow \text{inf} \rightarrow ((r.\text{Cons}.0 \leq \text{inf}, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\})$

## Need for a new abstract domain

```
let rec filter_lt inf l = match l with
| Cons(h, q) when h > inf -> filter_lt inf q
| Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
| Nil -> Nil
```

- Current summary:  $\text{filter\_lt} : l \rightarrow \text{inf} \rightarrow ((r.\text{Cons}.0 \leq \text{inf}, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\})$
- ✗ No relation between  $l$  and  $r$  !

## Relation on weak variables

```
let l = Cons(a+1, Cons(a, Cons(a-1, Nil))) (* a-1 <= l.Cons.0 <= a+1 *)
```

`l.Cons.0` is a *weak* variable: it represents multiple concrete elements!

## Relation on weak variables

```
let l = Cons(a+1, Cons(a, Cons(a-1, Nil))) (* a-1 <= l.Cons.0 <= a+1 *)
```

`l.Cons.0` is a *weak* variable: it represents multiple concrete elements!

```
let q = match l with Cons(h, q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

✓ `q.Cons.0` respects the same constraints as `l.Cons.0`

## Relation on weak variables

```
let l = Cons(a+1, Cons(a, Cons(a-1, Nil))) (* a-1 <= l.Cons.0 <= a+1 *)
```

`l.Cons.0` is a *weak* variable: it represents multiple concrete elements!

```
let q = match l with Cons(h, q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

- ✓ `q.Cons.0` respects the same constraints as `l.Cons.0`
- ✗ We do not have `q.Cons.0 = l.Cons.0`

## Relation on weak variables

```
let l = Cons(a+1, Cons(a, Cons(a-1, Nil))) (* a-1 <= l.Cons.0 <= a+1 *)
```

l.Cons.0 is a *weak* variable: it represents multiple concrete elements!

```
let q = match l with Cons(h, q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

✓ q.Cons.0 respects the same constraints as l.Cons.0

✗ We do not have q.Cons.0 = l.Cons.0

⇒ We perform an expand i.e. we copy the constraints on l.Cons.0

## Relation on weak variables

```
let l = Cons(a+1,Cons(a,Cons(a-1,Nil))) (* a-1 <= l.Cons.0 <= a+1 *)
```

`l.Cons.0` is a *weak* variable: it represents multiple concrete elements!

```
let q = match l with Cons(h,q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

✓ `q.Cons.0` respects the same constraints as `l.Cons.0`

✗ We do not have `q.Cons.0 = l.Cons.0`

⇒ We perform an *expand* i.e. we copy the constraints on `l.Cons.0`

⇒  $a - 1 \leq q.Cons.0 \leq a + 1$  *but* no relation between `l` and `q`

```
let q = match l with Cons(h,q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

- `q.Cons.0` is expanded from `l.Cons.0`
- We want to store this relation!

## DelExp domain

```
let q = match l with Cons(h,q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

- `q.Cons.0` is expanded from `l.Cons.0`
- We want to store this relation!

We create the DelExp domain, which stores constraints under the form  $x \triangleleft y$ .

## DelExp domain

```
let q = match l with Cons(h,q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

- `q.Cons.0` is expanded from `l.Cons.0`
- We want to store this relation!

We create the DelExp domain, which stores constraints under the form  $x \triangleleft y$ .

$x \triangleleft y$

- $\iff$  Elements of  $x$  forms a subset of  $y$
- $\iff$  The elements of  $x$  verifies all the constraints verified by the elements of  $y$
- $\iff$  We can always propagate constraints from  $y$  to  $x$
- $\iff$  We can always perform `expand(x,y)`

## DelExp domain

```
let q = match l with Cons(h,q) -> q (* a-1 <= q.Cons.0 <= a+1*)
```

- `q.Cons.0` is expanded from `l.Cons.0`
- We want to store this relation!

We create the DelExp domain, which stores constraints under the form  $x \triangleleft y$ .

$x \triangleleft y$

$\iff$  Elements of  $x$  forms a subset of  $y$

$\iff$  The elements of  $x$  verifies all the constraints verified by the elements of  $y$

$\iff$  We can always propagate constraints from  $y$  to  $x$

$\iff$  We can always perform `expand(x,y)`

✓ We can now infer: `q.Cons.0  $\triangleleft$  l.Cons.0`

```
let rec filter_lt inf l = match l with
| Cons(h, q) when h > inf -> filter_lt inf q
| Cons(h, q) when h <= inf -> Cons(h, (filter_lt inf q))
| Nil -> Nil
```

✓ New summary:  $\text{filter\_lt} : l \rightarrow \text{inf} \rightarrow r.\text{Cons}.0 < \text{inf} \wedge r.\text{Cons}.0 \triangleleft l.\text{Cons}.0$

```
let rec mult2 l = match l with
  | Cons(h, q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
```

✗ Current summary:  $\text{mult2} : l \rightarrow T$

## AffDelExp: imprecision on mult2

```
let rec mult2 l = match l with
  | Cons(h, q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
```

**X** Current summary:  $\text{mult2} : l \rightarrow T$

$\Rightarrow$  AffDelExp domain:  $x \triangleleft a \times y + b$

## AffDelExp: imprecision on mult2

```
let rec mult2 l = match l with
  | Cons(h, q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
```

✗ Current summary:  $\text{mult2} : l \rightarrow T$

$\Rightarrow$  AffDelExp domain:  $x \triangleleft a \times y + b$

✓ New summary:  $\text{mult2} : l \rightarrow r.\text{Cons}.0 \triangleleft 2 \times l.\text{Cons}.0$

## ApplyDelExp: Imprecision on map (no side-effects)

```
let rec map f l = match l with
  | Cons(h, q) -> Cons(f h, map f q)
  | Nil -> Nil
```

✗ Current summary:  $\text{map} : l \rightarrow f \rightarrow T$

## ApplyDelExp: Imprecision on map (no side-effects)

```
let rec map f l = match l with
  | Cons(h, q) -> Cons(f h, map f q)
  | Nil -> Nil
```

- ✗ Current summary:  $\text{map} : l \rightarrow f \rightarrow T$
- ApplyDelExp domain:  $x \triangleleft f(y)$

## ApplyDelExp: Imprecision on map (no side-effects)

```
let rec map f l = match l with
  | Cons(h, q) -> Cons(f h, map f q)
  | Nil -> Nil
```

- ✗ Current summary:  $\text{map} : l \rightarrow f \rightarrow T$
- ApplyDelExp domain:  $x \triangleleft f(y)$
- ✓ New summary:  $\text{map} : l \rightarrow f \rightarrow r.\text{Cons}.0 \triangleleft f(l.\text{Cons}.0)$

## A general framework

We have a general methodology to create domains for properties  $x \triangleleft \theta(y)$ !

## A general framework

We have a general methodology to create domains for properties  $x \triangleleft \theta(y)$ !

✓ No hypothesis on variable types  $\implies$  Polymorphism

# A general framework

We have a general methodology to create domains for properties  $x \triangleleft \theta(y)$ !

- ✓ No hypothesis on variable types  $\Rightarrow$  Polymorphism
- ✓ Family of properties  $\Rightarrow$  Higher-order

# A general framework

We have a general methodology to create domains for properties  $x \triangleleft \theta(y)$ !

- ✓ No hypothesis on variable types  $\Rightarrow$  Polymorphism
- ✓ Family of properties  $\Rightarrow$  Higher-order
- ✓ Operate directly on weak variables  $\Rightarrow$  Works for all kind of containers

## A general framework

We have a general methodology to create domains for properties  $x \triangleleft \theta(y)$ !

- ✓ No hypothesis on variable types  $\implies$  Polymorphism
- ✓ Family of properties  $\implies$  Higher-order
- ✓ Operate directly on weak variables  $\implies$  Works for all kind of containers
- ✓ Language-agnostic  $\implies$  Works both on OCaml and Python!

# Implementation

---

# MOPSA (Modular Open Platform for Static Analysis)



A modular and multi-language open-source platform:

# MOPSA (Modular Open Platform for Static Analysis)



A modular and multi-language open-source platform:

- Simplify the design of static analyzers

# MOPSA (Modular Open Platform for Static Analysis)



A modular and multi-language open-source platform:

- Simplify the design of static analyzers
- Implement relational abstract domains

# MOPSA (Modular Open Platform for Static Analysis)



A modular and multi-language open-source platform:

- Simplify the design of static analyzers
- Implement relational abstract domains
- Rely on cooperation and communication between them

# MOPSA (Modular Open Platform for Static Analysis)



A modular and multi-language open-source platform:

- Simplify the design of static analyzers
- Implement relational abstract domains
- Rely on cooperation and communication between them
- Support subsets of C and Python

<https://gitlab.com/mopsa/mopsa-analyzer>

# OCaml Analysis in MOPSA

Implementation steps:

- ✓ Injection of OCaml typed AST into MOPSA AST
- ✓ Domains for algebraic values
- ✓ Transfer functions (let, pattern-matching, etc.)
- ✓ Compositional function analysis
- ✓ Partitioning analysis
- ✓ Heterogeneous operators

# OCaml Analysis in MOPSA

Implementation steps:

- ✓ Injection of OCaml typed AST into MOPSA AST
- ✓ Domains for algebraic values
- ✓ Transfer functions (let, pattern-matching, etc.)
- ✓ Compositional function analysis
- ✓ Partitioning analysis
- ✓ Heterogeneous operators

3000 lines of OCaml, tested on hundred toy programs.

Limitations:

- ✗ Impure features (mutable arrays, references)
- ✗ Modules, functors...

## Experimental evaluation

Program name	Analysis (ms)	Program name	Analysis (ms)
numeric_loop3b.ml	12	filter_le.ml	131
binomial.ml	400	make_list.ml	131
mc91.ml	30	match_when.ml	9
tak.ml	923	is_exhaustive.ml	10
abs.ml	15	partial_app.ml	15
xor.ml	36	embed_fun.ml	9
rec_add.ml	44	to_fun.ml	11
non_terminate.ml	6	f_from_g.ml	11

**Figure 2:** Benchmarks for the OCaml analysis.

- ✓ Comparable with the analysis time of Salto (<1s)
- ✓ With the same precision (sometimes better) on most programs
- ✗ Toy programs so far, scalability to assess

Function	Delexp (s)	Overhead	Summary
hd.ml	0.008	2.6	$r \triangleleft 1.\text{Cons}.0$
tl.ml	0.008	2.5	$r.\text{Cons}.0 \triangleleft 1.\text{Cons}.0$
filter.ml	0.046	2.3	$r.\text{Node}.0 \triangleleft 1.\text{Node}.0$
map.ml	0.048	3.4	$r.\text{Cons}.0 \triangleleft f(1.\text{Cons}.0)$
copy.ml	0.056	3.0	$r.\text{Cons}.0 \triangleleft 1.\text{Cons}.0$
filterle.ml	0.160	3.7	$r.\text{Cons}.0 \triangleleft 1.\text{Cons}.0 \wedge r.\text{Cons}.0 < inf$
mult2.ml	0.065	3.4	$r.\text{Cons}.0 \triangleleft 2 \times 1.\text{Cons}.0$
mult2tree.ml	0.099	3.2	$r.\text{Node}.0 \triangleleft 2 \times 1.\text{Node}.0$
mult3plus4.ml	0.067	3.6	$r.\text{Cons}.0 \triangleleft 3 \times 1.\text{Cons}.0 + 4$
maptree.ml	0.061	2.8	$r.\text{Node}.0 \triangleleft f(1.\text{Node}.0)$
listtotree.ml	0.062	2.9	$r.\text{Cons}.0 \triangleleft 1.\text{Node}.0$

**Figure 3:** Analysis of ADT-manipulating OCaml functions

Function	Delexp (s)	Overhead	Summary
copy.list.py	0.069	1.3	$l_2 \triangleleft l_1$
copy.list.comprehension.py	0.069	1.4	$l_2 \triangleleft l_1$
copy.set.py	0.056	1.03	$s_2 \triangleleft s_1$
copy.set.comprehension.py	0.059	1.03	$s_2 \triangleleft s_1$
filter.list.py	0.092	1.5	$l_2 \triangleleft l_1$
filter.list.comprehension.py	0.091	1.5	$l_2 \triangleleft l_1$
filter.set.py	0.057	1.04	$s_2 \triangleleft s_1$
filter.set.comprehension.py	0.065	1.08	$s_2 \triangleleft s_1$
mult2.list.py	0.070	1.4	$l_2 \triangleleft 2 \times l_1$
mult2.list.comprehension.py	0.050	1.5	$l_2 \triangleleft 2 \times l_1$
mult2.set.py	0.057	1.1	$s_2 \triangleleft 2 \times s_1$
mult2.set.comprehension.py	0.062	1.06	$s_2 \triangleleft 2 \times s_1$
partition.list.py	0.124	1.7	$l_2 \triangleleft l_1 \wedge l_3 \triangleleft l_1$
partition.set.py	0.076	1.1	$s_2 \triangleleft s_1 \wedge s_3 \triangleleft s_1$
partition.set-list.py	0.104	1.4	$l_2 \triangleleft s_1 \wedge l_3 \triangleleft s_1$
partition.list-set.py	0.096	1.2	$s_2 \triangleleft l_1 \wedge s_3 \triangleleft l_1$
tail.list.py	0.076	1.3	$l_2 \triangleleft l_1$

**Figure 4:** Analysis of list-manipulating Python programs

## Conclusion and Future Work

---

Compositional, relational value analysis for pure functional language:

- A relational domain for algebraic values
- A domain for disjunctive function summaries
- Combined together thanks to their parametricity
- Implemented into MOPSA platform

## Conclusion and future work

Compositional, relational value analysis for pure functional language:

- A relational domain for algebraic values
- A domain for disjunctive function summaries
- Combined together thanks to their parametricity
- Implemented into MOPSA platform

And a language-agnostic domain for containers!

## Conclusion and future work

Compositional, relational value analysis for pure functional language:

- A relational domain for algebraic values
- A domain for disjunctive function summaries
- Combined together thanks to their parametricity
- Implemented into MOPSA platform

And a language-agnostic domain for containers! **Future work:**

- More refined properties on ADTs
- Extend to an impure fragment
- Overflows and exceptions... with compositionality!

# References

---

- [BH19] Rémy Boutonnet and Nicolas Halbwachs. “**Disjunctive relational abstract interpretation for interprocedural program analysis**”. In: *Verification, Model Checking, and Abstract Interpretation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings 20*. Springer. 2019, pp. 136–159.
- [BJM24] Santiago Bautista, Thomas Jensen, and Benoit Montagu. “**An input–output relational domain for algebraic data types and functional arrays**”. In: *Formal Methods in System Design (2024)*, pp. 1–74.
- [Bou92] François Bourdoncle. “**Abstract interpretation by dynamic partitioning**”. In: *Journal of Functional Programming 2.4 (1992)*, pp. 407–435.

- [CC02] Patrick Cousot and Radhia Cousot. “**Modular static program analysis**”. In: *International Conference on Compiler Construction*. Springer. 2002, pp. 159–179.
- [CC94] Patrick Cousot and Radhia Cousot. “**Higher-order abstract interpretation (and application to compartment analysis generalizing strictness, termination, projection and PER analysis of functional languages)**”. In: *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. IEEE. 1994, pp. 95–112.
- [Gop+04] Denis Gopan et al. “**Numeric domains with summarized dimensions**”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*. Springer. 2004, pp. 512–529.

- [Jou19] Matthieu Journault. “**Analyse statique modulaire précise par interprétation abstraite pour la preuve automatique de correction de programmes et pour l’inférence de contrats** | Theses. fr”. PhD thesis. Sorbonne université, 2019.
- [LM24] Pierre Lermusiaux and Benoît Montagu. “**Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation**”. In: *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*. Vol. 14577. LNCS. Springer, 2024, pp. 391–420.
- [MJ21] Benoit Montagu and Thomas Jensen. “**Trace-based control-flow analysis**”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 482–496.

- [PR21] Mário Pereira and António Ravara. “**Cameleer: A deductive verification tool for OCaml**”. In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 677–689.
- [Vaz+14] Niki Vazou et al. “**Refinement types for Haskell**”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 2014, pp. 269–282.

# Relationality

```
let rec add x y =  
  if x = 0 then y  
  else add (x-1) y
```

```
let z = add 42 12
```

# Impure features

We'd like to support arrays, references and mutable fields.

- Identify impure variables with types and abstract them to  $T$
- Give them as inputs to every functions
- Identify functions where impure variables don't escape to reduce the cost

## Grégoire Maires's internship: C/OCaml interface

In OCaml, some functions are implemented through C function calls:

```
external seek_in : in_channel -> int -> unit =  
    "caml_ml_seek_in"
```

- ✓ Similar work was done by Raphaël Monat with C/Python in MOPSA.
- ✓ This internship aimed at exploring C/OCaml interface!

⇒ Preliminary results