Static Value Analysis by Abstract Interpretation for Functional Programs manipulating Recursive Algebraic Data Types

Milla Valnet[1,3], Raphaël Monat[2], Antoine Miné[3]

JFLA 2023

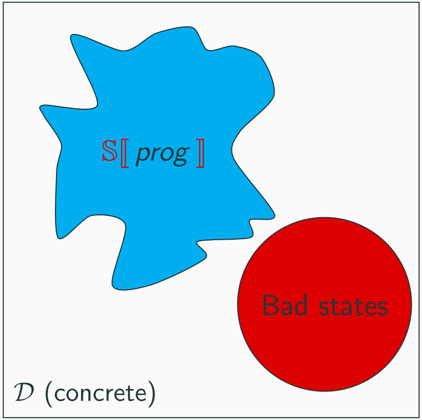
[1]ENS Ulm, [2]Inria Lille, [3]Sorbonne Université

Introduction

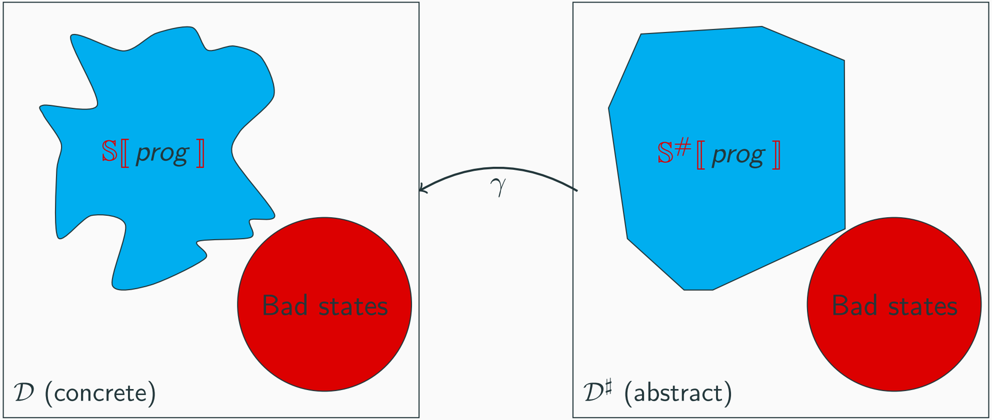
Software Verification

Software bugs can be costly... and testing is not enough!

Abstract interpretation

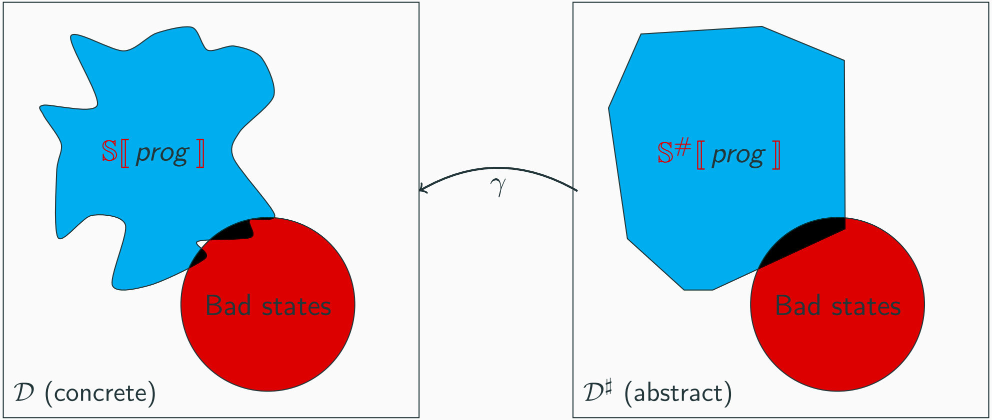


Abstract interpretation



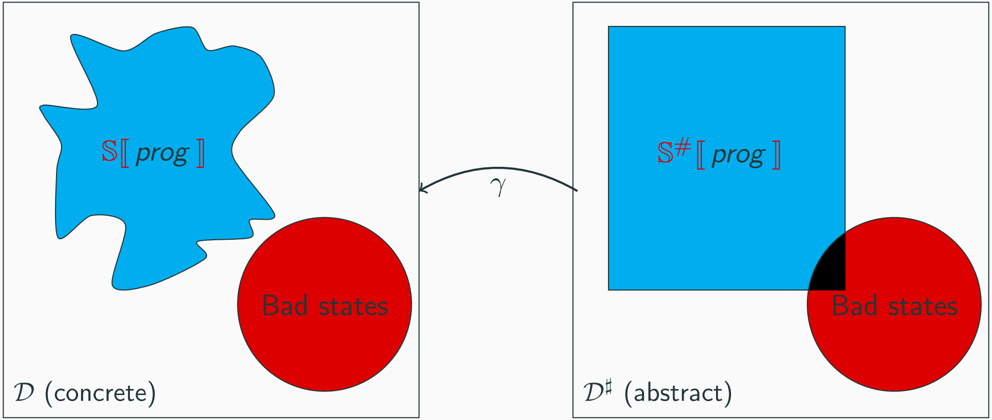
✓ Program correct

Abstract interpretation



✗ True alarm

Abstract interpretation



✗ False alarm (too unprecise)

Domains and relationality

```
x = 0 ; y = 1 ;
while (y < 1000){
  if (rand(0,1) == 0) { x++; } else { x--; } ;
  y++; }
```

Domains and relationality

```
x = 0 ; y = 1 ;
while (y < 1000){
  if (rand(0,1) == 0) { x++; } else { x--; } ;
  y++; }
```

Interval domain :

- $\mathcal{D} = \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z})$, $\mathcal{D}^\sharp = \{\ [a, b]\ \}^2$

Domains and relationality

```
x = 0 ; y = 1 ;
while (y < 1000){
  if (rand(0,1) == 0) { x++; } else { x--; } ;
  y++; }
```

Interval domain :

- $\mathcal{D} = \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z})$, $\mathcal{D}^\sharp = \{\ [a, b]\ \}^2$
- $y \rightarrow [1000, 1000]$, $x \in]-\infty, +\infty[$

Domains and relationality

```
x = 0 ; y = 1 ;
while (y < 1000){
  if (rand(0,1) == 0) { x++; } else { x--; } ;
  y++; }
```

Interval domain :

- $\mathcal{D} = \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z})$, $\mathcal{D}^\sharp = \{\ [a, b]\ \}^2$
- $y \to [1000, 1000]$, $x \in]-\infty, +\infty[$

Polyhedra domain :

- $\mathcal{D} = \mathcal{P}(\mathbb{Z}^n)$, $\mathcal{D}^\sharp = \{\ \bigwedge_{j \leq m} (\sum_{i=1}^{n} a_{i,j} V_i \geq \beta_j)\ \}$

Domains and relationality

```
x = 0 ; y = 1 ;
while (y < 1000){
  if (rand(0,1) == 0) { x++; } else { x--; } ;
  y++; }
```

Interval domain :

- $\mathcal{D} = \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z})$, $\mathcal{D}^\sharp = \{ \ [a, b] \ \}^2$
- $y \to [1000, 1000]$, $x \in]-\infty, +\infty[$

Polyhedra domain :

- $\mathcal{D} = \mathcal{P}(\mathbb{Z}^n)$, $\mathcal{D}^\sharp = \{ \bigwedge_{j \leq m} (\sum_{i=1}^{n} a_{i,j} V_i \geq \beta_j) \ \}$
- $y = 1000$, $-y < x < y$

Functional programming

Functional programming

New features to handle!

Functional programming

New features to handle!

Functional programming

New features to handle!

- ☐ Recursivity
- ☐ Algebraic Data Types
- ☐ Pattern-matching
- ☐ Higher Order
- ☐ Polymorphism

Functional programming

New features to handle!

- ✓ Recursivity
- ✓ Algebraic Data Types
- ✓ Pattern-matching
- ✗ Higher Order
- ✗ Polymorphism

Motivating example

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


let rec mult2 = fun l -> match l with
  | Cons(h,q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

Motivating example

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

let rec mult2 = fun l -> match l with
  | Cons(h,q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

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

Motivating example

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

let rec mult2 = fun l -> match l with
  | Cons(h,q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

- 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 mult2 = fun l -> match l with
  | Cons(h,q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

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

What about static analysis by abstract interpretation?

State of the art

For imperative and object-oriented languages, we have mature static value analyzers.

State of the art

For imperative and object-oriented languages, we have mature static value analyzers.

For functional languages:

- Type systems and deductive methods: SAT/SMT solvers, annotations

State of the art

For imperative and object-oriented languages, we have mature static value analyzers.

For functional languages:

- Type systems and deductive methods: SAT/SMT solvers, annotations
- Abstract interpreters for CFA, termination analysis, etc.: no value analysis

State of the art

For imperative and object-oriented languages, we have mature static value analyzers.

For functional languages:

- Type systems and deductive methods: SAT/SMT solvers, annotations
- Abstract interpreters for CFA, termination analysis, etc.: no value analysis
- Bautista et al. [2022]: domain for non recursive algebraic values

State of the art

For imperative and object-oriented languages, we have mature static value analyzers.

For functional languages:

- Type systems and deductive methods: SAT/SMT solvers, annotations
- Abstract interpreters for CFA, termination analysis, etc.: no value analysis
- Bautista et al. [2022]: domain for non recursive algebraic values
- Jhala et al. [2011]: HMC, translation into an imperative language

Domains for algebraic data

Algebraic Data Types

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

Algebraic Data Types

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

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

Algebraic Data Types

```
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})

Algebraic Data Types

```
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})

Algebraic Data Types

```
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: $\bot$, y.Cons.1: $\bot$), {Nil})

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

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

Algebraic Data Types

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

We choose as an abstract domain:

- We summarize non-recursive field i, j accessible from x : t in one variable $x.i.j$
- We summarize each recursive field by the set of constructors accessible from it.
- We keep track of x's constructor.

Algebraic Data Types

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

We choose as an abstract domain:

$$\mathcal{D}_{\mathtt{t}} = \prod_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n_i}} \mathcal{D}_{i,j}^{\perp} \times \mathcal{P}(\mathbb{C})$$

- We summarize non-recursive field i, j accessible from $x : \mathtt{t}$ in one variable $x.i.j$
- We summarize each recursive field by the set of constructors accessible from it.
- We keep track of x's constructor.

Transfer function: pattern-matching

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

We proceed iteratively:

- We evaluate e_i in an over-approximation of environments where e and p_i match.
- We remove p_i pattern and evaluate the result matching in one such that they can't.
- We join the results.

Transfer function: pattern-matching

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

We proceed iteratively:

- We evaluate e_i in an over-approximation of environments where e and p_i match.
- We remove p_i pattern and evaluate the result matching in one such that they can't.
- We join the results.

This method is *flow sensitive* and able to handle `when` clauses.

Transfer function: pattern-matching

```
match Cons(1, Nil) with
 | Cons(h,q) -> h
 | Nil -> 0
```

Transfer function: pattern-matching

```
match Cons(1, Nil) with
 | Cons(h,q) -> h
 | Nil -> 0
```

- `Cons`$(1,$ `Nil`$)$ and `Cons`(h, q) match in environments where $h = 1$. Then h evaluates to 1.

Transfer function: pattern-matching

```
match Cons(1, Nil) with
 | Cons(h,q) -> h
 | Nil -> 0
```

- `Cons`$(1,$`Nil`$)$ and `Cons`(h, q) match in environments where $h = 1$. Then h evaluates to 1.
- There is no remaining environment for the second pattern.

Transfer function: pattern-matching

```
match Cons(1, Nil) with
 | Cons(h,q) -> h
 | Nil -> 0
```

- `Cons`$(1,$ `Nil`$)$ and `Cons`(h, q) match in environments where $h = 1$. Then h evaluates to 1.
- There is no remaining environment for the second pattern.
- Then the result is 1.

Functions

Functions

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

A function is abstracted as a relation between the inputs and the output.

Functions

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

A function is abstracted as a relation between the inputs and the output.

- We initialize $x : (x.0, x.1)$ with $x.0$ and $x.1$ to $\top$.

Functions

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

A function is abstracted as a relation between the inputs and the output.

- We initialize $x : (x.0, x.1)$ with $x.0$ and $x.1$ to $\top$.
- We analyze the body of the function

Functions

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

A function is abstracted as a relation between the inputs and the output.

- We initialize $x : (x.0, x.1)$ with $x.0$ and $x.1$ to $\top$.
- We analyze the body of the function
- We deduce the relation between the result and x.

Functions

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

A function is abstracted as a relation between the inputs and the output.

- We initialize $x : (x.0, x.1)$ with $x.0$ and $x.1$ to $\top$.
- We analyze the body of the function
- We deduce the relation between the result and x.

Here, we have: $f : x \to x.0 + x.1$.

Functions

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

A function is abstracted as a relation between the inputs and the output.

- We initialize $x : (x.0, x.1)$ with $x.0$ and $x.1$ to $\top$.
- We analyze the body of the function
- We deduce the relation between the result and x.

Here, we have: $f : x \to x.0 + x.1$.

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

For application, we instantiate the input variables in the relation abstracting f by the abstraction of arguments.

Functions

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

A function is abstracted as a relation between the inputs and the output.

- We initialize $x : (x.0, x.1)$ with $x.0$ and $x.1$ to $\top$.
- We analyze the body of the function
- We deduce the relation between the result and x.

Here, we have: $f : x \rightarrow x.0 + x.1$.

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

For application, we instantiate the input variables in the relation abstracting f by the abstraction of arguments.

Here, we instantiate $x : (x.0, x.1)$ by $(42, 12)$ so we get $x.0 + x.1 = 42 + 12 = 54$.

Recursivity

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

For recursive functions, their concrete semantics is computed with a fixpoint, so their abstract semantics will use iterations, with a widening application.

Recursivity

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

For recursive functions, their concrete semantics is computed with a fixpoint, so their abstract semantics will use iterations, with a widening application.

- We start with $f : x_1 \rightarrow ... \rightarrow x_n \rightarrow \bot$ with x_i to $\top$.

Recursivity

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

For recursive functions, their concrete semantics is computed with a fixpoint, so their abstract semantics will use iterations, with a widening application.

- We start with $f : x_1 \to ... \to x_n \to \bot$ with x_i to $\top$.
- We evaluate the body of the function with this hypothesis and get a more precise abstraction for f.

Recursivity

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

For recursive functions, their concrete semantics is computed with a fixpoint, so their abstract semantics will use iterations, with a widening application.

- We start with $f : x_1 \rightarrow ... \rightarrow x_n \rightarrow \bot$ with x_i to $\top$.
- We evaluate the body of the function with this hypothesis and get a more precise abstraction for f.
- We iterate the body evaluation with this new hypothesis.

Recursivity

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

For recursive functions, their concrete semantics is computed with a fixpoint, so their abstract semantics will use iterations, with a widening application.

- We start with $f : x_1 \rightarrow \ldots \rightarrow x_n \rightarrow \bot$ with x_i to $\top$.
- We evaluate the body of the function with this hypothesis and get a more precise abstraction for f.
- We iterate the body evaluation with this new hypothesis.
- We ensure convergence in finite time by widening.

The analysis on our example

Example 1 - Non recursive function

```
let hd = fun l -> match l with
  | Cons(h,q) -> h
  | Nil -> 0
```

Example 1 - Non recursive function

```
let hd = fun l -> match l with
  | Cons(h,q) -> h
  | Nil -> 0
```

We abstract the right hand side.

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

We create variable $l : ((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons})$.

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

We create variable $l : ((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons})$.

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

We create variable $l : ((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons})$.

- l and $\texttt{Cons}(h, q)$ match when $l_{cons} = \{\texttt{Cons}\}$ and $l.\texttt{Cons}.0 = h$, then the result is $l.\texttt{Cons}.0$

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

We create variable $l : ((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons})$.

- l and $\texttt{Cons}(h, q)$ match when $l_{cons} = \{\texttt{Cons}\}$ and $l.\texttt{Cons.0} = h$, then the result is $l.\texttt{Cons.0}$
- l and Nil match when $l_{cons} = \{\texttt{Nil}\}$, then the result is 0.

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

We create variable $l : ((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons})$.

- l and $\texttt{Cons}(h, q)$ match when $l_{cons} = \{\texttt{Cons}\}$ and $l.\texttt{Cons}.0 = h$, then the result is $l.\texttt{Cons}.0$
- l and Nil match when $l_{cons} = \{\texttt{Nil}\}$, then the result is 0.
- The result is $0 \cup_{\mathbb{Z}} l.\texttt{Cons}.0$

Example 1 - Non recursive function

```
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil -> 0
```

We abstract the right hand side.

We create variable $l : ((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons})$.

- l and $\texttt{Cons}(h, q)$ match when $l_{cons} = \{\texttt{Cons}\}$ and $l.\texttt{Cons}.0 = h$, then the result is $l.\texttt{Cons}.0$
- l and Nil match when $l_{cons} = \{\texttt{Nil}\}$, then the result is 0.
- The result is $0 \cup_{\mathbb{Z}} l.\texttt{Cons}.0$

We can summarize the function $\texttt{hd} : l \to 0 \cup_{\mathbb{Z}} l.\texttt{Cons}.0$.

Example 2 - Recursive function

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

Example 2 - Recursive function

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

We initialize mult2 : $((l.\mathtt{Cons.0}, l.\mathtt{Cons.1}), l_{cons}) \rightarrow \bot$.

Example 2 - Recursive function

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

We initialize `mult2` : $((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons}) \rightarrow \perp$.

We iteratively analyze the body.

Example 2 - Recursive function

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

We initialize mult2 : $((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons}) \to \bot$.

We iteratively analyze the body.

Example 2 - Recursive function

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

We initialize $\texttt{mult2} : ((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons}) \rightarrow \bot$.

We iteratively analyze the body.

1. We analyze the pattern-matching:

Example 2 - Recursive function

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

We initialize $\texttt{mult2} : ((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons}) \to \bot$.

We iteratively analyze the body.

1. We analyze the pattern-matching:
   - With $l_{cons} = \{\texttt{Cons}\}$, $l.\texttt{Cons}.0 = h$, we get $((r.\texttt{Cons}.0 : 2 \times l.\texttt{Cons}.0 \cup^\sharp \bot, \bot), \{\texttt{Cons}\})$

Example 2 - Recursive function

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

We initialize mult2 : $((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons}) \to \bot$.

We iteratively analyze the body.

1. We analyze the pattern-matching:
    - With $l_{cons} = \{\texttt{Cons}\}$, $l.\texttt{Cons}.0 = h$, we get $((r.\texttt{Cons}.0 : 2 \times l.\texttt{Cons}.0 \cup^{\sharp} \bot, \bot), \{\texttt{Cons}\})$
    - With $l_{cons} = \{\texttt{Nil}\}$, we get $((r.\texttt{Cons}.0 : \bot, \bot), \{\texttt{Nil}\})$

Example 2 - Recursive function

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

We initialize mult2 : $((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons}) \rightarrow \bot$.

We iteratively analyze the body.

1. We analyze the pattern-matching:
   - With $l_{cons} = \{\texttt{Cons}\}$, $l.\texttt{Cons}.0 = h$, we get $((r.\texttt{Cons}.0 : 2 \times l.\texttt{Cons}.0 \cup^\sharp \bot, \bot), \{\texttt{Cons}\})$
   - With $l_{cons} = \{\texttt{Nil}\}$, we get $((r.\texttt{Cons}.0 : \bot, \bot), \{\texttt{Nil}\})$
   - By join, we have : $((r.\texttt{Cons}.0 : 2 \times l.\texttt{Cons}.0, \bot), \{\texttt{Cons}, \texttt{Nil}\})$

Example 2 - Recursive function

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

We initialize $\texttt{mult2} : ((l.\texttt{Cons}.0, l.\texttt{Cons}.1), l_{cons}) \rightarrow \bot$.

We iteratively analyze the body.

1. $\texttt{mult2} : l \rightarrow ((r.\texttt{Cons}.0 : 2 \times l.\texttt{Cons}.0, \bot), \{\texttt{Cons}, \texttt{Nil}\})$

Example 2 - Recursive function

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

We initialize $\texttt{mult2} : ((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons}) \to \bot$.

We iteratively analyze the body.

1. $\texttt{mult2} : l \to ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \bot), \{\texttt{Cons}, \texttt{Nil}\})$
2. By analyzing the pattern again, we get:
   $((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\})$

Example 2 - Recursive function

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

We initialize mult2 : $((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons}) \rightarrow \bot$.

We iteratively analyze the body.

1. mult2 : $l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \bot), \{\texttt{Cons}, \texttt{Nil}\})$
2. By analyzing the pattern again, we get:
   $((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}\}) \cup^\sharp ((\bot, \bot), \{\texttt{Nil}\})$

Example 2 - Recursive function

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

We initialize $\texttt{mult2} : ((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons}) \rightarrow \bot$.

We iteratively analyze the body.

1. $\texttt{mult2} : l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \bot), \{\texttt{Cons}, \texttt{Nil}\})$
2. $\texttt{mult2} : l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\})$

Example 2 - Recursive function

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

We initialize mult2 : $((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons}) \rightarrow \bot$.
We iteratively analyze the body.

1. mult2 : $l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \bot), \{\texttt{Cons}, \texttt{Nil}\})$
2. mult2 : $l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\})$
3. By analyzing the pattern again, we get the same result: this is a fixpoint.

Example 2 - Recursive function

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

We initialize $\texttt{mult2} : ((l.\texttt{Cons.0}, l.\texttt{Cons.1}), l_{cons}) \rightarrow \bot$.
We iteratively analyze the body.

1. $\texttt{mult2} : l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \bot), \{\texttt{Cons}, \texttt{Nil}\})$
2. $\texttt{mult2} : l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\})$
3. By analyzing the pattern again, we get the same result: this is a fixpoint.

Then $\texttt{mult2} : l \rightarrow ((r.\texttt{Cons.0} : 2 \times l.\texttt{Cons.0}, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\})$.

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

$$x : ((\texttt{x.Cons.0:[1, 2]}, \texttt{x.Cons.1:\{Nil, Cons\}}), \{\texttt{Cons}\})$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

$$\begin{cases} \text{mult2: } \text{l} \rightarrow ((r.\texttt{Cons}.0 : 2 \times l.\texttt{Cons}.0, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}) \\ \text{x: } (([1,2], \{\text{Cons, Nil}\}), \{\text{Cons}\}) \end{cases}$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```

$$\begin{cases} \text{mult2: } \mathsf{l} \rightarrow ((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}) \\ \text{x: } (([1,2], \{\text{Cons, Nil}\}), \{\text{Cons}\}) \end{cases}$$

$$\implies r_1 : \ (r_1.\text{Cons}.0 : [2,4], \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\})$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (r1) <= 4)
```

$$\begin{cases} \text{mult2: } \mathsf{l} \rightarrow ((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}) \\ \text{x: } (([1,2], \{ \text{Cons}, \text{Nil}\}), \{ \text{Cons}\}) \end{cases}$$

$$\Longrightarrow r_1 : \; (r_1.\text{Cons}.0 : [2,4], \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\})$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (r1) <= 4)
```

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert( hd (r1)  <= 4)
```

$$\begin{cases} \mathsf{hd}:\ \mathsf{l} \to 0 \cup_{\mathbb{Z}} \mathsf{l}.\mathtt{Cons}.0 \\ \mathsf{r}_1 :\ ([2,4], \{\mathtt{Cons}, \mathtt{Nil}\}), \{\mathtt{Cons}, \mathtt{Nil}\}) \end{cases}$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (r1) <= 4)
```

$$\begin{cases} \text{hd: } \mathsf{l} \rightarrow 0 \cup_{\mathbb{Z}} l.\texttt{Cons}.0 \\ \text{r}_1 : \ ([2,4], \{\texttt{Cons}, \texttt{Nil}\}), \{\texttt{Cons}, \texttt{Nil}\}) \end{cases} \implies r : [0, 4]$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(r <= 4)
```

$$\begin{cases} \text{hd: } \mathsf{l} \to 0 \cup_{\mathbb{Z}} l.\mathtt{Cons}.0 \\ \mathsf{r}_1 : \ ([2,4], \{\mathtt{Cons}, \mathtt{Nil}\}), \{\mathtt{Cons}, \mathtt{Nil}\}) \end{cases} \implies r : [0,4]$$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(r <= 4)
```

- $r : [0, 4]$

Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(r <= 4)
```

- $r : [0, 4]$

✓ The assertion is proved!

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:

- Aiming at simplifying the design of static analyzers

MOPSA (Modular Open Platform for Static Analysis)

A modular and multi-language open-source platform:

- Aiming at simplifying the design of static analyzers
- Implementing relational abstract domains

MOPSA (Modular Open Platform for Static Analysis)

A modular and multi-language open-source platform:

- Aiming at simplifying the design of static analyzers
- Implementing relational abstract domains
- Relying on cooperation and communication between them

MOPSA (Modular Open Platform for Static Analysis)

A modular and multi-language open-source platform:

- Aiming at simplifying the design of static analyzers
- Implementing relational abstract domains
- Relying on cooperation and communication between them
- Supporting subsets of C and Python

`https://gitlab.com/mopsa/mopsa-analyzer`

OCaml Analysis

We performed the following implementation steps:

- ✓ Injecting OCaml typed AST into MOPSA AST
- ✓ Designing domains for algebraic values and non-recursive functions
- ✓ Implementing transfer functions for all other constructs (let, type declarations, pattern-matching, etc.)

OCaml Analysis

We performed the following implementation steps:

- ✓ Injecting OCaml typed AST into MOPSA AST
- ✓ Designing domains for algebraic values and non-recursive functions
- ✓ Implementing transfer functions for all other constructs (let, type declarations, pattern-matching, etc.)

It represents about 2000 lines of OCaml, tested on a few dozens of toy programs, and still has limitations:

- ✗ Implementation to complete (recursive functions)
- ✗ Polymorphism, Higher-order
- ✗ Impure features (mutable arrays, references)
- ✗ But also modules, functors...

Experimental results

Program	Lines	Time (s)
list.ml	4	0.003
tree.ml	2	0.005
match.ml	6	0.004
match_alarm.ml	6	0.005
match_error.ml	6	0.004
add.ml	3	0.004

Figure 1: Execution time on a few toy programs

Conclusion

Conclusion

- A static value analysis for a first-order functional language
- Design of a relational domain for algebraic values
- Implementation into MOPSA platform
- Paving the way towards an analyzer for a higher-order functional language

Further works

- Add support for higher order and polymorphism
- Extend to an impure fragment
- Make the implementation scalable!
- Towards higher-order information: length, depth, or even more sophisticated properties (sort, balance)

Thank you for your attention

Polymorphism and higher-order

For polymorphism, we may:

- Analyze the function for each type instance encountered
- Develop equality and inequality domains for polymorphic data

For higher-order, we may:

- Analyze the function for each function summary in argument
- For numeric information, generalize the current analysis (functions and values are just points of numeric domains)

But we would need new methods for structural information on algebraic values.

Impure features

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

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