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

Milla Valnet\textsuperscript{1,3}, Raphaël Monat\textsuperscript{2}, Antoine Miné\textsuperscript{3}

JFLA 2023

\textsuperscript{1}ENS Ulm, \textsuperscript{2}Inria Lille, \textsuperscript{3}Sorbonne Université
Introduction
Software bugs can be costly... and testing is not enough!
Abstract interpretation

$S[\text{prog}]$

$\mathcal{D}$ (concrete)

Bad states
Abstract interpretation

\[ S[[\text{prog}]] \]

Bad states

\[ D \text{ (concrete)} \]

\[ S#[[\text{prog}]] \]

Bad states

\[ D^# \text{ (abstract)} \]

✓ Program correct
Abstract interpretation

$D$ (concrete)

$S \left[ \text{prog} \right]$

$\gamma$

$\mathcal{D}^\#$ (abstract)

$S^\# \left[ \text{prog} \right]$

$\times$ True alarm
Abstract interpretation

\[ S[[\text{prog}]] \]

\( D \) (concrete)

Bad states

\[ S^#[[\text{prog}]] \]

\( D^# \) (abstract)

\( \times \) False alarm (too unprecise)
x = 0 ; y = 1 ;
while (y < 1000) {
    if (rand(0,1) == 0) { x++; } else { x--; } ;
y++;
}
Domains and relationality

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

**Interval domain:**

- \( D = \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \), \( D^\#: \mathbb{R} \)
  - \( D^\# = \{ \mathbb{R} \}^2 \)
Domains and relationality

\[ x = 0 ; \ y = 1 ; \]

\[ \text{while} \ (y < 1000)\{
    \text{if} \ (\text{rand}(0,1) == 0) \ {x++;} \ \text{else} \ {x--;}\; ;
    y++; \}
\]

Interval domain:

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

\begin{verbatim}
x = 0 ; y = 1 ;
while (y < 1000){
    if (rand(0,1) == 0) { x++; } else { x--; } ;
y++;
}
\end{verbatim}

Interval domain:

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

Polyhedra domain:

- \( \mathcal{D} = \mathcal{P}(\mathbb{Z}^n) \), \( \mathcal{D}^\# = \{ \bigwedge_{j \leq m} (\sum_{i=1}^{n} a_{i,j} V_i \geq \beta_j) \} \)
x = 0 ; y = 1 ;
while (y < 1000){
    if (rand(0,1) == 0) { x++; } else { x--; } ;
    y++; }

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

Polyhedra domain : 
- \( \mathcal{D} = \mathcal{P}(\mathbb{Z}^n) \), \( \mathcal{D}^\# = \{ \bigwedge_{j \leq m} (\sum_{i=1}^{n} a_{i,j} V_i \geq \beta_j) \} \)
- \( y = 1000, -y < x < y \)
Functional programming
Functional programming

New features to handle!
Functional programming

New features to handle!

OCaml

Haskell

Clojure
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

```ocaml
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?
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
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
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
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
type list = Cons of int * list | Nil
Algebraic Data Types

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

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

definition of list type

definition of x

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

let y = Nil

let z = Cons(4, x)
```
type list = Cons of int * list | Nil

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

let y = Nil

let z = Cons(4, x)

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})
Algebraic Data Types

```haskell
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

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

We choose as an abstract domain:

\[ D_t = \prod_{1 \leq i \leq n} \prod_{1 \leq j \leq n_i} D_{i,j}^\perp \times \mathcal{P}(C) \]

- 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.
Transfer function: pattern-matching

\[
\text{match } e \text{ with } \mid p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n
\]

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

\[
\text{match } e \text{ with } | \ p1 \rightarrow e1 \ | \ ... \ | \ pn \rightarrow en
\]

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 \textit{flow sensitive} and able to handle \texttt{when} clauses.
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.
match \texttt{Cons(1, Nil)} with
| \texttt{Cons(h, q)} \rightarrow h
| \texttt{Nil} \rightarrow 0

- \texttt{Cons(1, Nil)} and \texttt{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
A function is abstracted as a relation between the inputs and the output.
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 $\top$. 
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
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 $x$.
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: $f : x \rightarrow x.0 + x.1$. 

```
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: $f : x \to x.0 + x.1$.

For application, we instantiate the input variables in the relation abstracting $f$ by the abstraction of arguments.
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 $x.0 + x.1 = 42 + 12 = 54$. 
For recursive functions, their concrete semantics is computed with a fixpoint, so their abstract semantics will use iterations, with a widening application.

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

```ocaml
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 \( \top \).
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 \( f : x_1 \rightarrow \ldots \rightarrow x_n \rightarrow \perp \) with \( x_i \) to \( \top \).
- We evaluate the body of the function with this hypothesis and get a more precise abstraction for \( f \).
Recursivity

```ocaml
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.
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 \( f \).
- 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

```ml
let hd = fun l -> match l with
    | Cons(h,q) -> h
    | Nil        -> 0
```
Example 1 - Non recursive function

```ml
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 \( l : ((l\cdot\text{Cons}.0, l\cdot\text{Cons}.1), l_{\text{cons}}) \).
Example 1 - Non recursive function

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

We abstract the right hand side.

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

- \( l \) and \( \text{Cons}(h, q) \) match when \( l_{cons} = \{ \text{Cons} \} \) and \( l.\text{Cons}.0 = h \), then the result is \( l.\text{Cons}.0 \)
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 $l : ((l.\text{Cons}.0, l.\text{Cons}.1), l_{cons})$.

- $l$ and $\text{Cons}(h, q)$ match when $l_{cons} = \{\text{Cons}\}$ and $l.\text{Cons}.0 = h$, then the result is $l.\text{Cons}.0$
- $l$ and $\text{Nil}$ match when $l_{cons} = \{\text{Nil}\}$, 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 \( l : ((l.\text{Cons}.0, l.\text{Cons}.1), l_{\text{cons}}) \).

- \( l \) and \( \text{Cons}(h, q) \) match when \( l_{\text{cons}} = \{ \text{Cons} \} \) and \( l.\text{Cons}.0 = h \), then the result is \( l.\text{Cons}.0 \)
- \( l \) and \( \text{Nil} \) match when \( l_{\text{cons}} = \{ \text{Nil} \} \), then the result is 0.
- The result is \( 0 \cup \mathbb{Z} \) \( l.\text{Cons}.0 \)
Example 1 - Non recursive function

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

We abstract the right hand side.

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

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

We can summarize the function \( \text{hd} : l \rightarrow 0 \cup_{\mathbb{Z}} l.\text{Cons}.0 \).
Example 2 - Recursive function

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

We initialize mult2 : ((/Cons.0, /Cons.1), l_{cons}) → ⊥.
Example 2 - Recursive function

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

We initialize $\text{mult2} : (\langle \text{Cons}.0, \text{Cons}.1 \rangle, \text{l}_{\text{cons}}) \rightarrow \bot$.

We iteratively analyze the body.
Example 2 - Recursive function

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

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

We iteratively analyze the body.
Example 2 - Recursive function

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

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

We iteratively analyze the body.

1. We analyze the pattern-matching:
Example 2 - Recursive function

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

We initialize \( \text{mult2} : ((l.\text{Cons}.0, l.\text{Cons}.1), \text{lcons}) \rightarrow \bot \).

We iteratively analyze the body.

1. We analyze the pattern-matching:
   - With \( \text{lcons} = \{\text{Cons}\} \), \( l.\text{Cons}.0 = h \), we get
     \[ ((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0 \cup \# \bot, \bot), \{\text{Cons}\}) \]

Example 2 - Recursive function

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

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

We iteratively analyze the body.

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

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

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

We iteratively analyze the body.

1. We analyze the pattern-matching:
   - With $l_{cons} = \{\text{Cons}\}$, $l.\text{Cons}.0 = h$, we get
     $((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0 \uplus \perp, \perp), \{\text{Cons}\})$
   - With $l_{cons} = \{\text{Nil}\}$, we get $((r.\text{Cons}.0 : \perp, \perp), \{\text{Nil}\})$
   - By join, we have : $((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \perp), \{\text{Cons, Nil}\})$
let rec mult2 = fun l -> match l with
  | Cons(h,q) -> Cons(2*h, mult2 q)
  | Nil -> Nil

We initialize mult2 : ((l.Cons.0, l.Cons.1), l_cons) \rightarrow \bot.
We iteratively analyze the body.

1. mult2 : l \rightarrow ((r.Cons.0 : 2 \times l.Cons.0, \bot), \{Cons, Nil\})
Example 2 - Recursive function

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

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

We iteratively analyze the body.

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

2. By analyzing the pattern again, we get:
   $((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}\})$
Example 2 - Recursive function

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

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

We iteratively analyze the body.

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

2. By analyzing the pattern again, we get:
   $((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}\}) \cup^\ast((\bot, \bot), \{\text{Nil}\})$
Example 2 - Recursive function

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

We initialize `mult2 : ((l.Cons.0, l.Cons.1), l_cons) → ⊥`.

We iteratively analyze the body.

1. `mult2 : / → ((r.Cons.0 : 2 × l.Cons.0, ⊥), {Cons, Nil})`
2. `mult2 : / → ((r.Cons.0 : 2 × l.Cons.0, {Cons, Nil}), {Cons, Nil})`

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

We iteratively analyze the body.

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

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

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

We iteratively analyze the body.

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

Then \( \text{mult2} : l \rightarrow ((r.\text{Cons.0} : 2 \times l.\text{Cons.0}, \{\text{Cons, Nil}\}), \{\text{Cons, Nil}\}). \)
Example 3

```plaintext
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```
Example 3

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

```
x : ([(x.Cons.0 : [1, 2]), x.Cons.1 : {Nil, Cons}), {Cons}])
```
Example 3

```
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```
Example 3

```plaintext
let x = Cons(1, Cons(2, Nil)) in
assert(hd (mult2 x) <= 4)
```
Example 3

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

\[
\begin{aligned}
\text{mult2: } l & \rightarrow ((r.\text{Cons.0} : 2 \times l.\text{Cons.0}, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\}) \\
x: (([1,2], \{\text{Cons}, \text{Nil}\}), \{\text{Cons}\})
\end{aligned}
\]
Example 3

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

\[
\begin{align*}
\text{mult2}: & \quad l \rightarrow ((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\}) \\
x: & \quad (\{1,2\}, \{\text{Cons}, \text{Nil}\}, \{\text{Cons}\})
\end{align*}
\]

\[\Rightarrow r_1 : (r_1.\text{Cons}.0 : [2,4], \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\})\]
Example 3

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

\[
\begin{align*}
\text{mult2: } & l \mapsto ((r.\text{Cons}.0 : 2 \times l.\text{Cons}.0, \{\text{Cons, Nil}\}), \{\text{Cons, Nil}\}) \\
x: & (\{[1,2], \{\text{Cons, Nil}\}\}, \{\text{Cons}\}) \\
\Rightarrow & r_1: (r_1.\text{Cons}.0 : [2, 4], \{\text{Cons, Nil}\}), \{\text{Cons, Nil}\})
\end{align*}
\]
Example 3

```plaintext
let x = Cons(1, Cons(2, Nil)) in
assert(hd (r1) <= 4)
```
Example 3

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

{ hd: l → 0 ⋃ \mathbb{Z} \setminus .Cons.0
  r1 : ([2,4],{Cons,Nil}), {Cons,Nil})
```
Example 3

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

\[
\begin{align*}
\text{hd: } & l \rightarrow 0 \cup \mathbb{Z} / .\text{Cons.0} \\
n_1: & ([2, 4], \{\text{Cons, Nil}\}, \{\text{Cons, Nil}\}) \quad \Rightarrow \quad r: [0, 4]
\end{align*}
\]
Example 3

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

\[
\begin{cases}
  \text{hd: } l \rightarrow 0 \cup \mathbb{Z} \setminus \text{Cons.0} \\
  r_1 : ([2, 4], \{\text{Cons, Nil}\}, \{\text{Cons, Nil}\}) \quad \implies \quad r : [0, 4]
\end{cases}
\]
Example 3

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

• $r : [0, 4]$
Example 3

```plaintext
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:

https://gitlab.com/mopsa/mopsa-analyzer
MOPSA (Modular Open Platform for Static Analysis)

A modular and multi-language open-source platform:

- Aiming at simplifying the design of static analyzers

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

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>list.ml</td>
<td>4</td>
<td>0.003</td>
</tr>
<tr>
<td>tree.ml</td>
<td>2</td>
<td>0.005</td>
</tr>
<tr>
<td>match.ml</td>
<td>6</td>
<td>0.004</td>
</tr>
<tr>
<td>match_alarm.ml</td>
<td>6</td>
<td>0.005</td>
</tr>
<tr>
<td>match_error.ml</td>
<td>6</td>
<td>0.004</td>
</tr>
<tr>
<td>add.ml</td>
<td>3</td>
<td>0.004</td>
</tr>
</tbody>
</table>

**Figure 1:** Execution time on a few toy programs
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
- 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 inputs to every functions
- Identify functions where impure variables don’t escape to reduce the cost