# Combinations of Reusable Abstract Domains for a Multilingual Static Analysis

Antoine Miné  Abdelraouf Ouadjaout  Matthieu Journault
Raphaël Monat  Aymeric Fromherz

APR team
LIP6
Sorbonne Université
Paris, France

13/07/2019

# Sound, semantic, static analysis

**Goal:** program verification by static analysis

source

```
int search(int* t, int n) {
  int i;
  for (i=0; i < n; i++) {

    if (t[i]) break;
  }

  return t[i];
}
```

- work directly on the source code

# Sound, semantic, static analysis

**Goal:** program verification by static analysis

source

```
int search(int* t, int n) {
  int i;
  for (i=0; i < n; i++) {
    // 0 ≤ i < n
    if (t[i]) break;
  }
  // (0 ≤ i ≤ n) ∨ (n < 0)
  return t[i];
}
```

- work directly on the source code
- infer properties on program executions
- automatically (cost effective)
- by constructing dynamically a semantic abstraction of the program

# Sound, semantic, static analysis

**Goal:** program verification by static analysis

source

```
int search(int* t, int n) {
  int i;
  for (i=0; i < n; i++) {
    // 0 ≤ i < n
    if (t[i]) break;          ✓
  }
  // (0 ≤ i ≤ n) ∨ (n < 0)
  return t[i];                ✗
}
```

- work directly on the source code
- infer properties on program executions
- automatically (cost effective)
- by constructing dynamically a semantic abstraction of the program
- deduce program correctness or raise alarms
  implicit specification: absence of RTE; or user-defined properties: contracts
- using approximate abstractions (efficient, but possible false alarms)
- soundly (no false positive)

# Modular Open Platform for Static Analysis

Goal: build a static analysis platform (in OCaml)
for research and education in abstract interpretation

- basic support for common abstractions and C analysis
- easy to extend to support novel abstractions and languages
- as few limitations as possible
  (simple abstractions should be easy, complex ones should be possible)
- try new ideas on how to engineer an abstract interpreter
- reuse more, experiment more easily

In this talk :

- work in progress. . .
- more engineering than science. . .

# Overview:

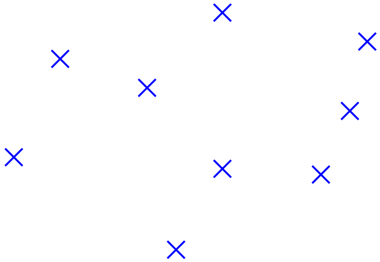
1. static analysis by Abstract Interpretation
2. MOPSA framework and desing choices
3. application to C analysis
   - analysis of run-time errors in C
   - stub language to model C libraries
4. application to Python analysis
   - value analysis for Python
   - type analysis for Python

# Abstract interpretation primer

# Abstract interpretation

Abstract interpretation: theory of the approximation of program semantics

Principle: be tractable by reasoning at an abstract level
keep soundness by considering over-approximations

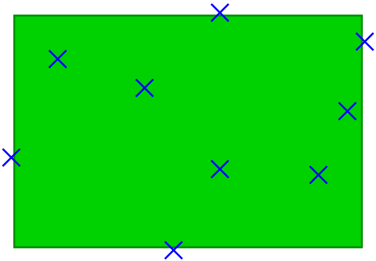


concrete executions $\mathcal{D}$ : $\{(0,3),(5.5,0),(12,7),\ldots\}$ (not practical)

# Abstract interpretation

Abstract interpretation: theory of the approximation of program semantics

Principle: be tractable by reasoning at an abstract level
keep soundness by considering over-approximations

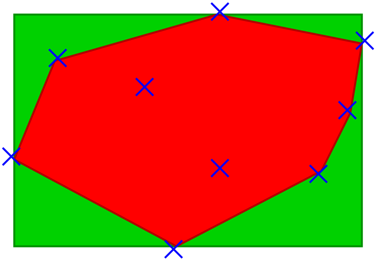


concrete executions $\mathcal{D}$ : $\{(0,3),(5.5,0),(12,7),\ldots\}$ (not practical)
box domain $\mathcal{D}^\sharp_b$ : $X \in [0,12] \wedge Y \in [0,8]$ (linear cost)

# Abstract interpretation

Abstract interpretation: theory of the approximation of program semantics

Principle: be tractable by reasoning at an abstract level
keep soundness by considering over-approximations

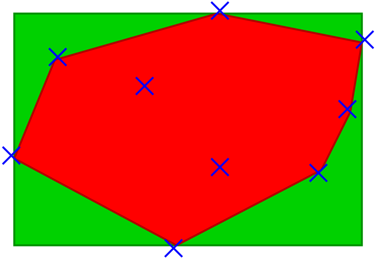


concrete executions $\mathcal{D}$ : $\{(0,3),(5.5,0),(12,7),\ldots\}$ (not practical)
box domain $\mathcal{D}^\sharp_b$ : $X \in [0,12] \land Y \in [0,8]$ (linear cost)
polyhedron domain $\mathcal{D}^\sharp_p$ : $6X + 11Y \geq 33 \land \cdots$ (exponential cost)

# Abstract interpretation

Abstract interpretation: theory of the approximation of program semantics

Principle: be tractable by reasoning at an abstract level
keep soundness by considering over-approximations



| | | |
|---|---|---|
| concrete executions $\mathcal{D}$ : | $\{(0,3),(5.5,0),(12,7),\ldots\}$ | (not practical) |
| box domain $\mathcal{D}^\sharp_b$ : | $X \in [0,12] \wedge Y \in [0,8]$ | (linear cost) |
| polyhedron domain $\mathcal{D}^\sharp_p$ : | $6X + 11Y \geq 33 \wedge \cdots$ | (exponential cost) |

Each abstract element represents a concrete element, via $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$

# Abstract computations

Define an interpretation of atomic statements in the abstract domain.

For each $\mathbb{S}[\![ s ]\!] : \mathcal{D} \to \mathcal{D}$, provide $\mathbb{S}^\sharp[\![ s ]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$.

Compose interpretations to analyze full programs.

Replace $\mathbb{S}[\![ s_1 ]\!] \circ \ldots \circ \mathbb{S}[\![ s_n ]\!]$ with $\mathbb{S}^\sharp[\![ s_1 ]\!] \circ \ldots \circ \mathbb{S}^\sharp[\![ s_n ]\!]$.

# Abstract computations

Define an interpretation of atomic statements in the abstract domain.

For each $\mathbb{S}[\![ s ]\!] : \mathcal{D} \to \mathcal{D}$, provide $\mathbb{S}^\sharp[\![ s ]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$.

Compose interpretations to analyze full programs.

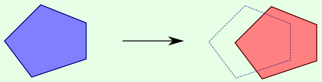
Replace $\mathbb{S}[\![ s_1 ]\!] \circ \ldots \circ \mathbb{S}[\![ s_n ]\!]$ with $\mathbb{S}^\sharp[\![ s_1 ]\!] \circ \ldots \circ \mathbb{S}^\sharp[\![ s_n ]\!]$.

## Polyhedra operators

### Assignments

• `X ← X + 1` •

translation



# Abstract computations

Define an interpretation of atomic statements in the abstract domain.

For each $\mathbb{S}[\![\, s \,]\!] : \mathcal{D} \to \mathcal{D}$, provide $\mathbb{S}^\sharp[\![\, s \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$.

Compose interpretations to analyze full programs.

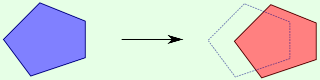
Replace $\mathbb{S}[\![\, s_1 \,]\!] \circ \ldots \circ \mathbb{S}[\![\, s_n \,]\!]$ with $\mathbb{S}^\sharp[\![\, s_1 \,]\!] \circ \ldots \circ \mathbb{S}^\sharp[\![\, s_n \,]\!]$.

## Polyhedra operators

### Assignments

• `X` ← `X + 1` •

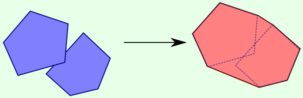
translation



### Branches: join

**if** ⋯ **then** ⋯ • **else** ⋯ • **fi** •

convex hull



# Abstract computations

Define an interpretation of atomic statements in the abstract domain.

For each $\mathbb{S}[\![\, s \,]\!] : \mathcal{D} \to \mathcal{D}$, provide $\mathbb{S}^\sharp[\![\, s \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$.

Compose interpretations to analyze full programs.

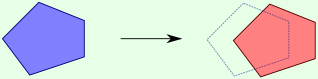
Replace $\mathbb{S}[\![\, s_1 \,]\!] \circ \ldots \circ \mathbb{S}[\![\, s_n \,]\!]$ with $\mathbb{S}^\sharp[\![\, s_1 \,]\!] \circ \ldots \circ \mathbb{S}^\sharp[\![\, s_n \,]\!]$.

## Polyhedra operators

### Assignments

• `X ← X + 1` •

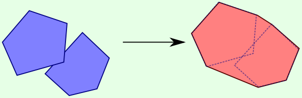
translation



### Branches: join

**if** ··· **then** ··· • **else** ··· • **fi** •

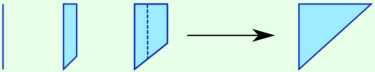
convex hull



### Loops: inductive invariants

**while** • ··· **do** ··· **done**

iteration with widening $\triangledown$



# A more complex example

source

```
int main( int argc, char *argv[]) {
  int i = 0;
  for (char **p = argv; *p; p++) {
    strlen(*p); // valid string
    i++; // no overflow
  }
  return 0;
}
```

Numeric:

$\texttt{argc} \in [1, \texttt{maxint}]$
$\text{size}(\texttt{argv}) = \texttt{argc} + 1$
$\text{size}(\texttt{@}) \in [1, \texttt{maxsize}]$
$0 \leq \text{offset}(\texttt{p}) \leq \text{size}(\texttt{argv}) - 1$
$\text{offset}(\texttt{p}) = \texttt{i}$

Memory:

argc: variable
argv: variable
p: variable
i: variable
@: summary block

Pointers:

$\texttt{argv}[0 \ldots \texttt{argc} - 1] \mapsto \{\texttt{@}\}$
$\texttt{argv}[\texttt{argc}] \mapsto \{\texttt{NULL}\}$
$\texttt{p} \mapsto \{\texttt{argv}\}$

Strings:

$\exists k \in [0 \ldots \texttt{size}(\texttt{@}) - 1] : \texttt{@}[k] = 0$

# A more complex example

**source**

```
int main( int argc, char *argv[]) {
  int i = 0;
  for (char **p = argv; *p; p++) {
    strlen(*p); // valid string
    i++; // no overflow
  }
  return 0;
}
```

Numeric:

$\texttt{argc} \in [1, \texttt{maxint}]$
$\text{size}(\texttt{argv}) = \texttt{argc} + 1$

Memory:

argc: variable
argv: variable

**Combining domains**

Combination of domains for different types (number, pointers, ...)
and different properties (relational domains for inductive invariants)
that can be composed and can communicate.

$\texttt{argv}[\texttt{argc}] \mapsto \{\texttt{NULL}\}$
$\texttt{p} \mapsto \{\texttt{argv}\}$

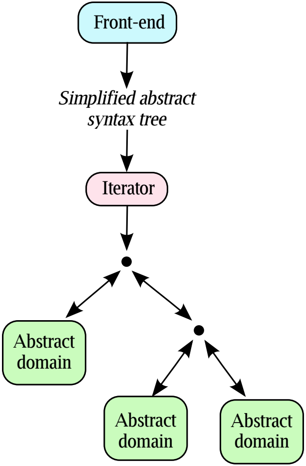
# Classic analyzer design

A classic analyzer (Astrée, Frama-C) has:

- one or several front-ends (one per language)
- a simplified target analysis language
  low-level: C light, JVM, LLVM bitcode, Jimple, etc.
- an iterator
- a tree-structure combination of abstractions
  with layered abstraction signatures
  heap / blocks / scalar values / numeric abstractions

Pros and cons:

+ fewer language constructs to abstract
+ easy to reuse domains across languages
− static simplifications in the front-end
  → cripple precision before the analysis
− restrictions to domain composition
  → no reuse across abstraction layers



# MOPSA Framework

# MOPSA characteristics

## MOPSA:

- unified AST for programs: high-level, extensible, multi-language
- lowering of complex statements dynamically, during analysis

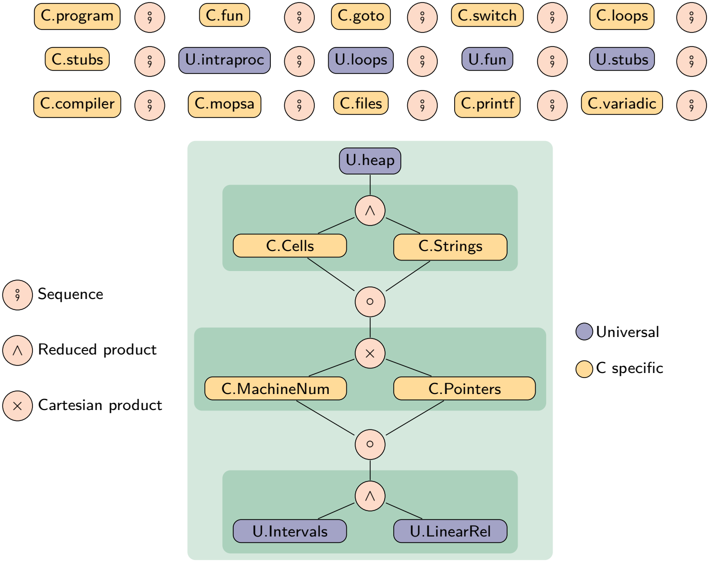
- common signature for all abstract domains
- domain communications, access to preconditions, reductions
- domain organisation in DAGs, sharing abstract information

- more general environment abstractions, handling optional variables

## Languages:

- toy-language "universal" (demonstration, factoring abstractions)
- full C language
- C function specification language (similar to ACSL / JML)
- large subset of Python 3
- language subsets (struct-less, dereference-less, pointer-less, pure arithmetic, etc.)

# C value analyzer configuration



# Extensible AST: Universal loops

We use extensible types and distributed iterators.

E.g., universal is a toy-language with only simple while loops

- extend stmt_kind with AST fragments

Universal.Ast

```
type stmt_kind += S_while of expr * stmt
```

# Extensible AST: Universal loops

We use extensible types and distributed iterators.

E.g., universal is a toy-language with only simple while loops

- extend stmt_kind with AST fragments

Universal.Ast

```
type stmt_kind += S_while of expr * stmt
```

- define an iterator exec for this fragment
  - handles some AST fragments, defaults to None for others
  - defined by induction on the AST
    by calling recursively the overall iterator man

$$\mathbb{S}^\sharp[\![\, \texttt{while (e) s} \,]\!] X^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\, \neg e \,]\!] (\text{lfp}\, \lambda Y^\sharp.\, X^\sharp \cup \mathbb{S}^\sharp[\![\, s \,]\!] \circ \mathbb{S}^\sharp[\![\, e \,]\!] Y^\sharp)$$

Universal.Iterators.Loops

```
let exec stmt man flow =
    match stmt_kind stmt with
    | S_while (cond, body) ->
       let i = lfp (fun f -> Flow.join f (man.exec (S_assume cond) f |>
                                          man.exec body)) flow
       Some (man.exec (S_assume (E_not cond) i))
    | _ ->
       None (* pass-through *)
```

# Extensible AST: C and Python loops

C AST

```
type stmt_kind += S_c_for of stmt * expr option * expr option * stmt
               |  S_c_do_while of stmt * expr
```

Python AST

```
type stmt_kind += S_py_for of expr * expr * stmt * stmt
               |  S_py_while of expr * stmt * stmt
```

- preserve the high-level AST of the source languages
- reuse universal AST when possible (no S_c_while)

# Extensible AST: C and Python loops

C AST

```
type stmt_kind += S_c_for of stmt * expr option * expr option * stmt
               |  S_c_do_while of stmt * expr
```

Python AST

```
type stmt_kind += S_py_for of expr * expr * stmt * stmt
               |  S_py_while of expr * stmt * stmt
```

- preserve the high-level AST of the source languages
- reuse universal AST when possible (no S_c_while)

C iterator

```
let exec stmt man flow = match stmt_kind stmt with
    | S_c_for (cond, body) ->
        let flow', body' = ... in Some (man.exec (S_while (cond, body')) flow')
```

- the iterator transforms the loops into a S_while universal loop and calls the overall iterator recursively
  $\Longrightarrow$ delegate the iteration strategy to universal (factor semantics)

# Extensible AST: C and Python loops

C AST

```
type stmt_kind += S_c_for of stmt * expr option * expr option * stmt
               |  S_c_do_while of stmt * expr
```

Python AST

```
type stmt_kind += S_py_for of expr * expr * stmt * stmt
               |  S_py_while of expr * stmt * stmt
```

- preserve the high-level AST of the source languages
- reuse universal AST when possible (no S_c_while)

C iterator

```
let exec stmt man flow = match stmt_kind stmt with
    | S_c_for (cond, body) ->
        let flow', body' = ... in Some (man.exec (S_while (cond, body')) flow')
```

- the iterator transforms the loops into a S_while universal loop and calls the overall iterator recursively
  $\Longrightarrow$ delegate the iteration strategy to universal (factor semantics)

The AST merges source languages and intermediate languages.

# Non-local control-flow

Handling of statements by induction on the syntax:

- $\mathbb{S}^\sharp[\![\, s_1\texttt{;}\ s_2 \,]\!] X^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\, s_2 \,]\!] \circ \mathbb{S}^\sharp[\![\, s_1 \,]\!]\, X^\sharp$
- $\mathbb{S}^\sharp[\![\, \texttt{if (e) s else t} \,]\!] X^\sharp \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![\, s \,]\!] \circ \mathbb{S}^\sharp[\![\, e \,]\!] X^\sharp) \cup^\sharp (\mathbb{S}^\sharp[\![\, t \,]\!] \circ \mathbb{S}^\sharp[\![\, \neg e \,]\!] X^\sharp)$

# Non-local control-flow

Handling of statements by induction on the syntax:

- $\mathbb{S}^\sharp[\![\, s_1;\ s_2 \,]\!] X^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\, s_2 \,]\!] \circ \mathbb{S}^\sharp[\![\, s_1 \,]\!]\, X^\sharp$
- $\mathbb{S}^\sharp[\![\, \texttt{if (e) s else t} \,]\!] X^\sharp \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![\, s \,]\!] \circ \mathbb{S}^\sharp[\![\, e \,]\!] X^\sharp) \cup^\sharp (\mathbb{S}^\sharp[\![\, t \,]\!] \circ \mathbb{S}^\sharp[\![\, \neg e \,]\!] X^\sharp)$
- adding gotos. . .

C AST

```
type stmt_kind += S_c_goto of string
               |  S_c_label of string
```

example

```
    x = 12;
    if (...) { x++; goto l1; }
    x = 99;
l1: return x;
```

How can we handle control flow that does not follow the AST structure?

$\Longrightarrow$ post-conditions are flows, containing several continuations.

# Flows as post-conditions

- environments $\mathcal{D}^\sharp$ abstract $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\text{memory state})$
- flows $\mathcal{F}^\sharp \stackrel{\text{def}}{=} \text{token} \to \mathcal{D}^\sharp$

**C goto flows**

```
type token += T_cur | T_goto of string
```

**example with flows**

```
    x = 12; [T_cur → 12]
    if (...) { x++; [T_cur → 13] goto l1; [T_goto l1 → 13] }
    [T_cur → 12, T_goto l1 → 13]
    x = 99;
    [T_cur → 99, T_goto l1 → 13]
l1: [T_cur → [13,99]] return x;
```

- $\mathbb{S}^\sharp[\![\,\texttt{goto l}\,]\!]X^\sharp \stackrel{\text{def}}{=} X^\sharp[cur \mapsto \bot, \texttt{l} \mapsto X^\sharp(cur) \cup^\sharp X^\sharp(\texttt{l})]$
- $\mathbb{S}^\sharp[\![\,\texttt{label l}\,]\!]X^\sharp \stackrel{\text{def}}{=} X^\sharp[cur \mapsto X^\sharp(cur) \cup X^\sharp(\texttt{l}), \texttt{l} \mapsto \bot]$

- also useful for `break`, `return`, exceptions, long jumps, generators
- *backward* jumps require fixpoint computations

# From universal numeric expressions. . .

Universal language integer expressions over $\mathbb{Z}$.

- $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \simeq \mathcal{P}(\mathbb{Z}^{|\mathcal{V}|})$
- $+$, $-$, $/$, $\times$ with mathematical semantics
  (no bit-size, no overflow, no wrap-around)
- natural setup for most numeric domains $\mathcal{D}^\sharp$
  (polyhedra, etc.)

# ...to C numeric expressions

C has machine integers, with bit-size and signedness.

- rewrite C numeric expressions into universal expressions
- evaluate with intervals to check for overflows (check the error flow)
  - if no overflow, $+_c = +_{universal}$
  - if overflow, add an explicit wrap operator (optionally signal an alarm)
- propagate the transformed expression to other domains (polyhedra)

# . . . to C numeric expressions

C has machine integers, with bit-size and signedness.

- rewrite C numeric expressions into universal expressions
- evaluate with intervals to check for overflows (check the error flow)
  - if no overflow, $+_c = +_{universal}$
  - if overflow, add an explicit wrap operator (optionally signal an alarm)
- propagate the transformed expression to other domains (polyhedra)

evaluation zones

```
type zone += Z_u_num | Z_c_scalar
```

C assignments to universal assignments

```
eval: zone -> exp -> man -> flow -> exp

let exec stmt man flow = match stmt with
  | S_assign(lval, rval) ->
    let lval' = man.eval ~zone:(Z_c_scalar, Z_u_num) lval flow
    and rval' = man.eval ~zone:(Z_c_scalar, Z_u_num) rval flow in
    man.exec ~zone:Z_u_num (S_Assign (lval',rval')) flow
```

- support for different interpretation zones ($\mathbb{Z}$, machine integers, etc.)

# ...to C numeric expressions

C has machine integers, with bit-size and signedness.

- rewrite C numeric expressions into universal expressions
- evaluate with intervals to check for overflows (check the error flow)
  - if no overflow, $+_{c} = +_{\mathit{universal}}$
  - if overflow, add an explicit wrap operator (optionally signal an alarm)
- propagate the transformed expression to other domains (polyhedra)

evaluation zones

```
type zone += Z_u_num | Z_c_scalar
```

C assignments to universal assignments

```
eval: zone -> exp -> man -> flow -> exp

let exec stmt man flow = match stmt with
  | S_assign(lval, rval) ->
    let lval' = man.eval ~zone:(Z_c_scalar, Z_u_num) lval flow
    and rval' = man.eval ~zone:(Z_c_scalar, Z_u_num) rval flow in
    man.exec ~zone:Z_u_num (S_Assign (lval',rval')) flow
```

- support for different interpretation zones ($\mathbb{Z}$, machine integers, etc.)

"evaluation" as dynamic rewriting into other expressions

# C pointers

- pointer value: $\mathcal{D} = \mathcal{P}(\mathcal{V}_{ptr} \rightarrow (\text{ base (variable, block)} \times \text{ offset (integer)}))$
- pointer arithmetic: byte-level offset arithmetic

Pointer abstraction $\mathcal{D}^\sharp \stackrel{\text{def}}{=} (\mathcal{V}_{ptr} \rightarrow \mathcal{P}(\mathcal{V})) \times \mathcal{N}um^\sharp$

- maintains internally the bases of each pointer
- create a numeric variable for each pointer to represent its offset
- "evaluate" pointer arithmetic into offset arithmetic
- delegate the offset abstraction to the numeric domains

```
char a[10] = "hello";
int i = _mopsa_rand(0,9);
char *p = &(a[i]); // ⟨p ↦ {a}, i ∈ [0,9] ∧ offset(p) = i⟩
```

$\Rightarrow$ infer relations between pointer offsets and numeric variables

# Expression evaluations into DNF

When transforming expressions, a domain can perform a case analysis:

- return a disjunction of expressions
- associate a subset of environments to each disjunct

eval signature

```
eval: zone -> exp -> man -> flow -> (exp * flow) DNF.t
```

# Expression evaluations into DNF

When transforming expressions, a domain can perform a case analysis:

- return a disjunction of expressions
- associate a subset of environments to each disjunct

eval signature

```
eval: zone -> exp -> man -> flow -> (exp * flow) DNF.t
```

Example:

evaluate `*(p+10)` in $X^\sharp$ where $p \in \{\text{NULL}, \&a, \&b\}$

return the disjunction:

$$(\mathit{error}, \mathbb{S}^\sharp[\![\, \mathit{assume\ base}(p) = \text{NULL} \,]\!] X^\sharp) \vee$$
$$(*(\&a + 10), \mathbb{S}^\sharp[\![\, \mathit{assume\ base}(p) = a \,]\!] X^\sharp) \vee$$
$$(*(\&b + 10), \mathbb{S}^\sharp[\![\, \mathit{assume\ base}(p) = b \,]\!] XX^\sharp)$$

- locality: disjunctions are merged at the end of the statement
- low coupling with other domains (eval mechanism)
- conjunctions are also possible thanks to reductions
  $\Longrightarrow$ use disjunctive normal forms

# Queries

Two scopes for data-types representing properties:

- abstract value: data-type private to each domain (locally available)
- queries: concrete data-type for communication (globally available)

# Queries

Two scopes for data-types representing properties:

- abstract value: data-type private to each domain (locally available)
- queries: concrete data-type for communication (globally available)

interval query

```
type _ query += Q_interval : expr -> IntItv.t with_bot query
```

- ability to evaluate any expression into an interval
- any domain can answer an interval query (intervals, polyhedra, etc.)
  request an interval and interpret its result

- concrete type with a lattice structure
  (the framework combines the answers from all domains)
- extensible, global data-type

# General domain reductions

## Application of queries:

Reduce the interval domain
using interval information from other domains.

global interval reduction

```
let reduce stmt man pre post =
  let vars = get_modified_vars stmt man pre in

  List.fold_left (fun post var ->
    let itv = man.get_value Itv.id var post in
    let itv' = man.ask (Q_interval (S_var var)) post in
    if I.subset itv itv' then post
    else man.set_value Itv.id var itv' post
  ) post vars
```

- applied after each statement
- focuses on the variables modified by the statement `stmt`
- independent from the domains, defined externally

# Heterogeneous environments

Instead of $\mathcal{P}(\mathcal{V} \to Val)$, abstract $\mathcal{P}(\mathcal{V} \rightharpoonup Val)$

- partial functions: not all variables have a value in an environment
- collect environment with heterogeneous supports

callee

```
int g;
void f(int* p) { •
  if (p) *p = g + 1;
}
```

caller 1

```
void g1() {
  int x;
  g(&x);
  // x == g + 1
}
```

caller 2

```
void g2() {
  int y;
  f(&y);
  // y == g + 1
}
```

Applications:

- merge stack contexts in inter-procedural analysis
- dynamic memory allocation (path-dependent allocation)
- optional variables (None in Python)

# Heterogeneous environment abstraction

How to lift $\mathcal{D}^\sharp_\mathcal{V}$ abstracting $\mathcal{P}(\mathcal{V} \to \mathit{Val})$ to $\mathcal{P}(\mathcal{V} \rightharpoonup \mathit{Val})$?

(classic solution: partitioning wrt. support $\to$ costly)

Use a single abstract element $(X^\sharp, L, U)$

- $L \subseteq U \subseteq \mathcal{V}$, lower and upper bounds on variables
- $X^\sharp \in \mathcal{D}^\sharp_U$ a single abstract element over $U$
- $\gamma(X^\sharp) \stackrel{\text{def}}{=} \{\, \rho_{|\mathcal{W}} \mid \rho \in \gamma_U(X^\sharp),\, L \subseteq \mathcal{W} \subseteq U \,\}$

Example:

$(0 \leq x \leq 10 \wedge y \leq x, \{x\}, \{x, y\})$

represents $\{\, [x \mapsto i] \mid i \in [0, 10] \,\} \cup \{\, [x \mapsto i, y \mapsto j] \mid i \in [0, 10], j \leq i \,\}$

# Heterogeneous environment abstraction

How to lift $\mathcal{D}^\sharp_\mathcal{V}$ abstracting $\mathcal{P}(\mathcal{V} \to Val)$ to $\mathcal{P}(\mathcal{V} \rightharpoonup Val)$?

(classic solution: partitioning wrt. support $\to$ costly)

Use a single abstract element $(X^\sharp, L, U)$

- $L \subseteq U \subseteq \mathcal{V}$, lower and upper bounds on variables
- $X^\sharp \in \mathcal{D}^\sharp_U$ a single abstract element over $U$
- $\gamma(X^\sharp) \stackrel{\text{def}}{=} \{\, \rho_{|\mathcal{W}} \mid \rho \in \gamma_U(X^\sharp),\ L \subseteq \mathcal{W} \subseteq U \,\}$

Example:

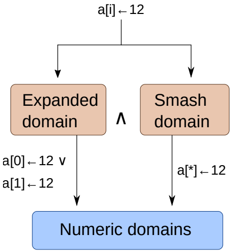
$(0 \leq x \leq 10 \wedge y \leq x,\ \{x\}, \{x, y\})$

represents $\{\, [x \mapsto i] \mid i \in [0, 10] \,\} \cup \{\, [x \mapsto i, y \mapsto j] \mid i \in [0, 10], j \leq i \,\}$

- any numeric domain $\mathcal{D}^\sharp$ can be lifted systematically
  (precise join and sound inclusion tests can be tricky)
- ability to represent relations involving optional variables
- all domains in MOPSA have this heterogeneous semantics

# Stacked domains: Issue

Powerful but complex interactions between reduction and evaluation.

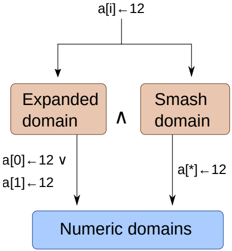



- both domains have a different view of the same concrete variables
- evaluation delegates the assignment independently for each domain
- the numeric domain collects both effects
  $\mathbb{S}^\sharp[\![\, eval_{cell}(\texttt{a[i]} \leftarrow 12) \,]\!] X^\sharp \wedge \mathbb{S}^\sharp[\![\, eval_{smash}(\texttt{a[i]} \leftarrow 12) \,]\!] X^\sharp$

This is not sound !

# Stacked domains: Solution

Powerful but complex interactions between reduction and evaluation.




Solution: domains inform other domains of side-effects (log and replay)

$\mathbb{S}^\sharp[\![ (\texttt{a[0]} \leftarrow 12 \vee \texttt{a[1]} \leftarrow 12); \texttt{a[*]} \leftarrow \top ]\!] X^\sharp \wedge$

$\mathbb{S}^\sharp[\![ \texttt{a[*]} \leftarrow 12; \texttt{a[0]} \leftarrow \top; \texttt{a[1]} \leftarrow \top ]\!] X^\sharp$

Other application : predicate domains, e.g.: $\forall i \in [0, n] : *(p + i) = *(q + i)$

- delegates the abstraction of $n$, $p$, $q$ to other domains (evaluation)
- sound reduction with cell and smash domains

# Application to C Analysis

# C analysis

- Clang front-end (C $\rightarrow$ OCaml faithful, high-level AST)
- support for integers, floats, pointers, structs, unions
- dynamic memory allocation with recency abstraction
- check for run-time errors
- limited support for the standard library
- inter-procedural analysis by inlining
  no recursivity
- no concurrency
- forward analysis only (no backward analysis)

Goal: a platform to help prototype new analyses on C codes

# Memory abstractions: cell domain

## Low-level memory abstraction

- handles structured types (arrays, struct, union)
- decompose the memory into scalar cells
  cell = (variable, offset, scalar-type)
- "evaluate" general C expressions into scalar expressions
  translate dereferences, structure and array accesses into cells

```
union { uint16 ax; struct { uint8 al; uint8 ah; } bytes; } regs;
regs.ax = 0xABCD; // regs[0 : 2] = 43981
x = reg.bytes.al; // x = 205
```

- supports type punning and pointer arithmetic
- represented in expansion (one cell per offset) or smashed (offset-insensitive cell)
- recency abstraction for dynamic allocation
  distinguish the most recent allocation, with strong update
  from a summary allocation, with weak update
  at each allocation site

# Memory abstractions: C strings

Domain to analyze low-level C string manipulation [SAS'18]

string copy

```
char *p = dst, *q = src;
while (*q != '\0') { *p = *q; p++; q++; }
*p = '\0';
```

- for each buffer $B$, remember the allocated size : $a_B$
- and the position of the first '\0' : $l_B$
- delegate the abstraction of $a_B$, $l_B$ by evaluation
  - evaluation to DNF is very useful for case analysis
  - infer relations between length, indices, offsets
- reduction with cell abstractions

# Memory abstractions: C strings

Domain to analyze low-level C string manipulation [SAS'18]

string copy

```
char *p = dst, *q = src;
while (*q != '\0') { *p = *q; p++; q++; }
*p = '\0';
```

- for each buffer $B$, remember the allocated size : $a_B$
- and the position of the first '\0' : $l_B$
- delegate the abstraction of $a_B$, $l_B$ by evaluation
  - evaluation to DNF is very useful for case analysis
  - infer relations between length, indices, offsets
- reduction with cell abstractions

Result: we can infer

- as loop invariant: $off_{\mathtt{p}} = off_{\mathtt{q}} \leq l_{\mathtt{src}} \leq a_{\mathtt{src}}$
- after the loop: $off_{\mathtt{p}} = off_{\mathtt{q}} = l_{\mathtt{src}} \leq a_{\mathtt{src}}$
- raise an alarm if $l_{\mathtt{src}} \geq a_{\mathtt{src}}$ or $l_{\mathtt{src}} \geq a_{\mathtt{dst}}$
- otherwise, we ensure that $l_{\mathtt{dst}} = l_{\mathtt{src}}$.

# Stub contract language

open

```
/*$
 * requires: exists int i in [0, size(__file) - 1]: __file[i] == 0;
 *
 * case "success":
 *   local:   void* fd = new FileDescriptor;
 *   ensures: return == (int)fd;
 *
 * case "failure":
 *   assigns: _errno;
 *   ensures: return == -1;
 */
int open (const char *__file, int __oflag, ...);
```

# Stub contract language

open

```
/*$
 * requires: exists int i in [0, size(__file) - 1]: __file[i] == 0;
 *
 * case "success":
 *   local:   void* fd = new FileDescriptor;
 *   ensures: return == (int)fd;
 *
 * case "failure":
 *   assigns: _errno;
 *   ensures: return == -1;
 */
int open (const char *__file, int __oflag, ...);
```

## Specification language:

- inspired from ACSL (Frama-C)
- targets stub modeling (not functional verification)
- yet another language in MOPSA (extending and sharing AST and domains)
- interpret formulas in abstract domains
  $\Longrightarrow$ domains dedicated to quantified formulas (strings, arrays)
- modeling of resources (memory, file descriptors, etc.)

# C benchmarks

- extracted from Juliet Test Suite (v 1.3) for C/C++
  - CWE476 on null pointer dereferences.
  - CWE369 on divisions by zero
  - CWE190 on integer overflows
- each test has a bad version and a correct version

| Category | Loc | Tests | Time | Alarms | Coverage |
|---|---|---|---|---|---|
| CWE476 | 25K | 522 | 2mn26s | 0 | 100% |
| CWE369 | 109K | 1368 | 7mn20s | 372 | 53% |
| CWE190 | 440K | 6840 | 34mn57s | 0 | 73% |

On-going work: analyzing actual C programs from GNU CoreUtils.

# Application to Python Analysis

# Python 3 language

Highly dynamic language:

- variables have no fixed type (only values have)
- everything is an object
- complex operator semantics (many cases, many ways to override)
- complex control-flow: exceptions, generators, lambdas
- rich built-in and standard libraries
- meta-programming (introspection, dynamic classes, eval)
- no formal semantics
- evolving language

$\Longrightarrow$ static analysis is challenging, but rewarding

# Python 3 semantics

- formalize the concrete semantics
  based on the Python manual and CPython implementation
- use a denotational-style semantics (easier to abstract)
- type-based cases (eval and DNF are useful)

$\Longrightarrow$ the abstract semantics has the same structure as the concrete one

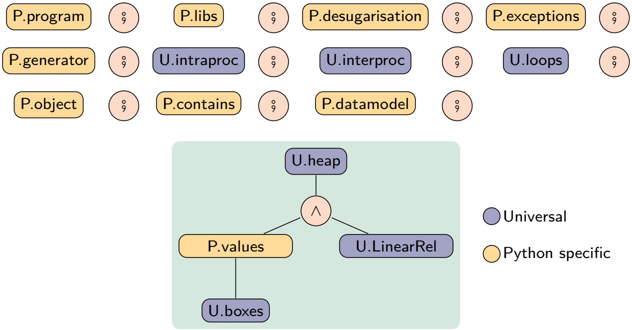
# Python 3 semantics

$$
\begin{array}{l}
\mathbb{E}[\![\, e_1 + e_2 \,]\!](f, \epsilon, \Sigma) \stackrel{\text{def}}{=} \\
\quad \text{let } (f_1, \epsilon_1, \Sigma_1, v_1) = \mathbb{E}[\![\, e_1 \,]\!]\,(f, \epsilon, \Sigma) \text{ in} \\
\quad \text{let } (f_2, \epsilon_2, \Sigma_2, v_2) = \mathbb{E}[\![\, e_2 \,]\!]\,(f_1, \epsilon_1, \Sigma_1) \text{ in} \\
\quad \text{if } \mathit{hasattr}(v_1, \texttt{\_\_add\_\_}, \Sigma_2) \text{ then} \\
\quad\quad \text{let } (f_3, \epsilon_3, \Sigma_3, v_3) = \mathbb{E}[\![\, v_1.\texttt{\_\_add\_\_}(v_2) \,]\!]\,(f_2, \epsilon_2, \Sigma_2) \text{ in} \\
\quad\quad \text{if } v_3 = \texttt{NotImpl} \wedge \mathit{typeof}(v_1) \neq \mathit{typeof}(v_2) \text{ then} \\
\quad\quad\quad \text{if } \mathit{hasattr}(v_2, \texttt{\_\_radd\_\_}, \Sigma_3) \text{ then} \\
\quad\quad\quad\quad \text{let } (f_4, \epsilon_4, \Sigma_4, v_4) = \mathbb{E}[\![\, v_2.\texttt{\_\_radd\_\_}(v_1) \,]\!]\,(f_3, \epsilon_3, \Sigma_3) \text{ in} \\
\quad\quad\quad\quad \text{if } v_4 = \texttt{NotImpl} \text{ then } \texttt{TypeError}(f_4, \epsilon_4, \Sigma_4) \text{ else } (f_4, \epsilon_4, \Sigma_4, v_4) \\
\quad\quad\quad \text{else } \texttt{TypeError}(f_3, \epsilon_3, \Sigma_3) \\
\quad\quad \text{else if } v_3 = \texttt{NotImpl} \text{ then } \texttt{TypeError}(f_3, \epsilon_3, \Sigma_3) \text{ else } (f_3, \epsilon_3, \Sigma_3, v_3) \\
\quad \text{else if } \mathit{hasattr}(v_2, \texttt{\_\_radd\_\_}, \Sigma_2) \wedge \mathit{typeof}(v_1) \neq \mathit{typeof}(v_2) \text{ then} \\
\quad\quad \text{let } (f_3, \epsilon_3, \Sigma_3, v_3) = \mathbb{E}[\![\, v_2.\texttt{\_\_radd\_\_}(v_1) \,]\!]\,(f_2, \epsilon_2, \Sigma_2) \text{ in} \\
\quad\quad \text{if } v_3 = \texttt{NotImpl} \text{ then } \texttt{TypeError}(f_3, \epsilon_3, \Sigma_3) \text{ else } (f_3, \epsilon_3, \Sigma_3, v_3) \\
\quad \text{else } \texttt{TypeError}(f_2, \epsilon_2, \Sigma_2)
\end{array}
$$

- formalize the concrete semantics
  based on the Python manual and CPython implementation
- use a denotational-style semantics (easier to abstract)
- type-based cases (eval and DNF are useful)

$\Longrightarrow$ the abstract semantics has the same structure as the concrete one

# Python value analyzer configuration




- hand-written parser in Menhir
- resolves `import` at parsing time
- reuse universal domains: numeric, heap abstractions, loops, etc.

# Concrete domains for Python semantics

Concrete collecting semantics in $\mathcal{P}(\mathcal{E} \times \mathcal{H})$:

- environments: $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathit{Val}$
- values: $\mathit{Val} \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathit{Addr} \cup \{\,\texttt{True}, \texttt{False}, \texttt{None}, \texttt{Undef}, \texttt{NotImplemented}\,\}$
- heap: $\mathcal{H} \stackrel{\text{def}}{=} \mathit{Addr} \rightharpoonup \mathit{Obj}$
  $\mathit{Obj} \stackrel{\text{def}}{=} \mathit{String} \rightharpoonup \mathit{Val}$

# Non-relational value analysis for Python

Follows the concrete semantics:

- environments: $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathcal{V} \to \mathit{Val}^\sharp$
- values: $\mathit{Val}^\sharp \stackrel{\text{def}}{=} \mathbb{Z}^\sharp \times \mathit{Bool}^\sharp \times \mathcal{P}(\mathit{Addr}^\sharp) \times \mathit{None}^\sharp \times \mathit{NotImplemented}^\sharp \times \mathit{Undef}^\sharp$

  (abstract disjoint unions as tuples)
- $\mathit{None}^\sharp, \mathit{NotImplemented}^\sharp, \mathit{Undef}^\sharp \stackrel{\text{def}}{=} \{\bot, \top\}$, $\mathit{Bool}^\sharp \stackrel{\text{def}}{=} \{\bot, \top, t, f\}$
- $\mathbb{Z}^\sharp$: non-relational domain (e.g., intervals)
- $\mathit{Addr}^\sharp$: allocation site abstraction
- heap: $\mathcal{H}^\sharp \stackrel{\text{def}}{=} \mathit{Addr}^\sharp \rightharpoonup \mathit{Obj}^\sharp$

Object abstraction: $\mathit{Obj}^\sharp \stackrel{\text{def}}{=} (\mathit{String} \rightharpoonup \mathit{Val}^\sharp) \times \mathcal{P}(\mathit{String})$

Attributes can be added to objects dynamically
$\Longrightarrow$ a set of objects can have heterogeneous sets of attributes

- $\mathit{String} \rightharpoonup \mathit{Val}^\sharp$ maps all possible attributes to their value
- $\mathcal{P}(\mathit{String})$: attributes that are guaranteed to exist in all objects

  necessary to prove that `AttributeError` cannot occur

# Built-ins in Python

Built-in data-structures:

- Strings: bounded sets of constant strings, or $\top$
- Lists: one summary element, and a length
- Dictionaries: as objects, or with a summary element

Example: model a list access `l[i]`

- $C1$: $\texttt{isinsance}(\texttt{l}, \texttt{list}) \wedge \texttt{isinsance}(\texttt{i}, \texttt{int})$
- $C2$: $-\texttt{len}(\texttt{l}) \leq \texttt{i} < \texttt{len}(\texttt{l})$
- $C3$: $\texttt{len}(\texttt{l}) = 1$

| case | evaluation |
|---|---|
| $\neg C1$ | `TypeError` |
| $C1 \wedge \neg C2$ | `IndexError` |
| $C1 \wedge C2 \wedge C3$ | summary variable $\ell$ |
| $C1 \wedge C2 \wedge \neg C3$ | weak copy of summary variable $\ell$ |

only partial support in MOPSA at the moment, to be improved

# Python benchmarks

- regression tests from the official Python 3.6.3 distribution.
- analyze only 9 out of 500 tests (limited coverage of the standard library)

| Regression test | Lines | Tests | Time | ✓ | ✗ | ✱ | Coverage |
|---|---|---|---|---|---|---|---|
| test_augassign | 273 | 7 | 645ms | 4 | 2 | 1 | 85.71% |
| test_baseexception | 141 | 10 | 20ms | 6 | 0 | 4 | 60.00% |
| test_bool | 294 | 26 | 47ms | 12 | 0 | 14 | 46.15% |
| test_builtin | 454 | 21 | 360ms | 3 | 0 | 18 | 14.29% |
| test_contains | 77 | 4 | 418ms | 1 | 0 | 3 | 25.00% |
| test_int_literal | 91 | 6 | 29ms | 6 | 0 | 0 | 100.00% |
| test_int | 218 | 8 | 88ms | 3 | 0 | 5 | 37.50% |
| test_list | 106 | 9 | 88ms | 3 | 0 | 6 | 33.33% |
| test_unary | 39 | 6 | 11ms | 2 | 0 | 4 | 33.33% |

- analyze performance benchmarks
- evaluate the impact of relational numeric domains

| Performance benchmark | Lines | Interval | | Octagon | | Polyhedra | |
|---|---|---|---|---|---|---|---|
| float | 37 | 1.5s | ✓ | 4.8s | ✓ | 3.4s | ✓ |
| fannkuch | 37 | 0.8s | ✗(3) | 4.7s | ✗(1) | 3.3s | ✓ |
| nbody | 66 | 1.0s | ✗(2) | 10min1s | ✗(2) | $\infty$ | |

# Types as abstraction of values

dynamic typing

```
def fspath(p):
    if isinstance(p, (str, bytes)):
        return p
    elif hasattr(p, "__fspath__"):
        res = p.__fspath__()
        if isinstance(res, (str, bytes)):
            return res
         else:
            raise TypeError(...)
    else:
         raise TypeError(...)
```

Python mixes:

- nominal typing: `isinstance`
- duck typing: `hasattr`

Both can be resolved in the abstraction $Val^\sharp$:

- nominal typing: value of the attribute `__class__`
- duck typing: presence of a specific attribute in $Obj^\sharp$

# Type analysis for Python

On-going work:

More scalable abstraction remembering only type information

- sets of the types of the values stored in each variable
  $\mathcal{V} \rightarrow \mathcal{P}(\textit{types})$
- top-down, flow-sensitive inference by propagation of abstract values
  $\Longrightarrow$ more of an Abstract Interpretation technique than regular typing
- types for built-in objects: `List[int]`
- types for nominal and duck typing: `Instance[class,attrs]`
- bounded parametric polymorphism: `List[`$\alpha$`]`, $\alpha \in \{\dots\}$
  $\Longrightarrow$ relational typing domain: V:`List[`$\alpha$`]` $\wedge$ W:`List[`$\alpha$`]` $\wedge$ $\alpha = \beta$

# Benchmarks for Python type analysis

- reuse MOPSA framework, change the abstract domains
- compare with
  - Typpete: type inference via SMT-solve
  - Fritz & Hafe: data-flow equations
  - Pytype from Google

| Program | Fritz & Hage | Pytype | Typpete | MOPSA |
|---|---|---|---|---|
| Analysis method | Dataflow analysis | Unclear | SMT-solver | AI |
| class_attr_ok | ✓ | ✗ | ✳ | ✓ |
| class_pre_store | ✓ | ✓ | ✓ | ✓ |
| default_args_class | ✓ | ✓ | ✓ | ✓ |
| except_clause | ✗ | ✳ | ✓ | ✓ |
| fspath | ✗ | ✗ | ✳ | ✓ |
| magic | ✓ | ✓ | ✓ | ✳ |
| polyfib | ✳ | ✗ | ✳ | ✳ |
| poly_lists | ✳ | ✓ | ✳ | ✓ |
| vehicle | ✓ | ✓ | ✓ | ✳ |
| widening | ✓ | ✗ | ✳ | ✓ |

# Conclusion

# Conclusion

## Features:

- compositional, flexible architecture to build static analyzers
- a few original choices
  unified AST, iterators, partial environments, evaluation, DNF, stacked domains
- used in research projects on C and Python analysis
- reusable abstract domains, language support, semantic abstractions
- extensible, with loose coupling
- additional features: interactive debug, interpreter, web-based GUI

## Future work:

- enhance coverage for C and Python built-in libraries
- test on larger, more realistic code bases
- release as open source with support
- mixing C and Python ?