Mopsa Tutorial

Raphaël Monat

`rmonat.fr/lipari24/`

Lipari Summer School
4 Sep. 2024

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Goals: explore new designs, ease development of (relational) analyses

One AST to rule them all	Unified domain signature	DAG of abstractions
Multilanguage support	Semantic rewriting	Relational domains
Expressiveness	Loose coupling	Composition
Reusability	Observability	Cooperation

A motivating example

Averaging tasks

```python
class Task:
  def __init__(self, weight):
    if weight < 0: raise ValueError
    self.weight = weight

def average(l):
  m = 0
  for i in range(len(l)):
    m = m + l[i].weight
  m = m // (i + 1)
  return m

l = [Task(randint(0, 20))
  for i in range(randint(5, 10))]
m = average(l)
```

Proved safe?

- `m // (i+1)`
- `l[i].weight`

Searching for a loop invariant

Stateless domains: list content, list length

Environment abstraction

$m \mapsto @^\sharp_{\mathtt{int}^\sharp}$ $\quad i \mapsto @^\sharp_{\mathtt{int}^\sharp}$ $\quad \underline{\mathrm{els}}(l) \mapsto @^\sharp_{\mathtt{Task}}$

$\underline{@^\sharp_{\mathtt{Task}} \cdot \mathtt{weight}} \mapsto @^\sharp_{\mathtt{int}^\sharp}$

Numeric abstraction (polyhedra)

$m \in [0, +\infty)$ $\quad \underline{\mathrm{els}}(l) \in [0, 20]$

$0 \leq i < \underline{\mathrm{len}}(l)$ $\quad 5 \leq \underline{\mathrm{len}}(l) \leq 10$

$0 \leq \underline{@^\sharp_{\mathtt{Task}} \cdot \mathtt{weight}} \leq 20$

Attributes abstraction

$@^\sharp_{\mathtt{Task}} \mapsto (\{\,\mathtt{weight}\,\}, \emptyset)$

Contributors (2018–2024, chronological arrival order)

- A. Miné
- A. Ouadjaout
- M. Journault
- A. Fromherz
- D. Delmas
- R. Monat
- G. Bau
- F. Parolini
- M. Milanese
- M. Valnet
- J. Boillot

Maintainers in bold.

Slides inspired from many contributors, mistakes are my own.

Works around Mopsa

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b]

Multilanguage Python+C [MOM21]

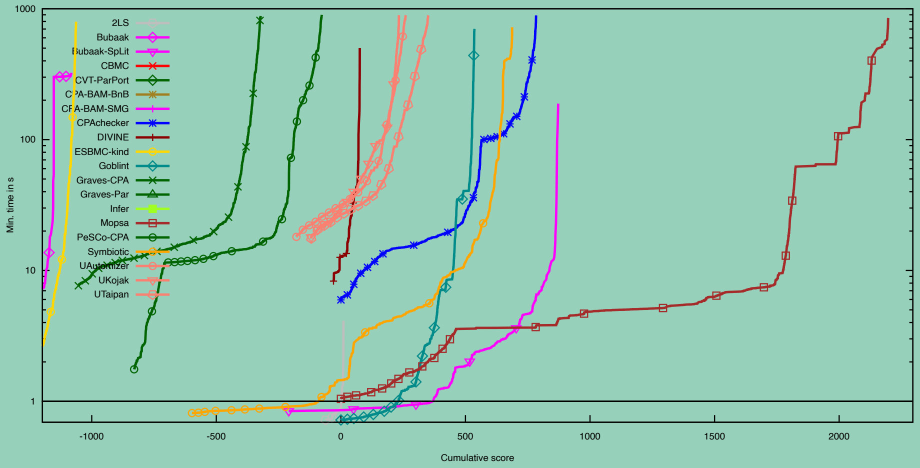
WIP: Michelson [Bau+22], OCaml [VMM23], Catala (date arithmetic [MFM24])...

Properties

- **Absence of RTEs**
- Patch analysis [DM19]
- Endianness portability [DOM21]
- Non-exploitability [PM24]
- Sufficient precondition inference [MM24]

Software Verification Competition

We won the “SoftwareSystems” track of SV-Comp 2024 [Mon+24]!



Ocaml tidbits

Mopsa is implemented in OCaml.

- Curried functions: `f x y z` $\leadsto f(x, y, z)$
- `fun x -> e` $\Leftrightarrow \lambda x.e$
- Variable binding `let x = e1 in e2`
- Algebraic datatypes and pattern matching

```
type 'a option = None | Some of 'a

match e with
| None -> e1
| Some y -> e2 y
```

Polymorphism = Type Variables

Outline

1. Introduction
2. Architecture of Mopsa
3. Current analyses in Mopsa
4. Easing development
5. Conclusion

Then: 60 minutes practical session

Objectives

- Understand Mopsa's core principles
- Ability to run C/Python analyses

Feel free to ask questions!

New users welcome!

Feel free to reach out in the future!

Architecture of Mopsa

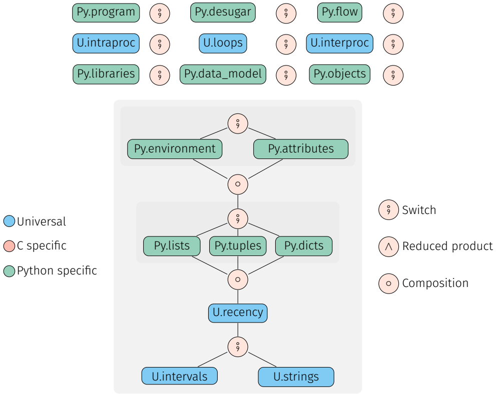
Defining analyses

Mopsa design

Analysis = composition of abstract domains

unified domain signature $\Longrightarrow$ iterators are abstract domains

- flexible architecture suitable for various programming paradigms
- separation of concerns
- allows reuse of domains across languages
- defined as json files in `share/mopsa/configs`



Abstract state & domain signature

Which type can we give to the abstract state?

- Polymorphism to the rescue: `'a` represents the abstract state
- Extended into `'a flow` to maintain additional info (more later)

Handling cases

- `type ('a, 'r) cases` as DNFs over `'a flow * 'r`
- `Cases.singleton : 'r -> 'a flow -> ('a, 'r) cases`
- Binding operator `cases >>$ fun r flow -> ...`

  `>>$ : ('a,'r) cases -> ('r -> 'a flow -> ('a,'s) cases) -> ('a,'s) cases`
- Side note: this is a monad

Abstract state & domain signature – II

The manager: interroperating the whole analysis and local domains

- Local domain has a private `type t`
- Whole abstract state of `type 'a`

$\Longrightarrow$ `type ('a, t) man`

From global analysis to local domain

- Get the domain's data
  `get : 'a -> t`
- Set the domain's data
  `set : t -> 'a -> 'a`

From local domain to global analysis

- Analyze a given expression
- Analyze a given statement
  `man.exec stmt` $\sigma \Leftrightarrow \mathbb{S}^\sharp[\![\text{stmt}]\!]\sigma$

Signatures later

Also: lattice operators

Abstract state & domain signature – III

Utilities

```
type ('a, 'r) cases (* ≃ DNF of 'a flow * 'r *)

type 'a eval = ('a, expr) cases
type 'a post = ('a, unit) cases

(* Manager, allowing interaction between a
   domain ('t) and whole analysis ('a) *)
type ('a, 't) man = {
  get : 'a -> 't;
  set : 't -> 'a -> 'a;
  exec : stmt -> 'a flow -> 'a post;
  eval : expr -> 'a flow -> 'a eval;
  (* [...] *)
}
```

Domain type overview

```
module type DOMAIN = sig

type t
(* private, opaque data of the domain *)
val name : string

val join : t -> t -> t (* and other lattice operators *)

(* Transfer functions *)
val exec : stmt -> ('a, t) man -> 'a flow -> 'a post option
val eval : expr -> ('a, t) man -> 'a flow -> 'a eval option

(* [...] *)
end
```

Focus on the domain-local transfer functions

```
val exec : stmt -> ('a, t) man -> 'a flow -> 'a post option
val eval : expr -> ('a, t) man -> 'a flow -> 'a eval option
```

- `('a, t) man` manager
- `'a flow` abstract state
- option: domains return `None` for unsupported statements/expressions.
  - `'a post = ('a, unit) cases`. DNF of abstract states.
  - `'a eval = ('a, expr) cases`. DNF of abstract states and symbolic expressions. Useful for rewriting, esp. for relational analyses

Example: loop iterator

Iterators are stateless domains:

- `type t = unit`, trivial lattice operators
- We have an automatic lifter for all that!

The loop iterator focuses on postfixpoint computation, and delegates the rest.

Universal.Iterators.Loops

```
let rec lfp man cond body flow_init flow =
  man.exec (mk_block [mk_assume cond; body]) flow >>$ fun () flow' ->
  if man.lattice.subset (man.lattice.join flow_init flow') flow
  then Cases.singleton () flow'
  else lfp man cond body flow_init (man.lattice.widen flow flow')

let exec stmt man flow = match stmt.skind with
  | S_while (cond, body) ->
    Some (lfp man cond body flow flow >>$ fun () lfp_flow ->
          man.exec (mk_assume (mk_not cond)) lfp_flow)
  | _ -> None
```

Architecture of Mopsa

Delegation-based support of multiple languages

Iterators to handle multiple languages

Traditional approaches

Desugar/compile programs to an intermediate representation (IR)

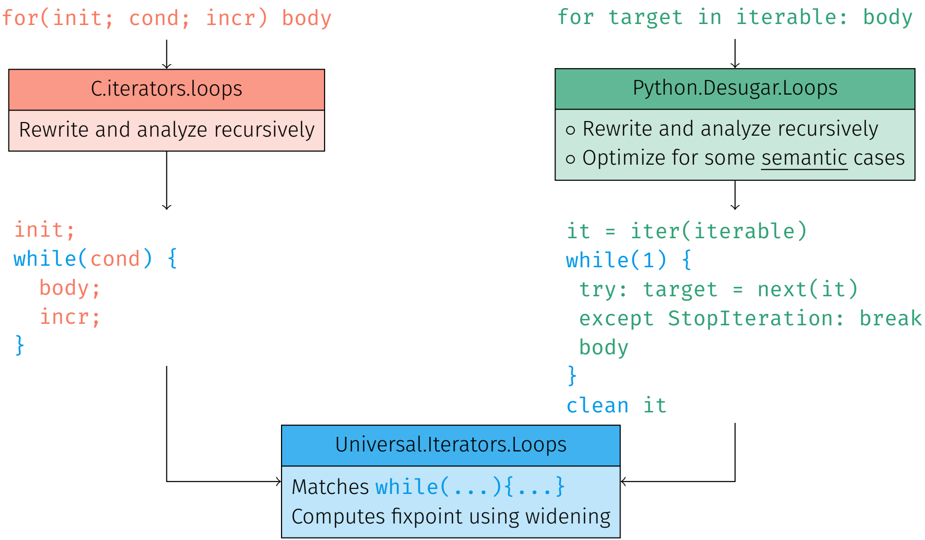
Example: Infer's IR has five (!) constructors

Mopsa

- No loss of precision from the frontend[1]
- Various programming paradigms supported!
- All constructs have to be handled – but rewritings are possible
- A single AST type which can be extended for new languages

[1] By default, 3-address code may result in precision loss [NP18]

Dynamic, semantic iterators with delegation



Architecture of Mopsa

Stateful domains through numerical examples

Numerical abstract values

$$\mathcal{P}(\mathcal{V} \to \mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{V} \to \mathcal{P}(\mathbb{Z}) \underset{\dot{\alpha}_I}{\overset{\dot{\gamma}_I}{\leftrightarrows}} \mathcal{V} \to \mathcal{I}$$

Cartesian abstraction — Lifting intervals

$$\alpha : \begin{cases} \mathcal{P}(\mathcal{V} \to \mathbb{Z}) & \to & \mathcal{V} \to \mathcal{P}(\mathbb{Z}) \\ \Sigma & \mapsto & \lambda v.\{\, \sigma(v) \mid \sigma \in \Sigma \,\} \end{cases} \qquad \gamma : \begin{cases} \mathcal{V} \to \mathcal{P}(\mathbb{Z}) & \to & \mathcal{P}(\mathcal{V} \to \mathbb{Z}) \\ f & \mapsto & \{\, \sigma \mid \forall v \in \mathcal{V}, \sigma(v) \in f(v) \,\} \end{cases}$$

$\Longrightarrow$ We can define abstract operations on values only.

The `nonrel` combinator will lift values to the mapping.

Numerical abstract values – II

Value signature

```
module type VALUE = sig
type t

val name : string

val bottom: t
val top: t

val subset: t -> t -> bool
val join: t -> t -> t
val meet: t -> t -> t
val widen: 'a ctx -> t -> t -> t
val constant : constant -> typ -> t
val binop : operator -> typ -> t -> typ ->
  t -> typ -> t
val filter : bool -> typ -> t -> t

val backward_binop : operator -> typ ->
  t -> typ -> t -> typ -> t -> t * t
val compare : operator -> bool -> typ ->
  t -> typ -> t -> (t * t)

val print: printer -> t -> unit
end
```

Implementations for intervals, congruences, powerset of integers

They all abstract the same object

Motivational example

```
// Hyp: a array, of size l ∈ [10, 20]
s = 0;
for(int i = 0; i < l; i++) {
  s += a[i];
}
```

i ∈ [0, 20], l ∈ [10, 20], unable to prove safe access ✗

The cartesian abstraction breaks relationality

$$\mathcal{P}(\mathcal{V} \to \mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{V} \to \mathcal{P}(\mathbb{Z})$$

$$\begin{aligned}(\gamma \circ \alpha)(\{\, 0 \le x < y \le 2 \,\}) &= (\gamma \circ \alpha)(\{\, (0,1); (0,2); (1,2) \,\}) \\ &= \gamma\,(x \mapsto \{\, 0,1 \,\}, y \mapsto \{\, 1,2 \,\}) \\ &= \{\, (0,1); (0,2); (\mathbf{1},\mathbf{1}); (1,2) \,\} \\ &= \{\, 0 \le x < 2, 0 < y \le 2 \,\}\end{aligned}$$

Relational domains - II

Mopsa relies on the Apron library [JM09], providing among others:

- Polyhedra [CH78], variants with PPL[BHZ08] or PPLite[BZ20] $\sum_i \alpha_i V_i \leq \beta_i$
- Octagons [Min06b] $\pm V_i \pm V_j \leq c_{i,j}$
- Grids [Bag+06] $\sum_i \alpha_i V_i \equiv \beta_i [n]$

Motivational example, with polyhedra

```
// Hyp: a array, of size l ∈ [10, 20]
s = 0;
for(int i = 0; i < l; i++) {
  s += a[i];
}
```

$0 \leq i < l$, ✓

Architecture of Mopsa

Leveraging relational abstract domains

Machine Numbers

- Numerical domains rely on mathematical numbers.
- C uses finite precision numbers with modular arithmetics.
- MachineNum lifts C statements to a math semantic.

Dynamic Lifting

Consider a variable x declared as unsigned char.

$$\mathbb{E}^\sharp[\![\texttt{x + 2}]\!]\langle \texttt{x} \mapsto [0, 255]\rangle \begin{cases} \xrightarrow{x + 2 \in [0, 255]} \wr \texttt{x + 2}, \langle \texttt{x} \mapsto [0, 253]\rangle, \varnothing \wr \\ \xrightarrow{x + 2 \notin [0, 255]} \wr \mathit{wrap}(\texttt{x + 2}, [0, 255]), \langle \texttt{x} \mapsto [254, 255]\rangle, \{\texttt{IntOverflow}\} \wr \end{cases}$$

C expression

Mathematical expression

N.B: expression evaluation required here

Machine Numbers – II

C.Memory.Machine_Numbers

```
let eval exp man flow =
  match exp.ekind with
    | E_var v -> Some (Cases.singleton exp flow)
    | E_binop(op, e1, e2) ->
      man.eval e1 flow >>$ fun n1 flow ->
      man.eval e2 flow >>$ fun n2 flow ->
      let vmin, vmax = rangeof exp.etyp in
      let nexp = mk_binop n1 op n2 in
      let ret = assume (mk_in nexp vmin vmax) man flow
          ~fthen:(fun flow ->
              let flow = safe_c_integer_overflow flow in
              Cases.singleton nexp flow)
          ~felse:(fun flow ->
              let nexp' = mk_wrap nexp vmin vmax in
              let flow = raise_alarm IntegerOverflow flow in
              Cases.singleton nexp' flow
            ) in Some ret
    | _ -> None
```

Variables do not overflow

Evaluate **e1** and bind each case ⦃n1⦄

Evaluate **e2** and bind each case ⦃n2⦄

Partition on condition $\texttt{nexp} \in [\texttt{vmin}, \texttt{vmax}]$

Abstracting containers (strings, arrays) lengths

Consider a variable-length container *a*.

Motivational example, with polyhedra

```
// Hyp: a container
s = 0;
for(int i = 0; i < container_length(a); i++) {
  s += a[i];
}
```

We track its length through the introduction of a ghost numerical variable <u>len</u>(*a*)

The relational domain will be able to infer relationships between *i* and <u>len</u>(*a*).

N.B: In a non-relational setting, we could track values directly.

Convention: ghost variables are underscored.

Abstracting containers (strings, arrays) lengths – II

Universal.Toy.String_length

```
let exec stmt man flow = let range = srange stmt in match skind stmt with
  | S_assign ({ekind = E_var (s, _); etyp=T_string}, e) ->
    Some (man.exec (mk_assign (mk_len_string_var s range)
                   (mk_expr (E_len e) range) range) flow)

  | S_assign ({ekind = E_subscript ({ekind = E_var (s, _)}, i); e) ->
    Some (
      assume (mk_in i (mk_zero range) (mk_len_string_var s range) range)
             man flow
             ~fthen:(safe_subscript_access_check)
             ~felse:(fun flow ->
                 let flow = raise_alarm invalid subscript access flow in
                 Cases.empty flow))

  | _ -> None
```

Case `s = e`

Rewrite and delegate into $\underline{\text{len}}(s) = \text{len}(e)$

Case `s[i] = e`

Case disjunction on $0 \leq i \leq \underline{\text{len}}(s)$

Safe, nothing to do

Return ⊥, with alarm metadata

Abstracting containers (strings, arrays) lengths – III

```
string s;
if (rand(0, 1)) { s = "abcd"; }
else { s = "ab"; }
```

- Intervals $\underline{\text{len}}(s) \in [2, 4]$
- Intervals $\wedge$ Congruences $\underline{\text{len}}(s) \in [2, 4] \wedge 2\mathbb{Z}$
- Powerset of integers $\underline{\text{len}}(s) \in \{ 2, 4 \}$

```
string s = rand();
string t = s + s;
```

- Intervals $\underline{\text{len}}(s) \in [0, +\infty], \underline{\text{len}}(t) \in [0, +\infty]$
- Polyhedra $0 \leq t = 2 \cdot \underline{\text{len}}(s)$

NB: in case of dynamic allocations, ghost variables are a bit more complicated

Abstracting containers (strings, arrays) contents

Variable-length containers have unbounded size

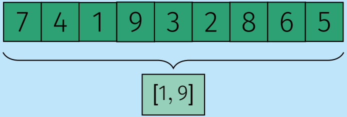
Not abstracting their contents $\Longrightarrow$ Non-terminating analysis

The "smashing abstraction"

Idea: summarize every concrete cell of the container into an abstract one

Concrete **a**

Smashed $\underline{\text{els}}(a)$



Weak update

Analyzing `a[2] = 12`, assuming $\underline{\text{els}}(a) \in [1, 9]$

✗ $\underline{\text{els}}(a) = [12, 12]$, as $\underline{\text{els}}(a)$ represents multiple concrete elements!

✓ $\underline{\text{els}}(a) \stackrel{\text{weak}}{=} [12, 12] \stackrel{\text{def}}{=} \underline{\text{els}}(a) \sqcup [12, 12] = [1, 12]$

$\mathbb{S}^\sharp [\![c \stackrel{\text{weak}}{=} e]\!] s \stackrel{\text{def}}{=} s \sqcup \mathbb{S}^\sharp [\![c = e]\!] (s)$

Abstracting containers (strings, arrays) contents – II

What about weak read?

`r = a[3]`, assuming $\underline{\text{els}}(a) \in [1, 9]$.

$r \in [1, 9]$, and similarly with non-relational domains.

Weak read and relational domains

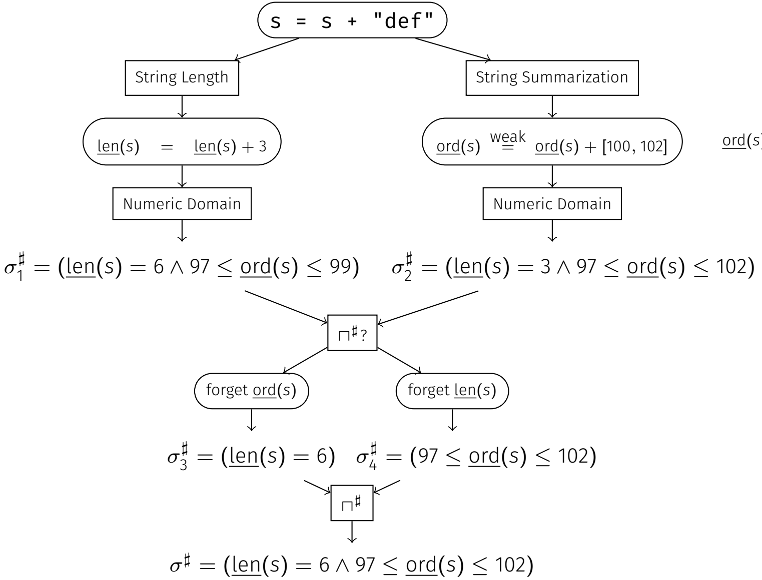
`r = a[3]`, assuming $1 \leq \underline{\text{els}}(a) \leq i$.

✗ $1 \leq r = \underline{\text{els}}(a) \leq i$, as $\underline{\text{els}}(a)$ represents multiple concrete elements!

✓ $\mathbb{S}^\sharp[\![\, \texttt{expand}(\underline{\text{els}}(a), r) \,]\!](1 \leq \underline{\text{els}}(a) \leq i) = 1 \leq \underline{\text{els}}(a) \leq i \wedge 1 \leq r \leq i$

Intuitively: expand copies constraints. Cf. [Gop+04]

The perils of reduced products with shared relational domains



$\underline{\text{ord}}(s)$: smashed ASCII codes of s.

Leveraging relational abstract domains: conclusion

Mopsa relies on rewriting, symbolic expressions and ghost variables

to leverage relational domains.

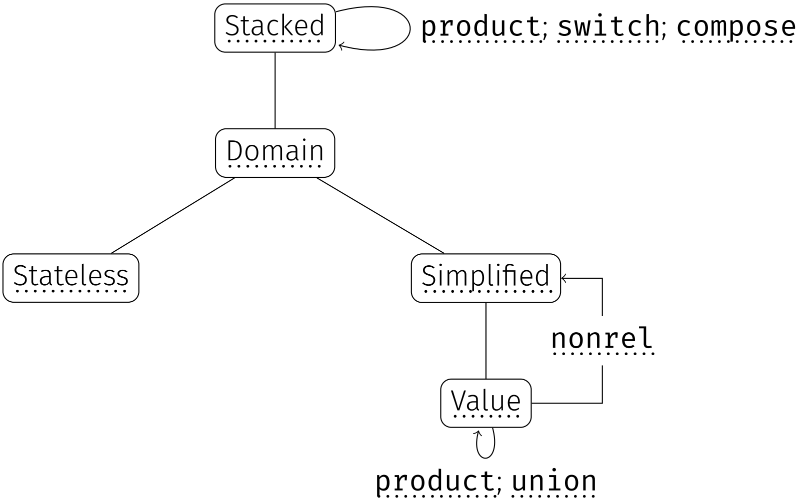
The great power of relational domains comes with

- Computational cost more than $\mathcal{O}(|\mathcal{V}|^3)$
- Force cohabitation of variables (cornerstone of Mopsa's design)
- Weak variables need specific operators (`expand`)
- Reduced product with sharing needs `merge` operator

Architecture of Mopsa

Domains & their combinators

Hasse diagram of domains



Combinators

Stacked Product

- Statements/expressions are dispatched to both sides
- Effects are collected to merge results soundly
- Reductions can be defined after evaluations of expressions or statements.

Switch

- Pointwise lattice lifting
- For **tf** $\in \{$ **eval**, **exec** $\}$:

```
Switch(D1, D2).tf args =
let o_r = D1.tf args in
match o_r with
 None -> D2.tf args
 Some r -> r
```

Compose

- Similar to switch
- Lattice operators can trigger operations on underlying domains

Architecture of Mopsa

Transparency in static analysis

Raising the bar in static analyzer transparency

```
$ static-analysis-tool file
...
No errors found
```

What has been checked? What has not?

Mopsa's approach to being transparent – implementation

C.Memory.Machine_Numbers

```
let eval exp man flow =
  match exp.ekind with
  | E_var v -> Some (Cases.singleton exp flow)
  | E_binop(op, e1, e2) ->
    man.eval e1 flow >>$ fun n1 flow ->
    man.eval e2 flow >>$ fun n2 flow ->
    let vmin, vmax = rangeof exp.etyp in
    let nexp = mk_binop n1 op n2 in
    let ret = assume (mk_in nexp vmin vmax) man flow
        ~fthen:(fun flow ->
            let flow = safe_c_integer_overflow flow in
            Cases.singleton nexp flow)
        ~felse:(fun flow ->
            let nexp' = mk_wrap nexp vmin vmax in
            let flow = raise_alarm IntegerOverflow flow in
            Cases.singleton nexp' flow
          ) in Some ret
  | _ -> None
```

Transparent: mark that `e1 + e2` does not overflow

Standard: report alarm
Stored as metadata in `'a flow`

Mopsa's approach to being transparent – example

Mopsa's approach to being transparent

- Reporting status of all proofs / checks in every analyzed context
- Quantitative precision measure

$$\text{Selectivity} = \frac{\#\text{checks proved safe}}{\#\text{checks}}$$

```c
int main() {
  int n;
  int y = -1;
  for(int x = 0; x < n; x++)
    y++;
}
```

Stmt	Itv	Poly
x++	Safe	Safe
y++	Alarm	Safe
Selectivity	50%	100%

Mopsa's approach to being transparent – output

Benefits of the approach

- Easy to implement
- "2,756 alarms" ⇝ 87% checks proved correct – "selectivity"
- ~~Program size~~ ⇝ "expression complexity"

Analysis of coreutils `fmt`

```
Checks summary: 21247 total, ✓18491 safe, ✗129 errors, ⚠2627 warnings
  Stub condition: 690 total, ✓513 safe, ✗3 errors, ⚠174 warnings
  Invalid memory access: 8139 total, ✓7142 safe, ✗4 errors, ⚠993 warnings
  Division by zero: 499 total, ✓445 safe, ⚠54 warnings
  Integer overflow: 11581 total, ✓10177 safe, ⚠1404 warnings
  Invalid shift: 163 total, ✓163 safe
  Invalid pointer comparison: 37 total, ✗37 errors
  Invalid pointer subtraction: 85 total, ✗85 errors
  Insufficient variadic arguments: 1 total, ✓1 safe
  Insufficient format arguments: 26 total, ✓25 safe, ⚠1 warning
  Invalid type of format argument: 26 total, ✓25 safe, ⚠1 warning
```

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently ✗
3. Assume and report: f has no effect
4. Assume and report: f has any effect on its parameters
5. Assume and report: f has any effect on its parameters and on globals

Related: soundiness paper [Liv+15]

Current analyses in Mopsa

Control-flow tokens

Control-flow tokens

- Astrée [Cou+06, footnote 4, page 6], Mopsa do not iterate over a CFG
- Proceeds by induction over the syntax
- Non-local control-flow is represented by <u>control-flow tokens</u>
  cur is the standard control-flow, *brk* states interrupted by a **`break`**
- Also applies to **`goto`**, **`return`**, **`raise`**, ...
- Implementation: `'a flow` contains a `'a TokenMap.t`

```
int i = 0;
while(i < 100) {
  i = i + rand(1, 3);
  if(i == 42) break;
}
```

$cur \mapsto i \in [1, 99]$

$cur \mapsto i \in [2, 102]$

$brk \mapsto i \in [42, 42]$

$cur \mapsto i \in ([2, 102] \sqcap [100, +\infty]) \sqcup [42, 42]$

More details: [Mon21, section 2.4.3]

Control-flow tokens – II

Universal.Iterators.Loops

```
let exec stmt man flow = match stmt.skind with
  | S_break ->
    Some (Cases.return () (Flow.rename T_cur T_break man.lattice flow))

  | S_while (cond, body) ->
    Some (
      lfp man cond body (Flow.rm T_break flow) flow >>% fun lfp_flow ->
      man.exec (mk_assume (mk_not cond)) lfp_flow >>% fun lfp_flow ->
      let lfp_flow = Flow.add T_cur (Flow.get T_break man.lattice lfp_flow)
                     man.lattice lfp_flow in
      let lfp_flow = Flow.set T_break (Flow.get T_break man.lattice flow)
                     man.lattice lfp_flow in
      Cases.return () lfp_return
    )

  | _ -> None
```

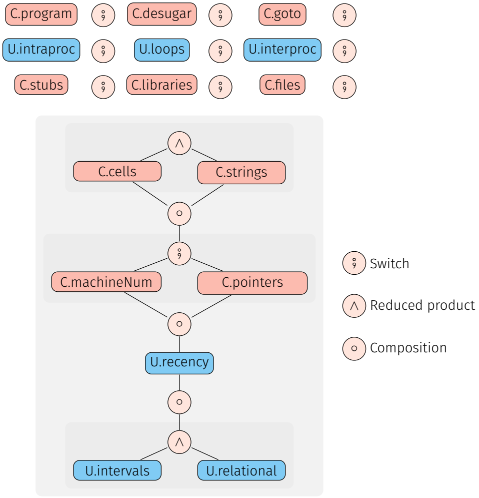
Standard flow transferred to *brk*

At loop end:
join *brk* and *cur* into *cur*

Current analyses in Mopsa

C analysis

C analysis overview



- Checks for run-time errors (integer overflows, invalid dereferences, . . .)
- Supports ints, floats, pointers, structs, . . .
- Inlining-based analysis
  ⚠ scalability
- No concurrency support

$$\textsc{Pointers} \stackrel{\text{def}}{=} \mathcal{V}_{ptr} \to \wp(\mathcal{V} \cup \{\texttt{NULL}, \texttt{INVALID}\})$$

1. Each pointer is mapped to the set of pointed bases (e.g, variables or dynamically allocated memory)
2. Offsets are ghost numerical variables: $\underline{offset}(\texttt{p})$
   Can express relations between offsets and numeric variables:

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

A low-level memory abstraction [Min06a]

- Translates memory accesses into scalar accesses
- `type cell = { base: var; offset: Z.t; typ: scalar_type }`
- Synthesis for memory dereferences
- Supports type-punning, pointer arithmetic. No bitfield support implemented

Lipari-themed example (little-endian case)

```c
uint32_t eax = 0xF0CACC1A;
uint8_t x = *((uint8_t *) &eax + 3);
```

$\mathbb{S}^\sharp[\![\,\texttt{uint32\_t eax = 0xF0CACC1A}\,]\!]$

$\quad \mathbb{S}^\sharp[\![\,\underline{\text{cell}}(\wr\texttt{eax}, 0, u32\wr) = \texttt{0xF0CACC1A}\,]\!]$

$\mathbb{S}^\sharp[\![\,\texttt{uint8\_t x = *((uint8\_t *) \&eax + 3);}\,]\!]$

$\quad \mathbb{E}^\sharp_{cells}[\![\,\texttt{*((uint8\_t *) \&eax + 3)}\,]\!]$

$\qquad \mathbb{S}^\sharp[\![\,\underline{\text{cell}}(\wr\texttt{eax}, 3, u8\wr) = \texttt{0xF0}\,]\!]$

$\quad \leftarrow \underline{\text{cell}}(\wr\texttt{eax}, 3, u8\wr)$

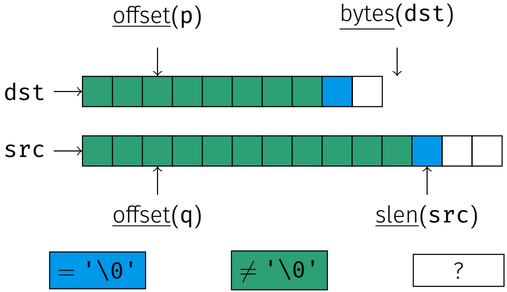
$\quad \mathbb{S}^\sharp[\![\, x = \underline{\text{cell}}(\wr\texttt{eax}, 3, u8\wr)\,]\!]$

String length domain [JMO18]

New ghost variables:

- bytes(var) size in bytes of memory block
- slen(var) position of first must 0.

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



String length domain

Conjunction of pre-conditions

The switch utility

Continuation (with pre-filtered state)

```
val switch : (expr list * ('a Flow.flow -> ('a,'r) cases)) list) ->
    ('a, 'b) man -> 'a flow -> ('a, 'r) cases
```

Transfer function of `base[offset] = rhs`

```
switch [
  (* set0 case *)
  (* Offset condition: offset ∈ [0, length] *)
  (* RHS condition: rhs = 0 *)
  (* Transformation: length := offset; *)
  [ mk_in offset zero length range;
    mk_eq rhs zero range ],
  man.exec (mk_assign length offset range)
  ;

  (* setnon0 case *)
  (* Offset condition: offset = length *)
  (* RHS condition: rhs ≠ 0 *)
  (* Transformation: length := [offset + 1, size]; *)
  [ mk_eq offset length range;
    mk_ne rhs zero range ],
  assign_length_interval (add offset one range) size
  ;
  (* First unchanged case *)
  (* Offset condition: offset ∈ [0, length - 1] *)
  (* RHS condition: rhs ≠ 0 *)
  (* Transformation: nop; *)
  [ mk_in offset zero (pred length range) range;
    mk_ne rhs zero range ],
  (fun flow -> Post.return flow)
  ;

  (* Second unchanged case *)
  (* Offset condition: offset >= length + 1 *)
  (* RHS condition: ⊤ *)
  (* Transformation: nop; *)
  [ mk_ge offset (succ length range) range ],
  (fun flow -> Post.return flow)

]
  man flow
```

libc stubs

- Annotation of Libc through a contract language [OM20]
- Inspired from Frama-C ACSL
- Contract language not restricted to C

strlen contract

```
/*$
 * requires: valid_string_or_fail(__s);
 * ensures : return ∈ [0, size(__s) - 1];
 * ensures : __s[return] == 0;
 * ensures : ∀ int i ∈ [0, return - 1]: __s[i] != 0;
 */
size_t strlen(const char *__s);
```

User-defined predicate, including
$\exists$`int i` $\in [0,$ `size(__s)` $-1]$: `__s[i] == 0`

Quantifier interpretation: delegated to domains

Some benchmarks

See SV-Comp 2024 results.

Benchmark	# Tests	Total LOC	Time	Precision
CWE121	2,508	234,930	3,064s	22.13%
CWE122	1,556	166,664	1,948s	25.84%
CWE124	758	93,372	961s	36.94%
CWE126	600	75,984	769s	46.83%
CWE127	758	89,022	963s	37.07%
CWE190	3,420	440,749	4,356s	78.13%
CWE191	2,622	340,884	3,236s	78.87%
CWE369	497	83,238	674s	70.42%
CWE415	190	17,990	228s	100.00%
CWE416	118	14,782	142s	67.80%
CWE469	18	1,520	22s	100.00%
CWE476	216	20,427	254s	100.00%

Table 1: Juliet benchmarks (non-relational configuration, no partitioning).

Benchmark	Time	Selectivity	# checks
basename	33.79s	98.65%	11,731
comm	42.67s	97.32%	12,654
dircolors	34.82s	99.74%	20,062
dirname	21.68s	99.61%	11,307
echo	19.26s	99.43%	11,010
false	14.50s	99.72%	10,774
getlimits	34.62s	98.54%	11,711
hostid	18.05s	99.65%	11,303
id	32.69s	99.04%	12,338
link	23.03s	99.52%	11,572
logname	20.36s	99.66%	11,307
mkfifo	34.87s	99.20%	11,807

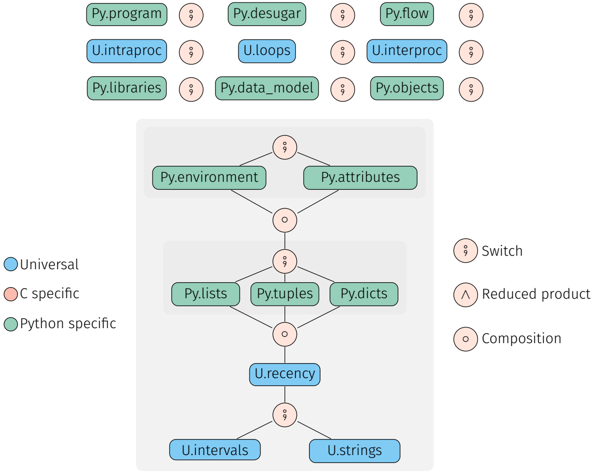
Table 2: `coreutils` benchmarks (fully symbolic arguments, relational analysis).

Current analyses in Mopsa

Python analysis

Python analysis overview

- Detects uncaught exceptions
- Type and value analyses available
- Supports crazy Python dynamic typing, semantics, . . .
- Does not support GC finalizers, `async`, `eval`



Python's dual type system

Nominal types: classes, MRO

Pointer-like domain

$\mathcal{V} \rightarrow \mathcal{P}(\text{Addr}^\sharp \cup \{\, \texttt{LocUndef}, \texttt{GlobUndef} \,\})$

Structural types: attributes

Attribute abstraction + ghost variables

$\text{Addr}^\sharp \rightarrow \text{ObjS}^\sharp$

Fspath (from standard library)

```python
class Path:
  def __fspath__(self): return 42

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

fspath("/dev" if random() else Path())
```

Attribute abstraction

Using an under and an over-approximation

$\mathsf{ObjS}^\sharp = \{ (l, u) \mid l \in \mathcal{P}(\mathit{string}), u \in \mathcal{P}(\mathit{string}) \cup \{ \top \}, l \subseteq u \vee u = \top \}$

Concretization

$$\gamma^\sharp_{\mathsf{ObjS}} : \begin{cases} \mathsf{ObjS}^\sharp & \rightarrow & \mathcal{P}(\mathcal{P}(\mathit{string})) \\ (l, \top) & \mapsto & \{ s \in \mathcal{P}(\mathit{string}) \mid l \subseteq s \} \\ (l, u) & \mapsto & \{ s \in \mathcal{P}(\mathit{string}) \mid l \subseteq s \subseteq u \} \end{cases}$$

Example

$\gamma^\sharp_{\mathsf{ObjS}}(\{ a \}, \{ a, b, c \}) = \{ \{ a \}, \{ a, b \}, \{ a, c \}, \{ a, b, c \} \}$

Recency abstraction

The recency abstraction [BR06]

- Precise analysis of object initialization
- Twofold partitioning:
  - by allocation site $l \in \mathbf{Loc}$
  - through a recency criterion: $(l, \boldsymbol{r})$ most recent allocation (with strong updates)
    $(l, \boldsymbol{o})$ older addresses (summarized)
- Initially designed for analysis of low-level code (binaries, C)
- Also used in Type Analysis for JavaScript [JMT09]

```python
class Task:
  def __init__(self, weight):
    if weight < 0: raise ValueError
    self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]
```

Return of ghost variables

Composed on top of address, for attribute "weight":

$\underline{@^\sharp(\text{Task}, r) \cdot \text{weight}} \mapsto [2, 2]$

$\left\{ \underline{@^\sharp(\text{Task}, r) \cdot \text{weight}} \mapsto [2, 2] \right.$

$\left\{ \begin{array}{l} \underline{@^\sharp(\text{Task}, r) \cdot \text{weight}} \mapsto [1, 1] \\ \underline{@^\sharp(\text{Task}, o) \cdot \text{weight}} \mapsto [2, 2] \end{array} \right.$

$\left\{ \begin{array}{l} \underline{@^\sharp(\text{Task}, r) \cdot \text{weight}} \mapsto [4, 4] \\ \underline{@^\sharp(\text{Task}, o) \cdot \text{weight}} \mapsto [1, 2] \end{array} \right.$

$\left\{ \begin{array}{l} \underline{\boldsymbol{@^\sharp(\text{Task}, r) \cdot \text{weight}}} \mapsto \mathbf{[5, 5]} \\ \underline{@^\sharp(\text{Task}, o) \cdot \text{weight}} \mapsto [1, 4] \end{array} \right.$

Recency abstraction – III

Task creation

```python
class Task:
  def __init__(self, weight):
    if weight < 0: raise ValueError
    self.weight = weight

m = [1, 2]
l = [Task(i) for i in m]
l.append(Task(3))
```

Type analysis

Nominal types used in abstract addresses. No need for allocation-site in `Tasks`. But helpful for lists!

Value analysis

Use allocation sites for `range` objects.

Variable allocation policies

- Type-based (nominal) and/or location-based partitioning.
- Different configurations depending on type/value analysis.

Comparison of the type and value analyses

Name	LOC	Type Analysis					Non-relational Value Analysis				
		Time	Mem.	Exceptions detected			Time	Mem.	Exceptions detected		
				Type	Index	Key			Type	Index	Key
`nbody.py`	157	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**1**	1
`scimark.py`	416	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	**0**	0
`richards.py`	426	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	**2**	0
`unpack_seq.py`	458	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	0
`go.py`	461	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	3	20	0
`hexiom.py`	674	1[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**21**	3
`regex_v8.py`	1792	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**145**	0
`processInput.py`	1417	1[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	7	**4**	1
`choose.py`	2562	1.[illegible]	[illegible]	[illegible]	22	7	2.9m	3.7GB	12	**13**	7
Total	9294	4.0m	2.8GB	59	2214	12	13m	9.1GB	59	228	12

Conclusion

The non-relational value analysis

- does not remove false type alarms
- significantly reduces index errors
- is $\simeq 3\times$ costlier

Heuristic packing and relational analyses

- Static packing, using function's scope
- Rules out all 145 alarms of **`regex_v8.py`** (1792 LOC) at $2.5\times$ cost

Selectivity of the non-relational value analysis

Name	Attributes	Types	Indexes	Keys	Values	Overflows	Divisions
scimark.py	746/746	844/844	**2**/5		**29**/30	**21**/43	**20**/21
richards.py	**352**/353	389/389	**2**/4		**2**/3		2/2
unpack_seq.py	807/807	1210/1210			1/1		
go.py	**664**/697	728/728	**2**/20		7/7	**6**/12	**4**/6
hexiom.py	598/598	672/672	**10**/32	**0**/3	**23**/24		
regex_v8.py	7357/7357	8349/8349	**1913**/2057		63/63		
processInput.py	**617**/619	**790**/792	12/12	**0**/1	**0**/1	2/2	
choose.py	**2519**/2521	**2997**/2999	**28**/39	**4**/8	**9**/24	**7**/17	

Selectivity of the analysis on some classes of exceptions

Selectivity = Number of proved safe operations / Total number of checks

An empty cell denotes a program where the kind of exception cannot happen

Current analyses in Mopsa

Python+C analysis

Python+C analysis overview

Assessment 20% of the 200 most popular Python libraries rely on C code

- Performance (numpy)
- System libraries (pygit2)

Dangers

- Different values ($\mathbb{Z}$ vs. `Int32`)
- Shared memory state

Our approach

- **Combined** analysis of C, Python and interface code
- Previous works [TM07; FF08; LLR20] : JNI ⚠⇝ Java, low precision

Multilanguage code – example

counter.c

```c
typedef struct {
    PyObject_HEAD;
    int count;
} Counter;

static PyObject*
CounterIncr(Counter *self, PyObject *args)
{
    int i = 1;
    if(!PyArg_ParseTuple(args, "|i", &i))
        return NULL;

    self->count += i;
    Py_RETURN_NONE;
}

static PyObject*
CounterGet(Counter *self)
{
    return Py_BuildValue("i", self->count);
}
```

count.py

```python
from counter import Counter
from random import randrange

c = Counter()
power = randrange(128)
c.incr(2**power-1)
c.incr()
r = c.get()
```

- $\texttt{power} \leq 30 \Rightarrow \texttt{r} = 2^{\texttt{power}}$
- $\texttt{power} = 31 \Rightarrow \texttt{r} = -2^{31}$
- $32 \leq \texttt{power} \leq 64$: OverflowError: signed integer is greater than maximum
- $\texttt{power} \geq 64$: OverflowError: Python int too large to convert to C long

High-level idea

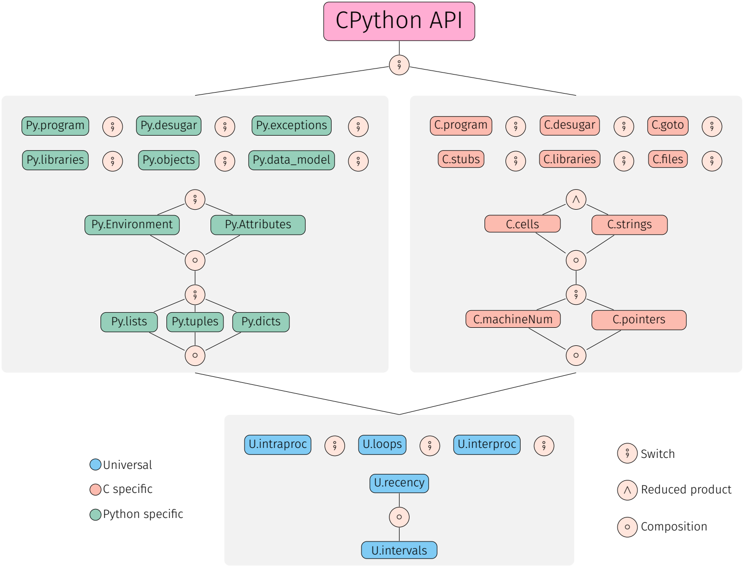
Difficulty: shared memory

- Each language may change the memory state, and has a different view of it
- Synchronization? We could perform a full state translation, but
  - the cost would be high in the analysis
  - some abstractions can be shared between Python and C

State separation ⇝ reduced synchronization

- Observation: structures are directly dereferenceable by one language only
- Switch to other language otherwise (`c.incr()` ⇝ `self->count += 1`)
  Additional hypothesis: C accesses to Python objects through the API
- Synchronization: only when objects change language for the first time
- Mopsa supports shared abstractions

From distinct Python and C analyses. . . to a multilanguage analysis!



Multilanguage analysis benchmarks

Corpus selection

- Popular, real-world libraries available on GitHub, averaging 412 stars.
- Whole-program analysis: we use the tests provided by the libraries.

Library	C + Py. Loc	Tests	🕒/test	$\frac{\text{\# proved checks}}{\text{\# checks}}$ %	# checks
noise	1397	15/15	1.2s	99.7%	6690
cdistance	2345	28/28	4.1s	98.0%	13716
llist	4515	167/194	1.5s	98.8%	36255
ahocorasick	4877	46/92	1.2s	96.7%	6722
levenshtein	5798	17/17	5.3s	84.6%	4825
bitarray	5841	159/216	1.6s	94.9%	25566

Easing development

General advice around static analysis development

Developing sound, precise, scalable, static analyzers is a challenge!

- Investing time in creating specific tools can help a lot!
  Examples in the remainder of this section.
- Reproducible science
  Experimental results should be reproducible, through artifacts
- Career
  - tool dev. vs papers
  - software life and changing jobs

Easing development

CI, tests & benchmarks

CI, tests & benchmarks

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

- third-party real code
- open-source – for the sake of reproducible science
- unmodified*
  - Underscores practicality of our approach
  - * stubs can be added in marginal cases

Comparing analysis reports

`mopsa-diff` script

- compares analysis report(s): either single output or set of outputs
- usecases: different configurations, different versions of Mopsa

```
--- baseline/touch-many-symbolic-args-a4.json
+++ pplite/touch-many-symbolic-args-a4.json

- time: 589.0760
+ time: 675.1761

+ parse-datetime.y:1399.44-46: alarm: Invalid memory access
- parse-datetime.y:965.56-71: alarm: Invalid memory access
- parse-datetime.y:980.25-52: alarm: Invalid memory access
- parse-datetime.y:1003.23-50: alarm: Invalid memory access
- parse-datetime.y:921.56-71: alarm: Invalid memory access
- parse-datetime.c:1733.2-8: alarm: Invalid memory access
- parse-datetime.y:781.26-41: alarm: Invalid memory access
- parse-datetime.y:772.23-38: alarm: Invalid memory access
- parse-datetime.y:755.23-38: alarm: Invalid memory access
- parse-datetime.y:973.25-52: alarm: Invalid memory access
- parse-datetime.y:610.8-41: alarm: Invalid memory access
- parse-datetime.y:743.25-40: alarm: Invalid memory access
```

```
139 reports compared
avg. time change       +52.065s
avg. speedup               -36%
new alarms                    2
removed alarms               32
new assumptions               0
removed assumptions           0
new successes                 0
new failures                  0
```

Easing development

Static analyzer interfaces

Where static analyzers usually start from

- Analysis output — Too coarse
- Printing abstract state using builtins — Not interactive
- Interpretation trace — Can be dozens of gigabytes of text

```
+ S [| set_program_name(argv[0]); |]
| | | + S [| add(argv0)
| | | |       argv0 = argv[0]; |]
| | | | + S [| add(argv0) |]
| | | | | + S [| add(argv0) |] in below(c.iterators.intraproc)
| | | | | | + S [| add(argv0) |] in C/Scalar
| | | | | | | + S [| add(offset⦃argv0⦄) |] in Universal
| | | | | | | o S [| add(offset⦃argv0⦄) |] in Universal done [0.0001s, 1 case]
| | | | | | o S [| add(argv0) |] in C/Scalar done [0.0001s, 1 case]
| | | | | | + S [| add(argv0) |] in below(c.memory.lowlevel.cells)
| | | | | | | + S [| add(offset⦃argv0⦄) |] in Universal
| | | | | | | o S [| add(offset⦃argv0⦄) |] in Universal done [0.0001s, 1 case]
| | | | | | o S [| add(argv0) |] in below(c.memory.lowlevel.cells) done [0.0001s, 1 case]
| | | | | o S [| add(argv0) |] in below(c.iterators.intraproc) done [0.0001s, 1 case]
| | | | o S [| add(argv0) |] done [0.0002s, 1 case]
| | | | + S [| argv0 = argv[0]; |]
| | | | | + S [| argv0 = (signed char *) @argv{0}:ptr; |] in below(c.iterators.intraproc)
| | | | | | + S [| argv0 = (signed char *) @argv{0}:ptr; |] in C/Scalar
| | | | | | | + S [| offset⦃argv0⦄ = (offset⦃@argv{0}:ptr⦄ + 0); |] in Universal
| | | | | | | | + S [| offset⦃argv0⦄ = (offset⦃@argv{0}:ptr⦄ + 0); |] in below(universal.iterators.intraproc)
```

An interactive engine acting as abstract debugger

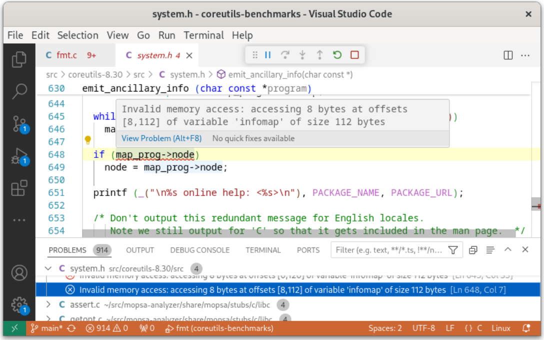
GDB-like interface to the abstract interpretation of the program

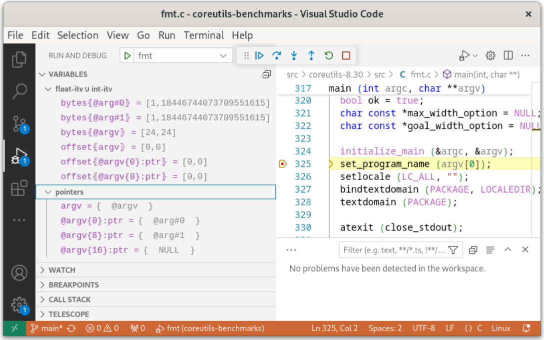
Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm
- Navigation
- Observation of the abstract state
  - Full state
  - Projection on specific variables
- Some scripting capabilities

IDE support

- Language Server Protocol for linters (report alarms)
- Debug Adapter Protocol providing interactive engine interface
- Both protocols introduced by VSCode, supported by multiple IDEs





Easing development

Plug-ins to observe the analysis

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis
- Thresholds for widening
- Coverage
- Heuristic unsoundness/imprecision detection
- Profiling

Coverage hooks

Coverage

- Global metric for the analysis' results
- Good to detect issues in the instrumentation of the fully context-sensitive analysis

No symbolic argument

```
./src/coreutils-8.30/src/fmt.c:
    'main' 76% of 72 statements analyzed
    'set_prefix' 100% of 12 statements analyzed
    'same_para' 100% of 1 statement analyzed
    'get_line' 100% of 30 statements analyzed
    'fmt' 100% of 7 statements analyzed
    'base_cost' 100% of 16 statements analyzed
    'line_cost' 100% of 10 statements analyzed
    'get_prefix' 100% of 18 statements analyzed
```

Symbolic arguments

```
./src/coreutils-8.30/src/fmt.c:
    'main' 100% of 72 statements analyzed
```

Heuristic unsoundness/imprecision detection

Detection of unsound transfer functions

Bottom shouldn't appear after some statements (such as assignments).

Goblint: ensure that at least one branch of a conditional is analyzed.

Detection of imprecise analysis

Warns when assignments to top are performed

Simplifies the search for sources of large imprecision

Profiling

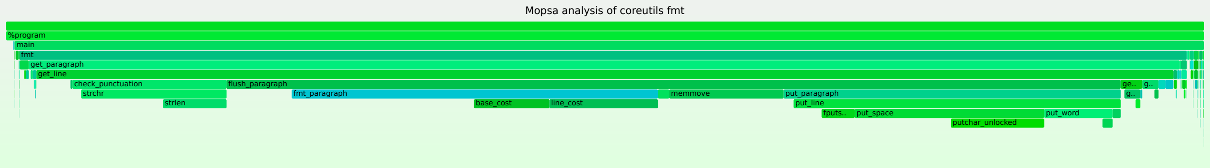
Standard profiling

Measures which parts of Mopsa are the most time-consuming

Abstract profiling hook

Measures which parts of the analyzed program are the most time-consuming

- Loop-level profiling
- Function-level profiling



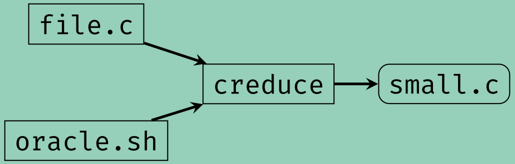
Easing development

Testcase reduction

Motivation

- Static analyzers are complex piece of code and may contain bugs
- In practice, some bugs will only be detected in large codebases
- Debugging extremely difficult: size of the program, analysis time

Automated testcase reduction using `creduce` [Reg+12]



Testcase reduction – II

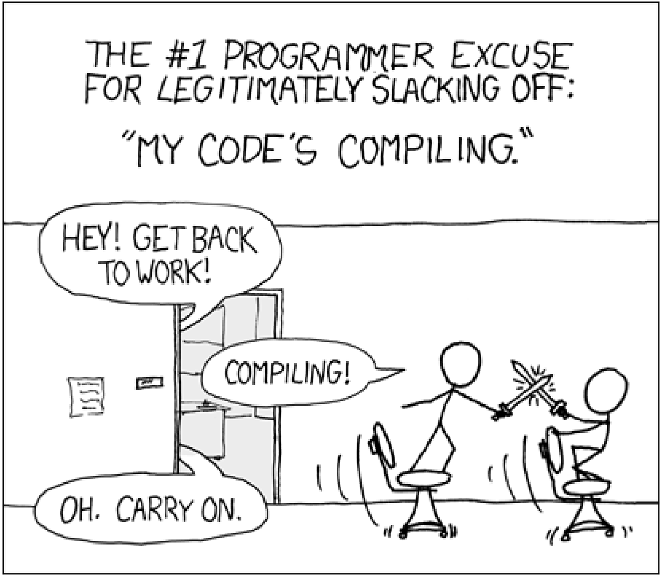
Internal errors debugging

- Highly helpful to significantly reduce debugging time of runtime errors (Apron mishandlings, raised exceptions, …)
- Has been applied to coreutils programs, SV-Comp programs of 10,000+ LoC

Differential-configuration debugging

```
$ mopsa-c -config=confA.json file.c
Alarm: assertion failure
$ mopsa-c -config=confB.json file.c
No alarm
```

Has been used to simplify cases in externally reported soundness issues


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

Handling multi-file projects

`creduce` limited to reducing a specific file

Mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`
  - Records compiler/linker calls and their options
  - Creates a compilation database

  ⇝ `mopsa-build make` drop-in replacement for `make`
- `mopsa-c` leverages the compilation database

```
mopsa-c mopsa.db -make-target=fmt
```

- Option to generate a single, preprocessed file

Conclusion

Some other approaches

External fixpoint engine

- Mopsa: each iterator (loops, gotos, calls) defines its fixpoint computation.
- Alternative: unified, external fixpoint engine
- Used by Goblint team [Saa+24], Lermusiaux and Montagu [LM24].

Systematic relationship between concrete and abstract domains

- See e.g, Michelland, Zakowski, and Gonnord [MZG24], Keidel and Erdweg [KE19]
- Lighter than a formally verified analyzer? [Jou+15]

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Goals: explore new designs, ease development of (relational) analyses

One AST to rule them all
- Multilanguage support
- Expressiveness
- Reusability

Unified domain signature
- Semantic rewriting
- Loose coupling
- Observability

DAG of abstractions
- Relational domains
- Composition
- Cooperation

Feedback wanted!

Anonymous survey, or come and talk to me!

References – I

[Bag+06] Roberto Bagnara et al. "Grids: A Domain for Analyzing the Distribution of Numerical Values". In: Lecture Notes in Computer Science. Springer, 2006, pp. 219–235.

[Bau+22] Guillaume Bau et al. "Abstract interpretation of Michelson smart-contracts". In: ed. by Laure Gonnord and Laura Titolo. ACM, 2022, pp. 36–43. DOI: 10.1145/3520313.3534660.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems". In: Sci. Comput. Program. 1-2 (2008), pp. 3–21.

References – II

[BR06] G. Balakrishnan and T. W. Reps. "Recency-Abstraction for Heap-Allocated Storage". In: LNCS. Springer, 2006, pp. 221–239.

[BZ20] Anna Becchi and Enea Zaffanella. "PPLite: Zero-overhead encoding of NNC polyhedra". In: Inf. Comput. (2020), p. 104620. DOI: 10.1016/J.IC.2020.104620.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: 1978.

[Cou+06] Patrick Cousot et al. "Combination of Abstractions in the Astrée Static Analyzer". In: 2006.

References – III

[DM19] David Delmas and Antoine Miné. "Analysis of Software Patches Using Numerical Abstract Interpretation". In: ed. by Bor-Yuh Evan Chang. Lecture Notes in Computer Science. Springer, 2019, pp. 225–246. DOI: 10.1007/978-3-030-32304-2_12.

[DOM21] David Delmas, Abdelraouf Ouadjaout, and Antoine Miné. "Static Analysis of Endian Portability by Abstract Interpretation". In: Lecture Notes in Computer Science. Springer, 2021, pp. 102–123.

[FF08] Michael Furr and Jeffrey S. Foster. "Checking type safety of foreign function calls". In: ACM Trans. Program. Lang. Syst. (2008).

References – IV

[Gop+04] D. Gopan et al. “Numeric Domains with Summarized Dimensions”. In: LNCS. Springer, 2004, pp. 512–529.

[JM09] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: Lecture Notes in Computer Science. Springer, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[JMO18] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. “Modular Static Analysis of String Manipulations in C Programs”. In: ed. by Andreas Podelski. Lecture Notes in Computer Science. Springer, 2018, pp. 243–262. DOI: 10.1007/978-3-319-99725-4_16.

References – V

[JMT09] S. H. Jensen, A. Møller, and P. Thiemann. "Type Analysis for JavaScript". In: LNCS. Springer, 2009, pp. 238–255.

[Jou+15] Jacques-Henri Jourdan et al. "A Formally-Verified C Static Analyzer". In: ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 247–259. DOI: 10.1145/2676726.2676966.

[Jou+19] M. Journault et al. "Combinations of reusable abstract domains for a multilingual static analyzer". In: New York, USA, July 2019, pp. 1–17.

[KE19] Sven Keidel and Sebastian Erdweg. "Sound and reusable components for abstract interpretation". In: Proc. ACM Program. Lang. OOPSLA (2019), 176:1–176:28. DOI: 10.1145/3360602.

References – VI

[Liv+15] Benjamin Livshits et al. "In defense of soundiness: a manifesto". In: Commun. ACM 2 (2015), pp. 44–46. DOI: 10.1145/2644805.

[LLR20] Lee, Lee, and Ryu. "Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis". In: 2020.

[LM24] Pierre Lermusiaux and Benoît Montagu. "Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation". In: ed. by Stephanie Weirich. Lecture Notes in Computer Science. Springer, 2024, pp. 391–420. DOI: 10.1007/978-3-031-57267-8_15.

References – VII

[MFM24] Raphaël Monat, Aymeric Fromherz, and Denis Merigoux. "Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law". In: ed. by Stephanie Weirich. Lecture Notes in Computer Science. Springer, 2024, pp. 421–450. DOI: 10.1007/978-3-031-57267-8_16.

[Min06a] Antoine Miné. "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics". In: ed. by Mary Jane Irwin and Koen De Bosschere. ACM, 2006, pp. 54–63. DOI: 10.1145/1134650.1134659.

[Min06b] Antoine Miné. "The octagon abstract domain". In: High. Order Symb. Comput. (2006).

References – VIII

[MM24] Marco Milanese and Antoine Miné. “Generation of Violation Witnesses by Under-Approximating Abstract Interpretation”. In: ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Lecture Notes in Computer Science. Springer, 2024, pp. 50–73. DOI: 10.1007/978-3-031-50524-9_3.

[MOM20a] R. Monat, A. Ouadjaout, and A. Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. In: LIPIcs. 2020.

[MOM20b] R. Monat, A. Ouadjaout, and A. Miné. “Value and allocation sensitivity in static Python analyses”. In: ACM, 2020, pp. 8–13. DOI: 10.1145/3394451.3397205.

References – IX

[MOM21] R. Monat, A. Ouadjaout, and A. Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. In: 2021.

[Mon+24] Raphaël Monat et al. “Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)”. In: ed. by Bernd Finkbeiner and Laura Kovács. Lecture Notes in Computer Science. Springer, 2024, pp. 387–392. DOI: 10.1007/978-3-031-57256-2_26.

[Mon21] Raphaël Monat. “Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries”. PhD thesis. Sorbonne Université, France, 2021.

References – X

[MZG24] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. "Abstract Interpreters: A Monadic Approach to Modular Verification". In: Proc. ACM Program. Lang. ICFP (Aug. 2024). DOI: 10.1145/3674646.

[NP18] Kedar S. Namjoshi and Zvonimir Pavlinovic. "The Impact of Program Transformations on Static Program Analysis". In: ed. by Andreas Podelski. Lecture Notes in Computer Science. Springer, 2018, pp. 306–325. DOI: 10.1007/978-3-319-99725-4_19.

References – XI

[OM20] A. Ouadjaout and A. Miné. "A Library Modeling Language for the Static Analysis of C Programs". In: ed. by David Pichardie and Mihaela Sighireanu. Lecture Notes in Computer Science. Springer, 2020, pp. 223–247. DOI: 10.1007/978-3-030-65474-0_11.

[PM24] Francesco Parolini and Antoine Miné. "Sound Abstract Nonexploitability Analysis". In: ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Lecture Notes in Computer Science. Springer, 2024, pp. 314–337. DOI: 10.1007/978-3-031-50521-8_15.

[Reg+12] John Regehr et al. "Test-case reduction for C compiler bugs". In: ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 335–346. DOI: 10.1145/2254064.2254104.

References – XII

[Saa+24] Simmo Saan et al. “Goblint: Abstract Interpretation for Memory Safety and Termination - (Competition Contribution)”. In: ed. by Bernd Finkbeiner and Laura Kovács. Lecture Notes in Computer Science. Springer, 2024, pp. 381–386. DOI: 10.1007/978-3-031-57256-2_25.

[TM07] Gang Tan and Greg Morrisett. “Ilea: inter-language analysis across Java and C”. In: 2007.

[VMM23] Milla Valnet, Raphaël Monat, and Antoine Miné. “Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs”. In: ed. by Timothy Bourke and Delphine Demange. Praz-sur-Arly, France, Jan. 2023, pp. 211–242.