MOPSA

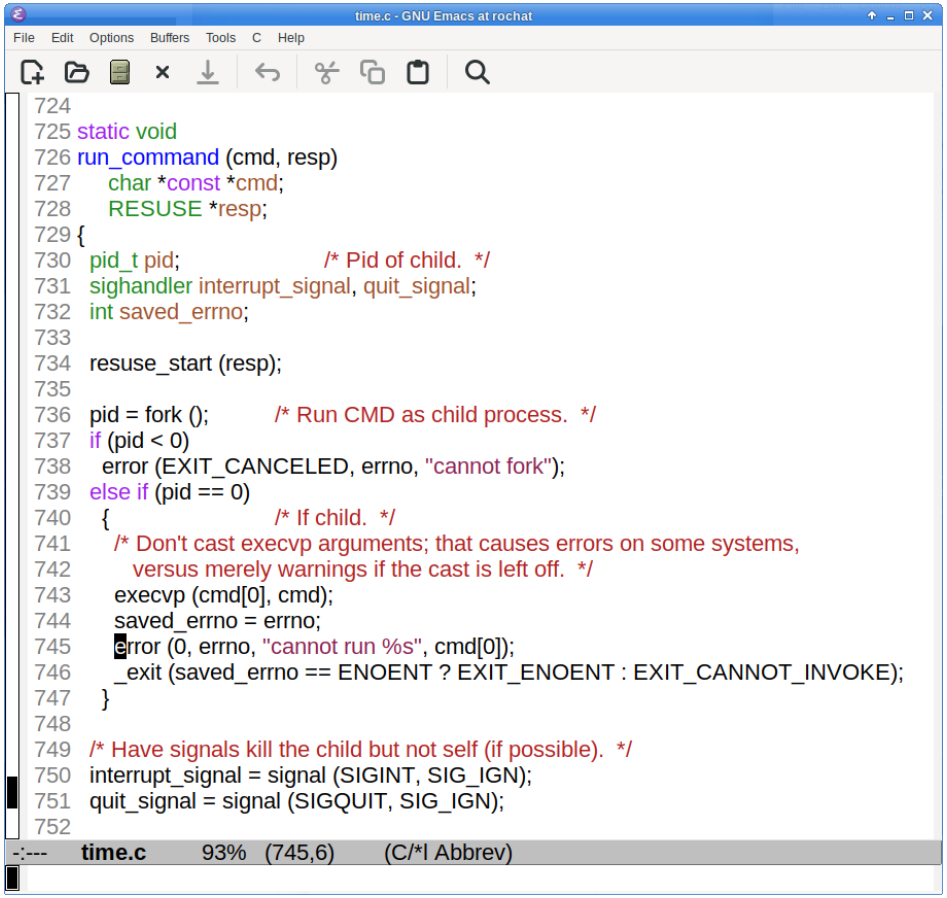
Multilingual Static Analysis for Program Verification using Abstract Interpretation

Guillaume Bau Jérôme Boillot David Delmas
Matthieu Journault Marco Milanese Antoine Miné
Raphaël Monat Abdelraouf Ouadjaout Francesco Parolini
Milla Valnet

APR team
LIP6, Sorbonne Université
Paris, France

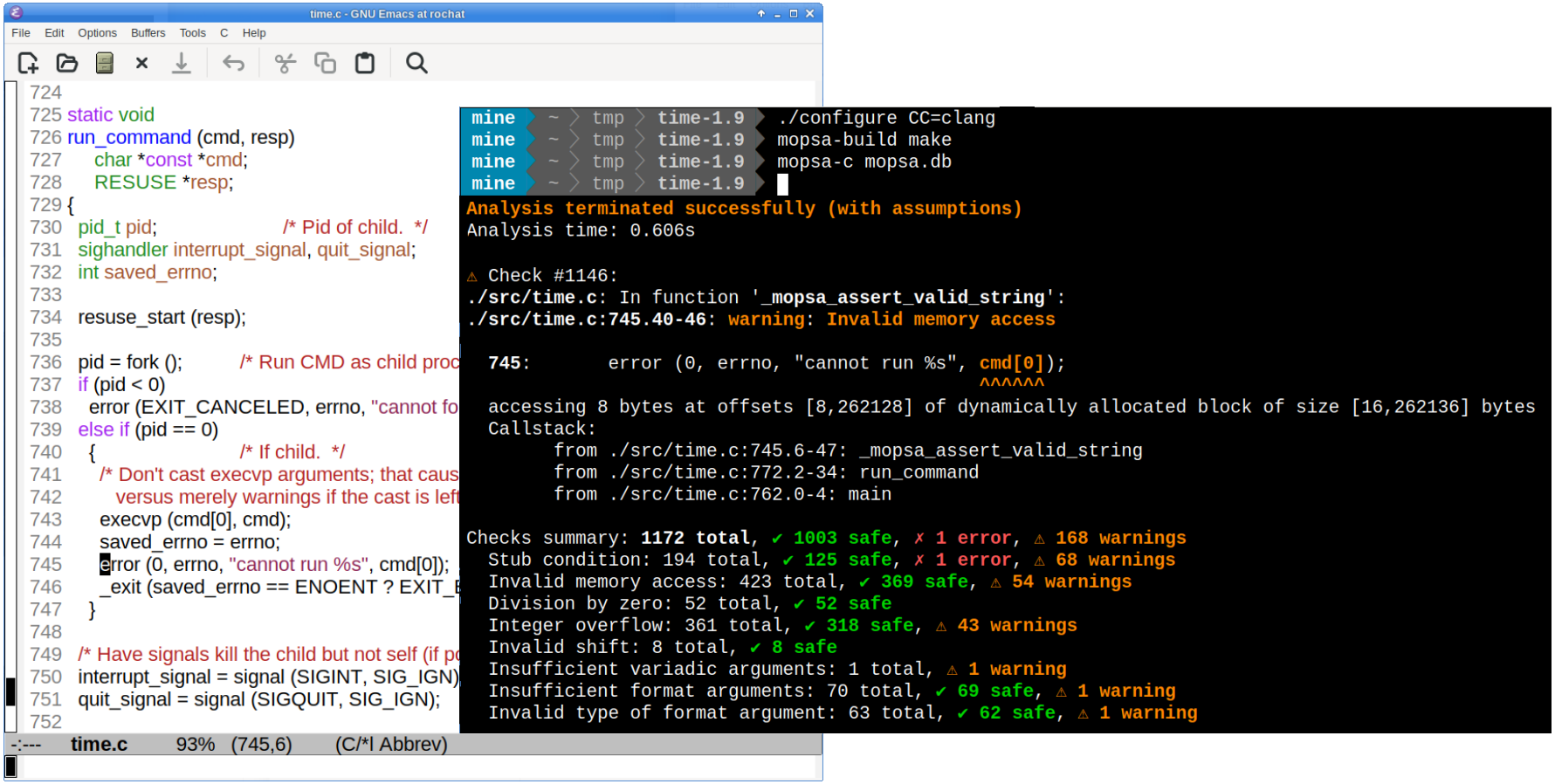
SWHSec Conference
06/06/2024
Cyber Campus, Paris

Program verification by static analysis



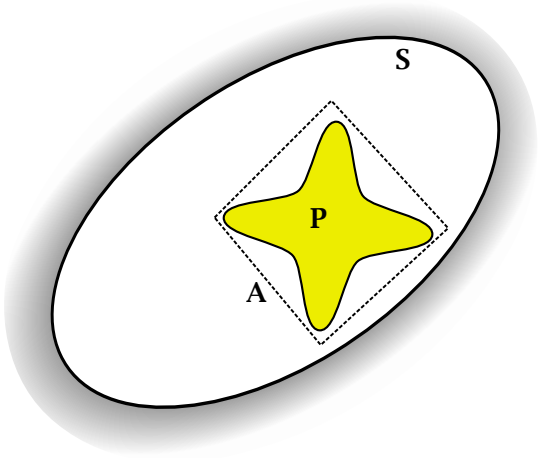
- analyze source code at compile time, without executing it
- fully automatic (no annotation, no interaction), moderately efficient
- report on non-functional correctness and assertion violations
  run-time errors, CWE, coding guidelines, etc.

Program verification by static analysis

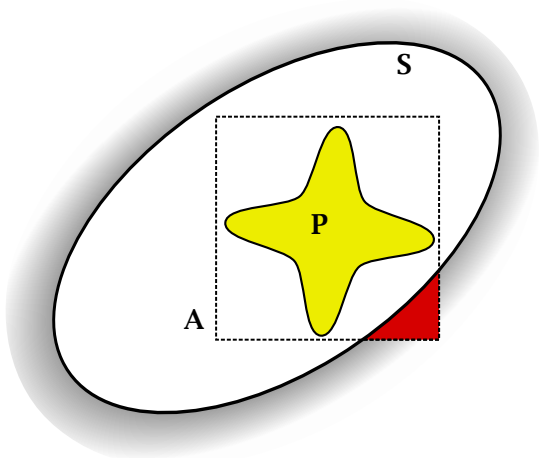


- analyze source code at compile time, without executing it
- fully automatic (no annotation, no interaction), moderately efficient
- report on non-functional correctness and assertion violations
  run-time errors, CWE, coding guidelines, etc.

Sound semantic static analysis



sound and precise analysis



false alarm

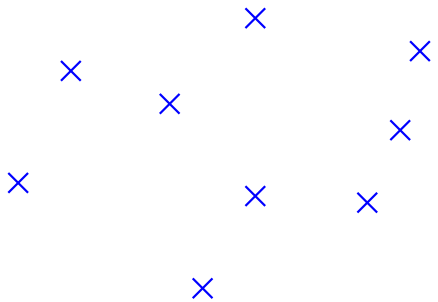
- compute an over-approximation A of reachable program states P and compare to a specification S
- $P \subseteq A$ by construction, guaranteed by abstract interpretation theory
- $A \subseteq S$: no bad state found $\Longrightarrow$ guarantee the absence of error!
- $A \not\subseteq S$: bad state found $\Longrightarrow$ inconclusive (true or false alarm)

A formal method sound and incomplete for the proof of absence of errors

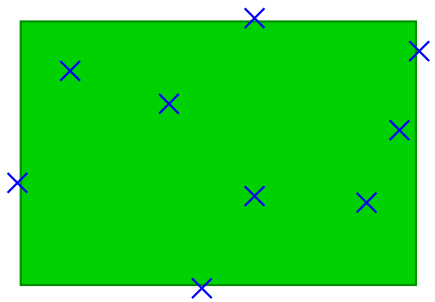
$\neq$ SAST, linters, but also $\neq$ model-checkers, deductive verification, theorem provers

Abstract domains:

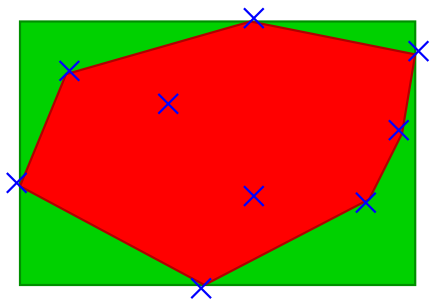
- computable, efficient symbolic representations of reachable states
  $\Longrightarrow$ non-representable sets are over-approximated



concrete memory states



box abstraction



polyhedral abstraction
relational abstraction

Abstract Interpretation

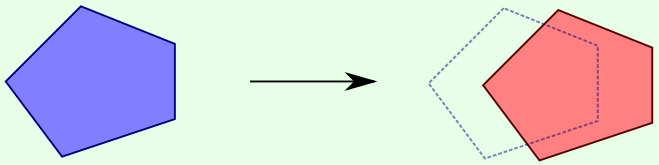
Efficiently computing over-approximations

Abstract domains:

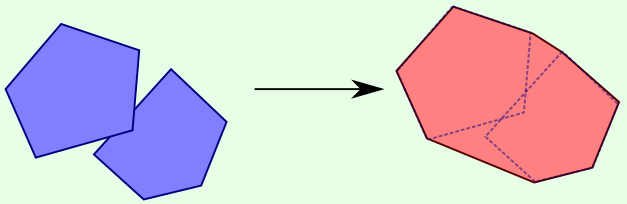
- computable, efficient semantic operators in the abstract world
  with soundness proof
- iterators traverse the program (CFG or AST)

Example: Polyhedra operations

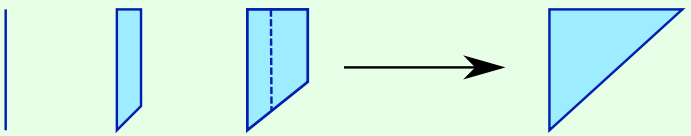
Assignments
• `X ← X + 1` •
translation



Branches: join
if ··· **then** ··· • **else** ··· • **fi** •
convex hull



Loops: inductive invariants
while • ··· **do** ··· **done**
iteration with widening ▽



MOPSA

MOPSA

Modular Open Platform for Static Analysis

Goal: open-source static analysis platform

- for research and education in abstract interpretation
- easy to extend for new languages and new properties
- develop, test, reuse abstractions
- open-source (LGPL)

Contributors

Guillaume Bau Jérôme Boillot David Delmas
Matthieu Journault Marco Milanese Antoine Miné
Raphaël Monat Abdelraouf Ouadjaout
Francesco Parolini Milla Valnet

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

MOPSA characteristics

Some classic aspects:

- whole-program forward abstract interpretation
- by induction on the AST
- in a collection of communicating abstract domains, sound by design
- analyze run-time errors in C programs

More original aspects:

- new languages: Python, OCaml
- multi-language: programs mixing C and Python
- new properties: non-regression for patches, endianess portability, exploitability, inferring counter examples
- experiments in new analysis engineering and architecture...
- academic implementation! not industrial-scale! various levels of maturity!

Feature: Extensible syntax tree for multiple languages

Classic design: common intermediate representation (JVM, LLVM, Clight,...)

- ✔ low-level: few and simple language constructs
- ✗ loss of high-level information and structure
- ✔ maximum sharing of abstract domains, easily retargetable (?)
- ✗ ...maybe not, if the language is too widely different! (Python?)

MOPSA design: extensible AST

- ✔ preserve the original AST of each source language
- ✔ and relevant fragments supported by well-known domains
  e.g., scalar fragment, pointer-free fragment, integer arithmetic
- ✔ translation is dynamic, more power (exploit abstract information)
- ✗ less efficient
- ✗ hard to keep track of sub-languages, possible match failures at run-rime

Feature: Extensible syntax tree for multiple languages

Classic design: common intermediate representation (JVM, LLVM, Clight,...)

- ✔ low-level: few and simple language constructs
- ✗ loss of high-level information and structure
- ✔ maximum sharing of abstract domains, easily retargetable (?)
- ✗ ...maybe not, if the language is too widely different! (Python?)

MOPSA design: extensible AST

- ✔ preserve the original AST of each source language
- ✔ and relevant fragments supported by well-known domains
  e.g., scalar fragment, pointer-free fragment, integer arithmetic
- ✔ translation is dynamic, more power (exploit abstract information)
- ✗ less efficient
- ✗ hard to keep track of sub-languages, possible match failures at run-rime

Feature: Distributed iterators and delegation

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

simple loops: `Universal.iterators.loops`

- matches `S_while`
- computes a fixpoint with acceleration ∇

C loops: `C.iterators.loops`

- matches `S_c_for (init, cond, incr, body)`
- rewrite into `S_while`
- and call interpreter recursively

```
init;
while (cond) {
  body;
  incr;
}
del init;
```

Python loops: `Python.desugar.loops`

- matches `S_py_for (target, iterable, body)`
- rewrite into `S_while`
- optimize simpler cases based on dynamic types (e.g., ranges)
- and call interpreter recursively

```
it = iter(iterable);
while (1) {
  try:  target = next(it);
  except:  StopIteration:  break;
  body;
}
del it, target;
```

In MOPSA, iterators are just abstract domains... without internal abstract state

Feature: Distributed iterators and delegation

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

simple loops: `Universal.iterators.loops`

- matches `S_while`
- computes a fixpoint with acceleration ∇

C loops: `C.iterators.loops`

- matches `S_c_for (init, cond, incr, body)`
- rewrite into `S_while`
- and call interpreter recursively

Python loops: `Python.desugar.loops`

- matches `S_py_for (target, iterable, body)`
- rewrite into `S_while`
- optimize simpler cases based on dynamic types (e.g., ranges)
- and call interpreter recursively

```
init;
while (cond) {
  body;
  incr;
}
del init;
```

```
it = iter(iterable);
while (1) {
  try:  target = next(it);
  except:  StopIteration:  break;
  body;
}
del it, target;
```

In MOPSA, iterators are just abstract domains... without internal abstract state

Feature: Distributed iterators and delegation

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

simple loops: `Universal.iterators.loops`

- matches `S_while`
- computes a fixpoint with acceleration ∇

C loops: `C.iterators.loops`

- matches `S_c_for (init, cond, incr, body)`
- rewrite into `S_while`
- and call interpreter recursively

```
init;
while (cond) {
  body;
  incr;
}
del init;
```

Python loops: `Python.desugar.loops`

- matches `S_py_for (target, iterable, body)`
- rewrite into `S_while`
- optimize simpler cases based on dynamic types (e.g., ranges)
- and call interpreter recursively

```
it = iter(iterable);
while (1) {
  try:  target = next(it);
  except:  StopIteration:  break;
  body;
}
del it, target;
```

In MOPSA, iterators are just abstract domains. . . without internal abstract state

Feature: Dynamic expression rewriting with case analysis

Flexible, modular way for domains to communicate:

- rewrite (part of) expressions, introduce new variables (embed abstractions)
- delegate to other domains (unaware of other abstractions, decoupling)
- possible perform case analysis with disjunctions (split the abstract state)

Example: string length domain

Abstracts character array a as $P_a : a[v_a] = 0 \wedge (\forall k \in [0, v_a[: a[k] \neq 0)$

- maintain internally a map $a \mapsto P_a$
- introduce an integer variable v_a for each P_a
- rewrite a sub-expression $a[i] == 0$ in abstract element $X^\sharp$ into a disjunction:
  - $(0, \mathbb{S}[\![i < v_a]\!] X^\sharp)$
  - $(1, \mathbb{S}[\![i = v_a]\!] X^\sharp)$
  - $([0, 1], \mathbb{S}[\![i > v_a]\!] X^\sharp)$

  keep some relationality between i, $a[i]$, and v_a
- $\mathbb{S}[\![i < v_a]\!]$, etc. are handled by (relational) numeric abstract domains

Feature: Dynamic expression rewriting with case analysis

Flexible, modular way for domains to communicate:

- rewrite (part of) expressions, introduce new variables (embed abstractions)
- delegate to other domains (unaware of other abstractions, decoupling)
- possible perform case analysis with disjunctions (split the abstract state)

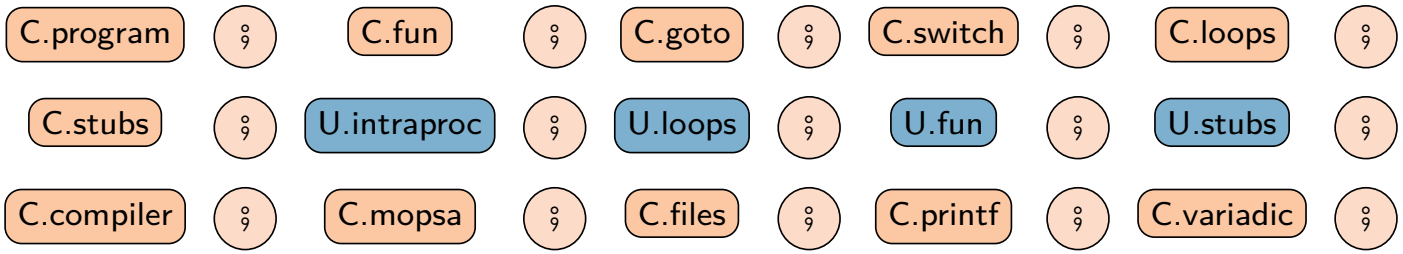
Example: string length domain

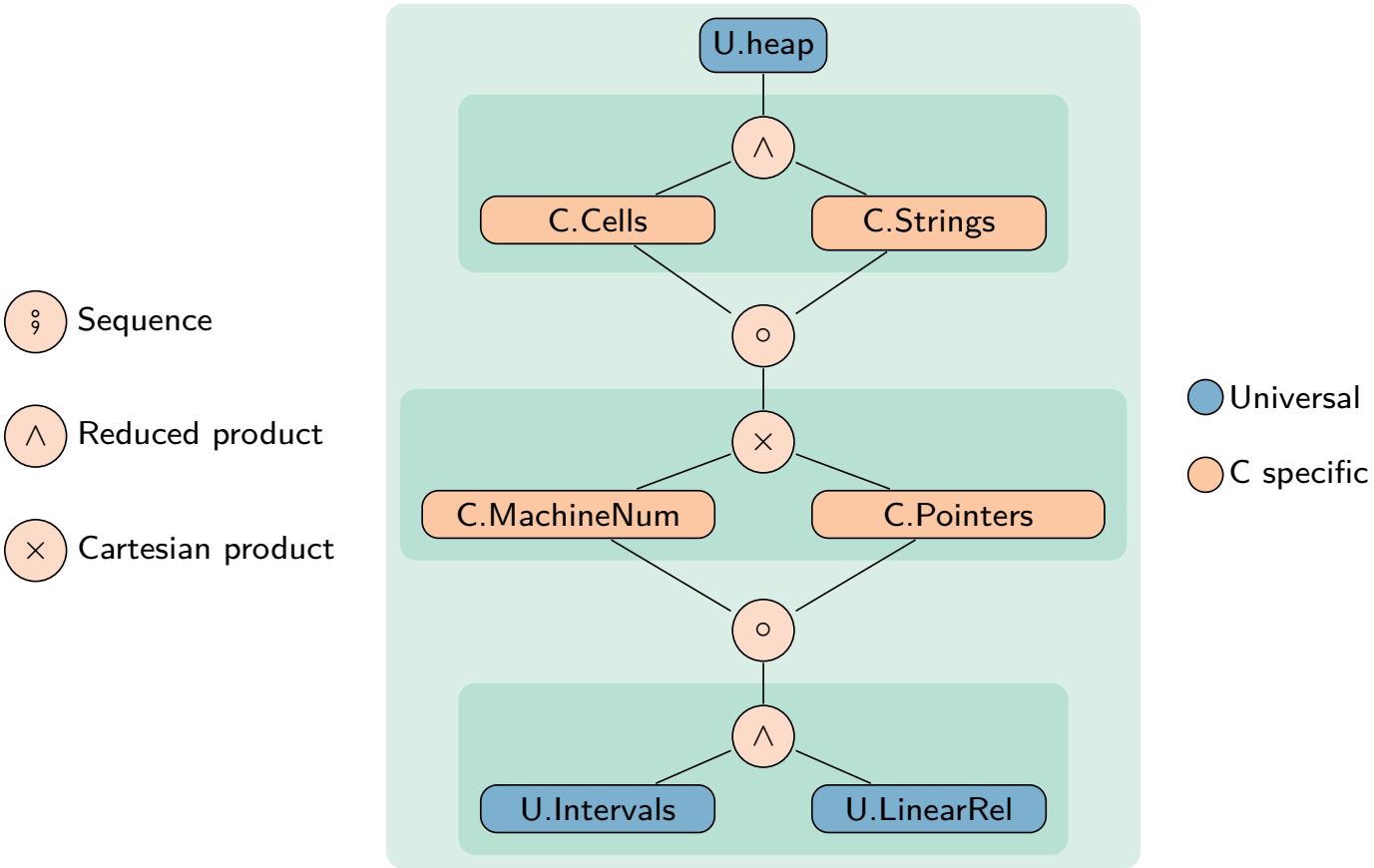
Abstracts character array a as $P_a : a[v_a] = 0 \wedge (\forall k \in [0, v_a[: a[k] \neq 0)$

- maintain internally a map $a \mapsto P_a$
- introduce an integer variable v_a for each P_a
- rewrite a sub-expression $a[i] == 0$ in abstract element $X^\sharp$ into a disjunction:
  - $(0, \mathbb{S}[\![i < v_a]\!] X^\sharp)$
  - $(1, \mathbb{S}[\![i = v_a]\!] X^\sharp)$
  - $([0, 1], \mathbb{S}[\![i > v_a]\!] X^\sharp)$

  keep some relationality between i, $a[i]$, and v_a
- $\mathbb{S}[\![i < v_a]\!]$, etc. are handled by (relational) numeric abstract domains

Example: Domains for a C analysis





Modeling C libraries

Requirement

For soundness, we need the C source or a model of every function called

stub for 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, ...);
```

- logic-based contract language, but with C expressions (sweet spot)
- stubs are executed at each call (not a specification to verify)
- for MOPSA: just another language (delegation, rewriting, etc.)
  - add abstract semantic operators for $\forall$, $\exists$, etc.
  - general notion of resources (generalizes dynamic memory allocation)

Modeling C libraries

Requirement

For soundness, we need the C source or a model of every function called

stub for 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, ...);
```

- logic-based contract language, but with C expressions (sweet spot)
- stubs are executed at each call (not a specification to verify)
- for MOPSA: just another language (delegation, rewriting, etc.)
  - add abstract semantic operators for $\forall$, $\exists$, etc.
  - general notion of resources (generalizes dynamic memory allocation)

C benchmarks: Juliet

Experiments

CWE	Lines	Time (h:m:s)	✅	⚠
Stack-based Buffer Overflow	234k	00:59:12	89%	11%
Heap-based Buffer Overflow	174k	00:37:12	86%	14%
Buffer Underwrite	93k	00:18:28	86%	14%
Buffer Over-read	75k	00:14:45	85%	15%
Buffer Under-read	89k	00:18:26	87%	13%
Integer Overflow	440k	01:24:47	52%	48%
Integer Underflow	340k	01:02:27	55%	45%
Divide By Zero	109k	00:13:17	55%	45%
Double Free	17k	00:04:21	100%	0%
Use After Free	14k	00:02:40	100%	0%
Illegal Pointer Subtraction	1k	00:00:24	100%	0%
NULL Pointer Dereference	21k	00:04:53	100%	0%

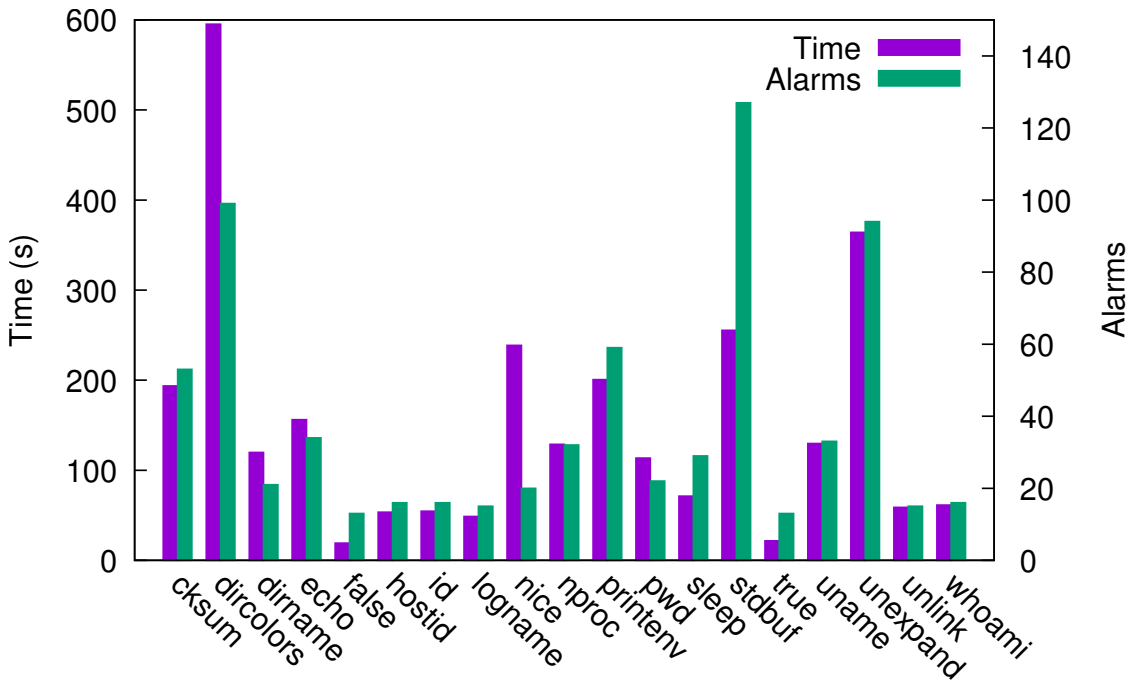
Analyzed 12 CWEs (13,261 tests) from NIST Juliet v1.3
each test comes with a good and a bad version

- ✅ good case safe **and** 1 error in bad case.
- ⚠ good case unsafe **or** many errors in bad case.

for all tests, the bad case reports an error (sound)

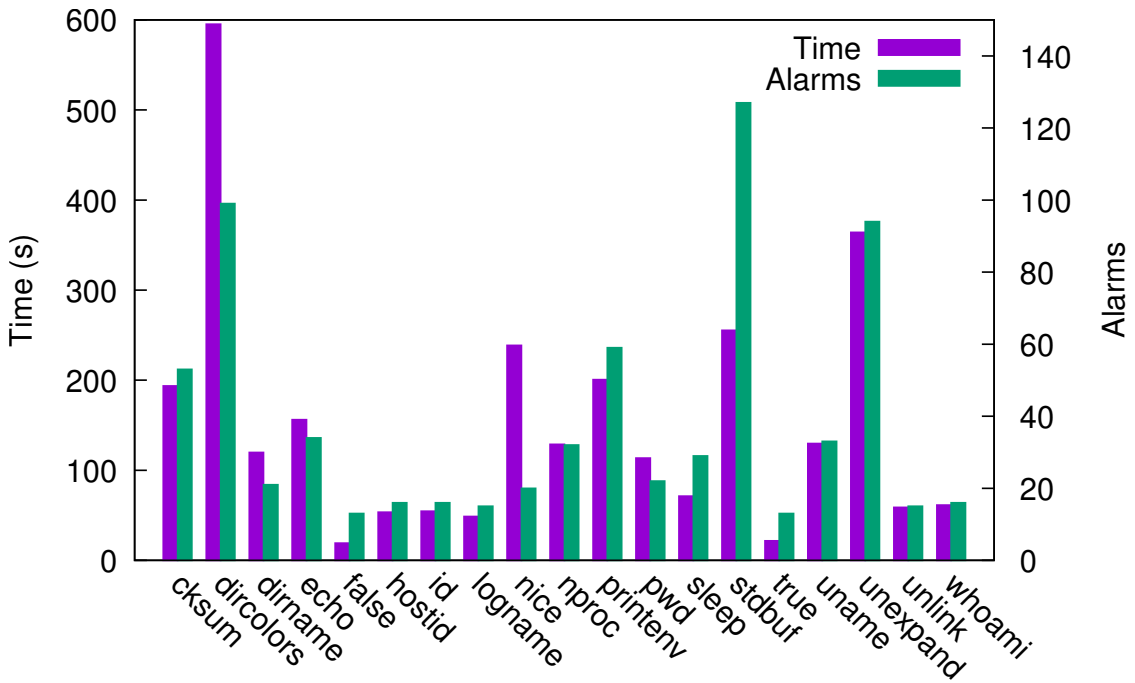
[with A. Ouadjaout @ SAS'20]

C benchmarks: Coreutils



- analyzed 19 programs from GNU Coreutils v8.30
- stub model for 1108 functions of the GNU libc
- `main` is called with a symbolic string array `argv` of arbitrary size

C benchmarks: Coreutils



- analyzed 19 programs from GNU Coreutils v8.30
- stub model for 1108 functions of the GNU libc
- `main` is called with a symbolic string array `argv` of arbitrary size

SV-COMP 2024 competition

1st place for Software Systems category !

Challenges in Python analysis

```
def f(a, b):
  return a + b
```

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

- control-flow very dependent on the dynamic type

  $\Rightarrow$ a precise flow- and context-sensitive value analysis is necessary!

- complex language to formalize!

  using a functional big-step semantics, close to the abstract interpreter implementation

  $\Rightarrow$ expression rewriting and case analysis is useful

Challenges in Python analysis

```
def f(a, b):
  return a + b
```

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

- control-flow very dependent on the dynamic type

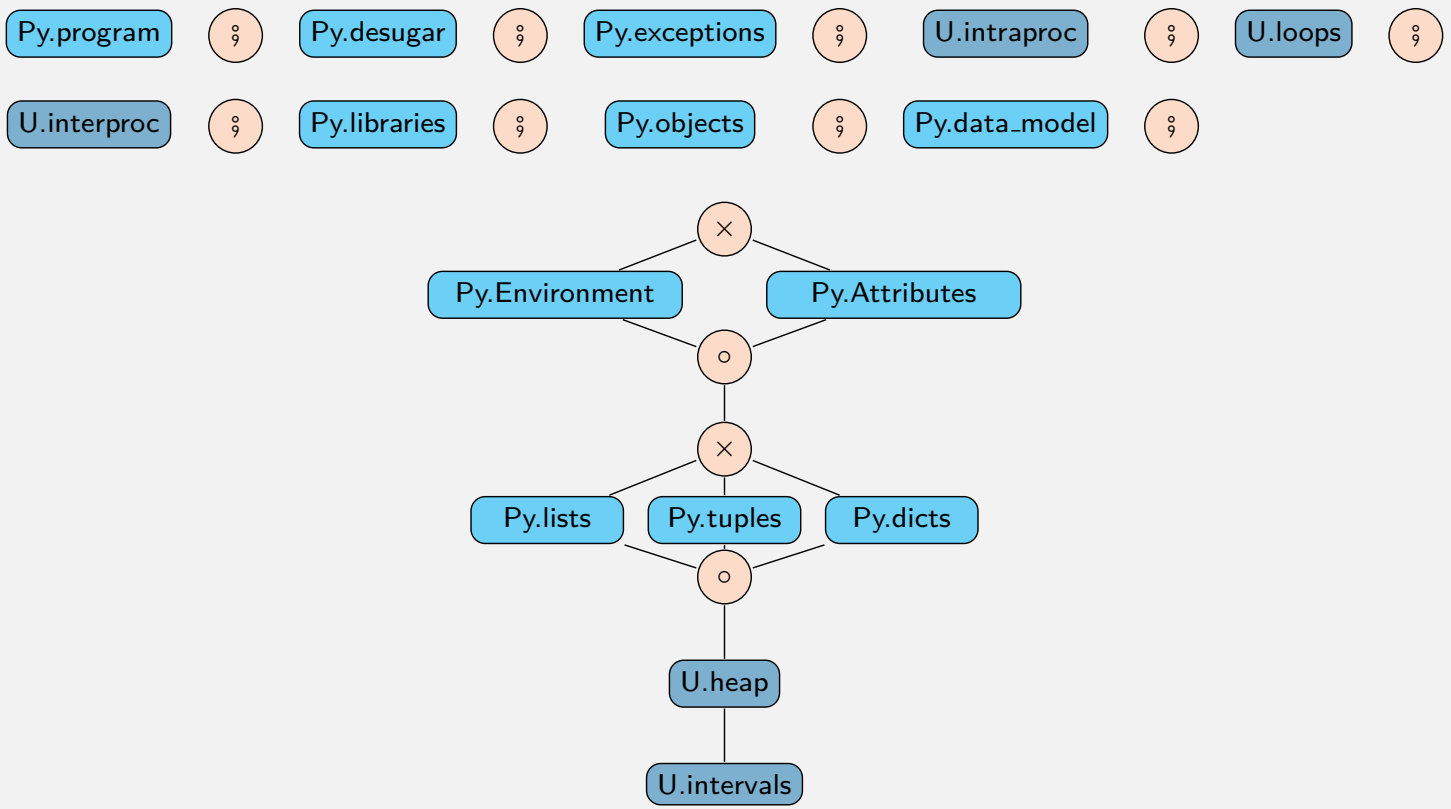
  $\Rightarrow$ a precise flow- and context-sensitive value analysis is necessary!

- complex language to formalize!

  using a functional big-step semantics, close to the abstract interpreter implementation

  $\Rightarrow$ expression rewriting and case analysis is useful

Domains for Python value analysis



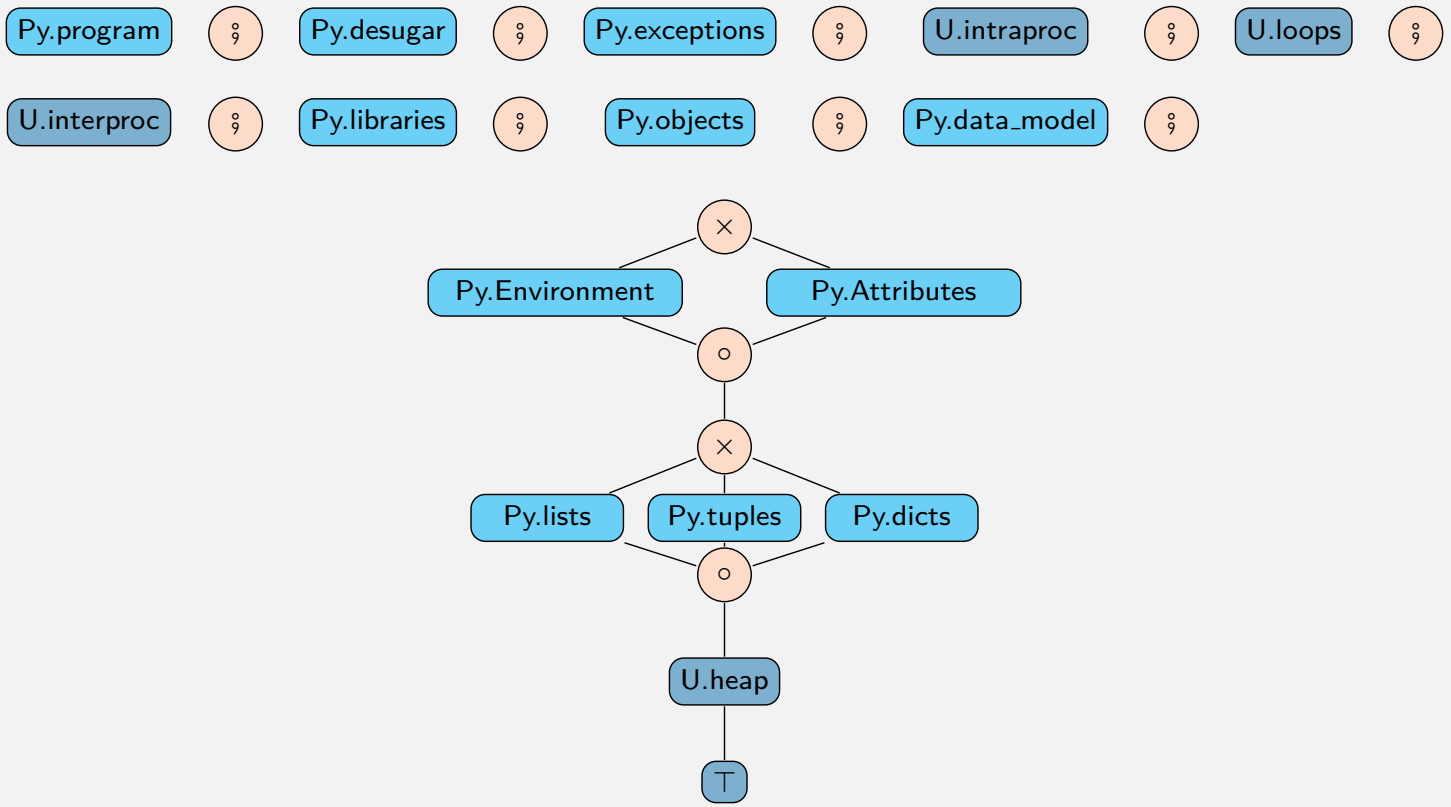
Example Python domain: Lists

- smash each list contents into a single "summary" variable
- keep the list length in a numeric variable

Application : analyze small programs (a few 100s of lines, few dependencies)

[with A. Fromherz & R. Monat @ SOAP'20, ECOOP'20]

Domains for Python type analysis



a type analysis is also easy to construct!

[with R. Monat @ SOAP'20, ECOOP'20]

Programs mixing C and Python

Python counter class in C

```
typedef struct {
  PyObject_HEAD;
  int counter;
} Counter;

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

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

Python client

```
from counter import Counter
from random import randrange

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

what can go wrong?

- power ≤ 30: r $= 2^{\text{power}}$
- power $= 31$: r $= 2^{-31}$: C overflow (silent wrap-around)
- power $\in [32, 62]$: Python `OverflowError` (overflow on int)
- power ≥ 63: Python `OverflowError` (overflow on long)

Programs mixing C and Python

Python counter class in C

```
typedef struct {
  PyObject_HEAD;
  int counter;
} Counter;

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

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

Python client

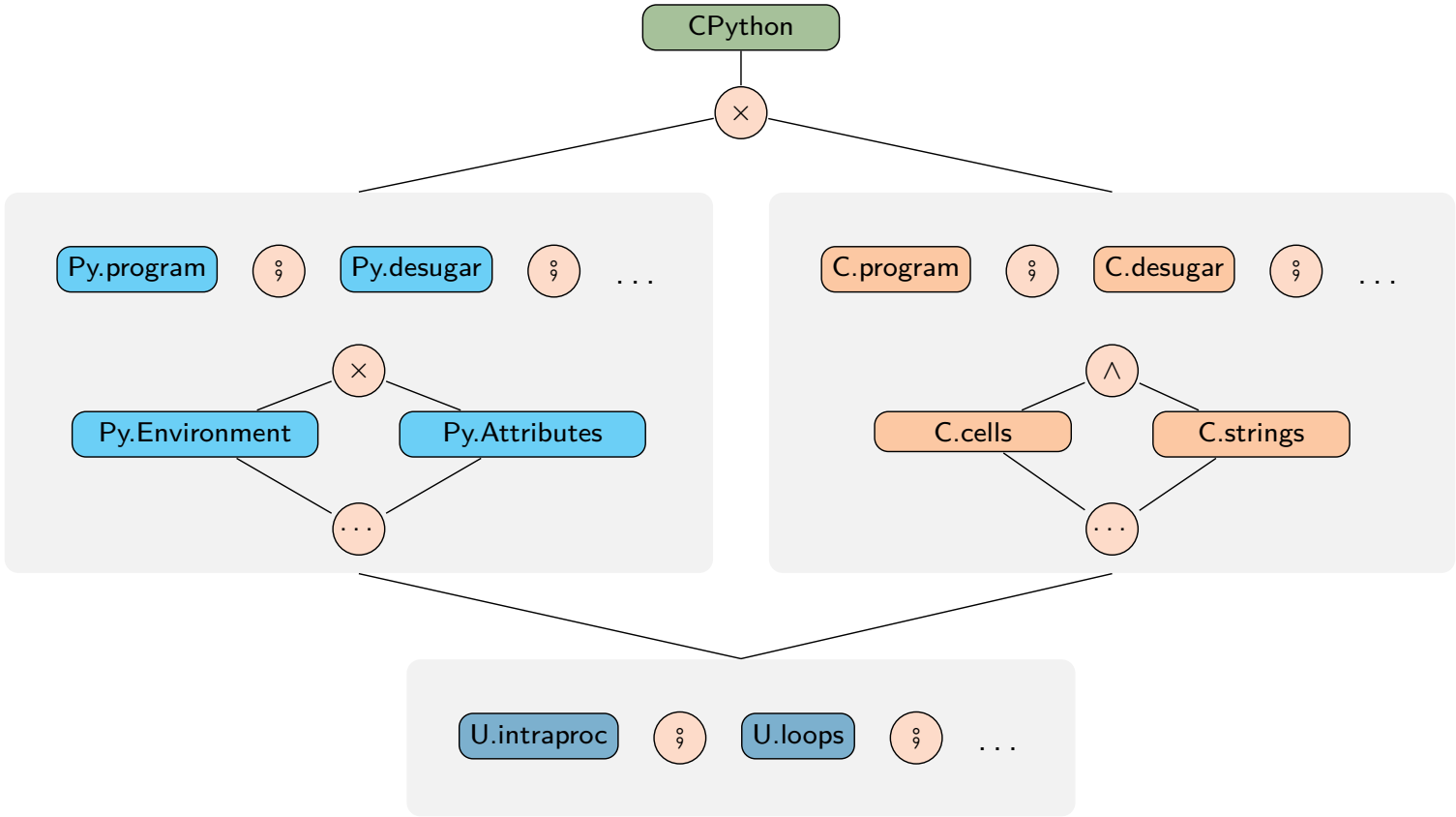
```
from counter import Counter
from random import randrange

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

what can go wrong?

- power ≤ 30: r $= 2^{\text{power}}$
- power $= 31$: r $= 2^{-31}$: C overflow (silent wrap-around)
- power $\in [32, 62]$: Python `OverflowError` (overflow on int)
- power ≥ 63: Python `OverflowError` (overflow on long)

Domains for a C–Python value analysis



Analyze an AST containing both Python and C sources using

- Python domains: for Python code (some CPython API calls are translated to Python instructions)
- C domains: for C code (includes part of CPython implementation in C)
- CPython domain: translate objects between shared C and Python heaps with complementary views (boundary functions)

Application: small C libraries with Python bindings and unit tests ($\simeq$ a few K lines)

[with R. Monat @ SAS'21]

Patch analysis for C

```
172 172    /* Like fstatat, but cache the result.  If ST->st_size is -1, the
173 173       status has not been gotten yet.  If less than -1, fstatat failed
174     -     with errno == -1 - ST->st_size.  Otherwise, the status has already
    174 +     with errno == ST->st_ino.  Otherwise, the status has already
175 175       been gotten, so return 0.  */
176 176    static int
177 177    cache_fstatat (int fd, char const *file, struct stat *st, int flag)
178 178    {
179 179      if (st->st_size == -1 && fstatat (fd, file, st, flag) != 0)
180     -      st->st_size = -1 - errno;
    180 +      {
    181 +        st->st_size = -2;
    182 +        st->st_ino = errno;
    183 +      }
181 184      if (0 <= st->st_size)
182 185        return 0;
183     -    errno = -1 - st->st_size;
    186 +    errno = (int) st->st_ino;
184 187      return -1;
185 188    }
```

Diff from `remove.c` in Coreutils

- analyze a pair of programs
- whole-program iteration on both versions
- compute semantic differences: new program semantic and new domains
- goal: prove the absence of regressions

Application: 100-1500 line patches from Coreutils and Linux

[with D. Delmas @ SAS'19]

Exploitability analysis

Exploitable

```
void use(char * input) {
  char dest[10];
  strcpy(dest, input); // alarm!
}

void main() {
  char buf[100];
  fgets(buf, sizeof(buf), stdin);
  use(buf);
}
```

Non-exploitable

```
void use(char * input) {
  char dest[10];
  strcpy(dest, input); // alarm!
}

void main() {
  char buf[10]; // fixed!
  fgets(buf, sizeof (buf), stdin);
  use(rand() ? buf
             : "123456789012345");
}
```

- discover whether a runtime error depends on a user-controllable value
  $\Longrightarrow$ possibly exploitable bug
- reduce alarm number by reporting only exploitable ones
- combination of forward value analysis and taint analysis

[with F. Parolini @ VMCAI'24]

Backward under-approximations

Finding witnesses for a bug

```
char buf[3];
int i = 0;
int j = input(0, 10); // input value?
int arr[3];
while (i < 5) {
  j += random(0, 1);
  i++;
}
buf[j] = '\0'; // bug
```

- given an alarm, find a witness proving the presence of the bug
- requires inferring sufficient preconditions for a given postcondition
  - infer user-input values
  - despite possible non-determinism (choices out of control of the user)
  - despite abstractions (approximations)
- the analysis proceeds backwards from the bug
- sufficient preconditions must be under-approximated

[with M. Milanese @ VMCAI'24]

Future work

Aspects of MOPSA to improve:

- **modular analyses** (beyond whole-program analyses)
  - analyze a function once, reuse the result many times
  - reuse across different git projects (libraries, files)
  - incremental analyses (add use context gradually)
  - develop more symbolic abstractions of the memory (separation properties)
- consider additional properties
  - further portability & impact analyses
  - **lightweight analyses**
- support **additional languages**, boost **multilanguage support**
  - work on OCaml support, OCaml-C bindings


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


The End