Design of the MOPSA Abstract Interpreter

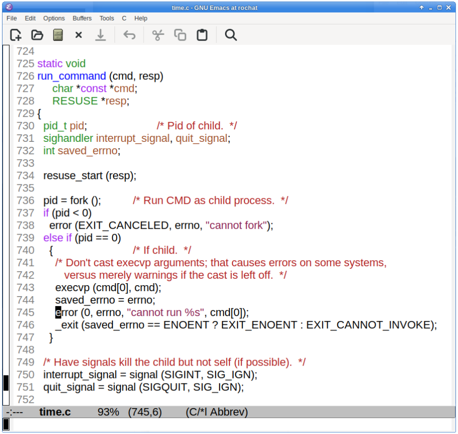
MPRI 2–6: Abstract Interpretation, application to verification and static analysis

Antoine Miné

Year 2024–2025

Course 5b
21 October 2024

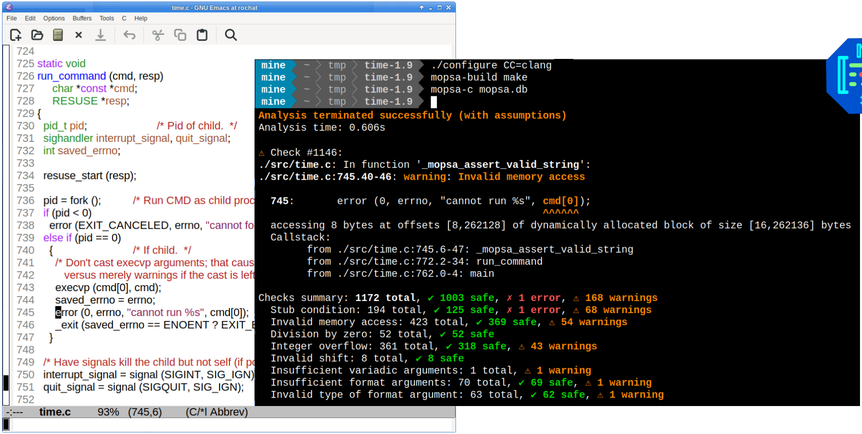
Program verification by static analysis



```
static void
run_command (cmd, resp)
     char *const *cmd;
     RESUSE *resp;
{
  pid_t pid;                    /* Pid of child.  */
  sighandler interrupt_signal, quit_signal;
  int saved_errno;

  resuse_start (resp);

  pid = fork ();          /* Run CMD as child process.  */
  if (pid < 0)
    error (EXIT_CANCELED, errno, "cannot fork");
  else if (pid == 0)
    {                     /* If child.  */
      /* Don't cast execvp arguments; that causes errors on some systems,
         versus merely warnings if the cast is left off.  */
      execvp (cmd[0], cmd);
      saved_errno = errno;
      error (0, errno, "cannot run %s", cmd[0]);
      _exit (saved_errno == ENOENT ? EXIT_ENOENT : EXIT_CANNOT_INVOKE);
    }

  /* Have signals kill the child but not self (if possible).  */
  interrupt_signal = signal (SIGINT, SIG_IGN);
  quit_signal = signal (SIGQUIT, SIG_IGN);
```

- 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



```
724
725 static void
726 run_command (cmd, resp)
727      char *const *cmd;
728      RESUSE *resp;
729 {
730   pid_t pid;                  /* Pid of child.  */
731   sighandler interrupt_signal, quit_signal;
732   int saved_errno;
733
734   resuse_start (resp);
735
736   pid = fork ();          /* Run CMD as child proc
737   if (pid < 0)
738     error (EXIT_CANCELED, errno, "cannot fo
739   else if (pid == 0)
740     {                     /* If child.  */
741       /* Don't cast execvp arguments; that caus
742          versus merely warnings if the cast is left
743       execvp (cmd[0], cmd);
744       saved_errno = errno;
745       error (0, errno, "cannot run %s", cmd[0]);
746       _exit (saved_errno == ENOENT ? EXIT_E
747     }
748
749   /* Have signals kill the child but not self (if po
750   interrupt_signal = signal (SIGINT, SIG_IGN);
751   quit_signal = signal (SIGQUIT, SIG_IGN);
752
-:---  time.c       93%  (745,6)      (C/*l Abbrev)
```

```
mine › ~ › tmp › time-1.9 › ./configure CC=clang
mine › ~ › tmp › time-1.9 › mopsa-build make
mine › ~ › tmp › time-1.9 › mopsa-c mopsa.db
mine › ~ › tmp › time-1.9 ›
Analysis terminated successfully (with assumptions)
Analysis time: 0.606s

⚠ Check #1146:
./src/time.c: In function '_mopsa_assert_valid_string':
./src/time.c:745.40-46: warning: Invalid memory access

  745:       error (0, errno, "cannot run %s", cmd[0]);
                                               ^^^^^^

  accessing 8 bytes at offsets [8,262128] of dynamically allocated block of size [16,262136] bytes
  Callstack:
        from ./src/time.c:745.6-47: _mopsa_assert_valid_string
        from ./src/time.c:772.2-34: run_command
        from ./src/time.c:762.0-4: main

Checks summary: 1172 total, ✔ 1003 safe, ✗ 1 error, ⚠ 168 warnings
  Stub condition: 194 total, ✔ 125 safe, ✗ 1 error, ⚠ 68 warnings
  Invalid memory access: 423 total, ✔ 369 safe, ⚠ 54 warnings
  Division by zero: 52 total, ✔ 52 safe
  Integer overflow: 361 total, ✔ 318 safe, ⚠ 43 warnings
  Invalid shift: 8 total, ✔ 8 safe
  Insufficient variadic arguments: 1 total, ⚠ 1 warning
  Insufficient format arguments: 70 total, ✔ 69 safe, ⚠ 1 warning
  Invalid type of format argument: 63 total, ✔ 62 safe, ⚠ 1 warning
```

- 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.

Domains for a more realistic language

Example: traversing a string array

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

Domains for a more realistic language

Example: traversing a string array

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

Memory:

`argc`: variable
`argv`: variable
`p`: variable
`i`: variable

Numeric:

$\texttt{argc} \in [1, \texttt{maxint}]$

Domains for a more realistic language

Example: traversing a string array

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

Memory:

`argc`: variable
`argv`: variable
`p`: variable
`i`: variable

Numeric:

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

Domains for a more realistic language

Example: traversing a string array

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

Memory:

argc: variable
argv: variable
p: variable
i: variable

Numeric:

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

Pointers:

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

Domains for a more realistic language

Example: traversing a string array

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

Memory:

argc: variable
argv: variable
p: variable
i: variable
@: summary block for all strings

Pointers:

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

Numeric:

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

Domains for a more realistic language

Example: traversing a string array

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

Memory:

argc: variable
argv: variable
p: variable
i: variable
@: summary block for all strings

Pointers:

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

Numeric:

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

C strings:

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

Domains for a more realistic language

Example: traversing a string array

```
int main( int argc, char *argv[]) {
```

Combining domains

- for different types (numbers, pointers, blocks, . . .)
- for different properties (intervals, relations, predicates, . . .)
- able to delegate & communicate information

Memory:

$\mathtt{argc}$: variable
$\mathtt{argv}$: variable
$\mathtt{p}$: variable
$\mathtt{i}$: variable
$\mathtt{@}$: summary block for all strings

Pointers:

$\mathtt{p} \mapsto \{\mathtt{argv}\}$
$\mathtt{argv}[0 \dots \mathtt{argc} - 1] \mapsto \{\mathtt{@}\}$
$\mathtt{argv}[\mathtt{argc}] \mapsto \{\mathtt{NULL}\}$

Numeric:

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

C strings:

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

MOPSA

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`

Some characteristics

Some classic aspects:

- whole-program forward abstract interpretation
- by induction on the AST (not using an equation solver on the CFG)
- 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, and with various levels of maturity. . .

Extensible syntax tree for multiple languages

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

- ✓ 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 factoring common, reusable language subsets
  e.g., scalar fragment, pointer-free, integer arithmetic, machine arithmetic
- ✓ translation is dynamic, more power (exploit abstract information)
- ✗ less efficient
- ✗ hard to keep track of sub-languages, possible match failures

Extensible syntax tree for multiple languages

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

- ✔ 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 factoring common, reusable language subsets
  e.g., scalar fragment, pointer-free, integer arithmetic, machine arithmetic
- ✔ translation is dynamic, more power (exploit abstract information)
- ✗ less efficient
- ✗ hard to keep track of sub-languages, possible match failures

Distributed iterators and delegation

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

simple loops : `Universal.iterators.loops`

- matches `S_while`
- computes a fixpoint with ∇

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 domains without any internal abstract state. . .

Distributed iterators and delegation

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

simple loops : `Universal.iterators.loops`

- matches `S_while`
- computes a fixpoint with ∇

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 domains without any internal abstract state...

Distributed iterators and delegation

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

simple loops : `Universal.iterators.loops`

- matches `S_while`
- computes a fixpoint with ∇

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;
```

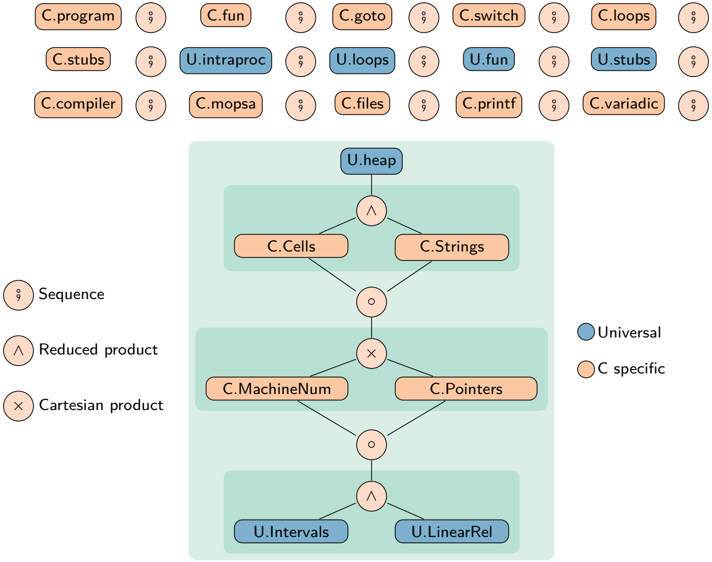
Python loops: `Python.desugar.loops`

- 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 domains without any internal abstract state. . .

Domains for a C analysis



C.program ⨾ C.fun ⨾ C.goto ⨾ C.switch ⨾ C.loops ⨾

C.stubs ⨾ U.intraproc ⨾ U.loops ⨾ U.fun ⨾ U.stubs ⨾

C.compiler ⨾ C.mopsa ⨾ C.files ⨾ C.printf ⨾ C.variadic ⨾

U.heap

$\wedge$

C.Cells

C.Strings

$\circ$

$\times$

C.MachineNum

C.Pointers

$\circ$

$\wedge$

U.Intervals

U.LinearRel

⨾ Sequence

$\wedge$ Reduced product

$\times$ Cartesian product

🔵 Universal

🟠 C specific

Simplified domain signature

Domain (simplified)

```
module type SIMPLIFIED = sig
  type t

  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 init : program -> t
  val exec : stmt -> ('a,t) man -> 'a ctx -> t
           -> t option
  ...
end
```

Framework (simplified)

```
type ('a, 't) man = {
  lattice : 'a lattice;
  get : 'a -> 't;
  set : 't -> 'a -> 'a;
  exec : stmt -> 'a -> 'a;
  ...
}
```

- `t`: abstract type of a values for a single domain
- `'a`: combination of all domains (map: domain id → abstract value in domain)
- `man`: to operate on the whole `'a` abstract state (e.g., recursive call to `exec`)
- `ctx`: global, polymorphic, extensible key/value store

Domain cooperation for memory analysis in C

`C.Pointers`: pointer value domain

- abstract data: a “base” for each pointer variable
- delegate: pointer offsets as numeric variables
- translate: pointer arithmetic into integer arithmetic

`C.cells`: memory block domain

- abstract data: split blocks (struct, array, malloc, . . .) into scalar cells
- delegate: scalar cells as variables
- translate: memory accesses into cell assignment and reads

Other abstractions are possible. . . (field insensitive, etc.)

`U.heap`: heap abstraction

- abstract data: allocation-site based map of dynamic blocks
- delegate: contents of the block
- translate: pointer values into finitely mainly abstract values

Recency abstraction: distinguish the block allocated last from the previous one
Other choices are possible (context-sensitivity, type-based abstraction, etc.)

Domain cooperation for memory analysis in C

`C.Pointers`: pointer value domain

- abstract data: a “base” for each pointer variable
- delegate: pointer offsets as numeric variables
- translate: pointer arithmetic into integer arithmetic

`C.cells`: memory block domain

- abstract data: split blocks (struct, array, malloc, . . .) into scalar cells
- delegate: scalar cells as variables
- translate: memory accesses into cell assignment and reads

Other abstractions are possible. . . (field insensitive, etc.)

`U.heap`: heap abstraction

- abstract data: allocation-site based map of dynamic blocks
- delegate: contents of the block
- translate: pointer values into finitely mainly abstract values

Recency abstraction: distinguish the block allocated last from the previous one
Other choices are possible (context-sensitivity, type-based abstraction, etc.)

Domain cooperation for memory analysis in C

`C.Pointers`: pointer value domain

- abstract data: a "base" for each pointer variable
- delegate: pointer offsets as numeric variables
- translate: pointer arithmetic into integer arithmetic

`C.cells`: memory block domain

- abstract data: split blocks (struct, array, malloc, . . .) into scalar cells
- delegate: scalar cells as variables
- translate: memory accesses into cell assignment and reads

Other abstractions are possible. . . (field insensitive, etc.)

`U.heap`: heap abstraction

- abstract data: allocation-site based map of dynamic blocks
- delegate: contents of the block
- translate: pointer values into finitely mainly abstract values

Recency abstraction: distinguish the block allocated last from the previous one
Other choices are possible (context-sensitivity, type-based abstraction, etc.)

Expression rewriting

Flexible, modular way for domains to communicate:

- rewrite (part of) expressions
- delegate to other domains (unaware of other abstractions, decoupling)
- dynamic rewriting: can exploit the current abstract information

Example: bounded to unbounded arithmetic

- many languages have bounded integers with wrap-around
- many domains only reason on mathematical integers (polyhedra)

Solution: a machine number domain

- evaluate sub-expressions in the abstract to check for overflows
- rewrite machine expressions into mathematical expressions
  - identity, if there is no overflow (best case)
  - adding an explicit wrap operator, otherwise (and reporting the overflow)
- propagate the mathematical expression to the other domains

Expression rewriting

Flexible, modular way for domains to communicate:

- rewrite (part of) expressions
- delegate to other domains (unaware of other abstractions, decoupling)
- dynamic rewriting: can exploit the current abstract information

Example: bounded to unbounded arithmetic

- many languages have bounded integers with wrap-around
- many domains only reason on mathematical integers (polyhedra)

Solution: a machine number domain

- evaluate sub-expressions in the abstract to check for overflows
- rewrite machine expressions into mathematical expressions
  - identity, if there is no overflow (best case)
  - adding an explicit wrap operator, otherwise
    (and reporting the overflow)
- propagate the mathematical expression to the other domains

Expression rewriting

Flexible, modular way for domains to communicate:

- rewrite (part of) expressions
- delegate to other domains (unaware of other abstractions, decoupling)
- dynamic rewriting: can exploit the current abstract information

Example: bounded to unbounded arithmetic

- many languages have bounded integers with wrap-around
- many domains only reason on mathematical integers (polyhedra)

Solution: a machine number domain

- evaluate sub-expressions in the abstract to check for overflows
- rewrite machine expressions into mathematical expressions
  - identity, if there is no overflow (best case)
  - adding an explicit wrap operator, otherwise
    (and reporting the overflow)
- propagate the mathematical expression to the other domains

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 $P_a : a[v_a] = 0 \land (\forall k \in [0, v_a[: a[k] \neq 0)$

- 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:
  - $(0, \mathsf{C}[\![\, i < v_a \,]\!] X^\sharp)$
  - $(1, \mathsf{C}[\![\, i = v_a \,]\!] X^\sharp)$
  - $([0,1], \mathsf{C}[\![\, i > v_a \,]\!] X^\sharp)$

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

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:
  - $(0, \mathsf{C}[\![\, i < v_a \,]\!]\, X^\sharp)$
  - $(1, \mathsf{C}[\![\, i = v_a \,]\!]\, X^\sharp)$
  - $([0,1], \mathsf{C}[\![\, i > v_a \,]\!]\, X^\sharp)$

keep some relationality between i, $a[i]$, and v_a

- $\mathsf{C}[\![\, i < v_a \,]\!]$, etc. are handled by (relational) numeric abstract domains

Simplified domain signature with cases

Domain (simplified)

```
module type DOMAIN = sig
  type t
  ...

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

Framework (simplified)

```
type 'r case =
  | Result of 'r
  | Empty | NotHandled

type 'a dnf = 'a list list

type ('a,'r) cases =
  { cases: ('r case * 'a) dnf.t;
    ctx: 'a ctx}

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

- `dnf`: disjunctive normal form (DNF)
- `cases`: DNF of arbitrary data associated to abstract sub-elements
- `eval`: DNF of expressions associated to abstract sub-elements
- `post`: post-condition

  a DNF is merged into a single abstract element at the end of each program instruction

Non-local control-flow

Handling of statements by induction on the syntax:

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

Goto in C:

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

C 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.

Non-local control-flow

Handling of statements by induction on the syntax:

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

Goto in C:

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

C 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.

Simplified domain signature with flows

Domain (simplified)

```
module type DOMAIN = sig
  type t
  ...
  val init : program -> ('a, t) man -> 'a flow
          -> 'a flow
  val exec : stmt -> ('a, t) man -> 'a flow
          -> 'a post option
  ...
end
```

Framework (simplified)

```
type token = ..
type token += T_cur

type 'a flow = {
  tmap : 'a TokenMap.t;
  ctx : 'a ctx;
}

type ('a,'r) cases =
  { cases: ('r case * 'a flow) dnf.t;
    ctx: 'a ctx}

type 'a post = ('a, unit) cases
```

- `token`: location where the execution continues
- `T_cur`: current flow, goto next instruction
- `flow`: map from tokens to data
- `exec` now manipulates several flows

Handling C gotos with flows

Tokens for C goto:

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

C example:

```
        x = 12; [T_cur → 12]
        if (...)  { x++; [T_cur → 13] goto l1; [T_cur → ⊥, 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;
```

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

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

Queries and reductions

Two scopes for property data-types:

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

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

module type DOMAIN = sig
  val ask : ('a,'r) query -> ('a, t) man -> 'a flow -> ('a, 'r) cases option
  ...
end
```

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

Application: reductions

- after each statement, query interval information and intersect
- focus on the variables modified by the statement (efficiency)
- independent from the domains, defined externally

Queries and reductions

Two scopes for property data-types:

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

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

module type DOMAIN = sig
  val ask : ('a,'r) query -> ('a, t) man -> 'a flow -> ('a, 'r) cases option
  ...
end
```

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

Application: reductions

- after each statement, query interval information and intersect
- focus on the variables modified by the statement (efficiency)
- independent from the domains, defined externally

Queries and reductions

Two scopes for property data-types:

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

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

module type DOMAIN = sig
  val ask : ('a,'r) query -> ('a, t) man -> 'a flow -> ('a, 'r) cases option
  ...
end
```

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

Application: reductions

- after each statement, query interval information and intersect
- focus on the variables modified by the statement (efficiency)
- independent from the domains, defined externally

Simple value reductions

Reduction: collaboration between domains

Example: intervals and congruences

$$\mathsf{C}[\![\, s \,]\!]_{i\times c}(I^\sharp, C^\sharp) \stackrel{\text{def}}{=} \rho(\mathsf{C}[\![\, s \,]\!]_i I^\sharp, \mathsf{C}[\![\, s \,]\!]_c C^\sharp)$$

```
$\rho([a,b], c\mathbb{Z}+d) \stackrel{\text{def}}{=}$
    let $a' = \min\{\, x \in c\mathbb{Z}+d \mid x \geq a \,\}$ in
    let $b' = \max\{\, x \in c\mathbb{Z}+d \mid x \leq b \,\}$ in
    if $a' > b'$ then $(\bot, \bot)$
    if $a' = b'$ then $([a', a'], 0\mathbb{Z}+a')$
    else $([a', b'], c\mathbb{Z}+d)$
```

In MOPSA, reductions are defined externally (outside domains)

```
let reduce man v =
    let c = man.get Cong.id v
    and i = man.get Itv.id v in
    let c', i' = meet_cgr_itv c i in
    man.set Itv.id i' v |> man.set Cong.id c'
```

- access domain information through setters / getters
- easy to plug in and out, independently from other domains
- *n*−array reductions are also possible

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
- stubs are executed at each call (not a specification to verify)
- for MOPSA: just another language (delegation, rewriting, etc.)
  - add transfer functions for ∀, ∃, etc.
  - beyond memory: general notion of resources (with recency abstraction)
  - delegation, expression rewriting, . . .

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%

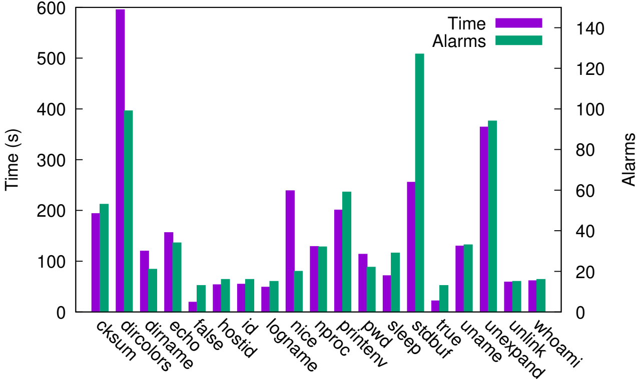
analyzed 12 CWEs (13,261 tests) from NIST Juliet v1.3
each test comes with a good and a bad version

✓ 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)

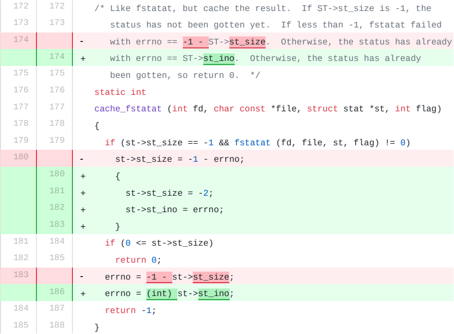
C benchmarks: Coreutils



- 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
- also, analysis of Juliet 1.3 benchmark from NIST, for 12 CWEs (13,261 tests)

[with A. Ouadjaout @ SAS'20]

Patch analysis for C



```diff
  /* Like fstatat, but cache the result.  If ST->st_size is -1, the
     status has not been gotten yet.  If less than -1, fstatat failed
-    with errno == -1 - ST->st_size.  Otherwise, the status has already
+    with errno == ST->st_ino.  Otherwise, the status has already
     been gotten, so return 0.  */
  static int
  cache_fstatat (int fd, char const *file, struct stat *st, int flag)
  {
    if (st->st_size == -1 && fstatat (fd, file, st, flag) != 0)
-     st->st_size = -1 - errno;
+     {
+       st->st_size = -2;
+       st->st_ino = errno;
+     }
    if (0 <= st->st_size)
      return 0;
-   errno = -1 - st->st_size;
+   errno = (int) st->st_ino;
    return -1;
  }
```

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]

Endianess portability for C

endianess

```
    unsigned short x;
    unsigned char b1, b2;
    // set next byte to 0xff
#if LITTLE_ENDIAN
    x = b1 | 0xff00;
#else
    x = (b1 << 8) | 0xff;
#endif
    // extract least significant byte
    b2 = *((unsigned char*) &x);
```

Prove that the program gives the same result on 32-bit machines with different byte orderings (endianess)

- analyze both versions of the program at the same time
- iterators for bi-programs and duplicated memory state
- add domains to efficiently represent equal/reversed memory portions
- scalable results on industrial code with Airbus

[with D. Delmas @ SAS'21]

Endianess portability for C

endianess

```
    unsigned short x;
    unsigned char b1, b2;
    // set next byte to 0xff
#if LITTLE_ENDIAN
    x = b1 | 0xff00;
#else
    x = (b1 << 8) | 0xff;
#endif
    // extract least significant byte
    b2 = *((unsigned char*) &x);
```

Prove that the program gives the same result on 32-bit machines with different byte orderings (endianess)

- analyze both versions of the program at the same time
- iterators for bi-programs and duplicated memory state
- add domains to efficiently represent equal/reversed memory portions
- scalable results on industrial code with Airbus

[with D. Delmas @ SAS'21]

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
  ⟹ possibly exploitable bug
- reduce alarm number by reporting only exploitable ones
- combination of forward value analysis and taint analysis

[with F. Parolini @ VMCAI'24]

Combining value and taint analysis

Example

```
$x \leftarrow$ input()
$y \leftarrow 1$
if $x = 0$ then
    $z \leftarrow$ rand()
    if $z = 0$ then $1/x$
    if $z = 1$ then $y \leftarrow z$
```

- the division by 0 depends on user input (implicit dependency)
  ⇒ it is exploitable
- y is assigned in a branch controlled by the input
  but its value is always 1 (spurious syntactic dependency)
  ⇒ y is not tainted
- a reduced product runs both taint and value analyses at the same time
  ⟹ more precise than running them in sequence

Experiments with non-exploitability analysis

Coreutils: 77 programs, ≃4188 Loc / program

	Alarms		Time	
	MOPSA	MOPSA-NEXP	MOPSA	MOPSA-NEXP
Intervals	4715	1217 (-74%)	1:17:06	1:28:42 (+14%)
Polyhedra	4651	1193 (-74%)	2:12:21	2:30:44 (+13%)

Juliet: 13261 programs, ≃ 2.8 MLoc total

	Alarms		Time	
	MOPSA	MOPSA-NEXP	MOPSA	MOPSA-NEXP
Intervals	49957	13906 (-72%)	11:32:24	11:48:51 (+2%)
Polyhedra	48256	13631 (-71%)	12:54:21	13:21:26 (+3.5%)

- effective in filtering alarms
- small cost overhead

Challenges in Python analysis

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

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

- control-flow very dependent on the dynamic type

  ⇒ 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

  ⇒ expression rewriting and case analysis is useful

Challenges in Python analysis

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

```
$\mathsf{E}\llbracket e_1 + e_2 \rrbracket (f, \epsilon, \Sigma) \overset{\text{def}}{=}$
  let $(f_1, \epsilon_1, \Sigma_1, v_1) = \mathsf{E}\llbracket e_1 \rrbracket (f, \epsilon, \Sigma)$ in
  let $(f_2, \epsilon_2, \Sigma_2, v_2) = \mathsf{E}\llbracket e_2 \rrbracket (f_1, \epsilon_1, \Sigma_1)$ in
  if $\mathit{hasattr}(v_1, \texttt{\_\_add\_\_}, \Sigma_2)$ then
    let $(f_3, \epsilon_3, \Sigma_3, v_3) = \mathsf{E}\llbracket v_1.\texttt{\_\_add\_\_}(v_2) \rrbracket (f_2, \epsilon_2, \Sigma_2)$ in
    if $v_3 = \texttt{NotImpl} \wedge \mathit{typeof}(v_1) \neq \mathit{typeof}(v_2)$ then
      if $\mathit{hasattr}(v_2, \texttt{\_\_radd\_\_}, \Sigma_3)$ then
        let $(f_4, \epsilon_4, \Sigma_4, v_4) = \mathsf{E}\llbracket v_2.\texttt{\_\_radd\_\_}(v_1) \rrbracket (f_3, \epsilon_3, \Sigma_3)$ in
        if $v_4 = \texttt{NotImpl}$ then $\texttt{TypeError}(f_4, \epsilon_4, \Sigma_4)$ else $(f_4, \epsilon_4, \Sigma_4, v_4)$
      else $\texttt{TypeError}(f_3, \epsilon_3, \Sigma_3)$
    else if $v_3 = \texttt{NotImpl}$ then $\texttt{TypeError}(f_3, \epsilon_3, \Sigma_3)$ else $(f_3, \epsilon_3, \Sigma_3, v_3)$
  else if $\mathit{hasattr}(v_2, \texttt{\_\_radd\_\_}, \Sigma_2) \wedge \mathit{typeof}(v_1) \neq \mathit{typeof}(v_2)$ then
    let $(f_3, \epsilon_3, \Sigma_3, v_3) = \mathsf{E}\llbracket v_2.\texttt{\_\_radd\_\_}(v_1) \rrbracket (f_2, \epsilon_2, \Sigma_2)$ in
    if $v_3 = \texttt{NotImpl}$ then $\texttt{TypeError}(f_3, \epsilon_3, \Sigma_3)$ else $(f_3, \epsilon_3, \Sigma_3, v_3)$
  else $\texttt{TypeError}(f_2, \epsilon_2, \Sigma_2)$
```

- control-flow very dependent on the dynamic type

  ⇒ 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

  ⇒ expression rewriting and case analysis is useful

Type abstractions for Python

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`

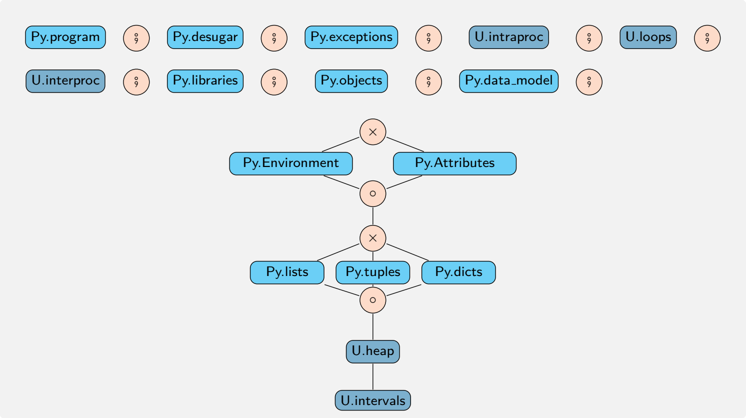
  $\Longrightarrow$ value of the attribute `__class__`
- duck typing: `hasattr`

  $\Longrightarrow$ presence of a specific attribute

Type domains:

- base types (`int`, `List[int]`, etc.)
- for custom objects: list of attribute names and their type
- bounded polymorphism: $\texttt{List}[\alpha]$, $\alpha \in \{\dots\}$

Domains for Python value analysis



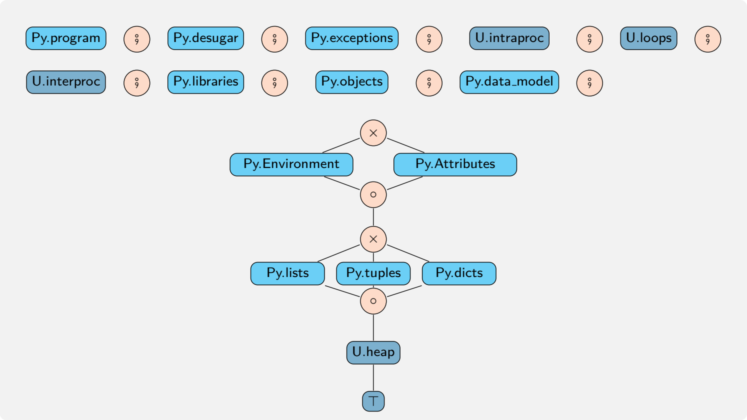
Example Python domain: Lists

- 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]

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)	∞	

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?

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

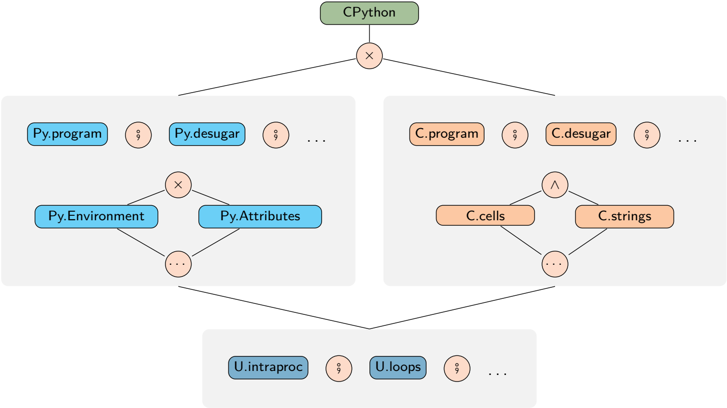
Python client

```
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` $= 2^{\texttt{power}}$
- `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


CPython
×
Py.program ⨟ Py.desugar ⨟ . . .
×
Py.Environment
Py.Attributes
. . .
C.program ⨟ C.desugar ⨟ . . .
∧
C.cells
C.strings
. . .
U.intraproc ⨟ U.loops ⨟ . . .

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]

Benchmark for C-Python analysis

Library	C locs	Py. locs	Tests	time	C checks	Py. checks	Asserts
noise	722	675	15/15	18s	99.6%	100%	0/21
ahocorasick	3541	1336	46/92	54s	93.1%	98.0%	30/88
levenshtein	5441	357	17/17	1.5m	79.9%	93.2%	0/38
cdistance	1433	912	28/28	1.9m	95.3%	98.3%	88/207
llist	2829	1686	167/194	4.2m	99.0%	98.8%	235/691
bitarray	3244	2597	159/216	4.6m	96.3%	94.6%	100/378

Analyze unit tests for Python libraries with C code:

- check C run-time-errors
- check Python exceptions
- check assertion violations

Future work

Some on-going work:

- **modular analysis** (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)
- **backward under-approximations**
- **OCaml** analysis

Future works: (internships & PhD opportunities!)

- **additional languages**
  - e.g. unsafe constructions in Rust
- **multi-language support**
  - binding analysis, such as OCaml-C bindings
- **more expressive properties**
  - functional properties
  - query languages (semantic CodeQL)

Bibliography

Bibliography

- Main page: https://mopsa.lip6.fr/

- **Sound abstract nonexploitability analysis.** F. Parolini, A. Miné. In VMCAI 2024.
- **Mopsa-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution).** R. Monat, M. Milanese, F. Parolini, J. Boillot, A. Ouadjaout, A. Miné (2024). In TACAS 2024.
- **Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs.** M. Valnet, R. Monat, A. Miné. In JFLA 2023.
- **Abstract interpretation of Michelson smart-contracts.** G. Bau, A. Miné, V. Botbol, M. Bouaziz. In SOAP 2022.
- **Multilanguage static analysis of Python programs with native C extensions.** R. Monat, A. Ouadjaout, A. Miné. In SAS 2021.
- **Static analysis of endian portability by abstract interpretation.** D. Delmas, A. Ouadjaout, A. Miné. in SAS 2021.

Bibliography

- **Static type analysis by abstract interpretation of Python programs.**
  R. Monat, A. Ouadjaout, A. Miné. In ECOOP 2020.
- **A library modeling language for the static analysis of C programs.**
  A. Ouadjaout, A. Miné. In SAS 2020.
- **Value and allocation sensitivity in static Python analyses.**
  R. Monat, A. Ouadjaout, A. Miné. In SOAP 2020.
- **Analysis of software patches using numerical abstract interpretation.**
  David Delmas, A. Miné. In SAS 2019.
- **Combinations of reusable abstract domains for a multilingual static analyzer.**
  M. Journault, A. Miné, R. Monat, A. Ouadjaout. In VSTTE 2019.