Combinations of Reusable Abstract Domains for a Multilingual Static Analysis

Antoine Miné     Abdelraouf Ouadjaout     Matthieu Journault
Raphaël Monat    Aymeric Fromherz

APR team
LIP6
Sorbonne Université
Paris, France

13/07/2019
Goal: program verification by static analysis

- work directly on the **source code**
Sound, semantic, static analysis

**Goal:** program verification by static analysis

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

- work directly on the source code
- infer properties on program executions
- automatically (cost effective)
- by constructing dynamically a semantic abstraction of the program
**Goal:** program verification by static analysis

- work directly on the **source code**
- infer properties on **program executions**
- **automatically** (cost effective)
- by constructing dynamically a **semantic abstraction** of the program
- deduce program **correctness** or raise **alarms**
  - implicit specification: absence of RTE; or user-defined properties: contracts
- using **approximate abstractions** (efficient, but possible false alarms)
- soundly (no false positive)
**Goal:** build a static analysis platform (in OCaml) for research and education in abstract interpretation

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

**In this talk:**
- work in progress...
- more engineering than science...
Overview:

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

Abstract interpretation: theory of the approximation of program semantics

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

Concrete executions $\mathcal{D}$: $\{(0, 3), (5.5, 0), (12, 7), \ldots\}$ (not practical)
Abstract interpretation: theory of the approximation of program semantics

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

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

Box domain \( \mathcal{D}_b^\# \): \( X \in [0, 12] \land Y \in [0, 8] \) (linear cost)
Abstract interpretation:

- **Theory:** theory of the approximation of program semantics

- **Principle:**
  - Be tractable by reasoning at an abstract level
  - Keep soundness by considering over-approximations

Concrete executions $\mathcal{D}$:

- Not practical

Box domain $\mathcal{D}_b^\#$:

- Linear cost
  - $X \in [0, 12] \land Y \in [0, 8]$

Polyhedron domain $\mathcal{D}_p^\#$:

- Exponential cost
  - $6X + 11Y \geq 33 \land \cdots$
Abstract interpretation

**Abstract interpretation:** theory of the **approximation** of program **semantics**

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

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

Each abstract element represents a concrete element, via $\gamma : \mathcal{D}^\# \rightarrow \mathcal{D}$
Abstract computations

Define an interpretation of atomic statements in the abstract domain. For each $S[s] : D \rightarrow D$, provide $S^\#[s] : D^\# \rightarrow D^\#$.

Compose interpretations to analyze full programs. Replace $S[s_1] \circ \ldots \circ S[s_n]$ with $S^\#[s_1] \circ \ldots \circ S^\#[s_n]$. 
Abstract computations

Define an **interpretation of atomic statements** in the abstract domain.

For each $S[s]: D \rightarrow D$, provide $S^# [s]: D^# \rightarrow D^#$.

Compose interpretations to analyze full programs.

Replace $S[s_1] \circ \ldots \circ S[s_n]$ with $S^# [s_1] \circ \ldots \circ S^# [s_n]$.

### Polyhedra operators

**Assignments**
- $X \leftarrow X + 1$
- translation
Define an interpretation of atomic statements in the abstract domain. For each $S[s]: D \rightarrow D$, provide $S^\#[s]: D^\# \rightarrow D^\#$.

Compose interpretations to analyze full programs. Replace $S[s_1] \circ \ldots \circ S[s_n]$ with $S^\#[s_1] \circ \ldots \circ S^\#[s_n]$.

**Polyhedra operators**

**Assignments**
- $X \leftarrow X + 1$
- translation

**Branches: join**
- if ··· then ··· else ··· fi
- convex hull
Abstract computations

Define an **interpretation of atomic statements** in the abstract domain.

For each \( S[s] : D \rightarrow D \), provide \( S^\#[s] : D^\# \rightarrow D^\# \).

Compose interpretations to analyze full programs.

Replace \( S[s_1] \circ \cdots \circ S[s_n] \) with \( S^\#[s_1] \circ \cdots \circ S^\#[s_n] \).

**Polyhedra operators**

**Assignments**
- \( X \leftarrow X + 1 \)
- translation

**Branches: join**
- if \( \cdots \) then \( \cdots \) else \( \cdots \) fi
- convex hull

**Loops: inductive invariants**
- while \( \cdots \) do \( \cdots \) done
- iteration with widening \( \nabla \)
A more complex example

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

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

**Memory:**
- \(\text{argc}: \text{variable}\)
- \(\text{argv}: \text{variable}\)
- \(p: \text{variable}\)
- \(i: \text{variable}\)
- @}: summary block

**Pointers:**
- \(\text{argv}[0...\text{argc} - 1] \mapsto \{@\}\)
- \(\text{argv}[\text{argc}] \mapsto \{\text{NULL}\}\)
- \(p \mapsto \{\text{argv}\}\)

**Strings:**
- \(\exists k \in [0...\text{size @} - 1] : @k = 0\)
A more complex example

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

**Numeric:**

- `argc` ∈ [1, maxint]
- `size(argv) = argc + 1`

**Memory:**

- `argc`: variable
- `argv`: variable

**Combining domains**

Combination of domains for **different types** (number, pointers, ...) and **different properties** (relational domains for inductive invariants) that can be composed and can communicate.
A classic analyzer (Astrée, Frama-C) has:

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

**Pros and cons:**

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

- unified AST for programs: high-level, extensible, multi-language
- lowering of complex statements dynamically, during analysis
- common signature for all abstract domains
- domain communications, access to preconditions, reductions
- domain organisation in DAGs, sharing abstract information
- more general environment abstractions, handling optional variables

Languages:

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

- 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 ⊗ C.Cells ⊗ C.Strings

Sequence

U.heap ⊗ C.MachineNum ⊗ C.Pointers

Reduced product

U.heap ⊗ U.Intervals ⊗ U.LinearRel

Cartesian product

Universal

C specific
Extensible AST: Universal loops

We use extensible types and distributed iterators.
E.g., universal is a toy-language with only simple while loops

- extend \texttt{stmt\_kind} with AST fragments

\begin{verbatim}
Universal.Ast

type stmt_kind += S\_while of expr * stmt
\end{verbatim}

\begin{verbatim}
let exec stmt man flow =
match stmt_kind stmt with
| S\_while (cond, body) ->
  let i = lfp (fun f -> Flow.join f (man.exec (S\_assume cond) f |> man.exec body)) flow
  Some (man.exec (S\_assume (E\_not cond) i))
| _ ->
  None (* pass-through *)
\end{verbatim}
Extensible AST: Universal loops

We use extensible types and distributed iterators.

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

- extend `stmt_kind` with AST fragments

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

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

```plaintext
S[#][ while (e) s ]X# def = S[#][ ¬e ] (lfp \lambda Y#. X# \cup S[#][ s ] \circ S[#][ e ] Y#)
```

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

Extensible AST: C and Python loops

**C AST**

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

**Python AST**

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

- preserve the high-level AST of the source languages
- reuse universal AST when possible (no S_c_while)
Extensible AST: C and Python loops

**C AST**

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

**Python AST**

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

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

**C iterator**

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

- the iterator transforms the loops into a S_while universal loop and calls the overall iterator recursively
  \[\rightarrow \text{delegate} \text{ the iteration strategy to universal (factor semantics)}\]
Extensible AST: C and Python loops

C AST

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

Python AST

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

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

C iterator

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

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

The AST merges source languages and intermediate languages.
Handling of statements by induction on the syntax:

- \( S^\#[s_1; s_2] X^\# \overset{\text{def}}{=} S^\#[s_2] \circ S^\#[s_1] X^\# \)

- \( S^\#[\text{if (e) s else t}] X^\# \overset{\text{def}}{=} (S^\#[s] \circ S^\#[e] X^\#) \cup^\# (S^\#[t] \circ S^\#[-e] X^\#) \)
Non-local control-flow

Handling of statements by induction on the syntax:

- \[ S^\#[ s_1; s_2 ] X^\# \overset{\text{def}}{=} S^\#[ s_2 ] \circ S^\#[ s_1 ] X^\# \]
- \[ S^\#[ \text{if (e) s else t} ] X^\# \overset{\text{def}}{=} (S^\#[ s ] \circ S^\#[ e ] X^\#) \cup^\# (S^\#[ t ] \circ S^\#[ \neg e ] X^\#) \]
- adding gotos...

```
C AST

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

```
example

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

How can we handle control flow that does not follow the AST structure?
\[ \Rightarrow \] post-conditions are **flows**, containing several continuations.
Flows as post-conditions

- environments $D^\#$ abstract $D \overset{\text{def}}{=} \mathcal{P}(\text{memory state})$
- flows $F^\# \overset{\text{def}}{=} \text{token} \rightarrow D^\#$

```
C goto flows

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

example with flows

x = 12; [T_cur \rightarrow 12]  
if (...) { x++; [T_cur \rightarrow 13] goto 11; [T_goto 11 \rightarrow 13] }  
[T_cur \rightarrow 12, T_goto 11 \rightarrow 13]  
x = 99;  
[T_cur \rightarrow 99, T_goto 11 \rightarrow 13]  
11: [T_cur \rightarrow [13,99]] return x;
```

- $S^\#[\text{goto 1}]X^\# \overset{\text{def}}{=} X^\#[\text{cur} \mapsto \bot, 1 \mapsto X^\#(\text{cur}) \cup^\# X^\#(1)]$
- $S^\#[\text{label 1}]X^\# \overset{\text{def}}{=} X^\#[\text{cur} \mapsto X^\#(\text{cur}) \cup X^\#(1), 1 \mapsto \bot]$

- also useful for break, return, exceptions, long jumps, generators
- *backward* jumps require fixpoint computations
Universal language integer expressions over $\mathbb{Z}$.

- $D \overset{\text{def}}{=} \mathcal{P}(\mathcal{V} \rightarrow \mathbb{Z}) \simeq \mathcal{P}(\mathbb{Z}^{|\mathcal{V}|})$

- $+, -, /, \times$ with mathematical semantics
  (no bit-size, no overflow, no wrap-around)

- natural setup for most numeric domains $D^\#$
  (polyhedra, etc.)
... to C numeric expressions

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

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

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

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

evaluation zones

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

C assignments to universal assignments

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

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

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

... to C numeric expressions

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

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

---

**evaluation zones**

```latex
type zone ::= Z_u_num | Z_c_scalar
```

---

**C assignments to universal assignments**

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

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

---

“evaluation” as **dynamic rewriting** into other expressions
C pointers

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

**Pointer abstraction**

Let $D^\# = (\mathcal{V}_{ptr} \rightarrow \mathcal{P}(\mathcal{V})) \times \mathcal{N}um^\#$

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

```c
char a[10] = "hello";
int i = _mopsa_rand(0,9);
char *p = &(a[i]); // (p \mapsto \{a\}, i \in [0,9] \land \text{offset}(p) = i)
```

⇒ infer relations between pointer offsets and numeric variables
Expression evaluations into DNF

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

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

**eval signature**

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

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

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

**eval signature**

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

**Example:**

evaluate \( *(p+10) \) in \( X\# \) where \( p \in \{ \text{NULL, } \&a, \&b \} \)

return the disjunction:

\[
(error, S\#[[ \text{assume base}(p) = \text{NULL} ]]X\#) \lor \\
(\ast(\&a + 10), S\#[[ \text{assume base}(p) = a ]]X\#) \lor \\
(\ast(\&b + 10), S\#[[ \text{assume base}(p) = b ]]XX\#)
\]

- **locality**: disjunctions are merged at the end of the statement
- **low coupling with other domains** (eval mechanism)
- **conjunctions** are also possible thanks to reductions
  \[ \implies \text{ use disjunctive normal forms} \]
Queries

Two scopes for data-types representing properties:

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

```plaintext
interval query

• ability to evaluate any expression into an interval
• any domain can answer an interval query (intervals, polyhedra, etc.)
• concrete type with a lattice structure (the framework combines the answers from all domains)
• extensible, global data-type
```
Two scopes for data-types representing properties:

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

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

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

- concrete type with a **lattice structure**
  (the framework combines the answers from all domains)

- **extensible**, global data-type
Application of queries:

Reduce the interval domain using interval information from other domains.

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

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

Instead of \( P(V \rightarrow Val) \), abstract \( P(V \rightarrow Val) \)

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

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

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

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

Applications:

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

How to lift $\mathcal{D}_V^\#$, abstracting $\mathcal{P}(V \to \text{Val})$ to $\mathcal{P}(V \rightarrow \text{Val})$?

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

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

1. $L \subseteq U \subseteq V$, lower and upper bounds on variables
2. $X^\# \in \mathcal{D}_U^\#$ a single abstract element over $U$
3. $\gamma(X^\#) \overset{\text{def}}{=} \{ \rho|_W \mid \rho \in \gamma_U(X^\#), L \subseteq W \subseteq U \}$

Example:

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

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

How to lift $D^\#_V$, abstracting $P(V \to \text{Val})$ to $P(V \twoheadrightarrow \text{Val})$?

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

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

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

Example:

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

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

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

Powerful but complex interactions between reduction and evaluation.

- both domains have a different view of the same concrete variables
- evaluation delegates the assignment independently for each domain
- the numeric domain collects both effects

\[ S^\# \[ eval_{cell}(a[i] \leftarrow 12) \] X^\# \land S^\# \[ eval_{smash}(a[i] \leftarrow 12) \] X^\# \]

This is not sound!
Stacked domains: Solution

Powerful but complex interactions between reduction and evaluation.

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

\[
S^\#[(a[0] \leftarrow 12 \lor a[1] \leftarrow 12); a[*] \leftarrow \top] X^\# \land \\
S^\#[a[*] \leftarrow 12; a[0] \leftarrow \top; a[1] \leftarrow \top] X^#
\]

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

- delegates the abstraction of \( n, p, q \) to other domains (evaluation)
- sound reduction with cell and smash domains
Application to C Analysis
C analysis

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

**Goal:** a platform to help prototype new analyses on C codes
Memory abstractions: cell domain

Low-level memory abstraction

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

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

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

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

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

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

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

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

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

Result: we can infer
- as loop invariant: $\text{off}_p = \text{off}_q \leq l_{\text{src}} \leq a_{\text{src}}$
- after the loop: $\text{off}_p = \text{off}_q = l_{\text{src}} \leq a_{\text{src}}$
- raise an alarm if $l_{\text{src}} \geq a_{\text{src}}$ or $l_{\text{src}} \geq a_{\text{dst}}$
- otherwise, we ensure that $l_{\text{dst}} = l_{\text{src}}$. 
Stub contract language

```c
int open (const char *__file, int __oflag, ...);

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

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

int open (const char *__file, int __oflag, ...);
```

**Specification language:**

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

- extracted from Juliet Test Suite (v 1.3) for C/C++
  - CWE476 on null pointer dereferences.
  - CWE369 on divisions by zero
  - CWE190 on integer overflows

- each test has a bad version and a correct version

<table>
<thead>
<tr>
<th>Category</th>
<th>Loc</th>
<th>Tests</th>
<th>Time</th>
<th>Alarms</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE476</td>
<td>25K</td>
<td>522</td>
<td>2mn26s</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>CWE369</td>
<td>109K</td>
<td>1368</td>
<td>7mn20s</td>
<td>372</td>
<td>53%</td>
</tr>
<tr>
<td>CWE190</td>
<td>440K</td>
<td>6840</td>
<td>34mn57s</td>
<td>0</td>
<td>73%</td>
</tr>
</tbody>
</table>

On-going work: analyzing actual C programs from GNU CoreUtils.
Application to Python Analysis
Python 3 language

Highly **dynamic** language:

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

⇒ static analysis is **challenging**, but **rewarding**
Python 3 semantics

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

\[ \Rightarrow \text{the abstract semantics has the same structure as the concrete one} \]
Python 3 semantics

\[
E[e_1 + e_2](f, \epsilon, \Sigma) \overset{\text{def}}{=} \\
\text{let } (f_1, \epsilon_1, \Sigma_1, v_1) = E[e_1](f, \epsilon, \Sigma) \text{ in} \\
\text{let } (f_2, \epsilon_2, \Sigma_2, v_2) = E[e_2](f_1, \epsilon_1, \Sigma_1) \text{ in} \\
\text{if } \text{hasattr}(v_1, \text{add}_-, \Sigma_2) \text{ then} \\
\text{let } (f_3, \epsilon_3, \Sigma_3, v_3) = E[v_1\text{.add}_-(v_2)](f_2, \epsilon_2, \Sigma_2) \text{ in} \\
\text{if } v_3 = \text{NotImpl} \land \text{typeof}(v_1) \neq \text{typeof}(v_2) \text{ then} \\
\text{if } \text{hasattr}(v_2, \text{radd}_-, \Sigma_3) \text{ then} \\
\text{let } (f_4, \epsilon_4, \Sigma_4, v_4) = E[v_2\text{.radd}_-(v_1)](f_3, \epsilon_3, \Sigma_3) \text{ in} \\
\text{if } v_4 = \text{NotImpl} \text{ then } \text{TypeError}(f_4, \epsilon_4, \Sigma_4) \text{ else } (f_4, \epsilon_4, \Sigma_4, v_4) \\
\text{else } \text{TypeError}(f_3, \epsilon_3, \Sigma_3) \\
\text{else if } v_3 = \text{NotImpl} \text{ then } \text{TypeError}(f_3, \epsilon_3, \Sigma_3) \text{ else } (f_3, \epsilon_3, \Sigma_3, v_3) \\
\text{else if } \text{hasattr}(v_2, \text{radd}_-, \Sigma_2) \land \text{typeof}(v_1) \neq \text{typeof}(v_2) \text{ then} \\
\text{let } (f_3, \epsilon_3, \Sigma_3, v_3) = E[v_2\text{.radd}_-(v_1)](f_2, \epsilon_2, \Sigma_2) \text{ in} \\
\text{if } v_3 = \text{NotImpl} \text{ then } \text{TypeError}(f_3, \epsilon_3, \Sigma_3) \text{ else } (f_3, \epsilon_3, \Sigma_3, v_3) \\
\text{else } \text{TypeError}(f_2, \epsilon_2, \Sigma_2)
\]

- **formalize** the concrete semantics
  based on the Python manual and CPython implementation

- **use a denotational-style semantics** (easier to abstract)

- **type-based cases** (eval and DNF are useful)

\[\Rightarrow\] the abstract semantics has the **same structure** as the concrete one
hand-written parser in Menhir
resolves import at parsing time
reuse universal domains: numeric, heap abstractions, loops, etc.
Concrete domains for Python semantics

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

- **environments**: $\mathcal{E} \overset{\text{def}}{=} \mathcal{V} \rightarrow \text{Val}$
- **values**: $\text{Val} \overset{\text{def}}{=} \mathbb{Z} \cup \text{Addr} \cup \{\text{True, False, None, Undef, NotImplemented}\}$
- **heap**: $\mathcal{H} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Obj}$
  $\text{Obj} \overset{\text{def}}{=} \text{String} \rightarrow \text{Val}$
Non-relational value analysis for Python

Follows the concrete semantics:

- **environments**: \( \mathcal{E} \mathbf{#} \overset{\text{def}}{=} \mathcal{V} \rightarrow \mathcal{V} \mathbf{#} \)
- **values**: \( \mathcal{V} \mathbf{#} \overset{\text{def}}{=} \mathbb{Z} \mathbf{#} \times \mathcal{B} \mathbf{#} \times \mathcal{P}(\mathcal{A} \mathbf{#}) \times \mathcal{N} \mathbf{#} \times \mathcal{U} \mathbf{#} \times \mathcal{U} \mathbf{#} \times \mathcal{U} \mathbf{#} \)
  (abstract disjoint unions as tuples)
  \[ \mathcal{N} \mathbf{#} = \{ \bot, \top \}, \mathcal{B} \mathbf{#} = \{ \bot, \top, t, f \} \]
  - \( \mathbb{Z} \mathbf{#} \): non-relational domain (e.g., intervals)
  - \( \mathcal{A} \mathbf{#} \): allocation site abstraction
  - **heap**: \( \mathcal{H} \mathbf{#} \overset{\text{def}}{=} \mathcal{A} \mathbf{#} \rightarrow \mathcal{O} \mathbf{#} \)

**Object abstraction**: \( \mathcal{O} \mathbf{#} \overset{\text{def}}{=} (\mathcal{S} \rightarrow \mathcal{V} \mathbf{#}) \times \mathcal{P}(\mathcal{S}) \)

Attributes can be added to objects dynamically
\( \Rightarrow \) a set of objects can have heterogeneous sets of attributes

- \( \mathcal{S} \rightarrow \mathcal{V} \mathbf{#} \) maps all possible attributes to their value
- \( \mathcal{P}(\mathcal{S}) \): attributes that are guaranteed to exist in all objects
  necessary to prove that \texttt{AttributeError} cannot occur
Built-ins in Python

Built-in data-structures:

- **Strings**: bounded sets of constant strings, or \( T \)
- **Lists**: one summary element, and a length
- **Dictionaries**: as objects, or with a summary element

Example: model a list access \( l[i] \)

- **C1**: `isinstance(l, list) \land instanceof(i, int)`
- **C2**: \(-\text{len}(l) \leq i < \text{len}(l)\)
- **C3**: \(\text{len}(l) = 1\)

<table>
<thead>
<tr>
<th>case</th>
<th>evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\neg C1</td>
<td>TypeError</td>
</tr>
<tr>
<td>( C1 \land \neg C2 )</td>
<td>IndexError</td>
</tr>
<tr>
<td>( C1 \land C2 \land C3 )</td>
<td>summary variable ( l )</td>
</tr>
<tr>
<td>( C1 \land C2 \land \neg C3 )</td>
<td>weak copy of summary variable ( l )</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Regression test</th>
<th>Lines</th>
<th>Tests</th>
<th>Time</th>
<th>✔</th>
<th>✗</th>
<th>✭</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_augassign</td>
<td>273</td>
<td>7</td>
<td>645ms</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>85.71%</td>
</tr>
<tr>
<td>test_baseexception</td>
<td>141</td>
<td>10</td>
<td>20ms</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>60.00%</td>
</tr>
<tr>
<td>test_bool</td>
<td>294</td>
<td>26</td>
<td>47ms</td>
<td>12</td>
<td>0</td>
<td>14</td>
<td>46.15%</td>
</tr>
<tr>
<td>test_builtin</td>
<td>454</td>
<td>21</td>
<td>360ms</td>
<td>3</td>
<td>0</td>
<td>18</td>
<td>14.29%</td>
</tr>
<tr>
<td>test_contains</td>
<td>77</td>
<td>4</td>
<td>418ms</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>25.00%</td>
</tr>
<tr>
<td>test_int_literal</td>
<td>91</td>
<td>6</td>
<td>29ms</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>test_int</td>
<td>218</td>
<td>8</td>
<td>88ms</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>37.50%</td>
</tr>
<tr>
<td>test_list</td>
<td>106</td>
<td>9</td>
<td>88ms</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>33.33%</td>
</tr>
<tr>
<td>test_unary</td>
<td>39</td>
<td>6</td>
<td>11ms</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>33.33%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Performance benchmark</th>
<th>Lines</th>
<th>Interval</th>
<th>Octagon</th>
<th>Polyhedra</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>37</td>
<td>1.5s</td>
<td>☑️ 4.8s</td>
<td>☑️ 3.4s</td>
</tr>
<tr>
<td>fannkuch</td>
<td>37</td>
<td>0.8s</td>
<td>X(3) 4.7s</td>
<td>X(1) 3.3s</td>
</tr>
<tr>
<td>nbody</td>
<td>66</td>
<td>1.0s</td>
<td>X(2) 10min1s</td>
<td>X(2) ∞</td>
</tr>
</tbody>
</table>
Types as abstraction of values

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

Python mixes:
- **nominal typing**: `isinstance`
- **duck typing**: `hasattr`

Both can be resolved in the abstraction $Val^\sharp$:
- **nominal typing**: value of the attribute `__class__`
- **duck typing**: presence of a specific attribute in $Obj^\sharp$
On-going work:

More scalable abstraction remembering only type information

- sets of the types of the values stored in each variable
  \[ V \rightarrow \mathcal{P}(\text{types}) \]

- top-down, flow-sensitive inference by propagation of abstract values
  \[ \implies \text{more of an Abstract Interpretation technique than regular typing} \]

- types for built-in objects: \texttt{List[int]}

- types for nominal and duck typing: \texttt{Instance[class,attrs]}

- bounded parametric polymorphism: \texttt{List[\alpha], \alpha \in \{\ldots\}}
  \[ \implies \text{relational typing domain: } V:\texttt{List[\alpha]} \land W:\texttt{List[\alpha]} \land \alpha = \beta \]
Benchmarks for Python type analysis

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Fritz &amp; Hage</th>
<th>Pytype</th>
<th>Typpete</th>
<th>MOPSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis method</td>
<td>Dataflow analysis</td>
<td>Unclear</td>
<td>SMT-solver</td>
<td>AI</td>
</tr>
<tr>
<td>class_attr_ok</td>
<td>✓</td>
<td>x</td>
<td>✱</td>
<td>✓</td>
</tr>
<tr>
<td>class_pre_store</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>default_args_class</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>except_clause</td>
<td>✱</td>
<td>✱</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fspath</td>
<td>✱</td>
<td>✱</td>
<td>✱</td>
<td>✓</td>
</tr>
<tr>
<td>magic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>polyfib</td>
<td>✱</td>
<td>x</td>
<td>✱</td>
<td>✱</td>
</tr>
<tr>
<td>poly_lists</td>
<td>✱</td>
<td>✓</td>
<td>✱</td>
<td>✓</td>
</tr>
<tr>
<td>vehicle</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>widening</td>
<td>✓</td>
<td>x</td>
<td>✱</td>
<td>✓</td>
</tr>
</tbody>
</table>
Conclusion
Conclusion

Features:

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

Future work:

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